DEVELOPING, TUNING, AND USING SCHEMA MATCHING SYSTEMS

BY

YOONKYONG LEE

DISSE ssrATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:
Associate Professor AnHai Doan, Chair
Professor Geneva Belford
Professor Marianne Winslett
Associate Professor ChengXiang Zhai
Abstract

This dissertation studies the schema matching problem that finds semantic correspondences (called matches) between disparate data sources. Examples of semantic matches include “location = address” and “name = concat(first-name,last-name).”

Schema matching is one of the key challenges for many data sharing and exchange applications. 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. Despite significant progress, many challenges remain. These include discovering complex matches, a prevalent problem in practice, tuning a matching system, and deploying a matching system effectively in an application.

In this dissertation, we develop solutions for the three challenges mentioned above. First, we develop a system that discovers both one-to-one and complex matches and provides a novel explanation facility that helps users analyze matches. Next, we develop a framework that automatically tunes multi-component matching systems by synthesizing a collection of matching scenarios. Finally, we show that we can efficiently exploit discovered semantic matches without extra user effort in certain applications.
To my family
Acknowledgments

During my graduate years, I have been truly fortunate to have the greatest mentors. I would especially like to thank my adviser, AnHai Doan, who demonstrated how to do good research, challenged me to grow into an independent researcher, and helped me struggle through graduation years. Thank you. I thank Kevin Chang, Jiawei Han, Mari- anne Winslett, and ChengXiang Zhai for the valuable lessons from classes and seminars. Thank you also to my committee – AnHai Doan, Geneva Belford, Marianne Winslett, and ChengXiang Zhai – who gave helpful comments and feedback, and helped me improve the quality of this dissertation.

I deeply thank Korea Ministry of Information and Telecommunication for supporting me with a fellowship for four years. I also thank my group mates, Pedro DeRose, Robert McCann, Mayssam Sayyadian, Robin Dhamankar, Rishi Sinha, and Warren Shen, for being great project partners, discussing research, and helping with practice talk.

I thank all my roommates, Seung-Won Hwang, Sooyoung Kim, Sun-A Kim, Jeong-Ah Shin, and Younhee Ko, for making the life in Urbana-Champaign fun. I thank my friends in Urbana-Champaign for being supportive, especially Heeyoun Cho, Youngjoo Lee, Yoonsook Mo, Soorim Noh, Haekyung Shin, and Insun Yoon. I thank all my friends in the computer science department. I thank Yoonsun Chung for her constant encouragement and precious friendship.

Finally, I thank my parent and my sister for their endless love and support. Without their support, I would not have finished, or would have lost my mind, or both. This dissertation is dedicated to you.
# Table of Contents

List of Tables ........................................................................ viii
List of Figures .......................................................................... ix

Chapter 1  Introduction .......................................................... 1
1.1 Schema Matching: Discovering Semantic Matches ............... 1
1.2 Challenges on the Practicality of Matching Solutions .......... 2
1.3 Goals of the Dissertation .................................................. 5
1.4 Overview of the Solutions ............................................... 5
  1.4.1 Developing a Schema Matching System to Discover Complex Matches 6
  1.4.2 Tuning Schema Matching Systems ................................... 7
  1.4.3 Using Schema Matching Systems ................................... 9
1.5 Contributions of the Dissertation ...................................... 9
1.6 Outline ............................................................................ 10

Chapter 2  Developing a Schema Matching System to Discover Complex Matches 11
2.1 Introduction ..................................................................... 11
2.2 Problem Definition ........................................................ 15
2.3 The iMAP Architecture .................................................. 17
  2.3.1 Candidate Match Generation ....................................... 18
  2.3.2 The Similarity Estimator ............................................ 25
  2.3.3 The Match Selector .................................................. 26
2.4 Exploiting Domain Knowledge ......................................... 27
2.5 Generating Explanations ................................................ 30
  2.5.1 Types of User Questions ............................................ 31
  2.5.2 The Explanation Module ........................................... 32
2.6 Empirical Evaluation ....................................................... 35
  2.6.1 Overall and 1-1 Matching Accuracy ............................. 37
  2.6.2 Complex Matching Accuracy ..................................... 38
  2.6.3 Explaining Match Predictions .................................... 44
2.7 Summary ........................................................................ 46

Chapter 3  Tuning Schema Matching Systems .......................... 48
3.1 Introduction ..................................................................... 48
3.2 The Match Tuning Problem ............................................ 53
### Chapter 3 Using Matching Systems

#### 3.2 Modeling 1-1 Matching Systems
- Tuning of Matching Systems

#### 3.3 The eTuner Approach
- Automatic Workload Creation
- User-Assisted Workload Creation

#### 3.4 Tuning with the Synthetic Workload
- Staged Tuning
- Tuning Subsystems of $M$
- Tuning to Select Features

#### 3.5 Empirical Evaluation
- Experimental Settings
- The Need for Tuning
- "Quick and Dirty" Tuning
- Domain- & Source-Dependent Tuning
- Tuning with eTuner
- Sensitivity Analysis
- Additional Experiments

#### 3.6 Summary

### Chapter 4 Using Schema Matching Systems

#### 4.1 Employing a Schema Matching System in an Application

#### 4.2 QSM: Extracting Structure from Unstructured Data
- The QSM Architecture
- Data Representation
- Querying Structures

#### 4.3 Benefits of QSM
- Scenario 1: Find capital cities whose population is greater than 10,000,000 people
- Scenario 2: Find capital cities whose elevation is greater than 1,000
- Scenario 3: Find capital cities whose area is greater than 600 km²
- Scenario 4: Find capital cities having famous palaces
- Scenario 5: Find capital cities in which subway is one type of public transportation
- Scenario 6: Find capital cities which are sister cities to Seoul, the capital of South Korea

#### 4.4 Efficiently Exploiting Semantic Matches
- Stages in Query Processor
- Schemes to Exploit Matches Efficiently

#### 4.5 Empirical Evaluation
- SQL-Like Query
- Focused Keyword Query

#### 4.6 Summary
List of Tables

2.1 Implemented searchers in iMAP ........................................... 21
2.2 Real-world domains for our experiments. ........................... 35
3.1 The performance of the staged tuner ................................. 88
List of Figures

1.1 An example of multi-component matching systems. .......................... 8

2.1 The schemas of two relational databases on house listing, and the semantic mappings between them. .................................................. 15
2.2 The iMAP architecture .................................................................. 17
2.3 A sample fragment of the dependency graph as generated by iMAP. ..... 33
2.4 Top-1 overall matching accuracy. .................................................. 37
2.5 Top-1 (top row) and top-3 (bottom row) matching accuracy for complex matches. ............................................................... 39
2.6 Top-1 (top row) and top-3 matching (bottom row) accuracy for partial complex matches. ...................................................... 39
2.7 Performance sensitivity .................................................................. 40
2.8 Generated explanation for pname vs. concat(first-name,last-name). .. 45
2.9 Generated explanation for num-rooms. ............................................. 45

3.1 An example of multi-component matching systems. ......................... 51
3.2 The LSD system: (a) library of matching components, (b) execution graph, and (c) sample knobs. ..................................................... 53
3.3 Execution graphs of (a) the SimFlood matching system, and (b) the LSD-SF matching system. ...................................................... 55
3.4 An illustration of the working of the LSD system. ............................ 56
3.5 The eTuner architecture. ............................................................... 61
3.6 Perturbing schema S to generate two schemas U and V₁ and the correct matches between them. .................................................. 62
3.7 High-level description of the workload generator. ........................... 66
3.8 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”. ......................................................... 70
3.9 High-level description of the wrapper feature selection method (called stepwise selection in [25]). ..................................................... 71
3.10 An example taxonomy for the Naive Bayes matcher. ....................... 71
3.11 High-level description of the wrapper method, as adapted to feature selection for text-based matchers. ........................................ 72
3.12 High-level description of the Relief-F algorithm [25], as adapted to feature selection in eTuner. ....................................................... 73
3.13 (a) Real world domains, (b) matching systems for our experiments, (c) a sample schema from Real Estate, and (d) a sample schema from Inventory.

3.14 The library of matching components of LSD.

3.15 The library of matching components of iCOMA.

3.16 The library of matching components of SimFlood.

3.17 The library of matching components of LSD-SF.

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

3.19 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.

3.20 The performance of the workload generator.

4.1 Wikipedia page of Seoul

4.2 System architecture

4.3 Default keyword search interface

4.4 QSM data structure

4.5 Mapping Table

4.6 SQL-like query interface

4.7 Focused keyword search interface

4.8 Query examples: $Q_1$ and $Q_2$

4.9 Snapshot of the back-end database

4.10 Re-written $Q_1$ and $Q_2$

4.11 Re-written $Q'_2$ after applying the Scheme 2

4.12 Re-written $Q'_2$ after applying the Scheme 3

4.13 The performance of the SQL-like queries

4.14 The performance of the focused keyword query
Chapter 1

Introduction

This dissertation studies schema matching, the problem of discovering semantic correspondences called matches between two disparate data sources. Example matches include “location = address” and “name = concat(first-name,last-name)”.

We begin this chapter by providing an overview of the schema matching problem (Section 1.1). Next, we describe key challenges in schema matching and show that while there has been significant progress, schema matching remains a difficult problem (Section 1.2). We then outline the solutions we have developed to address these challenges (Section 1.3 - 1.4). Finally, we list the contributions and offer a road map for the rest of the dissertation (Section 1.5 - 1.6).

1.1 Schema Matching: Discovering Semantic Matches

Semantic mappings specify the relationships among data stored in disparate sources. Establishing semantic mappings is a crucial problem in any data sharing architecture, such as a data integration system, data warehouse, peer-data management system or web-service based architecture. Data sharing systems are crucial for supporting a wide range of applications, such as enterprise data integration, scientific collaborations, data management on the WWW, and cooperation between government agencies.

Currently, semantic mappings are created by hand (typically supported by advanced graphical user interfaces), and in practice are extremely tedious and error-prone [84].

The problem of semi-automatically creating mappings has received significant at-
tention recently in both the database and AI communities (see [77] for a recent survey, and [44, 41] for several works since). The bulk of this previous work focused on the first phase of mapping, called schema matching. A match between two schemas specifies semantic correspondences between elements of both schemas [77]. These correspondences are later elaborated (e.g., using a system such as Clio [92]) to generate a mapping. For example, the mapping can be in the form of an SQL query that translates data from one source to another. It should be noted that both the process of generating matches and mappings are expected to involve human input.

This dissertation focuses on the first phase called schema matching. Schema matching finds semantic correspondences called matches between the schema of disparate data sources. Example matches include “location = address” and “name = concat(first-name, last-name)”.

There are two kinds of matches: one-to-one matches and complex matches. A one-to-one match specifies that the element location in one schema matches the area in the other. A complex match specifies that a combination of attributes in one schema corresponds to a combination in the other. For example, it may specify that name = concat(first-name, last-name).

1.2 Challenges on the Practicality of Matching Solutions

Most of the attention on schema matching has focused on developing matching solutions that discover accurate semantic matches semi-automatically (see [77, 34, 29, 69, 7] for recent surveys). Each individual matching technique has its own strengths and weaknesses [77, 31, 29, 28]. Hence, increasingly, matching tools are assembled from multiple components, each employing a particular matching technique [77, 31, 34, 29, 28].

At its core, discovering matches semi-automatically is a serious challenge because of semantic heterogeneity. Matching two schemas $S$ and $T$ requires deciding if two elements $s$
of $S$ and $t$ of $T$ represent the same real-world concept. While humans may be able to easily reason if two elements match, the semantic heterogeneity makes it difficult for machines to interpret. For example, $s$ and $t$ can have the same name but represent different concepts. The opposite is also possible: $s$ and $t$ can have different names but represent the same concept.

However, while developing semi-automatic matching solutions is certainly difficult, designing practical matching solutions involves many other difficult challenges. In the rest of this section, we discuss key challenges, focusing on the practicality of a schema matching system, and describe the state of the art with respect to these challenges.

First, to date, the work on schema matching has focused on discovering one-to-one matches between schema elements (e.g., relation attributes, XML tags). While one-to-one matches are common, relationships between real-world schemas involve many complex matches. In fact, in the schemas we consider in Chapter 2, complex matches compose up to half of the matches. Hence, the development of techniques to semi-automatically construct complex matches is crucial to any practical mapping effort.

Creating complex matches is fundamentally more difficult than one-to-one matches for the following reason: while the number of candidate one-to-one matches between a pair of schemas is bounded by the product of the sizes of the two schemas, the number of candidate complex matches is not. There is an unbounded number of functions for combining attributes in a schema, and each one of these could be a candidate match. Hence, in addition to the inherent difficulties of generating a match with which to start, the problem is exacerbated by having to examine an unbounded number of match candidates.

Second, although the multi-component nature of matching solutions is powerful in that it makes matching systems highly extensible and (with sufficient skills) customizable to a particular application domain [12, 78], 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 produce inferior accuracy. Indeed, in Chapter 3 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 [28, 27, 59, 52, 77].)

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 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 [25], a task that is difficult even for learned experts [25]. Furthermore, since we rarely know the ground truth for matches, it is not clear how to compare the quality of knob configurations.

For all of the 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 in which they are attractive in practice.

Finally, in order to build a practical matching system, it is not enough to only develop a matching system. A developer also needs to plug in the system in an application and decide how to use the discovered semantic matches. As we discussed in Section 1.1, in traditional data-sharing applications, there are two steps in creating semantic mappings. The schema matching problem is the first step of the process, and the results of a matching system are elaborated in the next step. While both processes are expected to involve human input, the second step requires more human effort than the first step. Most works on the second step focus on providing a framework which can efficiently handle user feedback [92]. Thus, after discovering semantic matches semi-automatically, a significant amount of interaction is still required. Therefore, in deciding how to use the discovered
semantic matches, one natural question to ask is, “Is there an application in which we can directly exploit matches?”

To answer this question, we must examine applications in which we directly exploit generated matches. As far as we know, no works have explored this issue. We believe studying this issue expands the practicality of matching systems.

1.3 Goals of the Dissertation

In this dissertation, we aim to develop solutions to the three key challenges mentioned above. Specifically, our goals are as follows:

- **Build a matching system to semi-automatically discover both one-to-one and complex matches.** This system should discover both types of matches accurately and effectively. It should also make it easier for system administrators to analyze the returned results.

- **Develop a framework to tune a matching system semi-automatically.** Ideally, the framework should tune a matching over future matching scenarios for which the ground truth matches are known. Furthermore, given matching scenarios, the framework must efficiently search for the best tuning configuration.

- **Explore applications which can directly exploit semantic matches.** As a first step towards understanding how to use a schema matching system in practice, another goal of this dissertation is to explore the question, “Can we directly use semantic matches in an application?” Then, if we can, we wish to explore the question, “How can we exploit matches efficiently in the application?”

1.4 Overview of the Solutions

An underlying theme of this dissertation is the *practicality* of schema matching systems. We now outline our solutions for the goals above.
1.4.1 Developing a Schema Matching System to Discover Complex Matches

As we noted, discovering complex matches requires examining an unbounded number of match candidates. To address this problem, we view the generation of complex matches as a *search* in the space of possible matches. To search the space effectively, we employ a set of search modules, called *searchers*, each of which considers a meaningful subset of the space, corresponding to specific types of attribute combinations.

As examples of searchers, a *text searcher* may consider only matches that are concatenations of text attributes, while a *numeric searcher* may consider combining attributes with arithmetic expressions. A *schema mismatch* searcher examines complex matches that involve the data instances of one schema and the schema elements of the other (such complex matches have been observed to be very common in practice [62]). Finally, a *date searcher* focuses on complex matches that involve date attributes, such as \( \text{date} = \text{concat(month," /",year)} \).

We use *beam search* [80] to control the search through the space of candidate matches. To *evaluate* the quality of each match candidate, we employ a set of techniques, including machine learning, statistics, and heuristic methods. Since the number of match candidates is often infinite, a key challenge in adapting the search to the matching context is that we do not know when the best match has been found, and thus the search should be terminated. We develop a simple termination criterion based on the diminishing-returns principle and show that it is often very effective in practice.

Given the matches produced by the search modules, we evaluate their quality further, using criteria that are impractical to be employed at the searcher level. For example, a numeric searcher uses mathematical transformations commonly employed in the equation discovery area [85] to quickly generate a ranked list of match candidates. We then re-rank the candidates, using also the *name similarity* between the attributes involved in a match. In the final step, we select the best matches from the re-ranked candidates, taking
into account domain knowledge and integrity constraints.

As schema matching systems employ more sophisticated techniques, the reasons for the predictions they make become rather involved. The complexity of the decisions is even more pronounced in the context of complex matches, in which the prediction for a complex match may depend on other predictions made for simpler or one-to-one matches.

We introduce a new feature that helps a human designer interacting with the system. We show how the system can offer an explanation of a predicted match, and we consider several fundamental types of explanations, such as knowing why a particular match is or is not created, and why a certain match is ranked higher than another.

### 1.4.2 Tuning Schema Matching Systems

Before developing a framework for tuning a matching system, we must first define the tuning problem. We view a matching system as a combination of matching components. Figure 1.1.a shows a matching system which has \((n + 2)\) components: \(n\) matchers, one combiner, and one selector (Chapter 3 describes these components in detail).

To the user, the components are blackboxes with “exposed knobs,” whose values can be adjusted. For example, a knob allows the user to set a threshold \(\alpha\) such that two schema attributes are declared matched if and only if their similarity score exceeds \(\alpha\). 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, 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 \(M\), how can we tune \(M\) 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 [77, 31].

Tuning the system $M$ (mentioned above) amounts to searching for the “best” knob configuration for matches to $S$. The quality of a particular knob configuration of $M$ is defined as an aggregate accuracy of the matching system when applied to that configuration. Accuracy metrics exist (e.g., precision, recall, and combinations thereof [27]). How can they be evaluated? How can we find a corpus of match problems in which ground truth (i.e., “true” matches) is known? This is clearly a major challenge for any effort toward 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 Figure 1.1.b to EMP, and the rule “replacing ,000 with K” to the column salary of this table. We note that these rules are created only once, independent of any schema $S$.

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

The space of knob configurations is often huge or infinite, making exhaustive searches impractical. Hence we implement a sequential greedy approach, denoted staged tuning. Consider the matching system $M$ in Figure 1.1.a. Here, we first tune each of the matchers $1 \ldots n$ in isolation, then tune the combination of the combiner and the matchers, assuming the knobs of the matchers have been set. Finally, we tune the entire matching system, assuming that the knobs of the combiner and matchers have been set. Many different types of knobs exist (e.g., discrete, continuous, set valued, ordered, etc.), each raising a different tuning challenge.

### 1.4.3 Using Schema Matching Systems

As we discussed in Section 1.2, exploiting discovered semantic matches directly in an application seems like an idea that is worth exploring. In this dissertation we outline a way to do so. First, we identify an application which exploits semantic matches without going through the second step. Next, we argue that exploiting semantic matches is useful in the specific application. We then turn our attention to the issue of exploiting matches efficiently.

### 1.5 Contributions of the Dissertation

This dissertation develops solutions for the challenges of the schema matching problem in terms of practicality. Specifically, this dissertation makes the following contributions:

- We develop a system discovering both one-to-one and complex matches. While complex matches are prevalent in practice, the vast majority of the research on schema matching has focused on one-to-one matches.
• This dissertation presents the first work that addresses the topic of tuning a schema matching system. The multi-component nature of current matching solutions requires tuning before deploying in practice, but currently matching systems are tuned manually.

• We provide the first study on directly using the semantic matches in practice. Traditionally, discovered matches are elaborated into mappings through a significant amount of user interactions, and then the final mappings are exploited in an application.

1.6 Outline

In the rest of this dissertation, we address the three challenges in schema matching that we described in Section 1.2. In Chapter 2, we describe iMAP, a matching system that discovers both one-to-one and complex matches. Then, in Chapter 3, we explain how to semi-automatically tune a matching system. Finally, in Chapter 4, we explore how to exploit the results of a schema matching system in practice.
Chapter 2

Developing a Schema Matching System to Discover Complex Matches

In this chapter, we describe the iMAP system that semi-automatically discovers both 1-1 and complex matches. We first provide an overview of this work in Section 2.1. We define the matching problem we are focused on (Section 2.2), and then describe our solution (Section 2.3-2.5). We then describe our experimental evaluation (Section 2.6) and summarize the chapter (Section 2.7).

2.1 Introduction

Creating semantic matches between disparate data sources is fundamental to numerous data sharing efforts. Manually creating matches is extremely tedious and error-prone. Hence many recent works have focused on automating the matching process. To date, however, virtually all of these works deal only with one-to-one (1-1) matches, such as address = location, which specify that element address in one schema matches location in the other.

While 1-1 matches are common, relationships between real-world schemas involve many complex matches. A complex match specifies that a combination of attributes in one schema corresponds to a combination in the other. For example, it may specify that address = concat(city, state) and room-price = room-rate * (1 + tax-rate). In fact, in the schemas we consider in Section 2.6, complex matches compose up to half of the matches. Hence, the development of techniques to semi-automatically construct complex matches

---

1This work has been previously published in a SIGMOD 2004 paper [24].
is crucial to any practical mapping effort.

Creating complex matches is fundamentally harder than 1-1 matches for the following reason. While the number of candidate 1-1 matches between a pair of schemas is bounded (by the product of the sizes of the two schemas), the number of candidate complex matches is not. There are an unbounded number of functions for combining attributes in a schema, and each one of these could be a candidate match. Hence, in addition to the inherent difficulties in generating a match to start with, the problem is exacerbated by having to examine an unbounded number of match candidates.

This chapter describes the iMAP system which semi-automatically discovers both 1-1 and complex matches between database schemas. Currently, iMAP considers matches between relational schemas, but the ideas we offer can be generalized to other data representations. Developing iMAP required several innovations.

**Generating Matches:** To address the problem of examining an unbounded number of match candidates, we view the generation of complex matches as a search in the space of possible matches. To search the space effectively, we employ a set of search modules, called searchers, each of which considers a meaningful subset of the space, corresponding to specific types of attribute combinations.

As examples of searchers, a text searcher may consider only matches that are concatenations of text attributes, while a numeric searcher considers combining attributes with arithmetic expressions. A schema mismatch searcher examines complex matches that involve the data instances of one schema and the schema elements of the other (such complex matches have been observed to be very common in practice [62]). Finally, a date searcher focuses on complex matches that involve date attributes, such as date = concat(month,”/”year).

We use beam search [80] to control the search through the space of candidate matches. To evaluate the quality of each match candidate, we employ a set of techniques, including machine learning, statistics, and heuristic methods. Since the number of match candidates
is often infinite, a key challenge in adapting search to the matching context is that we do not know when the best match has been found and thus the search should be terminated. We develop a simple termination criterion based on the diminishing-returns principle and show that it is often very effective in practice.

Given the matches produced by the search modules, iMAP evaluates their quality further, using criteria that are impractical to be employed at the searcher level. For example, a numeric searcher uses mathematical transformations commonly employed in the equation discovery area [85] to quickly generate a ranked list of match candidates. iMAP then re-ranks the candidates, using also the name similarity between the attributes involved in a match. In the final step, iMAP selects the best matches from the re-ranked candidates, taking into account domain knowledge and integrity constraints.

**Exploiting Domain Knowledge:** Several recent works [31, 28, 53] have noted the benefits of exploiting domain knowledge for schema matching. Intuitively, domain knowledge (e.g., keys) are useful for pruning some candidate matches. We show that in the context of complex matches the potential benefits of using domain knowledge are even greater: not only can domain knowledge be used to evaluate the accuracy of a proposed match, but it can also be used to prune which match candidates are even considered in the search phase.

In addition to exploiting domain knowledge in the form of integrity constraints and knowledge gleaned by learning from previous matches, iMAP exploits two new kinds of domain knowledge. First, if the databases being matched share some tuples, iMAP can utilize this overlap data to discover complex matches. Second, iMAP also exploits external data in the domain. For example, it can mine real estate listings to learn that the number of real estate agents in a specific area is bounded by 50. Now given the match `agent-name = concat(first-name, last-name)`, where `first-name` and `last-name` belong to the homeowner, iMAP can examine the data instances associated with the match to realize that `concat(first-name, last-name)` yields hundreds of distinct names, and hence is unlikely to
Finally, one of the important aspects of iMAP is that it tries to use domain knowledge as early as possible, in order to prune the consideration of matching candidates.

**Explaining Match Predictions:** As schema matching systems employ more sophisticated techniques, the reasons for the predictions they make become rather involved. The complexity of the decisions is even more pronounced in the context of complex matches, where the prediction for a complex match may depend on other predictions made for simpler or 1-1 matches.

In iMAP we introduce a new feature that helps a human designer interacting with the system. We show how the system can offer an *explanation* of a predicted match, and we consider several fundamental types of explanations, such as knowing why a particular match is or is not created, and why a certain match is ranked higher than another.

In summary, this chapter makes the following contributions:

- An architecture for semi-automatically discovering complex matches that combines search through a set of candidate matches and methods for evaluating each match in isolation, and a set of matches as a whole.

- Uses of new kinds of domain knowledge (overlap data and mining external data), and applying the knowledge as early as possible in the matching process.

- A mechanism for explaining the decisions made by the matching system.

- The iMAP system which embodies all these innovations, and a set of experiments on real-world schemas that illustrate the effectiveness of the system. Our experiments show that we can correctly match 43-92% of the complex matches in the schemas we considered.
2.2 Problem Definition

We discuss schema matching in terms of relational schemas, but the ideas we offer here carry over to other data representations (e.g., matching XML schemas and DTDs). As a running example, consider the two relational schemas $S$ and $T$ in Figure 2.1. Both databases store house listings and are managed by two different real-estate companies. The schema of database $T$, for example, has one table, LISTINGS, whereas database $S$ stores its data in two tables, HOUSES and AGENT.

Suppose the two real-estate companies have decided to merge. To cut costs, they eliminate database $S$ by transferring all house listings from $S$ to database $T$. Such data transfer is not possible without knowing the semantic mappings between the relational schemas of the databases. Below we show some of the mappings for the individual attributes of $T$, using SQL notation. Together, they specify how to create tuples for $T$ from data in $S$. In general, a variety of approaches have been used to specify semantic mappings (e.g., SQL, XQuery, GAV, LAV, GLAV [48]).
area = SELECT location from HOUSES
agent-address = SELECT concat(city, state) FROM AGENTS
list-price = SELECT price * (1 + fee-rate)

FROM HOUSES, AGENTS
WHERE agent-id = id

The process of creating mappings typically proceeds in two steps. In the first step, called schema matching, we find matches (a.k.a. correspondences) between elements of the two schemas. In the second step we elaborate the matches to create query expressions (as above) that enable automated data translation or exchange. The majority of the work in the area has considered algorithms for schema matching, with the significant exception of Clio [92], which is a nice example of a system that studies the second step of the process. We note that both steps of schema mapping may involve interaction with a designer. In fact, the goal of a schema mapping system is to provide a design environment where a human can quickly create a mapping between a pair of schemas. The human builds on the system’s suggestions where appropriate, and provides the system with feedback to direct it to the appropriate mapping.

There are two kinds of schema matches. The first, and the topic of the vast majority of past works on schema matching, is 1-1 matches. Such matches state that there is a correspondence between a pair of attributes, one in each schema. For example, attribute area in T corresponds to attribute location in table HOUSES of S.

The second kind, complex matches, specify that some combination of attributes in one schema corresponds to a combination in the other. In our example, an instance of agent-address in T is obtained by concatenating an instance of city and an instance of state in table AGENTS of schema S.

Complex matches may involve attributes from different tables. For example, list-price is obtained by the following combination of attributes: price * (1 + fee-rate). However, in order to obtain the appropriate pair of price and fee-rate, we need to specify that tables
HOUSES and AGENTS be joined by HOUSES.agent-id = AGENTS.id. In fact, discovering such join relationships was usually postponed to the second step of schema mapping [92].

In the following sections we describe the iMAP system which semi-automatically discovers complex matches for relational data. Initially, our goal was to discover complex matches that involve only attributes in a single table. However, by casting the problem of finding complex matches as search, we are sometimes able to find matches that combine attributes from multiple tables (and suggesting the appropriate join path, see Section 2.3.1). The key challenge that iMAP faces is that the space of possible match candidates is unbounded, corresponding to all the possible ways of combining attributes in expressions.

2.3 The iMAP Architecture

In our explanation of iMAP, we assume we are trying to find matches from a source schema (in our case $S$) to a target schema ($T$). In practice, we would typically generate matches in both directions.
The iMAP architecture is shown in Figure 2.2. It consists of three main modules: match generator, similarity estimator, and match selector. The match generator takes as input two schemas $S$ and $T$. For each attribute $t$ of $T$, it generates a set of match candidates, which can include both 1-1 and complex matches. As we explain below, the generation is guided by a set of search modules. The similarity estimator then computes for each match candidate a score which indicates the candidate’s similarity to attribute $t$. Thus, the output of this module is a matrix that stores the similarity score of $\langle$target attribute, match candidate$\rangle$ pairs. Finally, the match selector examines the similarity matrix and outputs the best matches for the attributes of $T$.

During the entire matching process, the above three modules also exploit domain knowledge and data to maximize matching accuracy, and interact with an explanation module to generate explanations for matches. The rest of this section describes the three modules. We discuss the use of domain knowledge in Section 2.4 and the explanation facility in Section 2.5.

### 2.3.1 Candidate Match Generation

Given an attribute in the target schema, the match generator must quickly discover a relatively small set of promising candidate matches for the attribute. The key idea underlying the match generator is that it recasts this discovery process as a search through the space of possible match candidates.

The space of match candidates can be extremely large or even infinite, since any expression for combining attributes of the source schema can potentially be a match candidate. Some combinations make sense (e.g., $\text{concat(city, state)}$), and others do not (e.g., $(\text{price} - \text{agent-id}) \times 2$). The match generator addresses this challenge by employing a set of special-purpose searchers. A searcher explores a specialized portion of the search space, based on knowledge of particular combination operators and attribute types.

The set of best match candidates is given by the union of the match candidates returned
by the specialized searchers. The following example illustrates searchers:

**Example 2.3.1** Consider the text searcher and the numeric searcher. Given a target attribute \( t \), the text searcher examines the space of all matches that are either attributes or concatenations of attributes in source schema \( S \), to find a small set of matches that best match attribute \( t \). The text searcher accomplishes this by analyzing the textual properties of the attributes of the two schemas. In the case of target attribute \texttt{agent-address} (Figure 2.1), this searcher may return the following matches, in decreasing order of confidence: \texttt{concat(city,state)}, \texttt{location}, \texttt{concat(name,state)}.

The numeric searcher exploits the values of numeric attributes to find matches that are arithmetic expressions over the attributes of source schema \( S \). Given target attribute \texttt{list-price} in Figure 2.1, this searcher may return the following matches: \texttt{price} \( \ast \left(1 + \texttt{fee-rate}\right)\), \texttt{price}, \texttt{price + agent-id}. □

In general, a searcher is applicable to only certain types of attributes. For example, the text searcher examines the concatenations of attributes, and hence is applicable to only textual ones. Except for data-type information that is available in the schema, this searcher employs a set of heuristics to decide if an attribute is textual. The heuristics examine the ratio between the number of numeric and non-numeric characters, and the average number of words per data value. As another example, the numeric searcher examines arithmetic expressions of attributes, and hence is applicable to only numeric attributes.

There are two main benefits to using multiple searchers in iMAP. First, the searchers enable considering a small and meaningful part of the space of candidate matches. Second, the system is easily extensible with additional searchers. For example, if we later develop a specialized searcher that finds complex matches for address attributes, then we can just plug the searcher into the system. In addition, particular domains are likely to benefit from specialized searchers for the domain.
The Internals of a Searcher

Applying search to candidate generation requires addressing three issues: search strategy, evaluation of candidate matches, and termination condition.

**Search Strategy:** Even the specialized search space of a searcher, such as the space of concatenations of the text searcher, could still be very large or even unbounded. Hence, we face the challenge of efficiently searching such spaces. In iMAP we propose to address this problem using a standard search technique called *beam search* [80]. The basic idea behind beam search is that it uses a *scoring function* to evaluate each match candidate, then at each level of the search tree, it keeps only \(k\) highest-scoring match candidates, where \(k\) is a pre-specified number. This way, the searcher can conduct a very efficient search in any type of search space.

**Match Evaluation:** To conduct beam search, given a match candidate such as concat(city, state), we must assign to it a score which approximates the semantic distance between it and the target attribute (say agent-address). In iMAP, we use a range of techniques, including machine learning, statistics, and heuristics, to compute such candidate scores. (We explain each implemented searcher in detail later.)

For example, to use learning techniques, we build a classifier for target attribute agent-address using the data in target schema \(T\), then apply it to classify candidate match concat(city, state). The classifier returns a confidence value which we can assign to be the candidate match’s score.

**Termination Condition:** Since the search space can be unbounded, we need a criterion for deciding when to stop the search. Our criterion is based on terminating when we start seeing diminishing returns from our search. Specifically, in the \(i\)th iteration of the beam search algorithm, we keep track of the highest score of any candidate matches (that have been seen up to that point), denoted by \(Max_i\). Then if the difference in the values of \(Max_i\) and \(Max_{i+1}\) (i.e., two consecutive iterations) is less than a pre-specified threshold \(\delta\), we
stop the search and return the $k$ highest-scoring match candidates as the most promising match candidates.

The following example illustrates the text searcher in more detail.

Example 2.3.2 Given a target attribute such as `agent-address`, the text searcher begins by considering all 1-1 matches, such as `agent-address = location`, `agent-address = price`, and so on (see Figure 2.1).

The text searcher then computes a score for each of the above matches. Consider the match `agent-address = location`. The searcher assembles a set of training examples for `agent-address`: a data instance in target schema $T$ is labeled “positive” if it belongs to `agent-address` and “negative” otherwise. Next, the searcher trains a Naive Bayes text classifier [35] on the training examples to learn a model for `agent-address`. The data instances are treated as text fragments in this training process. The searcher then applies the trained Naive Bayes text classifier to each data instance of attribute `location` (in source schema $S$) to obtain an estimate of the probability that that data instance belongs to `agent-address`. Finally, the searcher returns the average of the instance probabilities as the desired score.

After computing the scores for the 1-1 matches, the text searcher begins a beam search. It picks the $k$ highest-scoring 1-1 matches, then generates new matches by concatenating each of the $k$ matches with each attribute in $S$. For example, if `agent-address = city` is picked, then `agent-address = concat(city, state)` is generated as a new match. The searcher then computes the scores of the new matches as described above. In general, the score of match $t = f(s_1, \ldots, s_n)$ is
computed by comparing the column corresponding to \( t \) and the “composite column” corresponding to \( f(s_1, \ldots, s_n) \), and the comparison is carried out using the Naive Bayes text classifier. The searcher then picks the \( k \) best matches among all matches it has seen so far, and the process repeats until the diminishing-returns condition sets in, as described earlier. □

**Handling Join Paths:** Recall from Section 2.2 that a complex match can involve join paths. In our example, for the match \( \text{list-price} = \text{price} \times (1 + \text{fee-rate}) \), we need to discover that \( \text{price} \) and \( \text{fee-rate} \) should be joined via \( \text{HOUSES.agent-id} = \text{AGENTS.id} \).

We can find join paths for complex matches as follows. First, for each set of tables in \( S \), iMAP finds all possible join paths that can relate the tables. Note that the set of reasonable join paths for any group of tables is typically small, and can be discovered using a variety of techniques, including analyzing joins in queries that have been posed over the schemas and examining the data associated with the schemas [23]. The user can also suggest additional join paths for consideration.

Once iMAP has identified a small set of join paths per each group of tables, it modifies the search process to take the join paths into consideration. Consider the text searcher. Suppose it is in the process of generating new candidate matches. The current match is \( t = \text{concat}(a, b) \), where \( a \) and \( b \) are attributes of table \( X \) in schema \( S \). Now suppose the searcher is considering attribute \( c \) of table \( Y \) (also of \( S \)). Suppose iMAP has determined that tables \( X \) and \( Y \) can join via paths \( j_1 \) and \( j_2 \). Then the text searcher would create two candidate matches: \( \text{concat}(a, b, c) \) with \( a, b, c \) relating via \( j_1 \), and \( \text{concat}(a, b, c) \) with \( a, b, c \) relating via \( j_2 \). When materialized, each of the above two matches will likely form a different column of values, due to using different join paths.

**Implemented Searchers**

Table 2.1 characterizes the searchers currently implemented in iMAP. The searchers cover a variety of complex match types (text, numeric, category, etc.). They employ diverse
techniques to evaluate match candidates, and can exploit several forms of domain knowledge, such as domain constraints and overlap data. In what follows we describe the searchers that do not exploit overlap data. The rest will be described in Section 2.4. We do not discuss the text searcher any further.

**Numeric Searcher:** This searcher finds the best match for a target attribute judged to be numeric, such as `lot-area`. Building it raises the problem of how to compute the similarity score of a complex match, such as `lot-dimension1 * lot-dimension2`. We address this problem by considering the similarity between two value distributions: those of the values observed in column `lot-area`, and the values of the “composite column” created by “materializing” `lot-dimension1 * lot-dimension2`. We compute the similarity of value distributions using the Kullback-Leibler divergence measure [22, 55] (which has previously been used in other contexts, such as statistical natural language processing [55]).

The second problem we face is the type of matches the numeric searcher should examine. The searcher cannot consider an arbitrary space of matches, because this will likely lead it to overfit the data and find an incorrect match. Hence, we limit the numeric searcher to consider only a restricted space of common matches, such as those that add, subtract, multiply, or divide two columns. In Section 2.4 we discuss how the numeric searcher can exploit past complex matches or overlap data to find much more expressive matches, such as `price * quantity * (1 + fee-rate)`.

**Category Searcher:** This searcher finds “conversion” mappings between categorical attributes, such as `waterfront = f(near-water)`, where $f("yes") = 1$ (i.e., a data instance “yes” of `near-water` is converted into an instance “1” of `waterfront`) and $f("no") = 0$. Given a target attribute $t$, the searcher determines if $t$ is categorical, by counting the number of distinct values of $t$, and verifying that this number is below a threshold (currently set at 10). Next, it looks for category attributes on the source schema side, using the same technique. The searcher then discards the category source attributes whose number of distinct values are not the same as that of $t$, or whose similarity to $t$ is low (where similarity is computed
using the Kullback-Leibler measure on the two value distributions).

Let the remaining category source attributes be $s_1, \ldots, s_p$. Then for each $s_i$ the searcher attempts to find a conversion function $f_i$ that transforms the values of $s_i$ to those of $t$. Currently, the function $f_i$ that the searcher produces maps the value with the highest probability in the distribution of $s_i$ to that in the distribution of $t$, then the value with the second highest probability in the distribution of $s_i$ to that in the distribution of $t$, and so on. The searcher then produces as output attributes $s_1, \ldots, s_p$ together with the conversion functions $f_1, \ldots, f_p$.

**Schema Mismatch Searcher:** Schema-mismatch matches relate the data of a schema with the schema of the other. In the current iMAP implementation we focus on a particular type: a binary target attribute matches the data in a source attribute. For example, if a data instance of source attribute house-description contains (does not contain) the term “fireplace”, then the corresponding instance of target attribute fireplace is “yes” (“no”). This schema-mismatch type occurs very frequently in practice (e.g., product description, course listing). The fundamental reason is that often one schema chooses to mention a particular property (e.g., fireplace, zoom capability, hard copy edition) of an entity in the data, but another schema chooses to create an attribute modeling that property.

Given a target attribute $t$, this searcher determines if $t$ is a binary category attribute (using the same technique as in the category searcher). Next, it searches for the presence of the name of $t$ in the data instances of source attributes. If this name appears at least $p$ times (currently set at 5) in the data of $s$, then there may exist a schema mismatch between $t$ and $s$. The searcher then transforms $s$ into a category attribute $s'$, such that each data instance of $s$ is transformed into “1” if it contains the name of $t$, and 0 otherwise. Next, the searcher creates a conversion function $f$ that transforms data values of $s'$ into those of $t$ (similar to how it is done in the category searcher).

**Unit Conversion Searcher:** This searcher finds matches such as $\text{weight} = 2.2 \times \text{net-weight}$, a conversion between two different types of unit (pounds and kilogram in this case). It
first determines physical-quantity attributes, by looking for the presence of certain tokens (e.g., “hours”, “kg”, “$”, etc.) in the name and data of the attributes. The searcher then finds the best conversion from a set of conversion functions between commonly used units.

**Date Searcher:** This searcher finds complex matches for date attributes, using a set of terms in a simple ontology that captures date entities (e.g., day, month, year, week) and relationships among them (e.g., concatenation, generalization, specialization, subset). Suppose the searcher matches target attribute `birth-date` to ontology concept `DATE`, and source attributes `bday`, `bmonth`, and `byear` to ontology concepts `DAY`, `MONTH`, and `YEAR`, respectively. Suppose from the ontology we know that `DATE` is composed of `DAY`, `MONTH`, and `YEAR`, then we can infer that `birth-date = concat(bday, bmonth, byear)`.

### 2.3.2 The Similarity Estimator

For each target attribute $t$, the searchers suggest a relatively small set of promising match candidates. However, the scores assigned to each of the candidate matches is based only on a single type of information. For example, the text searcher considers only word frequencies via the Naive Bayes learner. Consequently, the accuracy reported by the searchers may not be very accurate.

The task of the similarity estimator is then to further evaluate these candidates, and assign to each of them a final score that measures the similarity between the candidate and $t$. In doing so, the similarity estimator tries to exploit additional types of information to compute a more accurate score for each match. To this end, it employs *multiple evaluator modules*, each of which exploits a specific type of information to suggest a score, and then combines the suggested scores into a final one. It is important to note that such an exhaustive evaluation would be prohibitively expensive to perform during the search phase.

Prior work suggests many evaluator modules, which exploit learning [31, 11, 36, 51],
statistical [44] linguistic and heuristic [28, 21, 63] techniques. The modules can be employed at this stage of iMAP. Currently, iMAP uses two modules:

- a name-based evaluator, which computes a score for a match candidate based on the similarity of its name to the name of the target attribute. The name of a match candidate is the concatenation of the names of the attributes appearing in that candidate, together with the names of the tables that contain the attributes.

- a Naive Bayes evaluator, which is the Naive Bayes classifier described earlier in Example 2.3.2.

These evaluators are similar to corresponding learner modules in [31].

2.3.3 The Match Selector

Once the similarity estimator has revised the score of the suggested matches of all target attributes, conceivably, the best global match assignment could simply be the one where each target attribute is assigned the match with the highest score. However, this match assignment may not be acceptable in the sense that it may violate certain domain integrity constraints. For example, it may map two source attributes to target attribute list-price, thus violating the constraint that a house has only one price.

The task of the match selector is to search for the best global match assignment that satisfies a given set of domain constraints. The match selector is similar in spirit to the constraint handler module described in [31], but extended to handle complex matches in addition to 1-1 matches.

A particularly interesting extension that we have developed allows the match selector to “clean up” complex matches using domain constraints. For example, in our experiments the overlap numeric searcher (described in the next section) frequently suggested matches such as \( \text{lot-area} = \) \( (\text{lot-sq-feet}/43560) + 1.3e-15 \times \text{baths} \). If the selector knows that source attribute baths maps
to target attribute num-baths, and that lot area and the number of baths are semantically unrelated and hence typically do not appear in the same formula, then it can drop the terms involving baths (provided that the value of the term is very small), thus transforming the above match into the correct one.

2.4 Exploiting Domain Knowledge

As we experimented with iMAP, we soon realized that exploiting domain knowledge can greatly improve the accuracy of complex matching. Indeed, past work (e.g., [31, 28, 53]) has noted the benefits of exploiting such knowledge in the context of 1-1 matching. There, the knowledge helps in evaluating matches and pruning unlikely matches. In the context of complex matching, however, exploiting domain knowledge brings even greater benefits, because it can also help to direct the search process and prune meaningless candidates early, avoiding costly evaluation. We now describe the use of domain knowledge in iMAP.

iMAP innovates in its use of domain knowledge in two ways. The first is the types of knowledge that it uses. Prior work on 1-1 matching has exploited domain constraints and past matches [31, 28, 53]. Here, in addition to these types of knowledge, iMAP also exploits overlap data between the databases and external data in the domain. Second, iMAP innovates in how to use domain knowledge. Specifically, iMAP uses domain knowledge at all levels of the system. In fact, (in the same spirit as pushing selections in query execution plans) we try to push the relevant domain knowledge to as early a point as possible in the match generation.

We now illustrate the above points by discussing how we exploit each particular type of domain knowledge.

**Domain Constraints:** Such constraints are either present in the schemas, or are provided by the domain experts or the user. In Section 2.6 we show that even with just a few
constraints iMAP can greatly improve matching accuracy.

Given a particular domain constraint, iMAP decides which system component should exploit it, trying to use it as early as possible. There are three cases:

- The constraint implies that two attributes of schema $S$ are unrelated, such as “name and beds are unrelated”, meaning that they cannot appear in the same match formula. Any searcher can use this constraint to never generate any match candidate that combines name and beds.

- The constraint involves a single attribute of $T$, such as “the average value of num-rooms does not exceed 10”. Any searcher can use this constraint to evaluate a match candidate for target attribute num-rooms. However, if the constraint is too expensive to be checked (e.g., when the searcher evaluates a very large number of match candidates), then it may have to be moved to the similarity estimator level, where the number of match candidates that it must be verified on will be far less than that at the searcher level.

- The constraint relates multiple attributes of $T$, such as “lot-area and num-baths are unrelated”. This constraint can only be exploited at the match selector level, as described earlier in Section 2.3.3, because the previous levels consider each attribute of schema $T$ in isolation.

**Past Complex Matches:** When mapping tasks are repetitive or done in closely related domains, we may often have examples of past matches. For example, in data integration settings, we map many sources into a single mediated schema [48, 31]. In enterprise data management we often find that we are mapping the same or similar schemas (and different versions thereof) repeatedly. iMAP currently extracts the expression template of these matches and uses those templates to guide the search process in the numeric searcher described in Section 2.3.1. For example, given the past match $price = pr \times (1 + 0.06)$, it
will extract the template VARIABLE * (1 + CONSTANT) and asks the numeric searcher to look for matches of that template.

**Overlap Data:** There are many practical mapping scenarios where the source and target databases share some data (e.g., when two sources describe the same company’s data or when two databases are views created from the same underlying database). Clearly, in such “overlap” cases the shared data can provide valuable information for the mapping process (as shown in [74]). Hence, we developed searchers to exploit such data. In what follows, we describe how searchers can incorporate overlap data.

*Overlap Text Searcher:* In the “overlap” case we can use this module instead of the text searcher in Example 2.3.2, to obtain improved matching accuracy. The module applies the text searcher to obtain an initial set of mappings. It then uses the overlap data to re-evaluate the mappings: the new score of each mapping is the fraction of the overlap data entities for which the mapping is correct. For example, suppose we know that databases $S$ and $T$ share a house listing (“Atlanta, GA,...”). Then, when re-evaluated, mapping \( \text{agent-address} = \text{location} \) receives score 0 because it is not correct for the shared house listing, whereas mapping \( \text{agent-address} = \text{concat(city, state)} \) receives score 1.

*Overlap Numeric Searcher:* In the “overlap” cases, this searcher can be used instead of the numeric searcher of Section 2.3.1. For each numeric attribute $t$ of schema $T$, this module finds the best arithmetic expression matches over numeric attributes of schema $S$. Suppose that the overlap data contains ten entities (e.g., house listings) and that the numeric attributes of $S$ are $s_1, s_2, s_3$. Then for each entity the searcher assembles a numeric tuple that consists of the values of $t$ and $s_1, s_2, s_3$ for that entity. Next, it applies an equation discovery system to the ten assembled numeric tuples in order to find the best arithmetic-expression match for attribute $t$. We use the recently developed LAGRAMGE equation discovery system [85]. This system uses a context-free grammar to define the search space of matches. As a result, this searcher can incorporate domain knowledge on numeric relationships in order to efficiently find the right numeric match. LAGRAMGE conducts a
beam search in the space of arithmetic matches. It uses the numeric tuples and the sum-of-squared-errors formula (commonly used in equation discovery) to compute match scores. 

**Overlap Category & Schema Mismatch Searchers:** Similar to the overlap text searcher, these searchers use their non-overlap counterparts to find an initial set of matches, then re-evaluate the matches using the overlap data.

**External Data:** Finally, another source of domain data is in sources external to the databases being matched. In principle, we can use external sources to mine properties of attributes (and their data values) that may be useful in schema matching. In fact, the mining can be completely decoupled from the matching system. In iMAP, given a target attribute (e.g., agent-name) and a feature that can be potentially useful in schema matching (e.g., number of distinct agent names), we mine external data (currently supplied by the domain experts) to learn a value distribution of the feature, then apply the learned distribution in evaluating match candidates for that target attribute.

### 2.5 Generating Explanations

As described earlier, the goal of a schema mapping system is to provide a design environment where a human user can quickly generate a mapping between a pair of schemas. In doing so, the user will inspect the matches predicted by the system, modify them manually and provide the system feedback. As mapping systems rely on more complex algorithms, there is a need for the system to explain to the user the nature of the predictions being made. Explanations can greatly help users gain insights into the matching process and take actions to converge on the correct matches quickly.

In iMAP we offer a novel explanation facility. We begin with an example that illustrates a possible scenario.

**Example 2.5.1** Suppose in matching real-estate schemas, for attribute list-price iMAP produces the ranked matches in decreasing order of confidence score:
The user is uncertain which of the two is the correct match, and hence asks iMAP to explain the above ranking.

iMAP can explain as follows. Both matches were generated by the overlap numeric searcher, and that searcher ranked the match \( \text{list-price} = \text{price} \times (1 + \text{monthly-fee-rate}) \) higher than \( \text{list-price} = \text{price} \). The similarity estimator also agreed on that ranking.

However, the match selector cannot rank \( \text{list-price} = \text{price} \times (1 + \text{monthly-fee-rate}) \) first because (a) it has accepted the match \( \text{month-posted} = \text{monthly-fee-rate} \) and (b) there is a domain constraint which states that the matches for \( \text{month-posted} \) and \( \text{price} \) do not share common attributes. Hence, the match selector must accept the match \( \text{list-price} = \text{price} \), and in essence flipped the ranking between the two matches.

When asked to explain the match \( \text{month-posted} = \text{monthly-fee-rate} \), which seems incorrect to the user, iMAP explains that the match is created because the date searcher has examined the data instances of source attribute \( \text{monthly-fee-rate} \) and concludes that it is a type of date.

At this point, the user examines \( \text{monthly-fee-rate} \) and tells iMAP that it is definitely not a type of date. iMAP responds by retracting the assumption made by the date searcher, and revising its match candidate ranking, to produce \( \text{list-price} = \text{price} \times (1 + \text{monthly-fee-rate}) \) as the top match. The user accepts this match with more confidence now that an explanation for the ranking exists.

We now explain the explanation facility in iMAP. We begin by describing the kinds of questions a user may want to ask of an explanation facility.

### 2.5.1 Types of User Questions

In principle, there are many question variations that a user may want to ask a matching system. In iMAP, we identified three main questions that are at the core of the rest:
1. **Explain existing match:** “why a certain match X is present in the output of iMAP?”

   An example is asking why the match month-posted = monthly-fee-rate is present in Example 2.5.1. In essence, the user wants to know how it was created and survived the evaluation and selection process, which components are most instrumental in getting the match to where it is in the output, and what important assumptions were made while generating it.

2. **Explain absent match:** Conversely, “why a certain match Y is *not* present in the iMAP’s output”.

3. **Explain match ranking:** “why match X is ranked higher than match Y in the output of iMAP”.

   In fact, it is interesting to note that as we were developing iMAP and experimenting with it, we were asking the same questions.

   One of our important design considerations is that we can ask the above questions from *each* component of iMAP (searchers, evaluator modules in the similarity estimator, and match selector). This greatly simplifies the construction of the explanation module, since questions can be reformulated recursively to the underlying components.

### 2.5.2 The Explanation Module

The key data structure underlying the explanation module of iMAP is the *dependency graph*, which is constructed during the matching process. The dependency graph records the flow of matches, data, and assumptions into and out of system components. The nodes of the graph are: schema attributes, assumptions made by system components, candidate matches, and pieces of domain knowledge such as domain constraints.

Two nodes in the graph are connected by a directed edge if one of them is the successor of the other in the decision process. The label of the edge is the system component that was responsible for the decision.
Figure 2.3: A sample fragment of the dependency graph as generated by iMAP.

Figure 2.3 shows a dependency graph fragment that records the creation and flow of match $\text{month-posted} = \text{monthly-fee-rate}$. A preprocessor finds that both $\text{month-posted}$ and $\text{monthly-fee-rate}$ have values between 1 and 12 and hence makes the assumptions that they represent months. The date searcher consumes these assumptions and generates $\text{month-posted} = \text{monthly-fee-rate}$ as a match candidate.

This candidate is then scored by the name-based evaluator and the Naive Bayes evaluator. The scores are combined by a combination module to produce a single score. The match selector acts upon the several mapping candidates generated to produce the final list of mappings. Here for the target attribute $\text{list-price}$ the selector reduces the rank of the mapping candidate $\text{price} \times (1 + \text{monthly-fee-rate})$ since it discovers that $\text{monthly-fee-rate}$ maps to $\text{month-posted}$.

The dependency graph is constructed as the system is run. Each of the components contributes nodes and edges during the execution of the system. At the end of the exe-
cution when the system has generated the matches the dependency graph is already in place.

**Generating Explanations:** We now briefly describe how iMAP generates explanations for the three types of predefined queries that have been described in Section 2.5.1. In each case, the system synthesizes an explanation in English for the user.

To answer the question “why match X is present”, iMAP selects the slice of dependency graph that records the creation and processing of match X. For example, the slice for month-posted = monthly-fee-rate is the portion of the graph where the nodes participated in the process of creating that match.

To answer the question “why match X is ranked higher than match Y”, the system compares the two slices of the dependency graph corresponding to X and Y. In comparing the slices, it focuses on places where the ranking is flipped and asks the relevant system component to explain why that component flips the ranking.

To answer the question “why match X is not present”, iMAP first examines the dependency graph to see if match X has been generated at all. If it has, then iMAP finds out where it has been eliminated, and asks the involved system component to explain why it eliminated X.

If match X has not been generated, then iMAP asks the searchers to see if any of them is capable of generating X. Suppose a searcher S indicates that it can generate X (but did not), then iMAP asks S for an explanation of why it did not generate X. These explanations are processed and then presented to the user.

**Performance:** Since each searcher produces only \( k \) top matches where \( k \) is the width of a beam search and hence is small, and since matching in iMAP goes through only three stages (searchers, similarity estimator, selector), it is easy to show that the dependency graph is relatively small. Hence, maintaining a dependency graph and traversing it to generate explanations incur negligible time and storage cost. We must exercise care, however, to make sure that each iMAP component can generate its explanations efficiently, in
order to efficiently obtain the global explanations.

### 2.6 Empirical Evaluation

We have evaluated iMAP on four real-world domains. Our goals were to evaluate the matching accuracy of iMAP, to measure the relative contributions of domain knowledge, and to examine the usefulness of match explanations.

#### Domains and Data Sources:

Table 2.2 describes the four real-world domains. Real Estate lists houses for sale. Inventory describes product inventories of a grocery business. Cricket describes cricket players, and Financial Wizard stores financial data of clients.

Obtaining data for schema matching experiments remains a challenge (though several benchmarks are currently being considered or built). In our work, we began by obtaining two independently developed databases for the Cricket domain (from cricinfo.com and cricketbase.com in December 2002), and used them as the source and target databases.

For the other three domains, we obtained one real-world database for each domain. The databases came from the Internet, the sample databases of Microsoft Access 97, the students in a large undergraduate database class, and from volunteers (who spent time populating the Financial Wizard database). The numbers of tables and attributes in each

<table>
<thead>
<tr>
<th>Domains</th>
<th>Source schema</th>
<th>Target schema</th>
<th># of 1-1 matches</th>
<th># of complex matches</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># tables</td>
<td># attributes</td>
<td># tables</td>
<td># attributes</td>
</tr>
<tr>
<td>Real estate</td>
<td>2</td>
<td>32</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>Inventory</td>
<td>2</td>
<td>44</td>
<td>38</td>
<td>27</td>
</tr>
<tr>
<td>Cricket</td>
<td>3</td>
<td>38</td>
<td>42</td>
<td>22</td>
</tr>
<tr>
<td>Financial wizard</td>
<td>4</td>
<td>41</td>
<td>22</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 2.2: Real-world domains for our experiments.
database are shown under the headline “Source Schema” of Table 2.2. Next, for each database $S$ we asked volunteers to create a target schema $T$. We asked the volunteers to examine and create complex matches between $T$ and $S$.

The target schemas $T$ are described in Table 2.2. The table shows the number of attributes of $T$ (under “Target Schema”), the number of 1-1 matches (between $T$ and $S$), and then the number of complex matches, broken down into different types. Below are a few examples of such matches:

- $\text{name} = \text{CONCAT(first\_name, last\_name)}$
- $\text{test\_economy\_rate} = 6 \times \text{t\_runs\_given} / \text{t\_balls}$
- $\text{ODI\_overs} = 0.1667 \times \text{o\_balls}$
- $\text{Marital\_status} = f(\text{person\_marital\_stats})$
  - Single=$f(\text{SIN})$ Married=$f(\text{MAR})$ Divorced=$f(\text{DIV})$
- $\text{fireplace} = 1$ if house\_description contains fireplace
- $\text{ODI\_debut} = (\text{o\_debut\_day}-\text{o\_debut\_month}-\text{o\_debut\_year})$

In the final step, we populated the schemas with data, using the database obtained for each domain. As discussed in Section 2.4, both the “overlap” and “disjoint” scenarios where the source and target databases do and do not share data occur frequently in practice. Hence we created both scenarios for experimental purposes. We took care to ensure that the source and target databases share some data in the “overlap” scenarios, but do not share any in the “disjoint” ones.

Data Processing: We performed only trivial data cleaning operations such as removing “unknown” and “unk”. Next, we specified domain constraints on the schemas. We specified only the most obvious constraints, such as “player-first-name does not appear in the same match as t-highest-score”, and “zip-code does not appear in the same match as account-number”.

Experiments: The above process in effect generated eight experimental domains, since
for each application (e.g., real estate, inventory, etc.) we have two domains, with disjoint and overlap data, respectively. We run experiments with several configurations of iMAP on all eight domains, as described in the next subsection.

**Performance Measure:** iMAP outputs for each target attribute a ranked list of best matches. We define the top-1 matching accuracy to be the fraction of target attributes whose top-1 match candidates are correct. The top-3 matching accuracy is then the fraction of target attributes whose top three candidates include the correct match. The top-3 accuracy is interesting because an interactive matching system typically proposes a ranked list of matches to a designer, and therefore the correct match needs not be the top 1.

Several prior works [28, 27] employ the notion of precision and recall to evaluate matching algorithms. Since iMAP finds matches for all target attributes, its precision and recall can be shown to be the same, and to be equivalent to the notion of matching accuracy used above.

### 2.6.1 Overall and 1-1 Matching Accuracy

Figure 2.4 shows the overall top-1 matching accuracy (that is, the fraction of all target attributes whose best match candidate is correct). Part (a) of the figure shows the accuracy for the four overlap domains. For each domain, the four bars from left to right represent respectively the accuracy of the iMAP configuration which exploits (a) no do-
main knowledge (i.e., the default system), (b) domain constraints, (c) overlap data, and (d) both domain constraints and overlap data.

The results show that iMAP achieves high matching accuracy 68-92% over all four overlap domains. The default iMAP achieves accuracy 58-74%. Exploiting domain constraints or overlap data further improves accuracy by 12-23%, and exploiting both domain constraints and overlap data further improves accuracy by as much as 11%.

Part (b) of Figure 2.4 shows the accuracy for the four disjoint domains. For each domain, the two bars from left to right represent respectively the accuracy of the default iMAP configuration and the one which exploits domain constraints. (Note that there is no overlap data and hence no bars representing the accuracy over exploiting overlap data.)

Here iMAP achieves accuracy rates 62-79%, slightly lower than those in the overlap domains. The default iMAP achieves accuracy 55-76%, and exploiting domain constraints improves accuracy by 9%.

In summary, Figure 2.4 shows that iMAP obtained high overall top-1 accuracy of 62-92% across domains. Its top-3 accuracy (not shown in the figure) is even higher, ranging from 64-95%. Finally, iMAP also achieves top-1 and top-3 accuracy of 77-100% over 1-1 matches (not shown in the figure). These results are competitive with those reported by existing 1-1 matching systems (e.g., [31, 28, 59, 54, 11]).

2.6.2 Complex Matching Accuracy

We now examine how well iMAP does in finding complex matches. Figure 2.5 shows matching accuracy in a format similar to that of Figure 2.4, but only for complex matches.

Part (a) of Figure 2.5 shows the top-1 accuracy 50-86% for the four overlap domains. The default iMAP achieves accuracy 33-55% on all domains, except for 9% on Inventory. At the end of this subsection we analyze the reasons that prevent iMAP from correctly identifying the remaining complex matches, in general and also in the Inventory domain.

As expected, exploiting domain constraints further improves accuracy up to 17%. Ex-
Figure 2.5: Top-1 (top row) and top-3 (bottom row) matching accuracy for complex matches.
Figure 2.6: Top-1 (top row) and top-3 matching (bottom row) accuracy for partial complex matches.
Ploiting overlap data improve accuracy over the default iMAP by up to 46%, and exploiting both domain constraints and overlap data improves accuracy by 10-64%.

In the four “disjoint” domains (Figure 2.5.b), the top-1 accuracy is lower, ranging 27-58%. The main reason for lower accuracy is that there is no overlap data to rely on. Hence accuracy for text matches slightly decreases and most numeric matches cannot be predicted. However, accuracy for categorical and schema mismatch matches remains high.

Figure 2.5.c-d shows that the top-3 accuracy over both overlap and disjoint domains are 43-92%, improving up to 42% over the top-1 accuracy of Figure 2.5.a-b. Thus, a significant number of correct complex matches are in the top three matches (per target attribute) produced by iMAP.

To examine exploiting past matches, we asked students in a database class to create database schemas in the same domain as Financial Wizard databases, then asked them to create 15 complex matches between the schemas. Next, we applied iMAP to exploit these 15 matches, as explained in Section 2.4. iMAP was able to find complex numeric matches and improve the top-1 matching accuracy on the disjoint Financial Wizard domain by 28%.

Discussion: There are several reasons that prevent the current iMAP system from identifying all complex matches. First, in many cases iMAP could not find “smaller components” of a complex match. In our example, when the correct match is agent-address = concat(apt-number, street-name, city, state), iMAP may return only the complex mapping concat(street-name, city, state). This is because the current learning and statistical techniques employed by iMAP are not sophisticated enough to detect such subtle difference of just a single number (apt-number). We believe adding format learning techniques may help in many such cases.

Second, the reverse problem also holds: in many cases iMAP added “small noise components” to a complex match. For example, in the Inventory domain, iMAP added...
agent-id (a single digit number) to many complex matches related to agents, thus reducing accuracy significantly. As we have shown earlier, this problem can be addressed by more aggressive match cleaning and enforcing of domain constraints. This underscores the importance of automatically learning domain constraints for complex matching.

Third, if the databases are disjoint, it was very difficult to discover meaningful numeric relationships. This is a fundamental problem that underlies any system that finds complex matches. Here, we have proposed a preliminary solution of exploiting past numeric matches and showed its promise. We believe more work is needed on this topic in particular, and on the issue of constructing and re-using domain knowledge in general.

Finally, many complex matches are not in the top one, but somewhere in the top three (and more general, in the top ten) of the matches predicted by iMAP. Given the fact that finding a complex match requires gluing together so many different components, perhaps this is inevitable and inherent to any complex matching solution. This underscores the importance of generating explanations and building effective mapping design environment, so that humans can examine the top ranked matches to create mappings.

Finding Partial Complex Matches: So far we have considered only cases where iMAP produces the exact complex matches, that is, finding the exact attributes, expression, and relationship. We note, however, that even when iMAP finds only partial complex matches, these matches would still be useful, because the user can elaborate on them to find the exact matches.

Hence, we examine how well iMAP finds these partial complex matches. The first type of such partial matches finds only the right attributes. Figure 2.6 shows the accuracy for this type. The top-1 accuracy is 73-86% over overlap domains and 36-75% over disjoint domains. The top-3 accuracy is high, ranging from 82 to 100% over overlap domains and 70-83% over disjoint domains. These results suggest that in a significant number of cases iMAP finds the correct set of attributes in a complex match.

Finding Value Correspondences: Even when iMAP does not find the correct match, in
many cases it would still be easy for user to examine the ranked list of candidate matches and find the correct expression. For example, in the Real Estate domain, iMAP generated the following top three matches for attribute num-rooms:

\[
\begin{align*}
    &2 \times \text{dining-rooms} + \text{bed-rooms} + \text{bath-rooms} \\
    &\text{bath-rooms} + \text{bed-rooms} + 2 \times \text{dining-rooms} \\
    &\text{bath-rooms} + 2 \times \text{living-rooms} + \text{bed-rooms}
\end{align*}
\]

For this attribute, the overlap numeric matcher converged before generating the correct complex match having all four terms. Nevertheless, a user can easily elaborate on the above top three incorrect matches to arrive at the correct expression.

Finding the correct expression is useful because such expressions, known as value correspondences, can be fed directly into a schema refinement tool such as Clio [92], to produce the final correct mapping. Hence, we are interested in knowing how well iMAP does in finding value correspondences. We asked several volunteers to examine the top three of iMAP’s results, on all eight domains, and count the cases where it would be fairly obvious from the top three matches that what the correct value correspondences should be. iMAP found 75-93% of correct value correspondences over the overlap domains and 57-75% over the disjoint domains. The results, while somewhat subjective, due to the judgment of the volunteers, do suggest that iMAP can find value correspondences in a large number of cases.

**Performance Sensitivity:** Figure 2.7.a-b shows the variation of the top-1 and top-3 matching accuracy as a function of the number of data tuples available from each source, for the “disjoint” Real Estate and “overlap” Inventory domains, respectively. The results show that on these domains the performance of iMAP stabilizes fairly quickly and almost levels off after 100 data tuples. Experiments with other domains show the same phenomenon. iMAP thus appears to be robust, and can work well with relatively little data. One of the reasons this observation is important is that we can reduce the running time
Figure 2.7: Performance sensitivity

Efficiency: Our unoptimized iMAP version took 5-20 minutes on the experimental domains. (iMAP spent most of the time searching for promising match candidates.) We found that by applying a variety of techniques, including preliminary preprocessing to break the schemas into independent chunks on each of which iMAP can work in isolation, we can significantly reduce iMAP’s runtime. In the long term, scaling up matching systems to very large schemas is an important and interesting topic that we are planning to pursue.

2.6.3 Explaining Match Predictions

We now provide anecdotal evidence that iMAP can generate meaningful explanations that provide insight into the working of the system and that are actionable to the user.

We examined the matches produced by iMAP in various experimental domains, and asked for explanations in many cases of incorrect or suspicious matches. Figures 2.8 and 2.9 show the explanations in two such cases, slightly edited for space reasons.

Figure 2.8 shows the explanation for question “why $\text{pname} = \text{last-name}$ ranks higher than $\text{concat( first-name, last-name)}$” in the Cricket domain. The figure shows that at the searcher level $\text{concat( first-name, last-name)}$ was ranked higher than $\text{last-name}$. It also clearly shows that things went wrong at the similarity estimator level. Here, the Naive
Figure 2.8: Generated explanation for **pname** vs. **concat(first-name,last-name)**.

USER: Why **pname** = last-name ranks higher than **concat(first-name,last-name)**?

**iMAP:**
1. **Searcher Level:**
   - Text searcher generated last-name
   - Text searcher generated **concat(first-name,last-name)**
   - Text searcher ranked **concat(first-name,last-name)** higher by 0.102
2. **Similarity Estimator Level:**
   - Name-based evaluator ranked last-name higher by 0.0419
   - Naïve Bayes evaluator ranked **concat(first-name,last-name)** higher by 0.01
   - Final score for last-name: 0.434
   - Final score for **concat(first-name,last-name)**: 0.420
3. **Match Selector Level:**
   - Match selector did not modify two candidates
4. **Greatest influence on top five candidates for **pname**: Name-based evaluator**

Figure 2.9: Generated explanation for **num-rooms**.

USER: Why **num-rooms** = bath-rooms + bed-rooms + dining-rooms + living-rooms does not appear?

**iMAP:**
1. Overlap numeric matcher **CAN** generate
   - bath-rooms + bed-rooms + dining-rooms + living-rooms for **num-rooms**.
2. Overlap numeric matcher **DID NOT** generate it.
3. A reason: the match has length of 4 terms.
   - Overlap numeric searcher has considered only candidates of length up to 3 terms.
4. Characteristics of the search space for **num-rooms**:
   a. Number of considered numeric attributes: 7
   b. Considered numeric attributes: building-area lot-dimension1 lot-dimension2
      bath-rooms bed-rooms dining-rooms living-rooms
   c. Used successor functions: Addition Multiplication
   d. Max. number of terms: 3
   e. Max. number of elements in a term: 3
Bayes evaluator still ranked matches correctly, but the name-based evaluator flipped the ranking. This flipping was clearly responsible for the final ranking, since the explanation shows that the match selector did not modify the ranking that came from the similarity estimator.

The last line of the explanation also confirmed the above conclusion, since it states that the name-based evaluator has the greatest influence on the top five match candidates for \textit{pname}. The influence of an evaluator is computed by (a) simulating the matching process on the dependency graph without that evaluator, and (b) compute the difference between the new ranking of candidate matches and the original ranking. The difference in ranking is currently computed as \( \sum_{i} |n_i - o_i| \), where \( n_i \) and \( o_i \) are the positions of match candidate \( i \) in the new ranking and old ranking, respectively.

Thus, the main reason for the incorrect ranking for \textit{pname} appears to be that the name-based evaluator has too much influence. This explanation would allow the user to fine tune the system, possibly by reducing the weight of the name-based evaluator (in the score combination step).

In the second example in Figure 2.9, when asked why a particular match for \textit{numrooms} (in the Real Estate domain) does not appear in the output, \textsc{iMAP} did not find that match on the dependency graph, so it asked the searchers. The overlap numeric searcher explained that it could, but did not generate the match because the match has length four and the searcher converged before generating candidates of that length.

The above explanation suggests that the convergence criterion of the overlap numeric searcher was set too loosely. This provides grounds for the future actions of the user.

\section*{2.7 Summary}

Semantic matches are key for enabling a wide variety of data sharing and exchange scenarios. The vast majority of the research on schema matching has focused on 1-1 matches.
This chapter described a solution to the problem of finding complex matches, which are prevalent in practice. The key challenge with complex matches is that the space of possible matching candidates is possibly unbounded, and evaluating each candidate is harder. iMAP uses two main techniques to search the space effectively. First, it employs a set of specialized searchers that explore meaningful parts of the space. Second, it makes aggressive use of various types of domain knowledge to guide the search and the evaluation where possible. Keeping in spirit with recent works, the architecture of iMAP is modular and extensible. New searchers and new evaluation modules can be added easily. Finally, iMAP offers a novel explanation facility that helps human users interact with the system to generate mappings quickly. Our experimental results show that iMAP achieves 43-92% accuracy on several real-world domains, thus demonstrating the promise of the approach.
Chapter 3

Tuning Schema Matching Systems

In this chapter, we describe eTuner, an approach to automatically tune schema matching systems. We first provide an overview of this work in Section 3.1. We define the problem of tuning matching systems (Section 3.2), and then describe our solution (Section 3.3-3.4). We then describe our experimental evaluation (Section 3.5) and summarize the chapter (Section 3.6).

3.1 Introduction

Most recent schema matching systems assemble multiple components, each employing a particular matching technique. The multi-component nature is powerful in that it makes matching systems highly extensible and (with sufficient skills) customizable to a particular application domain [12, 78]. 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 correctly adjust their numerous “knobs” (e.g., threshold, coefficients, weights, etc.)?

Without tuning, matching systems often fail to exploit domain characteristics, and produces inferior accuracy. Indeed, in Section 3.5 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 [28, 27, 59, 52, 77]; see Section 3.5 for more details.)

1This work has been previously published in a 2007 VLDB Journal paper [47].
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 the correctness of each data value transmitted, so erroneous matches can 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 [92] to elaborate matches into semantic mappings (e.g., in the form of SQL queries [92] which specify exact relationships between elements of different schemas; see a more detailed description in [24, 92, 83]). Here, improving the accuracy of the automated match phase can significantly reduce peoples’ workloads, and also the likelihood that they could overlook or introduce mistakes.

Second, large-scale data integration, peer-to-peer, and distributed IR systems (e.g., on the Web [3]) often involve tens or hundreds of sources, thus discovering 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 [82], for example, develops Kite, a system that enables a 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 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 [25], a task that is difficult even for learned experts [25]. Furthermore, since we rarely know the ground truth for matches, it is not clear how to compare the quality of
knob configurations.

For all of the 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 in which they are attractive in practice.

In this chapter we describe eTuner, an approach to automatically tune schema matching systems. Developing eTuner required several innovations.

**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 3.1.a shows a matching system which has \((n + 2)\) components: \(n\) matchers, one combiner, and one selector (Section 3.2 describes these components in detail).

To the user (and eTuner) the components are *blackboxes*, with “exposed knobs” whose values can be adjusted. For example, a knob allows the user to set a threshold \(\alpha\) such that two schema attributes are declared matched if and only if their similarity score exceeds \(\alpha\). 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 chapter 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 \(M\), how to tune \(M\) 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 [77, 31].

**Synthesize Workload with Known Ground Truth:** Tuning the system \(M\) mentioned above amounts to searching for the “best” knob configuration for matches to \(S\). The
quality of a particular knob configuration of $M$ is defined as an aggregate accuracy of the matching system, when applied with that configuration. Accuracy metrics exist (e.g., precision, recall, and combinations thereof [27]). 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 Figure 3.1.b to EMP, and the rule “replacing ,000 with K” to the column salary of this table. We note that these rules are created only once, independent of any schema $S$.

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

Figure 3.1: An example of multi-component matching systems.
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 $M$ in Figure 3.1.a. Here, we first tune each of the matchers $1 \ldots n$ in isolation, then tune the combination of the combiner and the matchers, assuming the knobs of the matchers have been set. Finally, we tune the entire matching system, assuming that the knobs of the combiner and matchers have been set. 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 Section 3.4.

In summary, this chapter makes 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 Sections 3.6.

- 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.
3.2 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 [77]. Hence, in this chapter we focus on the problem of tuning such systems, leaving those that finds complex matches (e.g., address = concat(city, state) [24, 91, 43]) as future work. In this chapter we handle only relational schemas, and defer handling other data representations (e.g., XML schemas) to the future work.

3.2.1 Modeling 1-1 Matching Systems

We define an 1-1 matching system $\mathcal{M}$ to be a triple $(L, G, K)$, where $L$ is a library of matching components, $G$ is a directed graph that specifies the flow of execution among the components of $\mathcal{M}$, and $K$ is a collection of control variables (henceforth knobs) that the user (or a tuning system such as eTuner) can set. The description of each component in $L$ lists the set of knobs available for that component.

In what follow we elaborate on the above concepts, using the matching system in Figure 3.2 as a running example. This system is a version of LSD, a learning-based multi-component matching system described in [31, 30, 29].

![Figure 3.2: The LSD system: (a) library of matching components, (b) execution graph, and (c) sample knobs.](image)
Library of Matching Components

Such a library contains the following four types of components, variants of which have often been proposed in the literature [77, 34]:

- **Matcher (schemas $\rightarrow$ 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 chapter, we will use “matrix” as a shorthand for “similarity matrix”.) Library $L$ in Figure 3.2.a has five matchers. The first two compare the names of two attributes (using q-gram and TF/IDF techniques, respectively) to compute their similarity score [28, 31]. The remaining three matchers exploit data instances [31].

- **Combiner (matrix $\times \ldots \times$ matrix $\rightarrow$ matrix):** A combiner merges multiple similarity matrices into a single one. Combiners can take the average, minimum, maximum, or a weighted sum of the similarity scores (Figure 3.2.a) [28, 36, 31]. More complex types of combiner include stacking (an ensemble learning method, employed for example in LSD [31]), decision tree [36], and elaborate (often hand-crafted) scripts (e.g., in Protoplasm [12]).

- **Constraint Enforcer (matrix $\times$ constraints $\rightarrow$ matrix):** Such an enforcer exploits pre-specified domain constraints or heuristics to transform a similarity matrix (often coming from a combiner) into another one that better reflects the true similarities. 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 Figure 3.2.a has a single constraint enforcer, which exploits integrity constraints such as “lot-area cannot be smaller than house-area” [31].

- **Match Selector (matrix $\rightarrow$ matches):** This component selects matches from a given similarity matrix. The simplest selection strategy is thresholding: all pairs of attributes
with similarity score exceeding a given threshold are returned as matches [28]. More complex strategies include formulating the selection as an optimization problem over a weighted bipartite graph [59] (Figure 3.2.a).

Execution Graph

This is a directed graph whose nodes specify the components of $M$ 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 chapter. (Section 3.5 gives a complete description of the four systems.)

**LSD:** The execution graph of LSD [31] is shown in Figure 3.2.b and has four levels. It states that LSD first applies the $n$ matchers, then combines their output similarity matrices using a combiner. Next, LSD applies a constraint enforcer, followed finally by a match selector. (We omit displaying domain constraints as an input to the enforcer, to avoid
The following example illustrates the working of LSD:

**Example 3.2.1** Consider matching the schemas of data sources *realtor.com* and *homes.com* in Figure 3.4. 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 Figure 3.4.b.

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 Figure 3.4.a), and outputs another similarity matrix (see the lower half of Figure 3.4.b). 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 the combiner simply takes the average of the corresponding scores, then it outputs a similarity matrix as shown in Figure 3.4.c.
Notice the combined 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 Figure 3.4.d. 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 Figure 3.4.e), the enforcer will adjust the similarity scores to best reflect this constraint (see [31] for more detail). Finally, the match selector returns the matches with the highest score, as shown in Figure 3.4.f.

**COMA & SimFlood:** Figure 3.1.a shows the execution graph of the COMA system [28], 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 3.3.a shows the execution graph of the SimFlood matching system [59]. SimFlood employs a single matcher (a name matcher [59]), 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 [59]).

**LSD-SF:** We can combine LSD and SimFlood to build a system called LSD-SF, whose execution graph is shown in Figure 3.3.b. 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 [28, 31, 77]. 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 [28, 31]). Since our current focus is on automating the entire tuning process (allowing optional user feedback only in creating the synthetic workload, but not during the staged tuning, see Section 3.3.2), 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.

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

For example, Figure 3.2.c shows a decision tree matcher that has four knobs. The first knob, characteristics-of-attr, is set-valued. The matcher has defined a broad set of salient characteristics of schema attributes, such as the type of the attribute (integer, string, etc.), the min, max, average value of the attribute, and so on (see [50, 36] 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 [25]. Figure 3.8 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., [31]). Specifically, the combiner assigns to each matcher \( m_k \) a weight \( w_k \), then compute the
combined score:

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

where $$score(m_k, a_i, a_j)$$ 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 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 Figure 3.2.b. The system $$M$$ may specify that this node can be assigned either the q-gram name matcher or TF/IDF name matcher from the library (Figure 3.2.a).

Consequently, each node of an execution graph can be viewed as a unordered discrete knob. Note that it is conceptually possible to define “data flow” knobs, e.g., to change the topology of the execution graph. However, most current matching systems (with the possible exception of [12]) 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 [11, 33, 67], those in [36, 52, 66], and COMA++ and Protoplasm, industrial-strength matching systems under development at the University of Leipzig\(^2\) and Microsoft Research [12], respectively.

### 3.2.2 Tuning of Matching Systems

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

**Definition 3.2.1 (Match Tuning Problem)** Given

- $$\text{matching system } M = (L, G, K), \text{ as defined above;}$$

\(^2\)http://dbs.uni-leipzig.de/en/Research/coma.html
workload $W$ consisting of schema pairs $(S_1, T_1), (S_2, T_2), \ldots, (S_n, T_n)$ (often the range of schemas will be described qualitatively, e.g., “future schemas to be integrated with our warehouse”); and

- utility function $U$ defined over the process of matching a schema pair using a matching system; $U$ 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 $M(C)$ be the matching system $M$ using the knob configuration $C$, and let $C$ be the space of all knob configurations, as defined by $M$, then

$$C^* = \arg\max_{C \in C} \left[ \frac{1}{n} \sum_{i=1}^{n} U(M(C), (S_i, T_i)) \right] / n$$

(3.1)

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

In this chapter we restrict the above general problem. First, we use just one utility function $U$ accuracy. Specifically we use F-1, a combination of precision and recall formalized in Section 3.5. 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 [28, 27, 59, 52, 77].

As a second restriction, we tune $M$ 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 [31, 77]. In the next two sections we describe the eTuner solution to this problem.
3.3 The eTuner Approach

The eTuner architecture (see Figure 3.5) 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 $M$ using the synthetic workload and tuning procedures stored in an eTuner repository. The tuned system $M$ can now be applied to match schema $S$ with any subsequent schema. It is important to note that the transformation rules and the tuning procedures are created only once, independently of any application domain, when implementing eTuner.

While the tuning process is completely automatic, eTuner can also exploit user assistance to generate an even higher quality synthetic workload. Specifically, the user can “augment” schema $S$ with information on the relationships among attributes (see the dotted arrows in Figure 3.5).

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

3.3.1 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
Figure 3.6: Perturbing schema $S$ to generate two schemas $U$ and $V_1$ and the correct matches between them.

Each schema $V_i$, $i \in [1, n]$, it traces the perturbation process to create the set of correct semantic matches $\Omega_i$ 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.

**Creating 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 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 [24, 13] and thus may significantly bias the tuning process.

The above step is illustrated in Figure 3.6.a, which shows a schema $S$ with three tables. The schemas $V$ and $U$ generated from $S$ also have three tables with identical structures. However, table 3 of $S$, which we show in detail as table EMPLOYEES in Figure 3.6.a, 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.

**Creating 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 pre-specified, 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 $\alpha_t$ 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 Figure 3.6.a, which has three tables 1, 2, 3, has been transformed into schema $V_1$, which has only two tables 12 and 3. The tables 1 and 2 of $V$ have been merged into table 12 of $V_1$.

- **Perturbing the Structure of Each Table:** For each table of schema $V_1$, the generator now perturbs its structure. It randomly selects $column$-$transformation$ rules to apply to the columns of the table, exactly $\alpha_c$ 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 Figure 3.6.b, for table $EMPLOYEES$, column first is
dropped and two columns last and id are swapped.

- **Perturbing Table and Column Names:** In the next step, the name of each table and its columns in schema $V_1$ are perturbed. eTuner has implemented a set of rules that capture common name transformations [54, 24, 77]. 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 $\alpha_n$ times (currently set to two).

In Figure 3.6.b, the name of table EMPLOYEES has been abbreviated to EMPS (the first three letters plus “S” for plurality). The name of column last has been added the new table name as a prefix, to become 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 $\alpha_f$ times (currently set to two).

Once the format of a column $c$ has been perturbed, the generator perturbs the data values. If the values are numeric (e.g., price, age, etc), then they are assumed to have been generated from a normal distribution with mean $\mu_c$ and variance $\sigma_c^2$. Thus, the
generator estimates $\mu_c$ 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 $\mu'_c$ and $\sigma'_c^2$, respectively. Then each value $x$ is now generated according to the normal distribution:

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

(3.2)

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 $\mu_c$ and variance $\sigma_c^2$, the generator perturbs $\mu_c$ and $\sigma_c^2$ by random amounts in the range $\pm [10,100]\%$ to generate new mean $\mu'_c$ 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 $\mu'_c$ 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 Figure 3.6.b (the rightmost table). Its format has been perturbed so that the signs “$” and “,” are dropped, and its values have been changed, so that “40,000$” is now “45200”.

**Creating Semantic Matches between $V_1$ 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 $a = b_1, \ldots, a = b_n$. 

65
as correct matches between $V_1$ and $U$. Figure 3.6.c lists the correct matches between table EMPS of $V_1$ and table EMPLOYEES of $U$. As another example, suppose attributes first-name and last-name of $V$ are merged to create attribute name of $V_1$, then the generator derives the matches name = first-name and name = last-name.

Let $\Omega_i$ 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 $M$.

Figure 3.7 gives the pseudo code of the workload generator.

### 3.3.2 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 \texttt{emp-phone = phone-1} correct (between $V_1$ and $U$), but will not recognize \texttt{emp-phone = phone-2} as also correct (since \texttt{emp-phone} is not derived from \texttt{phone-2}). 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 $G = \{a_{i_1}, \ldots, a_{i_n}\}$ of schema $S$ are \textit{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, \texttt{phone-1} and \texttt{phone-2} are match equivalent. As another example, (depending on application) a user may also judge \texttt{first-name} and \texttt{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 $G = \{a_{i_1}, \ldots, a_{i_n}\}$ 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 3.5 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 chapter.

### 3.4 Tuning with the Synthetic Workload

We now describe how to tune a matching system $M$ with a synthetic workload $W$ as created in the previous section.
3.4.1 Staged Tuning

Our goal is to find a knob configuration of $M$ that maximizes the average accuracy over $W$ (see Definition 3.2.1). 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 Section 3.5 has 21 knobs, with at least 2 possible values per knob, resulting in at least $2^{21}$ configurations.

To address this problem, we propose a staged, greedy tuning approach. Assume the execution graph of $M$ has $k$ levels. We first tune each node at the bottom, i.e., at the $k$-th level, in isolation. Next, we tune subsystems that consist of nodes at the $(k-1)$th and $k$-th levels. While tuning such subsystems (described in detail in the following subsection), we assume the nodes at the $k$-th level have been tuned, so their knob values are fixed, and we only need to tune knobs at level $(k-1)$. If there is a loop of $m$ components, then the loop is treated as a single component when being considered for addition to a subsystem. This staged tuning repeats until we have reached the first level and hence have tuned the entire system.

Consider for example tuning the LSD system in Figure 3.2.b. We first tune each of the matchers $1 \ldots n$. Next, we tune the subsystem consisting of the combiner and the matchers, but assuming that the matchers have been tuned. Then we tune the subsystem consisting of the constraint enforcer, combiner, and matchers, assuming that the combiner and matchers have been tuned, and so on.

Suppose the execution graph has $k$ levels, $m$ nodes per level, and each node can be assigned one of the $n$ components in the library. Assume each component has $p$ knobs, and each knob has $q$ values. Then staged tuning examines only a total of $k \times (m \times (n \times p \times q))$ out of $(n \times p \times q)^{k \times m}$ knob configurations, a drastic reduction. Section 3.5 shows that while not guaranteeing to find the optimal knob configuration, staged tuning still outperforms currently possible tuning methods.
3.4.2 Tuning Subsystems of $M$

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

We then tune the knobs of $S$ as follows. Recall from Section 3.2.1 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., [36, 31]). Hence, we assume that there is just one type-III knob for subsystem $S$, 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 $S$ 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 Section 3.4.3 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 $S$. 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 [31]. They can be tuned using linear or logistic regression (over the synthetic workload) [31]. However, such tuning often requires that all other knobs of $S$ have been
Figure 3.8: Sixteen sample features that \( \texttt{eTuner} \) uses in selecting a best set of features for the schema attributes. CV stands for “coefficient of variation” and SD for “standard deviation”.

set (otherwise \( S \) 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 \( S \). Then in Step 2 we run procedures to tune interrelated knobs (if any, these procedures are stored in \( \texttt{eTuner} \)). If this tuning results in a better knob configuration, then we take it; otherwise we use the knob configuration found in Step 1.

### 3.4.3 Tuning to Select Features

The only thing that remains is to describe how to tune the type-III knob that selects features for subsystem \( S \). Without loss of generality, assume \( S \) is a matcher.

Recall from Section 3.2.1 that a matcher often transforms each schema attribute into a feature vector, then uses these vectors to compare attributes. In \( \texttt{eTuner} \) 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., [50, 28, 59, 36, 11, 54, 31, 33]). Figure 3.8 shows 16 sample features. The goal of tuning is then to select from the set \( F \) of all
enumerated features a subset $F^*$ that best assists the matching process.

The simplest solution to find $F^*$ is to enumerate all subsets of $F$, run $S$ 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** [25], 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 $S$ over the synthetic workload, and the best change is made. Then a new set of changes is considered. Figure 3.9 describes the wrapper method [25], as adapted to our context.

However, the wrapper method can still be very expensive. For example, even just for 20 features, it would run $S$ 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 [25] and shown in Figure 3.12) to select a small subset $F'$. **Relief-
Figure 3.11: High-level description of the wrapper method, as adapted to feature selection for text-based matchers.

$F$ detects relevant features well, and runs very fast, as it examines only the synthetic workload, not running any matching algorithms [25]. We then apply the above greedy wrapper algorithm to the much smaller set $F'$ to select the final set of features $F^\ast$.

**Selecting Features for Text-Based Matchers:** Features as described above are commonly used by learning methods such as decision tree, neural network [50, 36, 31, 33] and also by many rule-based methods (e.g., [28, 54, 59]). However, many learning-based (e.g., Naive Bayes, SVM) as well as IR-based matching methods (e.g., [21, 31]) view data instances as *text fragments*, and as such operate on a different space of features. We now consider generating such feature spaces and the associated feature selection problem.

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

Figure 3.10 shows a sample taxonomy of features over text for eTuner (adapted from [14]). 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
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 $E = \emptyset$ be the set of examples, $<f_1, ..., f_p, c>$,
      where $f_1, ..., f_p$ 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, $<f_1, ..., f_p>$ using $R$
        $E = \langle <f_1, ..., f_p, c> \rangle \cup E$

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 $e' = e_c$
   c. Foreach class $c \neq c_e$ in $S$ do
      Let $M_i$ be a set of examples, $e''$, nearest to $e$, where $e'' = e_c$
   d. Foreach feature $f$ in $F$ do
      Let $w_f = w_f - 1/(LB \left[ \Sigma_{i \in M_i} \delta(e''_j, e_f) \right] + \Sigma_{i \in H} \left[ \Sigma_{j \in M_i} \delta(e'_j, e_f) \right])$
      where $\delta$ is the dot product of two vectors

5. Let $F^* = \emptyset$ be the set of relevant features
   Foreach feature $f$ in $F$ do
   If $w_f \geq W$, then $F^* = \{f\} \cup F^*$

6. Return $F^*$

Figure 3.12: High-level description of the Relief-F algorithm [25], as adapted to feature selection in eTuner.
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 Figure 3.11), 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.

### 3.5 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 Sections 3.5.2- 3.5.4): (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 Section 3.5.2), by 1-15%, compared to using matching systems tuned with the source-dependent tuning method. eTuner yields lower matching accuracy in only 2 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.
3.5.1 Experimental Settings

Domains: We obtained publicly available schemas in four domains. The schemas have been used in recent schema matching experiments [31, 24, 52]. The domains have varying numbers of schemas (2-10) and diverse schema sizes (10-50 attributes per schema, see Figure 3.13.a). Real Estate lists houses for sale. Courses contains time schedules for several universities. Inventory describes business product inventories, and Product stores product descriptions of groceries. Figures 3.13.c-d show sample schemas in Real Estate and Inventory.

Matching Systems: Figure 3.13.b summarizes the four matching systems in our experiments. We began by obtaining three multi-component systems that were proposed recently. The LSD system was originally developed by one of us [31] to match XML DTDs. We adapted it to relational schemas. The SimFlood system [59] was downloaded from the Web³. The COMA system was described in [28]. Since we did not have access to COMA, we implemented a version of it called iCOMA. The iCOMA library includes all components described in [28], 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 Section 3.2), to obtain LSD-SF, the fourth matching system. Figure 3.13.b 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 (Figure 3.14). Both the decision tree matcher [36] and the Naive Bayes matcher [31, 64] exploit data instances. All knobs for the decision tree matcher have been discussed in Section 3.2.1 (and see [64] for further detail). The Naive Bayes matcher has one knob, abstraction-level, for choosing the abstraction level of

³http://www-db.stanford.edu/~melnik/mm/sfa
text tokens, as described in Section 3.4.3.

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 [24]. It has one knob for choosing a tokenizing rule. Currently, there are five rules: use each word as a token (no-stemming), use the Porter’s stemmer (stemming), and generate q-gram tokens after using the 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, \( \text{gram-size} \), to select the value of \( q \) among 2, 3, and 4.

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 Section 3.2.1. The threshold-based selector has a knob for setting the threshold
Figure 3.17: The library of matching components of LSD-SF.

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 \textit{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).

- \textbf{iCOMA}: Figure 3.15 lists components of iCOMA: 10 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 [28].

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 execu-
tion 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 3.16 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 enforcer, and the type&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 [59], 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 [59] for further detail).

The type&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 3.17 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 3.3.b), and LSD (the big box at the lower-right corner in Figure 3.3.b). 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.

**Experimental Methodology:** For each of the four domains described in Figure 3.13, 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 2 sources. We then report the average
accuracy per domain. For *eTun*er, 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 [28, 27, 59, 52, 77],
we use the $F_1$ score to evaluate matching accuracy. Given a set of candidate matches for
$S$ and $T$, we have $F_1 = (2PR)/(P + R)$, where precision $P$ is the percentage of candidate
matches that are correct, and recall $R$ is the fraction of all correct matches discovered. The
goal of tuning is to find the knob configuration that maximizes $F_1$ score.

### 3.5.2 The Need for Tuning

We begin by demonstrating the need for tuning, using Figures 3.18.a-d. The figures show
the results for LSD, iCOMA, SimFlood, and LSD-SF, respectively. Each figure shows the
results over four domains: Real Estate, Product, Inventory, and Course. Thus we have a
total 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 accu-
curacy as the first bar in each group. For instance, for LSD and Real Estate (the first group
of Figure 3.18.a), the first bar is 33%. The “as is” accuracy is 14-62% across all 16 cases,
demonstrating that “off-the-shelf” matching systems are quite brittle.

Next, we did our best to tune each system independently of any domain, in effect
imitating a vendor tuning a system before release. (We found graduate student volunteers
not suitable for this task, suggesting that administrators will also have difficulty tuning.
See below for details). We examined literature about each matching system, leveraged
our knowledge of machine learning and schema matching, and tweaked the systems on
pairs of schemas not otherwise used in the experiments. The *second bar* in each group
Figure 3.18: Matching accuracy for (a) LSD, (b) iCOMA, (c) SimFlood, and (d) LSD-SF.

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.

3.5.3 “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 they can tune effectively. Consider for example the decision tree matcher described in Section 3.2.1. We found that the tuned version of this matcher improves accuracy significantly, so tuning it is necessary. However, it was very difficult to explain the meaning of its knobs (see Section 3.2.1) 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.

### 3.5.4 Domain- & Source-Dependent Tuning

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

We tuned each matching system for each domain, in a manner similar to domain-independent tuning, but taking into account the characteristics of the domain sources. (For example, if a domain has many textual attributes, then we assigned more weight to the Naive Bayes text classifier [31].) The third bar in each group (Figures 3.18.a-d) shows accuracy 19-78%.
We then explored source-dependent tuning. Given a source $S$, we assume that we already know matches between $S$ and two other sources $S_1 - S_2$ in the same domain. We used staged tuning of $\text{eTuner}$ over these known matches to obtain a tuned version of the matching system. Next, we manually tweaked the system, trying to further improve its accuracy over matching $S$ with $S_1 - S_2$. The fourth bar in each group (Figures 3.18.a-d) shows accuracy 22-81%.

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

### 3.5.5 Tuning with $\text{eTuner}$

The fifth bar (second bar from the right) of each group (Figures 3.18.a-d) then shows the accuracy of matching systems tuned automatically with $\text{eTuner}$. The results show accuracy 23-82% across all 16 groups. $\text{eTuner}$ is better than source-dependent tuning (the best tuning method so far) in 14 out of 16 cases, by 1-15%, and is slightly worse in 2 cases, by 2%. The cost of using $\text{eTuner}$ 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 $\text{eTuner}$ 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 (Figures 3.18.a-d) shows the accuracy of $\text{eTuner}$ with user-assisted workload creation (Section 3.3.2), 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
Figure 3.19: 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.

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.

### 3.5.6 Sensitivity Analysis

**Synthetic Workload:** Figure 3.19.a shows the accuracies of automatic eTuner, as we vary the size (i.e., number of schemas generated) of the synthetic workload. The accuracies are for LSD over Real Estate and Inventory, though we observed similar trends in other cases. As the workload size increases, the number of schema/data perturbation rules that...
it captures increases. This improves accuracy. After size 25-30, however, accuracy starts decreasing. This is because at this point, all perturbation rules have been captured in the workload. As the workload’s size increases, its “distance” from real workloads increases, and so tuning overfits the matching system. Thus, for the current set of perturbation rules (as detailed in Section 3.3.1), 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 [24] contains a far richer set of perturbation rules than eTuner. Nevertheless, its accuracy on 1-1 matching (as reported in [24] on a different domain) is only 62-71%.

Exploiting Prior Match Results: Figure 3.19.b shows the accuracy of LSD over Inventory, as we replaced 0%, 22%, etc. of the synthetic workload with real schema pairs that have been matched in the same domain. The results show that exploiting previously matched schema pairs indeed improves the quality of the synthetic workload, thereby matching accuracy. This is important because such prior match results are sometimes available [31, 28]. However, while such match results can complement the synthetic matching scenarios, exploiting them alone does not work as well, as we demonstrated with source-dependent tuning described in Section 3.5.4.

Runtime Complexity: Our unoptimized version of eTuner took under 30 minutes to tune a schema S, spending the vast majority of time in the staged tuning step. We expect that tuning matching systems will often be carried out offline, e.g., overnight, or as a background task. In general, the scalability of tuning techniques such as eTuner will benefit from scaling techniques developed for matching very large schemas [78] as well
as optimization within the tuning module, such as reusing results across matching steps and more efficient, specialized procedures for knob tuning.

3.5.7 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.

Figures 3.20.a-d show the results for the four matching systems, respectively. Each figure comprises of four groups, showing the results for the four domains, respectively. The first two bars in each group are reproduced from Figure 3.18. 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 Section 3.4, 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 goal is to examine how close is the knob configuration found by staged tuning to the optimal one (as found by exhaustive
Table 3.1: The performance of the staged tuner

<table>
<thead>
<tr>
<th></th>
<th>LSD</th>
<th>iCOMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inventory</td>
<td>eTuner: Automatic 0.66</td>
<td>eTuner: Human-assisted 0.669</td>
</tr>
<tr>
<td>Course</td>
<td>0.743</td>
<td>0.746</td>
</tr>
</tbody>
</table>

Table 3.1 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.

3.6 Summary

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 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 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.
Chapter 4

Using Schema Matching Systems

In the previous two chapters, we focused on generating correct semantic matches. However, in practice, employing a matching system in an application is challenging, as well. In this chapter, we look into how to exploit semantic matches (the results of a matching system) in an application. We first elaborate on the question we examine (Section 4.1), and then we describe QSM, which is an application we build to show how to exploit semantic matches (Section 4.2). We then argue that exploiting semantic matches is useful in QSM (Section 4.3) and present schemes to efficiently use matches (Section 4.4). We then present the experimental evaluation of our schemes (Section 4.5) and summarize this chapter (Section 4.6).

4.1 Employing a Schema Matching System in an Application

There are two steps in creating semantic mappings. The schema matching problem, which is finding semantic matches among several sources, is the first step of the process. The matches (the result of a matching system) are elaborated on in the next step. Thus, matches which are the result of a matching system are not directly used in an application.

There are several solutions which can discover semantic matches automatically (see [77, 34, 29, 69, 7] for recent surveys). However, the second step requires human efforts to elaborate matches into mappings [92]. At that point, one natural question to ask is, “Can we directly exploit matches in an application without going through the second step?” As
far as we know, no works have explored this issue thoroughly. As a first step to address this problem, we must build an application in which we can directly exploit semantic matches and explore this issue in depth.

Our application QSM directly exploits semantic matches, and using QSM, we show that exploiting semantic matches is beneficial. However, if exploiting semantic matches is significantly time-consuming, we cannot take advantage of exploiting them in practice. In addition to showing the usefulness of QSM, we propose some schemes to use matches efficiently.

4.2 QSM: Extracting Structure from Unstructured Data

There is an enormous amount of unstructured data on the Web. To find information from it, we mostly rely on keyword searches, such as a Web search engine. We have also seen that we can go farther by exploiting embedded structures in text [20]. For example, consider the above page about Seoul, the capital of South Korea, in Wikipedia [2]. (Figure 4.1) It contains sections of text titled “History,” “Architecture,” “Transportation,” etc. It also displays an Infobox in the right panel, which lists attribute-value pairs such as (population, 10421782) and (area_total_km, 605.25). If we can access these structures, we can provide more concrete querying features, such as a focused keyword search. For instance, we might want to find cities which had an historical fire. Using extracted structures, we can pose the keyword query “fire” over the “History” section. To do this, we implement an application, QSM (Querying Structures with Matches), which extracts structures from downloaded Wikipedia pages and provides several querying features over the extracted structures.

In our application, we have added one more step called “matching” to find semantic matches among extracted structures. Two different extracted structures can represent a semantically equivalent concept. For example, the “Budapest” city page has a section
Figure 4.1: Wikipedia page of Seoul

Figure 4.1: Wikipedia page of Seoul

titled “Timeline of the history of Budapest,” which describes the history of the city similar to the “History” section in the “Seoul” city page. Without this information that both sections describe the history of each city, we would miss the “Budapest” city page when we run the above focused keyword query (finding “fire” over the “History” attribute). In QSM, we discover semantic matches among extracted structures and exploit them when we run a query.

In the rest of this section, we first describe the architecture of QSM, which is available at http://debrecen.cs.uiuc.edu/cgi-bin/QSM/index.cgi. Then, we describe the back-end database, which stores both plain text and extracted structures. We also store matches in the back-end DB. Finally, we explain querying features provided by QSM and how we implement them.

4.2.1 The QSM Architecture

QSM consists of five components: crawler, extractor, integrator, back-end database, and query processor. The overall architecture is illustrated in Figure 4.2.
Crawler

The *crawler* loads unstructured data into the back-end DB. We download city pages from Wikipedia, but although this component is named as *crawler*, we do not actually crawl Web pages. The Wikipedia site does not allow crawling. However, it supports the exporting feature. We export city pages through this service ([http://meta.wikimedia.org/wiki/Special:Export](http://meta.wikimedia.org/wiki/Special:Export)). Currently, we export fresh city pages whenever we need to build a new database. Later, if we wish, we can easily change this component to use the Wikipedia database dump to reduce download time.

Extractor

The *extractor* discovers structures implicit in unstructured data, takes extracted structures as attributes of each Web page, and stores them into the back-end DB. To extract structures, the *extractor* employs several extraction algorithms. Each extraction algorithm generates a different set of attributes. Each set of attributes extracted by a different algorithm does not overlap; each set keeps a unique set of identifiers. However, if attributes extracted by the same extraction algorithm have the same name, we take them as the same attributes.

We implement two kinds of extraction algorithms: *SectionName* and *InfoBox*. The *SectionName* algorithm extracts the first-level section names in each Wikipedia page. Some
Wikipedia pages contain infoboxes which are general templates containing predefined attributes for specific domains. (Figure 4.1) All attributes extracted by the SectionName algorithm are taken as text attributes. The InfoBox algorithm extracts infoboxes. Since attributes in infoboxes are predefined, we specify numeric attributes beforehand and make sure that the values of the specified attributes are numerical. Other attributes extracted by the InfoBox algorithm are taken as text attributes.

**Integrator**

The integrator is a matching system which consists of two matchers, one combiner, one constraint enforcer, and one match selector.

For each pair of the attributes, a matcher calculates a similarity score. Thus, if there are $N$ extracted attributes, a matcher outputs a $N \times N$ similarity matrix. Since the name of each extracted attribute is representative in QSM, both the Name matcher and the WordNet matcher exploit attribute names only. The Name matcher examines the similarity of the names and the WordNet matcher returns the semantic relatedness (between 0 and 1) based on information found in the lexical database WordNet [38].

After applying both the Name matcher and the WordNet matcher, the Max combiner takes the maximum of the similarity scores and merges the similarity matrices into a single one.

The constraint enforcer adjusts the returned similarity matrix according to two rules. First, if two attributes have different types, they are not a match. Second, if two attributes were extracted by different extraction algorithms, they are not a match.

Finally, using the threshold-based selector, the selector returns the matches whose similarity score is greater than or equal to the specified threshold.
Back-End Database

We use a relational DBMS as our back-end storage system. Due to this choice, we can take advantages of features provided by a DBMS such as query execution, query optimization, recovery, and concurrency control. In QSM, we have selected PostgreSQL as our RDBMS. The back-end database is fully described in the next section (Section 4.2.2).

Query Processor

The query processor accepts and parses the user query. Then, it rewrites the query into a format which can be run in the back-end database. QSM supports three kinds of queries: keyword search, SQL-like query, and focused keyword query.

Keyword Search: QSM provides a simple keyword search on downloaded Wikipedia pages. Figure 4.3 shows the QSM keyword search interface. The user types a list of keywords separated by a space, then QSM returns a list of pages, which include all keywords specified by the user. The result will be sorted by the score function, rank(), which is provided by PostgreSQL [1]. The score function considers how often the keywords appear in the page and how close together the keywords are in the page. To speed up the running time, we build full-text indexes on downloaded pages while the crawler loads Wikipedia pages into the back-end database.

SQL-Like Query & Focused Keyword Query: Both SQL-like query and focused keyword query exploit extracted structures with the match information. Since these are the main querying features in QSM, we describe them in detail in Section 4.2.3.
4.2.2 Data Representation

Since relational database management systems (RDBMs) are well-known for their ability to handle structures, we use a RDBMS as our back-end storage system and store all data in the tables. The idea of using relational systems as the basis workbench for extracting and querying structure from unstructured data was initially proposed in [20]. We use data representations which are similar to the data representations described in that paper.

QSM uses four physical tables (Page table, Attribute Catalog table, Text Attribute table, and Numeric Attribute table) to store both unstructured and structured data, one table (Mapping table) to record semantic matches, and one logical table (Wide table) to let users write simple queries. Figure 4.4 and Figure 4.5 illustrate these data structures.

The Page table holds unstructured data. Each crawled page is loaded into the table as a row. Each record represents a Web page. We also store the metadata of each page:
the unique page identifier and the title of the page.

The **Attribute Catalog table** stores the metadata of each extracted attribute. We first populate the table with three attributes: AttrId1 representing the `PageId` column in the *Page table*, AttrId2 representing the `PageName` column, and AttrId3 representing the `PageContents` column. The origin of these three attributes is *Document*. Also, for other attributes, we keep a unique identifier, the attribute name, the origin, and the type. The origin of an attribute represents the extraction algorithm employed to extract the attribute, which would be either *SectionName* or *CityInfoBox*. (Both algorithms are explained in Section 4.2.1.) For the attribute types, we distinguish text attributes and numeric attributes. Considering other attribute types will be interesting future work.

Each extracted attribute instance is stored in either the **Text Attribute table** or the **Numeric Attribute table**, depending on its type. If we store all instances in one table, we are not be able to distinguish different types of attributes; therefore, all of them are stored in text format. Thus, for each different type of attribute, we need to create a different attribute table. Since we only currently consider text-type attributes and numeric attributes, we have created two attribute tables: the **Text Attribute table** and the **Numeric Attribute table**. In each attribute table, we keep the source information that shows from which page the attribute instance is extracted.

The **Mapping table** is described in Figure 4.5. For each attribute, we store a set of matches. To keep the system simple, we only consider 1-1 matches. Each match has only one attribute in each side.

Finally, we provide the **Wide table**, which is a view of these physical tables. Users write a query over the **Wide table** so that they do not need to worry about joining tables. Basically, it assumes that each row represents a Wikipedia page and that each attribute is stored in a column.
4.2.3 Querying Structures

QSM provides two kinds of queries which query structures: the SQL-like query and the focused keyword query. Both query over extracted attributes, which are the embedded structures in Pages. The SQL-like query is similar to the simple SQL query, which consists of SELECT, FROM, and WHERE. The focused keyword query is a specialized version of the SQL-like query, which focuses on querying a text (extracted) attribute.

SQL-Like Query

Syntax: The interface of the SQL-like query is shown in Figure 4.6. Since there is only one table (Wide table) to which the query refers, the FROM clause is already filled with the Wide table. The SELECT clause has the following syntax:

\[
\text{SELECT expression [ [ AS ] output.name ]}, \ldots
\]

in which expression refers to an attribute in the Wide table and output.name specifies a new name for an output column. The WHERE clause has the following syntax:

\[
\text{WHERE condition}
\]

in which condition is any arithmetic expression that evaluates to a result of type boolean.

Semantics: Before we explain the semantics of the SQL-like query, we introduce two new concepts, the extended wide table \((W^*)\) and the association table \((S)\). First, assume that the wide table \((W)\) has \(N\) rows and \(M\) columns. For each column \(c_j(j = 1 \ldots M)\) in \(W\),
\( m_j \) is a set of columns which match to the column \( c_j \) including the column \( c_j \) itself \( (m_j = \{ m_{j1} \ldots m_{jk} \} \) i.e., \( m_j \) has \( k \) elements.). The match information is given in the mapping table.

**Definition 4.2.1** The extended wide table \( (W^\ast) \) is the extended version of \( W \) with semantic matches among columns. \( W^\ast \) has the same cardinality as \( W \); it has \( N \) rows and \( M \) columns. However, \( W^\ast \) has set-valued entries instead of atomic entries. Formally, the entry of the row \( i \) and the column \( j \) in \( W^\ast \) is \( \bigcup_{l=1\ldots k} a_{il} \), in which \( a_{il} \) represents the value of the row \( i \) and the column \( m_{jl} \) in \( W \).

**Definition 4.2.2** The association table \((S)\) is the flattened version of \( W^\ast \) (Definition 4.2.1). \( S \) has more than or equal to \( N \) rows and \( M \) columns. Each row in \( W^\ast \) is represented as a set of rows in \( S \) and each row in \( S \) has an atomic value in each column. For each row in \( W^\ast \), we create a set of rows by taking the Cartesian product among column values and insert them into \( S \). Basically, each row in \( W^\ast \) is transformed into a set of rows having atomic values.

Although the user writes a SQL-like query with \( W \), QSM runs the query over the association table, \( S \) (Definition 4.2.2). Any row in \( S \) that does not satisfy the condition in the WHERE clause will be eliminated from the output. A row in \( S \) satisfies the condition if it returns true when the actual row values are substituted for any variable references. Then, QSM displays the columns specified in the SELECT clause from the output rows in a table format.

**Implementation:** Given a SQL-like query \((Q)\), the query processor generates \( S \) on-the-fly and executes the query over it. To generate \( S \), we need to generate \( W \) first. However, to reduce the execution time, we actually generate a projected version of \( W \). Let \( A \) be the union of the set of attributes used in \( Q \) and the set of attributes which match to an attribute in \( Q \). We generate \( W_A \) which is the part of \( W \) projected over \( A \), create \( S \) from \( W_A \), and run the query over it on-the-fly.
Focused Keyword Query

Syntax: Since the SQL-like query only allow arithmetic operators, we introduce a specialized SQL-like query, focused keyword query, which exploits text attributes. Figure 4.7 shows the focused keyword search interface. It has a different syntax from the SQL-like query. The user specifies a list of keywords separated by a space in the SEARCH clause, and specifies a text column name in the OVER clause.

Assume that the user has entered the following focused keyword query.

```
SEARCH keyword_1 ... keyword_k
OVER column_name
FROM WIDE_TABLE
```

Semantics: Although the user writes the focused keyword query with W, QSM runs the query over the association table, S (Definition 4.2.2). Any row in S that does not contain all keywords (keyword_1, ..., keyword_k) in the column column_name will be eliminated from the output. A column value v contains a keyword k, if the cleaned version of v contains the cleaned version of k, where a cleaned version of a string s is created by stemming each word in s first and then removing any stop words. Then, QSM displays two columns, attrId1 and column_name, from the output rows in a table format.

Implementation: Given a focused keyword query (Q), the query processor generates S on-the-fly and executes the query over it. To generate S, we need to generate W first. However, to reduce the execution time, we actually generate a projected version of W. Let A be the union of {attrId1, column_name} and the set of attributes which match to column_name.
We generate $w_A$ which is the part of $W$ projected over $A$ and create $S$ from $w_A$. Then we return rows in $S$ which contain the keywords $(\text{keyword}_1, \ldots, \text{keyword}_k)$ in column name.

4.3 Benefits of QSM

To study the usefulness of QSM, we build QSM with 242 capital city pages from Wikipedia. For the purpose of comparison, we also implement QS which is similar to QSM, except that it does not exploit semantic matches. We study the usefulness of QSM by comparing the results of QSM and QS in eight search scenarios: three SQL-like queries and five focused keyword queries. In six out of eight search scenarios, we observe that if we exploit semantic matches between attributes, we can find more results which would be missed otherwise. In the following sections, we look into six search scenarios in detail. In the other two search scenarios, both QSM and QS return the same result set.

4.3.1 Scenario 1: Find capital cities whose population is greater than 10,000,000 people

A city Wikipedia page has a population_total attribute (attrId13) in the infobox. Therefore, the query can be written as the following SQL-like query.

```
SELECT attrId1, attrId2, attrId13
FROM WIDE_TABLE
WHERE attrId13 > 1000000
```

QS returns three cities: Beijing, Kinshasa, and Seoul. However, a city infobox includes other attributes which also specify the population information of the city. Thus, attrId13 matches to the following attributes: attrId10 (population_density_km2), attrId74 (populationMetro), attrId87 (populationUrban), attrId115 (population_density_sqmi), attrId181 (population_densityUrban_sqmi), attrId253 (population_densityMetro_km2), attrId308 (population_densityUrban_km2), attrId878 (population_density). Then QSM returns 13
capitals: Bangkok, Beijing, Buenos Aires, Cairo, Dhaka, Jakarta, Kinshasa, London, Mexico City, New Delhi, Seoul, Taipei, Tehran. By exploiting semantic matches, we find 9 more results in this scenario.

4.3.2 Scenario 2: Find capital cities whose elevation is greater than 1,000

A city Wikipedia page has an elevation attribute (attrId73) in the infobox. Then, the query can be written as the following SQL-like query.

```sql
SELECT attrId1, attrId2, attrId73
FROM WIDE_TABLE
WHERE attrId73 > 1000
```

QS returns 12 cities. However, a city infobox includes other attributes which also specify the elevation information of the city. Thus, attrId73 matches to the following attributes: attrId704(elevation_max_m) and attrId708(elevation_max_ft). Then QSM returns two additional capitals: Islamabad and Zagreb. By exploiting semantic matches, we can find 2 more results in this scenario.

4.3.3 Scenario 3: Find capital cities whose area is greater than 600 km²

The area_total_km2 attribute (attrId34) in the infobox represents the area information of a city. Then the query can be written as the following SQL-like query:

```sql
SELECT attrId1, attrId2, attrId34
FROM WIDE_TABLE
WHERE attrId34 > 600
```

QS returns 41 cities. However, a city infobox includes other attributes which also specify the area information of the city. Thus, attrId34 matches to 13 other attributes extracted from the infobox. Then QSM returns 67 capitals. By exploiting semantic matches, we find
4.3.4 Scenario 4: Find capital cities having famous palaces

A city Wikipedia page can also have an Architecture section title (attrId478). Then the query can be written as the following focused keyword query:

```
SEARCH palaces
OVER attrId478
FROM WIDE_TABLE
```

QS returns seven cities. However, the architecture information of a city can be described in sections with different titles, such as “Architecture and Cityscape” or “Buildings.” By exploiting matches, QSM returns eight more capitals than QS.

4.3.5 Scenario 5: Find capital cities in which subway is one type of public transportation

A city Wikipedia page can have a Transportation section title (attrId91). Then the query can be written as following focused keyword query:

```
SEARCH subway
OVER attrId91
FROM WIDE_TABLE
```

QS returns 16 cities. However, in some city pages, the transportation information is described in the section Transport. By exploiting this match, QSM returns seven more capitals than QS.
4.3.6 Scenario 6: Find capital cities which are sister cities to Seoul, the capital of South Korea

A city Wikipedia page can have a Sister cities section title (attrId71). Then the query can be written as following focused keyword query:

SEARCH Seoul
OVER attrId71
FROM WIDE_TABLE

QS returns eight cities. However, the architecture information of a city can be described in sections with different titles, such as Sister relationships. By exploiting matches, QSM returns three more capitals than QS.

4.3.7 Summary

As analyzed in Section 4.3.1 through Section 4.3.6, by exploiting semantic matches, we can increase the query quality. Both QSM and QS show the same precision. The QSM result set always includes the result set of QS when we run the same query. On the other hand, QSM shows the better recall. In six out of eight search scenarios, QSM returns capital cities which QS misses. In summary, QSM returns a better result set than QS.

4.4 Efficiently Exploiting Semantic Matches

In Section 4.3, we show that by exploiting semantic matches, we can improve the effectiveness of query results. However, exploiting semantic matches requires extra steps - finding semantic matches of columns and generating the association table. It is, therefore, useful to consider whether it is possible to exploit matches efficiently.

In this section, we explore schemes for reducing the execution time of queries which exploit semantic matches. First, we examine how the query processor executes a query in
Section 4.4.1. We then propose several approaches that are aimed at increasing efficiency in Section 4.4.2. In Section 4.5 we empirically show that these approaches reduce the query execution time.

4.4.1 Stages in Query Processor

The query processor executes a query in three stages: the mapping stage, the preprocessing stage, and the querying stage. To describe each step in detail, we use the two user queries in Figure 4.8. Assume that Figure 4.9 shows the current snapshot of the back-end Database.

Mapping Stage

For each attribute \( A \) used in the query, the query processor retrieves a set of attributes which match the attribute \( A \) by accessing the mapping table (Figure 4.9) in the back-end database. For example, the query \( Q_1 \) returns \( \{attrId7\} \) for \( attrId4 \) and an empty set, \( \{\}\), for \( attrId1 \). \( Q_2 \) returns \( \{attrId6\} \) for \( attrId5 \). To efficiently access the mapping table, we build indexes over the AttrId column in the mapping table. This stage takes 0.10 seconds on average, which accounts for about 1.5% of the total execution time (Section 4.5).

Preprocessing Stage

In this stage, the query processor rewrites user queries to run them in the back-end database using the matches retrieved in the mapping stage. In our examples, the user queries \( Q_1 \)
and $Q_2$ are rewritten as in Figure 4.10. This stage takes 0.01 seconds on average, which accounts for about 0.1% of the total execution time (Section 4.5).

### Querying Stage

This stage is where the query processor actually executes the query in the back-end database. This stage takes 6.81 seconds on average, which accounts for about 98.4% of the total execution time (Section 4.5). This is the most time-consuming stage. To reduce the overall runtime, we focus on reducing the runtime of this stage. In the next section, we describe several schemes to improve the efficiency of this stage.

#### 4.4.2 Schemes to Exploit Matches Efficiently

In the previous section, we analyzed each stage in the querying process and identified that the querying stage is the most expensive step. Thus, to increase the efficiency of the
query processor, we focus on reducing the runtime of the querying stage. In particular, we propose schemes to help the query planner in the back-end database select a faster query plan.

Our first scheme is for both the SQL-like query and the focused keyword query. After applying the first scheme, the cost of the SQL-like query is reduced significantly. (Section 4.5) We then introduce two more schemes for the focused keyword query. In Section 4.5, we show which combination of schemes performs the best for the focused keyword query.

**Scheme 1: Enforcing the Use of Indexes**

Initially, we built indexes on every table in the back-end database. Specifically, we built the following indexes on our tables:

- In the page table, we built index on the Pageld column.
- In the attribute catalog table, we built index on the AttrId column.
• In the text attribute table, we built two indexes on each column of PagId and AttrId.

• In the numeric attribute table, we built two indexers on each column of PagId and AttrId.

However, when we analyze the physical query plan executed in the back-end database, we observe that sometimes the query planner in the back-end database selects a plan which does not use indexes. This is due to the inherent characteristic of our back-end database. To avoid this problem, we enforce the use of indexes in the query planner in the back-end database to use indexes.

**Scheme 2: Reducing the Join Size**

This scheme is based on the idea of reducing the cost of join by reducing the size of join tables, which is commonly used by the traditional query optimizer [87].

To generate an association table on-the-fly, our query processor rewrites the focused keyword query into a new SQL query which has joins in the FROM clause. (See $Q'_2$ in Figure 4.10) We can reduce the cost of this join by reducing the size of the first join table which is attrId1 in our example.

Also, we observe that the written-focused keyword query has one selection as follows:

```
to_tsvector(attrId5) @@ to_tsquery(‘revolution’)
```

The `@@` operator represents the text search operator defined in the back-end database. The selection finds attributes which contain the given keyword. In other words, if the attribute value is extracted from a page which does not contain the keyword, the attribute value cannot satisfy the selection condition.

By employing the above idea, we can reduce the size of the first join table. Basically, we repeat the selection inside the first join table SELECT clause to reduce the join cost. In our example, by applying this scheme, the query $Q'_2$ is rewritten as Figure 4.11.
Scheme 3: Pushing Selections Down

This scheme is based on the idea of pushing a selection down, which is also commonly used by the traditional query optimizer [87].

To generate an association table on-the-fly, our query processor creates two intermediate tables: one for the page table and another for the keyword attribute. In our example, \(Q'_2\) in Figure 4.10 generates two intermediate tables called attrId1 and attrId5, where the keyword attribute is attrId5.

Our idea is to apply the selection when we generate the intermediate table for the keyword attribute. Then we join intermediate tables. In our example, by applying this scheme, the query \(Q'_2\) is rewritten as Figure 4.12.

This scheme has one more benefit: we can build inverted indexes beforehand. Previously, we could not build inverted indexes beforehand, since we did the text search over a column in an intermediate table. However, with this scheme, we do the text search over
the AttrContents column in the text attribute table. Thus, we build inverted indexes over this column offline and re-use them whenever we run a focused keyword query.

4.5 Empirical Evaluation

We conducted a variety of experiments to evaluate whether the schemes in Section 4.4 led to reduced query evaluation time.

Data Sources: We first built QSM with 1,000 Wikipedia city pages. Our preliminary results showed that the proposed schemes reduced query evaluation time, but not as much as we expected. We decided to increase the number of pages to 10,000. Instead of building QSM from the scratch, we synthetically generated new pages by reusing the 1,000 pages which were already downloaded. The reason was that if we downloaded 10,000 pages, we would need to find semantic matches among approximately 20,000 attributes extracted from fresh pages. This would have taken a huge amount of time. However, since we reused downloaded pages, we did not need to go through the matching process. Since the performance of the matching process was not our focus, we used 10,000 synthetically generated pages.

User Queries: We asked volunteers who do not know the internal architecture of QSM to propose queries. We selected 10 SQL-like queries and 10 focused keyword queries.

Experimental Methodology: For each query type, we run experiments with several configurations. For the SQL-like query, we run experiments with two configurations: (a) the default QSM and (b) the QSM with Scheme 1. For the focused keyword query, we add six more configurations: (c) QSM with Scheme 2, (d) QSM with Scheme 3, (e) QSM with Scheme 1 & 2, (f) QSM with Scheme 1 & 3, (g) QSM with Scheme 2 & 3, and (h) QSM with all schemes.

For each experiment set, we run each query 30 times. To simulate the actual querying activity, we run queries randomly, not sequentially. Thus, instead of running queries in
the order of \((\text{Query}_1^1, \text{Query}_1^2, \ldots, \text{Query}_1^{30}, \text{Query}_2^1, \text{Query}_2^2, \ldots, \text{Query}_2^{30}, \ldots, \text{Query}_{10}^1, \text{Query}_{10}^2, \ldots, \text{Query}_{10}^{30})\), in which \(\text{Query}_i^j\) represents the \(j\)th execution of \(\text{Query}_i\), we run them in an order similar to \((\text{Query}_4^1, \text{Query}_1^1, \ldots, \text{Query}_i^j, \ldots, \text{Query}_7^{30})\).

**Performance Measures:** For each query type, we report the average run time of each stage in the querying process with the average total run time.

### 4.5.1 SQL-Like Query

Figure 4.13 shows our results for the SQL-like query. It shows the execution time of each stage during the query process, including the total time. (Each stage is described in Section 4.4.1.) For each group, the left bar represents the execution time of the default QSM, and the right bar represents the execution time of QSM with the Scheme 1.

The results show that exploiting the Scheme 1 reduces the querying stage time by 0.76 seconds. Also, both the mapping time and the preprocessing time account for a small portion of the total run time as mentioned in Section 4.4.1.

In summary, Figure 4.13 shows that exploiting Scheme 1 reduces the query evaluation time for the SQL-like query.

### 4.5.2 Focused Keyword Query

We now examine how proposed schemes reduce the runtime of a focused keyword query. Figure 4.14 shows the runtime in a format similar to that of Figure 4.13, but with more configurations. For each stage group, the eight bars from left to right respectively represent the run time of the QSM configuration which exploits (a) no schemes (i.e., the default system), (b) Scheme 1, (c) Scheme 2, (d) Scheme 3, (e) both Scheme 1 and Scheme 2, (f) both Scheme 1 and Scheme 3, (g) both Scheme 2 and Scheme 3, and (g) all schemes - Scheme 1, Scheme 2, and Scheme 3.

We first look at the effectiveness of each scheme. Unlike the SQL-like query, exploit-
Figure 4.13: The performance of the SQL-like queries

Scheme 1 does not reduce the querying stage time of the focused keyword query, but each of the other schemes reduces the querying stage time. Exploiting Scheme 2 reduces the time by 4.6 seconds and exploiting Scheme 3 reduces it by 11.15 seconds. This shows that exploiting proposed schemes reduces the query evaluation time. Also, both the mapping time and the preprocessing time account for small portion of the total run time, as mentioned in Section 4.4.1.

Next, we look into other configurations which combine the proposed schemes. As expected, combining schemes further reduces the query execution time. Among the combination configurations which combine the proposed schemes, the configuration exploiting both Scheme 1 and Scheme 3 reduces the querying stage time the most, by 11.19 seconds.

From our experimental results, we observe three characteristics of proposed schemes. First, Scheme 1 is a complementary scheme. Although Scheme 1 itself does not reduce the query evaluation time, when it is combined with other schemes, it reduces the time further than when the other scheme is exploited alone. For example, when we exploit Scheme 2 only, we reduce the querying stage time by 4.6 seconds. But, if we exploit both Scheme 1 and Scheme 2, the querying stage time is reduced by 4.82 seconds, which shows
Second, although both Scheme 2 and Scheme 3 reduce the execution time, exploiting Scheme 3 shows better performance than Scheme 2. The reason is that by pushing the selection down, Scheme 3 actually generates a smaller intermediate table to be joined than Scheme 2.

Finally, exploiting both Scheme 2 and Scheme 3 shows better performance than exploiting Scheme 2 only, but worse than exploiting Scheme 3 only. Adding Scheme 3 to Scheme 2 means adding one more condition for the selection in the second intermediate table. (See Section 4.4.2.) This actually reduces the size of the selection operation and also the join cost. However, adding Scheme 2 to Scheme 3 means introducing a new selection operation in the first intermediate table. Thus, it causes extrac cost. This also explains why combining all schemes does not show the best performance among combination configurations.

In conclusion, combining Scheme 1 and Scheme 3 shows the best performance among all combination configurations.
4.6 Summary

Traditionally, semantic matches discovered by schema matching systems are elaborated into mappings to be exploited in practice. However, elaborating matches to mappings is a very time-consuming process. This chapter showed that semantic matches can be directly exploited in practice. First, it described an application QSM which provides two querying features exploiting semantic matches: a SQL-like query and a focused keyword query. Using QSM, we showed that exploiting semantic matches improves the quality of query results. Thus, by exploiting matches, QSM returns more correct results, which would be missed otherwise. Finally, we argued that although exploiting semantic matches demands extra execution time, we can reduce the execution time to a reasonable amount with three schemes. Our experimental results showed that our proposed schemes can reduce the query execution time up to 11.19 seconds, thus demonstrating that QSM can exploit semantic matches efficiently.
Chapter 5

Related Work

This chapter discusses recent search efforts related to the work presented in this dissertation.

5.1 Schema Matching Techniques

Schema matching has received increasing attention over the past two decades (see [77, 34, 29, 26, 69, 7, 6] for recent surveys). In this section, we give an overview of techniques for schema matching that have been developed.

5.1.1 Rule- vs. Learning- Based Solutions

A wealth of matching solutions has been developed. The solutions can be roughly divided into two groups: rule-based and learning-based solutions (though several solutions which leverage ideas from the fields of information retrieval and information theory have also been developed [21, 44]).

Rule-Based Solutions: Many of the early, as well as current, matching solutions employ hand-crafted rules to match schemas [63, 71, 16, 65, 54, 59].

In general, hand-crafted rules exploit schema information, such as element names, data types, structures, numbers of subelements, and integrity constraints. A broad variety of rules has been considered. For example, the TranScm system [63] employs rules such as “two elements match if they have the same name (allowing synonyms) and the same number of subelements.” The DIKE system [71, 70, 72] computes the similarities 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 system MOMIS [16, 9]
compute the similarities of schema elements as the weighted sum of the similarities of
names, data types, and substructures. The CUPID system [54] employs rules that cate-
gorize 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 learning-based techniques do. Second, they typically op-
erate only on schemas (not on data instances) and, hence, are fairly fast. Third, they can
work very well in certain types of applications and in domain representations that are
amenable to rules [67]. Finally, rules can provide a quick and concise method to capture
valuable user knowledge about the domain. For example, the user can write regular ex-
pressions that encode times or phone numbers, or quickly compile a collection of county
names or zip codes that help recognize these 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 for-
mat, distribution, frequently occurring words in the attribute values, and so on) that could
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 tex-
tual 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 ef-
forts to assist in the current ones. Thus, in a sense, systems that rely solely on rule-based
techniques have difficulties learning from the past and improving over time. The above
Learning-Based Solutions: Many such solutions have been proposed in the past decade (e.g., [50, 21, 10, 11, 31, 24, 36, 66, 13]). The solutions have considered a variety of learning techniques and exploited both schema and data information. For example, the SemInt system [50] 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 [31] employs Naive Bayes over data instances, and develops a novel learning solution to exploit the hierarchical nature of XML data. The iMAP system in Chapter 2, as well as the ILA and HICAL systems developed in the AI community [75, 81], match the schemas of two sources by analyzing the descriptions of objects that are found in both sources. The Autoplex and Automatch systems [10, 11] use a Naive Bayes learning approach that exploits data instances to match elements.

5.1.2 Discovering 1-1 and Complex Semantic Matches

The vast majority of current solutions focus on finding 1-1 semantic matches. To the best of our knowledge, the only work completed on complex matching other than iMAP in Chapter 2 is the work found in [91]. This work considers finding complex matches between two schemas by first mapping the matches into a domain ontology, then constructing the matches based on the relationships inherent in that ontology. Such ontology-based matching would work very well in certain contexts (e.g., see the date searcher in Section 2.3.1) and can be added to iMAP as additional searchers.

5.1.3 Exploiting Multiple Types of Information

There is also a growing realization that schema- and data-related evidence regarding the matching of two schemas is often inadequate for the matching process. Hence, several
works have advocated learning from external evidence beyond the two current schemas. Several types of external evidence have been considered. Some recent works advocate exploiting past matches [31, 28, 11, 77, 36, 12]. The key idea is that a matching tool must be able to learn from past matches and to successfully predict matches for subsequent, unseen matching scenarios.

The work [52] goes farther 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 match two specific real estate source schemas. Similarly, the works [41, 90] describe settings in which one must match multiple schemas all at once. Here the knowledge gleaned from each matching pair can help match other pairs, and as a result, we can obtain better accuracy than simply matching a pair in isolation. The work [57, 58] 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 the system by answering relatively simple questions. Those answers can be used to further build the system, including matching the schemas of the data sources in the system. Through this method, the enormous burden of schema matching is lifted from the system builder and spreads “thinly” over a mass of users.

### 5.1.4 Multi-Component Matching Solutions

The synergistic nature of matching solutions (e.g., based on rules, learning, information retrieval, information theory, graph algorithms, etc.) suggests that an effective matching solution should employ many techniques, each involving the types of information that can be effectively exploited. To this end, several recent works [12, 28, 31, 36, 78, 24] have described a system architecture that employs multiple modules called matchers, each of which efficiently exploits a certain type of information to predict matches. The system then combines the matchers’ predictions to arrive at a final prediction for the matches. Each matcher can employ one individual technique or a set of matching techniques as
described earlier (e.g., hand-crafted rules, learning methods, IR-based solutions). Combining the predictions of matchers can be manually specified [28, 12] or automated to some extent by using learning techniques [31].

Besides being able to exploit multiple types of information, the multimatcher 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 can be easily added when they become available. Our work in Chapter 2 also shows that the above solution architecture can be extended successfully to handle complex matches.

5.1.5 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 form 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 to the matching of multiple elements (e.g., “at most one element matches house-address”). To address this problem, several recent works [59, 54, 31, 32] have advocated moving the handling of constraints to after the matchers. In this way, 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 condition of a heuristic is that “two nodes match if their neighbors also match,” variations of which have been exploited in many systems (e.g., [63, 54, 59, 68]). A common scheme is to iteratively change the matching of a node based on those of its neighbors. The iteration is carried out once or
twice, or until some convergence criteria are reached.

5.1.6 Further Related Work

An important current research direction is to evaluate the above multi-component architecture in real-world settings. The works [12, 78] take some initial steps in this direction. The work [12] builds Protoplasm, an industrial-strength schema matching system, while the work [78] examines the scalability of matching systems to very large XML schemas.

A related direction focuses on creating robust and widely useful matcher operators, as well as developing techniques to quickly and efficiently combine operators for a particular matching task. The next logical direction is to make the frameworks easy to customize for a particular set of matching tasks. eTuner in Chapter 3 aims at automating this customization.

Finally, eTuner can also be seen as part of the trend toward self-tuning databases, to reduce the high total cost of ownership [4, 18, 17].

5.2 Leveraging Synthetic Workloads

As far as we know, our work in Chapter 3 is the first to employ synthetic workloads in schema matching. Synthetic workloads and inputs have also been recently exploited in several other contexts. The Recovery-Oriented Computing (ROC) project [73] 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 target systems in order to evaluate their robustness [15]. The work [37] constructs synthetic text documents that exhibit certain properties. It then examines how various information retrieval methods work with respect to these documents. The goal is to examine formal properties of these information retrieval methods. Finally, several learning approaches have also exploited artificial inputs. The work [60] 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 Chapter 3. Second, synthetic data can help significantly in improving 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., [56, 46, 49, 61, 19, 88]), as the data at the sources evolves (see [56] for more detail). We plan to adapt the same idea to improve record linkage systems (e.g., [40, 5, 89]).

As described in Section 5.1.3, several recent works exploit previously matched schema pairs to improve matching accuracy. 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, are often more comprehensive, and can be tailored to a particular matching situation. In Chapter 3, 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.
5.3 Compositional Approaches

Arguably, the success of relational data management derives partially from the following three factors: (1) it is possible to define a small set of core operators (e.g., select, project, join) so 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), and (3) everything is made as “declarative” as possible in order to enable effective user interaction, customization, and rapid development of applications.

It can be argued that the development of schema matching solutions has followed a similar compositional approach. First, monolithic solutions were developed. Next, they were “broken down,” and multi-component solutions were developed. eTuner in Chapter 3 suggests that these solutions can be “distilled” to extract a core set of operators, and that solutions composed of these operators can be tuned; in other words, they can be partially optimized. There are then two interesting future directions: 1) can we develop better tuning/optimization techniques? and 2) can we make schema matching solutions as “declarative” as possible?

Compositional solutions have also been considered and are currently under development in several other contexts, most recently in record linkage [8], data integration [79], and text data management [86]. Other contexts include information extraction (e.g., [39]), solving crossword puzzles [45], and identifying phrase structure in NLP [76].
Chapter 6
Conclusions and Future Work

Schema matching is the problem of discovering semantic correspondences between disparate sources. It is an important problem that will continue to play a prominent role in a variety of data sharing and exchange applications, such as data integration on the World-Wide Web. In this dissertation, we identified several key challenges in schema matching focused on practicality.

As the first such work, we developed a matching solution that discovers complex matches, which are prevalent in practice, along with one-to-one matches. The main idea was to employ multiple search modules to find complex matches and use various types of domain knowledge to guide the search.

Second, we showed that tuning a matching system is important to improve matching solutions to a point in which they are attractive in practice. Our key idea was to synthesize a collection of matching scenarios, for which we already knew the ground truth matches, and then use this collection to tune a matching system.

Finally, we argued that we can directly exploit discovered semantic matches without any extra user efforts. We developed an application in which we directly exploit semantic matches. Also, we showed that we can exploit them efficiently.

We made significant inroads into understanding and developing practical schema matching solutions, but substantial work remains toward the goal of achieving a comprehensive matching solution. In what follows, we discuss several directions for future work.

**Efficient User Interaction:** As noted before, schema matching requires user interaction. Any practical matching solution must handle this problem. Even when an almost-perfect
matching solution is available, the system administrator still has to verify the discovered matches. In practice, there could be a large number of matches. Hence, efficient user interaction is crucial. The main purpose of this problem is to minimize user interaction while maximizing the impact of user feedback.

**Efficient Large-Scale Schema Matching Systems:** Traditionally, the schema matching problem has focused on identifying semantic matches between two schemas. However, in some settings, we need to find semantic correspondences among a large number of sources. In particular, this problem arises when we integrate multiple sources over the Internet. A few works have examined this problem [42, 90], but the efficiency of their matching solutions have not addressed all issues. However, in practice, the efficiency of a matching system is critical. The key challenge of this problem is discovering semantic matches efficiently enough to be run in reasonable amount of time, and yet effectively enough to be used right away.
References


