Providing Best-Effort Services in Dataspace Systems

Xin Dong

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2007

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Xin Dong

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

____
Alon Y. Halevy

Reading Committee:

____
Philip A. Bernstein

____
Alon Y. Halevy

____
Dan Suciu

Date: ______________________

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature______________________________

Date______________________________

University of Washington

Abstract

Providing Best-Effort Services in Dataspace Systems

Xin Dong

Chair of the Supervisory Committee:
Professor Alon Y. Halevy
Computer Science and Engineering

Nowadays many data sharing applications need to manage a *dataspace* [68], which contains a number of heterogeneous data sources and partially unstructured data. Such scenarios include large enterprises, collaborative scientific projects, digital libraries, personal information, and the Web. Understanding the relationships between the data sources requires specifying schema mappings, such as one stating that full-name in one data source corresponds to the concatenation of first-name and last-name in another data source. However, the data sources in a dataspace are only loosely coupled, so we may not have schema mappings specified up front. This dissertation studies *how to provide best-effort search, querying and browsing services in a dataspace system*, even when precise schema mappings are not present.

To provide useful services over *all* data in a dataspace, we need to resolve heterogeneity in the data. Heterogeneity exists at three levels in a dataspace. At the instance level, the same real-world entity can be referred to using different values; for example, a person can be referred to as "Mike" in some data sources and as "Michael" in others. At the schema level, the same domain can be described using different schemas; for example, a person can be described by his first-name and last-name in one data source and by his full-name and other-name in another data source. At the query level, user queries can be composed according to a schema different from the source schema, or even in a language that is not supported by the data model of the source data; for example, a user may compose a SQL

query whereas some data sources are unstructured.

In this dissertation we describe solutions for resolving heterogeneity in a dataspace. To resolve heterogeneity at the instance level, we describe an algorithm that reconcile references that refer to the same real-world entity. Our algorithm can be applied to references that belong to multiple classes where rich associations between the references exist. To resolve heterogeneity at the schema level, we propose the concept of probabilistic schema mapping, with which we can return approximate answers even when precise mappings do not exist. We study the complexity of query answering with respect to probabilistic mappings. To resolve heterogeneity at the query level, we design an index over heterogeneous data in a dataspace. Our index extends inverted lists to capture both text and structure of the data to facilitate efficient answering of queries that combine keywords and structure. In addition, we design an algorithm that answers structured queries on unstructured data, such that we can provide seamless search on both structured and unstructured data.

Finally, we have grounded all our technical solutions to a particular system, the Semex Personal Information Management System. Semex provides a logical view of one's personal information, such that it supports associative browsing and provides seamless search and querying over one's personal data.

TABLE OF CONTENTS

Page

List of Figures . . . iii

List of Tables . . . vi

Chapter 1: Introduction . . . 1
- 1.1 Traditional Data Integration Systems . . . 1
- 1.2 Dataspaces . . . 3
- 1.3 Levels of Heterogeneity in Dataspaces . . . 6
- 1.4 Contributions of the Dissertation . . . 9
- 1.5 An Application: Personal Information Management . . . 13
- 1.6 Outline . . . 16

Chapter 2: Resolving Instance-Level Heterogeneity: Reference Reconciliation . . . 17
- 2.1 Problem Definition and Overview of Our Approach . . . 17
- 2.2 Reference Reconciliation Algorithm . . . 25
- 2.3 Computing Similarity Scores . . . 37
- 2.4 Experimental Evaluation . . . 41
- 2.5 Related Work . . . 52
- 2.6 Discussion . . . 54
- 2.7 Summary . . . 58

Chapter 3: Resolving Query-Level Heterogeneity I: Indexing Dataspaces . . . 59
- 3.1 Problem Definition and Overview of Our Approach . . . 59
- 3.2 Indexing Structure . . . 68
- 3.3 Indexing Hierarchies . . . 73
- 3.4 Experimental Evaluation . . . 81
- 3.5 Related Work . . . 93
- 3.6 Discussion . . . 95
- 3.7 Summary . . . 96

Chapter 4: Resolving Query-Level Heterogeneity II: Answering Structured Queries on Unstructured Data . . . 97
4.1 Problem Definition and Overview of Our Approach . . . 98
4.2 Constructing Query Graphs . . . 101
4.3 Extracting Keywords . . . 105
4.4 Experimental Evaluation . . . 111
4.5 Related Work . . . 116
4.6 Discussion . . . 117
4.7 Summary . . . 118

Chapter 5: Resolving Schema-Level Heterogeneity: Probabilistic Schema Mapping 119
5.1 Overview of Our Results . . . 119
5.2 Definition . . . 123
5.3 Complexity of Query Answering . . . 130
5.4 Representation of Probabilistic Mappings . . . 141
5.5 Broader Classes of Mappings . . . 147
5.6 Related Work . . . 148
5.7 Discussion . . . 148
5.8 Summary . . . 149

Chapter 6: An Application: The SEMEX Personal Information Management System 151
6.1 Browsing and Querying in SEMEX . . . 151
6.2 System Architecture . . . 156
6.3 Related Work . . . 160
6.4 Summary and Overarching PIM Themes . . . 162

Chapter 7: Conclusions and Future Directions . . . 164
7.1 Key Contributions . . . 164
7.2 Future Work . . . 165

Bibliography . . . 169

Appendix A: Algorithm of Building a Hybrid-ATIL . . . 180

Appendix B: Proofs from Chapter 5 . . . 185
B.1 Proof for Theorem 5.15 . . . 185
B.2 Proofs for Other Results in Chapter 5 . . . 190

LIST OF FIGURES

Figure Number Page

1.1 Architecture of data integration systems. 2

1.2 Heterogeneity at various levels in a dataspace. 7

2.1 Heterogeneity at the instance level in a dataspace. 18

2.2 An example association network: (a) the domain model; (b) the list representation of the association network; (c) the graph representation of the association network. 20

2.3 Example for reference reconciliation: (1) references extracted from personal data; (2) reconciliation results. 22

2.4 Algorithm for constructing the dependency graph. 29

2.5 The dependency graph for references in Figure 2.2(b): (a) the subgraph for references $a_1, a_2, p_1, p_2, p_3, p_4, p_5, p_6, c_1$ and c_2. (b) the subgraph for references p_5 and p_8. 30

2.6 Algorithm for information propagation in reference reconciliation. 33

2.7 Enrichment of references in Figure 2.5(b): (a) the original subgraph representing the similarity of references p_5 and p_8 (node m_6), and the similarity of p_5 and p_9 (node m_8); (b) the subgraph after the first step of reference enrichment; (c) the subgraph after the second step of reference enrichment. . 34

2.8 Algorithm for reference enrichment in reference reconciliation. 35

2.9 Algorithm for enforcing constraints in reference reconciliation. 37

2.10 Algorithm for reference reconciliation. 38

2.11 Domain Model for references from Cora. 41

2.12 Contribution of each type of evidence and each algorithmic feature. The top-left most point represents IndepDec and the bottom-right most point represents DepGraph. 49

3.1 Heterogeneity at the query level in a dataspace. 60

3.2 An example association network: (a) list representation; (b) graph representation. 62

3.3 The framework of our index: an extended inverted list over a collection of heterogeneous data. 65

3.4 Algorithms for (a) constructing an AAIL for a given association network, and (b) answering a predicate query using an AAIL. 72
3.5 The algorithm for looking up a prefix in a Hybrid-ATIL. 78
3.6 Efficiency of answering predicate queries. In each column, the longer bar shows the overall query-answering time and the shorter bar shows index-lookup time. 83
3.7 Efficiency of looking up different types of indexes in answering predicate queries with attribute clauses and neighborhood keyword queries (a) on shallow-hierarchy association network, and (b) on deep-hierarchy association network. 87
3.8 Efficiency of index updates: (a) instance updates; (b) structure updates. . . . 89
4.1 Searching and querying a dataspace. The left side of the graph shows the various models of queries and the right side shows the various models of data. The lines in the middle show the possible combination of data and query models and the communities that work on them. 98
4.2 Framework of our approach to keyword extraction. 100
4.3 The query graph for the query in Example 4.1. 103
4.4 Constructing the query graph for the SQL query in Example 4.3: (a) the preliminary graph constructed in the first step; (b) the compact graph constructed in the second step 105
4.5 The query graph with i-scores and r-scores for the query in Example 4.1: (a) the initial (i-score, r-score) pairs; (b) the information flow representing the effect of the "Dataspaces" label; (c) the information flow representing the effect of the "Paper" label. 105
4.6 Algorithm for i-score update. 108
4.7 Extracting keywords from query graph in Figure 4.5(a): (a) the i-scores of the labels after selecting the labels "Dataspaces" and "2005"; (b) the i-scores of the labels after selecting the label "Paper". 108
4.8 Algorithm for label selection. 109
4.9 Top-2 and top-10 precision for queries over different schemas without applying domain knowledge. 112
4.10 Top-10 precision for queries with length 0 in (a) the movie domain and (b) the geography domain, and with length 1 in (c) the movie domain, and (d) the geography domain. In (a) and (b) the ValueTable line and the QueryGraph line overlap, as the two methods extracted the same keywords. 113
4.11 Top-10 precision of queries with one attribute value in (a) the movie domain and (b) the geography domain, and with two attribute values in (c) the movie domain and (d) the geography domain. 114

4.12 Top-10 precision for queries over different schemas without applying domain knowledge (QueryGraph) and with applying domain knowledge (QueryGraph_DK). 115

5.1 Heterogeneity at the schema level in a dataspace. 120

5.2 Example 5.1: (a) a probabilistic schema mapping between S and T; (b) a source instance D_S; (c) the answers of Q over D_S with respect to the probabilistic mapping. 121

5.3 Example 5.11: (a) a source instance D_S; (b) a target instance that is by-table consistent with D_S; (c) a target instance that is by-tuple consistent with D_S; (d) $Q^{table}(D_S)$; (e) $Q^{tuple}(D_S)$. 126

5.4 Example 5.14: (a) $Q_1^{tuple}(D)$ and (b) $Q_2^{tuple}(D)$. 132

5.5 Example 5.29: the p-mapping in (a) is equivalent to the 2-group p-mapping in (b) and (c). 142

5.6 Example 5.34: the p-mapping in (a) corresponds to the p-correspondence in (b). 144

6.1 A sample screenshot of the Semex interface. The browsing trace in the middle pane answers the query elaborated in Example 6.1. 152

6.2 Semex architecture. 157

A.1 The algorithm for building a hybrid-ATIL. 181

A.2 The *stack* and *counter* array after (a) processing keyword “tian//desc//” and “tian//desc//name//firstName//”; (b) processing keyword “tian//desc//name//lastName//”; (c) processing keyword “tian//desc// name//nickName//”; (d) popping up prefix “tian// desc//name//nickName//”; (e) popping up prefix “tian//desc//name//”; (f) popping up prefix “tian//desc//”. The difference of each sub-graph from the former one is highlighted using the italic font. 183

LIST OF TABLES

Table Number Page

2.1 Properties of our data sets: the number of extracted references, the number of real-world entities and the reference-to-entity ratio. 41

2.2 Average precision, recall and F-measure for each class of references. 44

2.3 Average precision, recall and F-measure for Person references when only the email or paper subsets are considered and when the full data sets are considered. 45

2.4 Performance for different PIM data sets measured in occurrence-based precision, recall and F-measure. 46

2.5 Performance for different PIM data sets measured in representation-based precision, recall and F-measure. 46

2.6 Performance for different PIM data sets measured in diversity and dispersion. 47

2.7 The number of Person reference partitions obtained by different variations of the algorithm on PIM data set A. For each mode, the last column shows the improvement in recall (measured as the percentage reduction in the difference between the number of result partitions and the number of real-world entities) from *Attr-wise* to *Contact* by considering all the additional available evidence. For each evidence variation, the last row shows the recall improvement by applying reconciliation propagation and reference enrichment. The bottom-right cell shows the overall recall improvement of DepGraph over IndepDec. 48

2.8 Effect of considering constraints on reconciliation: the precision, recall, the number of real-word entities that are involved in erroneous reconciliations (false-positives), and the graph size in terms of the number of nodes. 51

2.9 Precision, recall and F-measure for the Cora data set. 52

3.1 The inverted list for the association network in Example 3.1. 66

3.2 The ATIL for the association network in Example 3.1. 69

3.3 The AAIL for the association network in Example 3.1. 71

3.4 The Dup-ATIL for the association network in Example 3.1. The difference from Table 3.2 is highlighted using bold font. 74

3.5 The Hier-ATIL for the association network in Example 3.1. The difference from Table 3.2 is highlighted using bold font. 75

3.6 The Hybrid-ATIL with threshold t=1 for the association network in Example 3.1. The difference from Table 3.5 is the row for “tian//name////”. . . . 77

3.7 The KIL with threshold $t = 1$ for the association network in Example 3.1. To save space, we only show the rows where the indexed keywords start with "birch". 81

3.8 Comparison of search efficiency using the KIL, using separate indexes as proposed in [85], and using a simple inverted list. 84

3.9 Comparison of indexing efficiency for different types of inverted lists. 88

3.10 Index-lookup time for answering (a) the original queries and (b) suffix queries on 250MB data sets with perturbed keywords. For the purpose of comparison, we also list the index-lookup time on the 25MB data. 91

3.11 Indexing time and index-lookup time for 10GB XML data sets. Note that some index-lookup time is not reported as the type of queries does not apply. 92

ACKNOWLEDGMENTS

I was led to the world of computer science by my mother, when I was eight and when personal computers were first introduced to China. Finding how fascinated I was with those mysterious electronic boxes, my mother decided to learn programming herself and then teach me. I soon reached the point when I could learn programming by myself and now my mother is still a beginner-level Windows user; however, I will always remember that my first programming teacher is my mother. The support and encouragement from my mother and also from my father are always with me. I am deeply indebted to their love and their faith in me.

I am grateful to Wenqi Fang, my computer teacher in high school, who introduced Algorithms and Artificial Intelligence to me. I still remember that afternoon when he handed me a newspaper article that introduces A* algorithm. I was impressed by the intelligence a computer can gain and that article opened a door for me. My deep interest in algorithms started from that article and from his guide during my high-school study.

I would like to thank Xiao'ou Chen, my adviser for my Master's degree at Peking University. Although my area was Graphics at that time, he suggested me learning XML as he saw it promising. That suggestion leads to my interest in database.

I owe a special debt to my Ph.D. adviser, Alon Halevy. I am indeed lucky in having him as my adviser. He guided my entrance into the database world, sparked my interest in data integration, taught me how to do research and how to think critically, discussed with me on my research projects, advised me on how to write crisp papers and give punchy presentations, answered my questions with patience, forgave me for the English errors I made in my writing, encouraged me when I felt down, and supported me in all aspects of my graduate life. He set a role model for me with his passion, sharpness, knowledge, and sense of humor. This work would not have been possible without the guidance and insight

from him.

I am extremely grateful to Dan Suciu and Philip Bernstein for discussing with me my research ideas, giving me invaluable feedback on my work, and mentoring me on how to pursue a good career and be a successful researcher. I would like to thank Pedro Domingo, from whom I learned Machine Learning, Oren Etzioni, from whom I learned Artificial Intelligence, and Efthimis Efthimiadis, from whom I learned Information Retrieval. The knowledge I obtained from their classes and the inspiring discussions with them on my research are very beneficial to this dissertation. I would also like to express my sincere gratitude to Magda Balazinska in the database group for helpful suggestions, and to James Landay for agreeing on serving in my supervisory committee.

Special thanks to Jayant Madhavan, Michelle Liu, and Yuhan Cai for interesting discussions and hard work on the SEMEX project. I would thank Igor Tatarinov for guiding me in my first research project. I owe many thanks to Rachel Pottinger, Zack Ives, Anhai Doan, and Gerome Miklau, the first several database students at UW, for being good resources and support. I want to thank Luke McDowell, Nilesh Dalvi, Chris Re, Michael Gubanov, Nodira Khoussainova, and other members in the database group at UW for their discussions and feedback. My appreciation also goes to my officemates, Sahngyun Hahn, Raphael Hoffman, Miryung Kim, Pradeep Shenoy, and so on, for the wonderful time I spent with them.

My husband, Jun Zhang, has been accompanying me during these years through my success and failures. No matter whether I am excited, inspired, discouraged, or stressed, he is the person whom I can always talk to and resort to. He gave me the strength to overcome the obstacles I encountered during my Ph.D. study. My son, Franklin Zhang, who is five month old at this time, seems to understand that his mom needs to graduate and try to coordinate from the beginning of his life. This dissertation is dedicated to them.

Chapter 1

INTRODUCTION

Nowadays many data sharing applications need to manage a *dataspace* [68], which contains a number of heterogeneous data sources and partially unstructured data. Such scenarios include large enterprises, collaborative scientific projects, digital libraries, personal information, and the Web. Heterogeneity exists in various aspects of the data in a dataspace. It can exist at the *schema* level; for example, a person can be described by his first-name and last-name in one data source and by his full-name and used-name in another data source. It can also exist at the *instance* level; for example, the person can be referred to as "Mike" in some data sources and as "Michael" in others. Understanding the relationships between the data sources requires specifying schema mappings, such as one stating that full-name in one data source corresponds to the concatenation of first-name and last-name in another data source. However, the data sources in a dataspace are only loosely coupled, so we may not have schema mappings specified upfront. This dissertation studies *how to provide best-effort search, querying and browsing services in a dataspace system*, even when precise schema mappings are not present.

1.1 Traditional Data Integration Systems

We begin this chapter by reviewing the state of the art for managing heterogeneous data. As a consequence of the advent of modern networking technologies and the rapid evolution of data management technologies, many applications need to manage a multitude of data sources, where data sets are produced independently by different organizations. These data sets can be highly heterogeneous, describing the same domain using different schemas and referring to the same real-world entity using different attribute values. To provide users a unified view of these data, we need to combine data residing at autonomous and

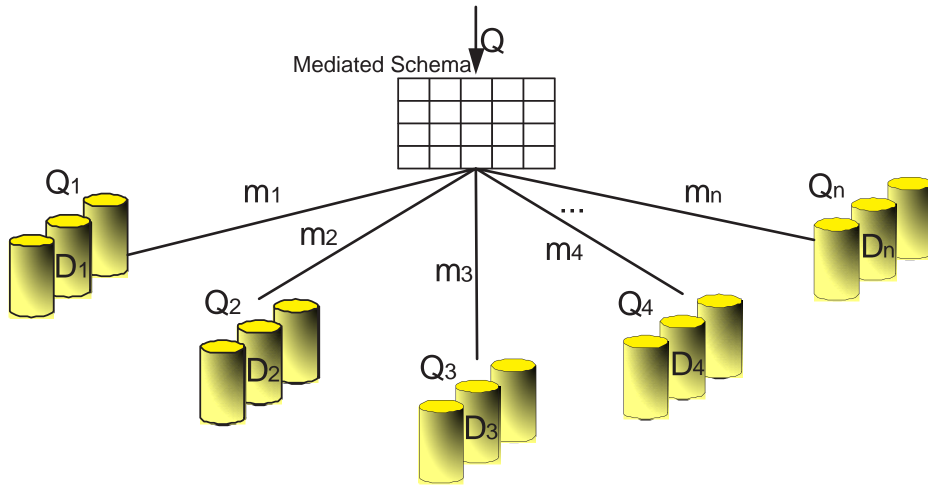


Figure 1.1: Architecture of data integration systems.

heterogeneous sources.

As depicted in Figure 1.1, a traditional data integration system (surveyed in [128, 90, 69, 70]) obtains this goal by specifying a *mediated schema*, which provides an integrated and virtual view of the disparate sources and captures the salient aspects of the domain being considered. Users query the underlying data by composing queries over the mediated schema. To retrieve answers from the various data sources, the system builds schema mappings between the source schemas and the mediated schema, and uses these mappings to reformulate a user query into a set of queries on the data sources.

A *schema mapping* specifies the semantic relationship between the contents of different data sources. As a simple example, consider the following source schema S and mediated schema T.

```
S: Author(aID, aName)
   Paper(pID, pTitle, pYear)
   AuthoredBy(aID, pID)
T: Paper(title, year, author)
```

The source schema describes papers and their authors using three tables. The mediated schema describes such information using a single table. The schema mapping between them,

specified as follows, states that the Paper table in the mediated schema can be obtained by joining the three tables in the data source.

```
SELECT P.pTitle AS title, P.pYear AS year, A.aName AS author
FROM Author AS A, Paper AS P, AuthoredBy AS B
WHERE A.aID=B.aID AND P.pID=B.pID
```

A data integration system heavily relies on the mappings between the data sources and the mediated schema for query reformulation. However, it is well known that creating and maintaining such mappings is non-trivial and requires significant resources, upfront effort, and technical expertise. Recently, there has been significant research on semi-automatically generating these schema mappings. For our example source and target schemas, the schema mapping tool first creates correspondences between the attributes, such as the correspondence between pTitle in the source schema and title in the mediated schema, and the one between aName in the source schema and author in the mediated schema. Then, according to the attribute correspondences, the tool generates the query that specifies the schema mapping. However, even with such tools, domain experts need to get involved to refine the automatically generated mappings. Therefore, generating schema mappings is still the major bottleneck in building a data integration system.

1.2 Dataspaces

Recently, [68] proposed the concept of *dataspace*, which is a collection of loosely-coupled heterogeneous data sources. Schema mappings may not exist between the data sources, either because it is hard and tedious to create them, or because it is even impossible to generate precise mappings. Before we describe dataspace systems in detail, we first give three example scenarios that motivate us for studying dataspace management.

Personal information management: Owing to the affordability of large amounts of storage, individual computer users have developed their own vast collections of data on their desktops. Personal data typically contain unstructured data, such as text documents, and also some structured or semi-structured data, such as address books and spreadsheets.

Those data are created and managed by different applications and are heterogeneous by nature. Currently, operating systems organize personal information into directory hierarchies, where data created by different applications are often stored in different directories for the particular applications.

Personal Information Management (PIM) [79] aims at offering easy access and manipulation of all of the information on one's desktop, with possible extension to mobile devices, personal information on the Web, and even all the information accessed during a person's life time. Recent desktop search tools such as Google Desktop Search [63], Yahoo! Desktop Search [140] and Windows Desktop Search [135] are an important first step towards PIM; however, they are limited to keyword search. The next step for PIM is to allow the user to search the desktop in more meaningful ways, asking questions such as "find the restaurant where we went to celebrate Joan's birthday," "find all emails with comments on my dissertation sent by my committee members," and "list all software I have participated in developing and count the number of lines of code I wrote." To answer such questions, a PIM system needs to seamlessly mesh the disparate personal data created by different applications. However, typical PIM users are not skilled enough to provide mappings between data sources; sophisticated users, on the other hand, may not be motivated to go through the tedious process of mapping generation. Hence, we often do not have mappings between the data sources.

Web-scale information management: The World Wide Web has been dominated by unstructured data since its inception; however, recently we are witnessing an increase both in the volume and in the variety of structured data on the web. The prime example of such data is the *deep web*, referring to content on the web that is stored in databases and served by querying HTML forms. More recent examples of structure are a variety of annotation schemes (*e.g.*, Flickr [53], the ESP game [131], Google Co-op [64]) that enable people to add labels to content (pages and images) on the web, and Google Base [65], a service that allows users to upload structured data about any domain into a central repository.

Ideally, a web search engine should provide one uniform interface that enables users to search all web data. To take a concrete example, suppose a user poses the query "The Da

Vinci Code" to a web search engine. We would like the engine to return relevant webpages, and in addition links to web forms where users can buy the book or the movie DVD, entries from Google Base that sell second-hand books, and links to special sites that have been annotated by movie enthusiasts as relevant. Providing such a search service requires an understanding of the semantic relationships between the heterogeneous web sources. However, the heterogeneity of the structure is at a scale that we have never seen before. Completely reconciling heterogeneity in this context is inconceivable, especially given that the structure and contents of the websites can evolve over time.

Bio-informatics data management: The last ten years has witnessed an explosion of biological data, including sequenced human genome, gene arrays, protein structure, and scientific literature. Biological databases typically consist of a mixture of data files, metadata, sequences, annotations, and relational data obtained from various sources [126].

To do global analysis, biological researchers often need to access data from multiple archival databases. Biology encompasses many domains of knowledge (molecular and cell biology, genetics, structural biology, pharmacology, physiology, etc.), where each domain has its own terminology and data needs and different domains are concerned with overlapping or complementary entity types. Furthermore, the domains themselves are a subject of study and knowledge on these domains is still evolving over time. Consequently, by nature there will always be parts of the schema mappings that cannot be precisely specified.

Whereas in such applications it is critical for providing quality search and browsing, we may not have mappings between the data sources. A DataSpace Support Platform (DSSP) solves this problem by taking a *data co-existence* approach. It emphasizes *pay-as-you-go* data management: provide some services from the outset and evolve the schema mappings between the different sources on an as-needed basis. Given a query, a DSSP generates *best-effort* or approximate answers from data sources where perfect mappings do not exist. When a DSSP discovers a large number of sophisticated operations (*e.g.*, answering relational queries, data mining) required over certain sources, it will guide the users to make additional effort to integrate those sources more closely.

The goal of my dissertation is to *provide best-effort search, querying and browsing as a*

dataspace system evolves, even when we do not have schema mappings or have only imperfect mappings between the data sources. This is an important first step towards pay-as-you-go data management, because only if we can offer "mapping-later" and even "mapping-never" services, can we realize the promise of the pay-as-you-go principle.

1.3 Levels of Heterogeneity in Dataspaces

Managing dataspaces raises many new challenges that have not been addressed in previous data-sharing systems. In particular, to provide useful services over *all* data in a dataspace, the DSSP needs to resolve heterogeneity at three levels: *instance level, schema level,* and *query level* (see Figure 1.2).

Instance-level heterogeneity: The first level of heterogeneity is the instance level: the same real-world entity can be referred to using different values. In Figure 1.2, "Stonebraker, M." in D_1, "Mike Stonebraker" in D_2, and "Michael Stonebraker" in D_3, indeed refer to the same real-world person. Variations in representation arise for multiple reasons: misspellings, use of abbreviations, different naming conventions, naming variations over time, and the presence of several values for particular attributes.

To provide seamless search, querying and browsing of a dataspace, it is crucial that a DSSP can detect the different object instances that refer to the same real-world entity. This problem is known as *reference reconciliation.* This problem is hard in general, and has been the subject of study in the Statistics, Database and Machine Learning communities. Most of the previous work considered techniques for reconciling instances referred to by tuples in a *single* database table, where typically each tuple contains a fair number of attributes and all tuples contain the same set of attributes. A DSSP, instead, often needs to tackle complex information spaces that contain instances of multiple classes and rich relationships between the instances; in addition, instances may have only few attributes and instances of the same class may have values for different sets of attributes.

Schema-level heterogeneity: The second level of heterogeneity is the schema level: the same domain can be described using different schemas. Since the schemas are often independently developed by different people in different real-world contexts, they often use different

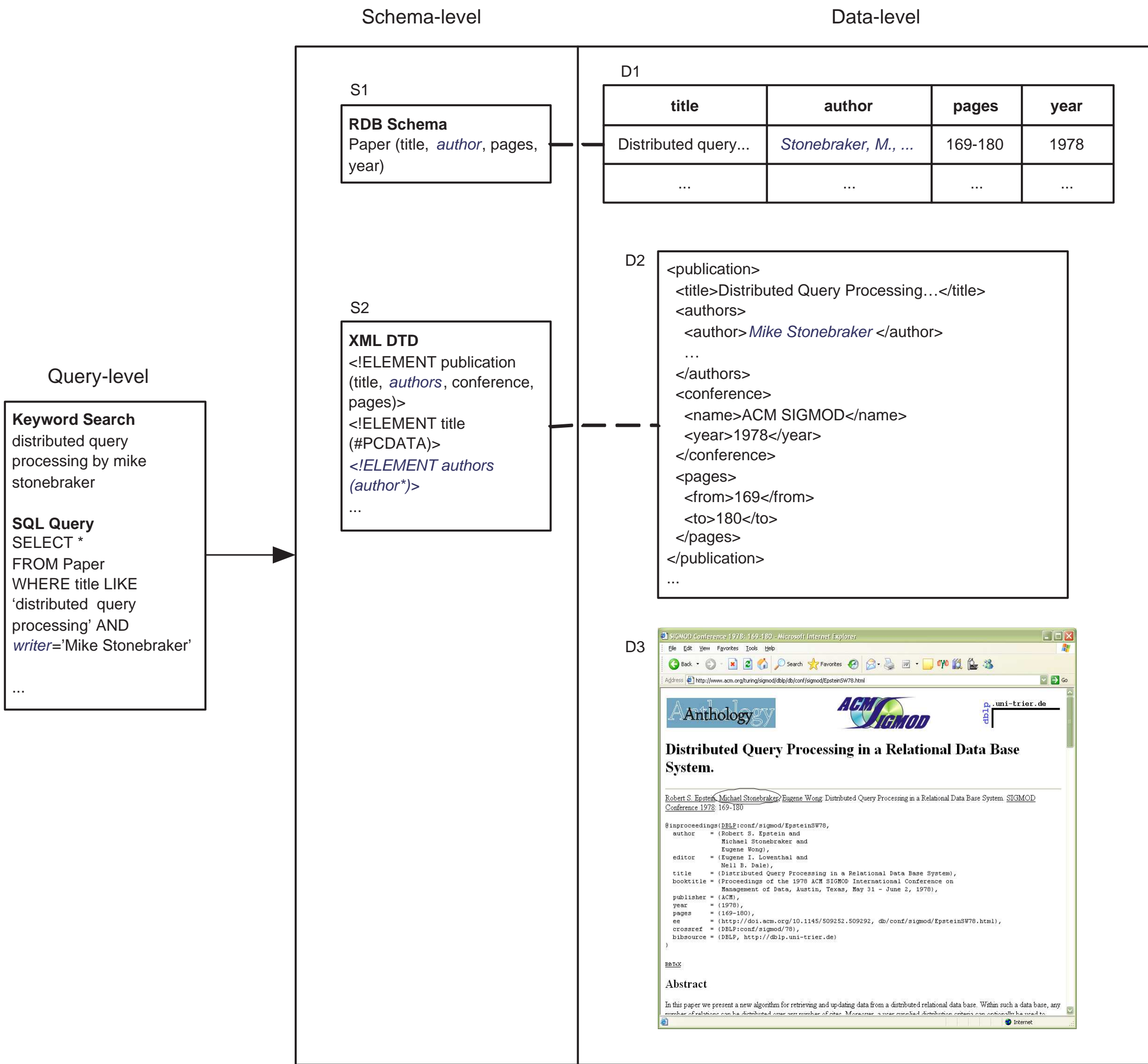


Figure 1.2: Heterogeneity at various levels in a dataspace.

structure and terminology. Consider Figure 1.2 as an example. To describe authors of a paper, D_1 uses the author attribute of the Paper table, D_2 nests the authors element under the publication element and each authors element can have a set of author sub-elements, and D_3 uses text to describe a paper and its authors.

Recall from Section 1.1 that to answer a query on heterogeneous data sources, we need to understand the semantic relationships between schemas, described by schema mappings. Although there has been active research on semi-automatic schema mapping, specifying schema mappings remains a hard and tedious process and in many dataspace applications it is even not possible to find precise mappings.

Query-level heterogeneity: The third level of heterogeneity is the query level: user queries can be composed not only according to a schema different from the source schema, but also in a language that is not supported by the data model of the source data (*e.g.*, a SQL query over unstructured data). For example, to search for a paper titled "distributed query processing" and authored by "Stonebraker", a user can either do a keyword search, or compose a SQL query, as shown in Figure 1.2. No matter what kind of queries the users ask, they wish to apply the queries to all content in the dataspace, and retrieve information from both structured and unstructured data sources. In addition, the users may not know or remember the exact structural terms or attribute values in the data sources, but rather use those that they feel best express their information needs.

A DSSP aims to provide seamless search and querying over *all* data in the dataspace. Recently the database community has studied how to do keyword search on structured data such as relational data or XML data [73, 4, 15, 139, 74]. However, providing seamless querying is far beyond supporting keyword search on structured and unstructured data: the system needs to identify the data sources that are relevant to the query, it needs to provide a meaningful ranking for answers from different data sources, and it needs to be able to answer structured queries on unstructured data. Furthermore, structured querying is often too strict in requiring detailed knowledge of the underlying schemas, whereas keyword search is often inadequate for sophisticated users who wish to specify structural requirements. To improve users' querying experience, a DSSP needs to support new kinds of querying

paradigms that combine structured and unstructured querying in a fundamental way.

1.4 Contributions of the Dissertation

This dissertation makes the following contributions.

Reference reconciliation

Chapter 2 describes a reference reconciliation algorithm to resolve heterogeneity at the instance level. It studies the problem for a complex information space, which does not just consist of a set of tuples as in previous work, but can be viewed as a network of instances of multiple classes and associations between the instances, where each instance in itself may contain only limited amount of information.

Our algorithm extends traditional reference reconciliation to complex information spaces. First, we make extensive use of *context information* (the associations between references) to provide evidence for reconciliation decisions. For example, given two references to persons, we will consider their co-authors and email contacts to help decide whether to reconcile them. Second, we *propagate* information between reconciliation decisions for different pairs of references. For example, when we decide to reconcile two papers, we obtain additional evidence for reconciling the person references to their authors. This, in turn, can further increase the confidence in reconciling other papers authored by the reconciled persons. Third, we address the lack of information in each reference by *reference enrichment*. For example, when we reconcile two person references, we gather the different representations of the person's name, collect her different email addresses, and enlarge her list of co-authors and email-contacts. This enriched reference can later be reconciled with other references where we previously lacked information for the reconciliation.

To incorporate these three strategies, we construct a *dependency graph*, where a node represents the similarity between a pair of references or attribute values, and an edge represents the dependency between the similarities. Our framework captures the dependency between similarities of different reference pairs, allows propagating information from one reconciliation decision to another, and helps enriching references right at the time of reconciliation.

We evaluate our reconciliation algorithm on several personal information data sets and on the Cora citation data set. In the PIM context, the experiments show that our approach significantly improves on standard reference reconciliation techniques. On the Cora data set, the results show that our approach is comparable to recent adaptive approaches for paper reconciliation, and at the same time produces accurate reconciliation on person and publisher instances.

Indexing heterogeneous data

As one step in resolving heterogeneity at the query level, Chapter 3 describes a generic index for loosely-coupled heterogeneous data to support efficiently answering queries that contain keywords and are structure aware.

In particular, we define two types of queries that allow users to combine keywords and structural requirements in a fundamental way. The two types of queries are *predicate queries* and *neighborhood keyword queries*. A predicate query allows the user to specify both keywords and simple structural requirements, such as "a paper with title 'Birch', authored by 'Raghu', and published in 'Sigmod 1996'". A neighborhood keyword query is specified by a set of keywords, but differs from traditional keyword search in that it also explores *associations* between data items, and so it leverages additional structure that may exist in the data or may have been automatically discovered. For example, searching for "Birch" returns not only the papers and presentations that mention the Birch project, but also people working on Birch and conferences in which Birch papers have been published. For both types of queries, we emphasize returning *possibly related* data in answers to queries rather than only the data that strictly satisfy the query.

Our contribution is a framework that indexes heterogeneous data from multiple sources through a (virtual) central association network, so as to support predicate queries and neighborhood keyword queries. Our index extends the traditional inverted list by capturing not only text values, but also structural information when it is present. Specifically, we describe extensions to inverted lists that capture attribute information and associations between data items. We also explore several methods for extending inverted lists such that they can incorporate various types of heterogeneity, including synonyms and hierarchies of

attributes and associations.

Our experimental results show that our techniques improve search efficiency by an order of magnitude and perform better than competing alternatives. In addition, the experiments show that our technique scales well and supports efficient index updates.

Answering structured queries on unstructured data

Chapter 4 proposes seamless querying of structured and unstructured data. Querying structured and unstructured data in isolation has been the main subject of research for the fields of Databases and Information Retrieval. Recently the Database Community has studied the problem of answering keyword queries on structured data such as relational data or XML data. The only combination that has not been fully explored is answering structured queries on unstructured data. We design an algorithm that answers structured queries on unstructured data by constructing a keyword query from a given structured query, and submitting the keyword query to the search engine for retrieving unstructured data. This is another contribution we make in resolving heterogeneity at the query level.

The key element in our solution is to construct a *query graph* (essentially the association network) for the structured query. Our algorithm selects node labels and edge labels (representing attribute values and schema elements that appear in the query) that best summarize the query graph, and uses them as keywords to the search engine. Our goal is to include only *necessary* labels to construct the keyword query, so keyword search returns exactly the query results and excludes irrelevant documents.

We describe an algorithm that extracts keywords based on the *informativeness* and *representativeness* of a label: the former measures the amount of information provided by the label, and the latter is the complement of the distraction that can be introduced by the label. One important observation that guides our algorithm is that the informativeness of a label also depends on the already selected keywords. For example, consider searching a paper instance. The term “paper” is informative if we know nothing else about the paper, but its informativeness decreases if we know the paper is about “dataspaces”, and further decreases if we also know the paper is by “Halevy”. In other words, in a query graph, once we select a label into the keyword set, the informativeness of other labels is reduced. Our algorithm

uses the query graph to model the effect of a selected label on the informativeness of the rest of the labels, and select the labels with the highest informativeness and representativeness in a greedy fashion.

The experimental results show that our algorithm works fairly well for a large number of query schemas from various domains even if we do not have domain knowledge, and the results improve when knowledge of the schema and the data is available.

Probabilistic schema mapping

Finally, Chapter 5 proposes the concept of *probabilistic schema mapping* to accommodate query answering in the presence of imprecise schema mappings and resolve heterogeneity at the schema level.

We define probabilistic schema mapping as a set of possible (ordinary) mappings between a source schema and a target schema, where each possible mapping has an associated probability. We begin by considering a simple class of mappings, where each mapping describes a set of correspondences between the attributes of a source relation and the attributes of a target relation. We introduce two possible semantics of probabilistic mappings. In the first, called *by-table* semantics, we assume there exists a single correct mapping between the source and the target, but we don't know which one it is. In the second, called *by-tuple* semantics, the correct mapping may depend on the particular tuple in the source to which it is applied. In both cases, the semantics of query answers is a generalization of certain answers [2] for data integration systems.

Beyond the definition of probabilities schema mappings, we make the following contributions. First, we study query answering with respect to probabilistic mappings. We show that the data complexity of answering select-project-join queries in the presence of probabilistic mappings is PTIME for by-table semantics and #P-complete for by-tuple semantics. We identify a large subclass of real-world queries for which we can still obtain all the by-tuple answers in PTIME.

The size of a probabilistic mapping may be quite large, since it essentially enumerates a probability distribution by listing every combination of events in the probability space. In practice, we can often encode the same probability distribution much more concisely. Our

second contribution is to identify two concise representations of probabilistic mappings for which query answering can be done in PTIME in the size of the mapping. We also examine the possibility of representing a probabilistic mapping as a Bayes Net, but show that query answering may still be exponential in the size of a Bayes Net representation of a mapping.

Finally, we show we can extend our results to several more powerful mapping languages, such as arbitrary GLAV mappings and complex mappings (where the correspondences are between sets of attributes). We show how our definitions and formal results carry over to these cases.

1.5 An Application: Personal Information Management

Whereas our technical contributions apply to dataspace applications in general, we have grounded them into a particular system, the SEMEX (short for SEMantic EXplorer) Personal Information Management System [43, 24]. To further motivate the technical problems we solve in the dissertation, we next describe the goals of SEMEX and the challenges we face to achieve these goals. We describe the SEMEX system in detail in Chapter 6.

As early as in 1945, Vannevar Bush [22] described the vision of a Personal Memex, which was motivated by the observation that our mind does not think by way of directory hierarchies, but rather by following *associations* between related objects. For example, a user may think of a person, emails sent by the person, then jump to thinking of one of her papers, papers cited by that paper, etc. However, currently operating systems organize personal information into directory trees, thus users need to examine different directories or open specific applications to access the data. Desktop search tools such as Google Desktop Search [63], Yahoo! Desktop Search [140] and Windows Desktop Search [135] provide keyword search on data across applications, but it does not allow users to browse or search their information by following associations either.

We built the SEMEX System, which offers users a flexible platform for personal information management. SEMEX has two main goals. The first goal is to enable browsing personal information by association. The challenge is to *automatically* create associations between data items on one's desktop. The second goal is to leverage the associations we created to provide a better search tool to increase users' productivity. For this purpose, SEMEX enables

lightweight information integration and provides a uniform interface to search across both structured and unstructured data. We next elaborate on each of these goals.

1.5.1 Browsing by Association

The key impediment to browsing personal information by association is that data on the desktop is stored *by application* and in directory hierarchies, whereas browsing by association requires a *logical view* of the objects on the desktop and the relations between them. As a simple example, information about people is scattered across our emails, address book, and text and presentation files. Even answering a simple query, such as finding all of one's co-authors, requires significant work. SEMEX provides a logical view of one's personal information, based on *meaningful* objects and associations. The instantiation of the logical view is indeed an association network. For example, users of SEMEX can browse their personal information by objects such as Person, Publication and Message and associations such as AuthoredBy, Cites and AttachedTo. Importantly, since users are typically not willing to tolerate any overhead associated with creating additional structure in their personal data, SEMEX attempts to create the logical view automatically.

It is impossible to anticipate in advance all the sources of associations between objects in one's personal information. Hence, it is important that SEMEX be extensible in the ways in which associations can be added. SEMEX obtains objects and associations from multiple types of sources. First, some associations are obtained by programs that are specific to particular file types. In the simple case, SEMEX extracts objects and associations from Address books and email clients (*e.g.*, senders and recipients, phone numbers and email addresses). In more complex cases, SEMEX extracts some associations (*e.g.*, AuthorOf) by analyzing Latex and Bibtex files. Second, associations can be obtained from external lists or databases (*e.g.*, a list of one's graduate students or departmental colleagues). Finally, complex associations can be derived from simple ones (*e.g.*, one's co-authors).

Extracting associations from multiple sources raises one of the important technical challenges for PIM. An association relates two objects in the world, and the objects are represented by references. In order to combine multiple sources of associations and to support

effective browsing and querying, SEMEX needs to *reconcile references*; that is, to decide if two references represent the same object in the world. Unlike previous work, the reconciliation problem is exacerbated in our context because each of the references typically contains only little information. For example, a person can be referred to by her full name or an abbreviated name in a citation, or only by an email address in emails. In Chapter 2 we will discuss how we leverage the association network to resolve such instance-level heterogeneity.

1.5.2 Seamless Search and Querying

One of the key services of SEMEX is to support efficient search and querying over one's personal data, with extension to organizational data and the Web. Personal information contains both unstructured data, such as text documents, and structured data, such as address books and spreadsheets. In addition, a large volume of application data are a mix: they contain some metadata that indicates the structure, and also rich text bodies (*e.g.*, emails and Latex files). In addition to the data already on the desktop, a user may often access some organizational data, such as the DBLP data, which contain useful information about publications and researchers, and a departmental database that lists the courses and instructors in the department.

Whereas data in the personal information space can be highly heterogeneous, typical users are not skilled enough to provide schema mappings between the data sources. To understand the semantic relationship between data sources and resolve such schema-level heterogeneity, SEMEX needs to rely on automatic schema mapping tools, which may generate imprecise mappings. We need to make the best use of such imprecise mappings and generate approximate answers with respect to them even when the users are not in the loop to refine the mappings. Chapter 5 proposes probabilistic schema mapping for this purpose.

Finally, to satisfy users' various needs, SEMEX allows a spectrum of search strategies, ranging from simple keyword search to expressive but sophisticated querying. Meanwhile, personal data vary from unstructured to structured, and well-defined mappings may not exist between different sources. To provide a uniform search interface through which users can search *all* data unaware of the underlying heterogeneity, SEMEX needs to tackle the

possible mismatch between the user query and the data model of a data source. In Chapter 3 and Chapter 4 we describe how we leverage the association network to resolve such query-level heterogeneity.

1.6 Outline

The following four chapters–Chapter 2 to 5–describe reference reconciliation, indexing heterogeneous data, answering structured queries on unstructured data, and probabilistic schema mappings, respectively. They elaborate on the ideas outlined in Section 1.4. Then, Chapter 6 describes how we incorporate these technical solutions in the Semex Personal Information Management system. Finally, Chapter 7 concludes the dissertation and discusses directions for future research.

Parts of this dissertation have been published in conferences. In particular, the reference reconciliation algorithm (Chapter 2) is described in [45], the indexing method (Chapter 3) is described in [44], the algorithm for answering structured queries on unstructured data (Chapter 4) is described in [91], and the concept of probabilistic schema mapping is described in [47]. Finally, the Semex system is described in [43] and demonstrated in [24].

Chapter 2

RESOLVING INSTANCE-LEVEL HETEROGENEITY: REFERENCE RECONCILIATION

One of the major impediments to integrating data from multiple sources in a dataspace is resolving references at the instance level (see Figure 2.1). Data sources have different ways of referring to the *same* real-world entity. For example, recall from Section 1.5 that in personal information the same person can be referred to using his full name, abbreviated name, email addresses, and so on. To join data from multiple sources, and therefore, to provide seamless search, querying and browsing, we must detect when different references refer to the same real-world entity. This problem is known as *reference reconciliation*.

This chapter studies resolving heterogeneity at the instance level by reference reconciliation. We begin by defining the reconciliation problem and giving an overview of our approach. Then, Section 2.2 describes our framework for reference reconciliation, and Section 2.3 describes the computation of similarities. Section 2.4 presents experimental evaluation and Section 2.5 describes related work. Finally, Section 2.6 discusses the limitations and extensions of our algorithm and Section 2.7 summarizes this chapter.

2.1 Problem Definition and Overview of Our Approach

Reference reconciliation has received significant attention in the literature, and its variations have been referred to as record linkage [137], merge/purge [72], de-duplication [115], hardening soft databases [34], reference matching [96], object identification [125] and identity uncertainty [97]. Most of the previous work considered techniques for reconciling references to a *single* class, where typically the data contain many attributes with each instance. However, a dataspace system often needs to tackle complex information spaces where instances of multiple classes and rich relationships between the instances exist, a class may have only a few attributes, and references typically have unknown attribute values.

In the rest of this section, we first define our notion of association network. We then

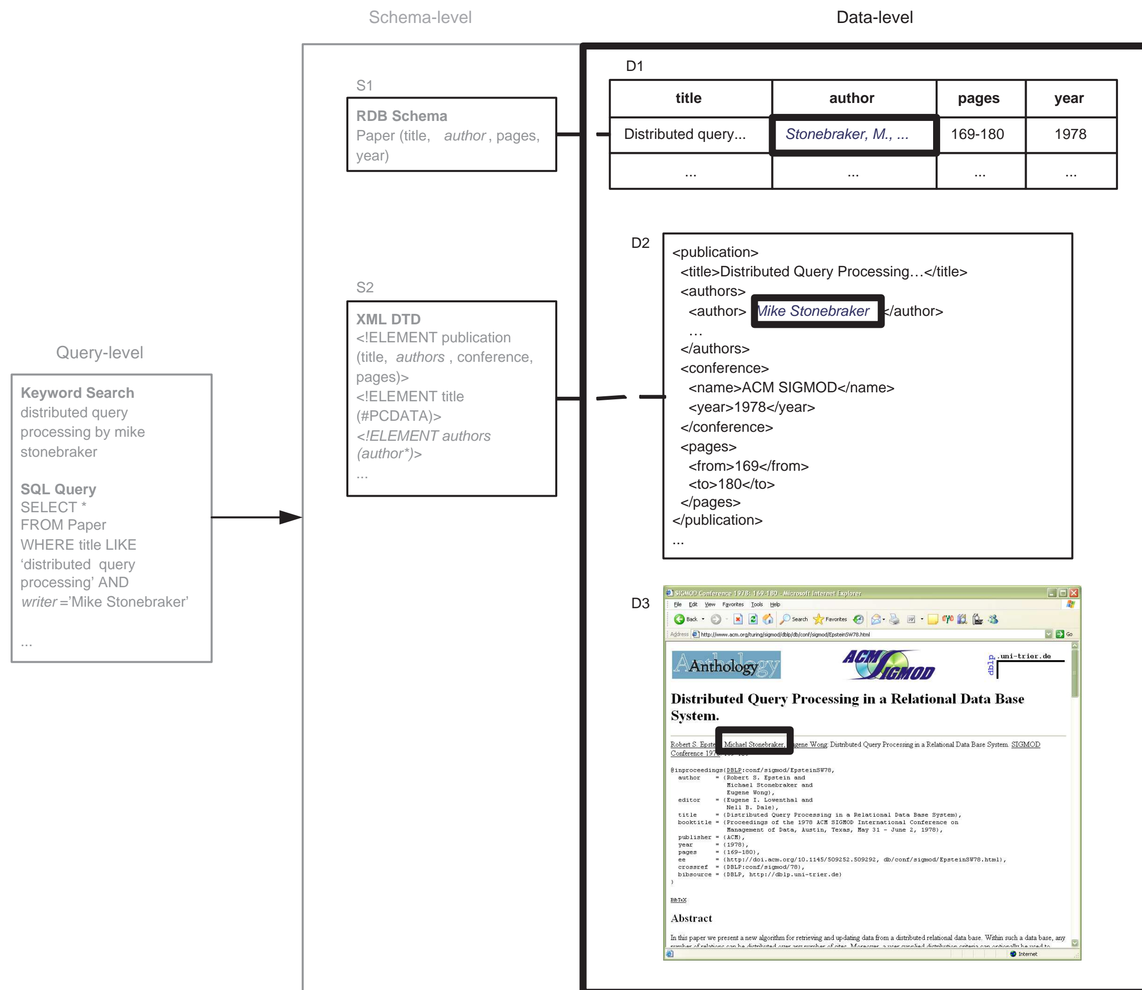


Figure 2.1: Heterogeneity at the instance level in a dataspace.

formally define the reference reconciliation problem and give an overview of our approach using an example from the application of personal information management.

2.1.1 Association network

We describe the data from different sources using a *domain model*, which is close in spirit to an E-R model. A domain model contains object *classes* such as Person, Publication and Message, and *associations* such as AuthorOf, Cites, Sender, MentionedIn. Each object class contains a set of *attributes*, and an attribute value is of atomic type (*e.g.*, string, integer, etc.). An object instance can have multiple values for the same attribute. An association is a relationship between two instances (being of different classes or of the same class). We assume that associations are directional, and in particular, each association has a *domain* and a *range*, both being classes. For convenience, for a particular class C, we call both the attributes of C and the associations whose domain is C as the *properties* of C. For example, both attribute title and association author (whose domain is Paper and range is Person) are properties of the class Paper.

Example 2.1. *Figure 2.2(a) shows a subset of a domain model for a personal information management application. The domain model contains four classes:* Person, Article, Conference *and* Journal*, each with a particular set of properties. The association properties are denoted with "*". As an example, the* Person *class has two attribute properties,* name *and* email*, and two association properties,* coAuthor *and* emailContact*, whose values are links to other* Person *instances. These associations link a* Person *instance with other* Person *instances that they have co-authored with or have exchanged emails with.* □

To further accommodate heterogeneity, our domain model also models (1) *synonyms* among class, attribute and association names (as well as an association being synonymous with an attribute), and (2) class or property *hierarchies*. Taking property hierarchies as an example, hierarchies can be of *sub-property* type or *sub-field* type. The sub-property type describes the *is-a* relationship between properties; for example, father is a sub-property of parent. The sub-field type describes the *is-a-part-of* relationship between properties; for example, city is a sub-field of address. Heterogeneity often arises in the way data sources

Person (name, email, *coAuthor, *emailContact)

Article (title, year, pages, *authoredBy, *publishedIn)

Conference (name, year, location)

Journal (name, year, volume, number)

(a)

a_1={title=“Distributed Query Processing in a Relational Data Base System”, pages = “169-180”, authoredBy = p_1, authoredBy = p_2, authoredBy = p_3, publishedIn = c_1}

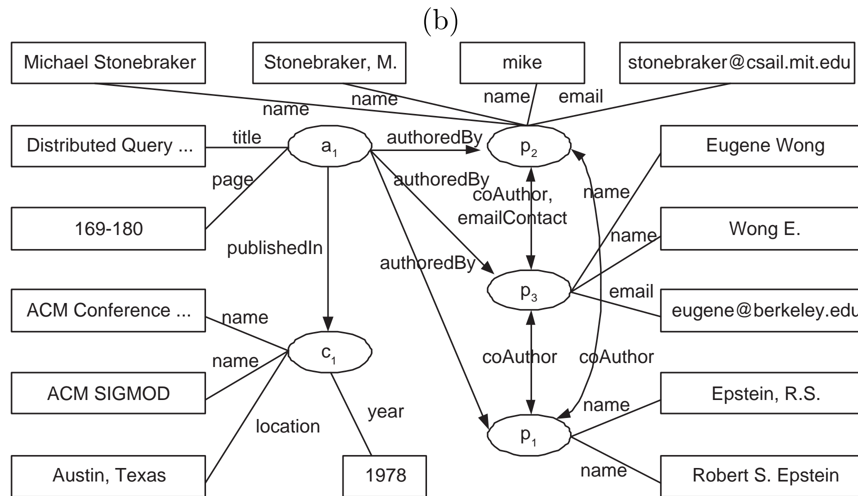
p_1={name = “Robert S. Epstein”, name = “Epstein, R.S.”, coAuthor = p_2, coAuthor = p_3}

p_2={name = “Michael Stonebraker”, name = “Stonebraker, M.”, name = “mike”, email = “stonebraker@csail.mit.edu”, coAuthor = p_1, coAuthor = p_3, emailContact = p_3}

p_3={name = “Eugene Wong”, name = “Wong, E.”, email = “eugene@berkeley.edu”, coAuthor = p_1, coAuthor = p_2, emailContact = p_2}

c_1={name = “ACM Conference on Management of Data”, name = “ACM SIGMOD”, year = “1978”, location = “Austin, Texas”}

(b)



(c)

Figure 2.2: An example association network: (a) the domain model; (b) the list representation of the association network; (c) the graph representation of the association network.

model structure hierarchies (*e.g.*, different ways of modeling addresses and people).

A data instance of a domain model is called an *association network*, consisting of *object instances* and the *associations* between the instances. In the rest of this dissertation, we represent an association network in two ways: the *list representation* represents an association network by listing for each object its attribute and association properties; the *graph representation* represents an association network as a graph, where ellipses nodes represent instances, rectangles nodes represent attribute values, undirectional edges represent attributes, and directional edges represent associations. We illustrate by an example.

Example 2.2. *Consider the association network depicted in Figure 2.2(b). It contains one* Article *instance* a_1*, three* Person *instances* p_1, p_2, p_3*, and one* Conference *instance* c_1*. Figure 2.2(b) lists for each instance its attribute values and associated instances. For example,* Paper a_1 *has title "Distributed Query Processing..."; it is associated with* Person *instances* p_1*,* p_2 *and* p_3*, and* Conference *instance* c_1*. Figure 2.2(b) describes the association network using a graph representation.* □

Finally, a *reference* partially specifies an instance of a particular class: it has a set of values (possibly empty set) for each attribute of that class. Typically a reference is extracted from the data by some extractors.

Example 2.3. *Figure 2.3(a) shows a set of references extracted from a personal data set. The* Article *references* a_1 *and* a_2*,* Person *references* p_1 *to* p_6*, and* Conference *references* c_1 *and* c_2 *are extracted from two Bibtex items. The other three* Person *references,* p_7*,* p_8 *and* p_9*, are extracted from emails. Note that they are indeed references to the instances shown in Figure 2.2.* □

2.1.2 Problem definition

Ultimately, our goal is to populate the association network according to the domain model such that each instance of a class refers to a single real-world entity, and each real-world entity is represented by at most a single instance. However, what we are given as input are *references* to real-world objects, obtained by some extractor programs.

a_1={title="Distributed Query Processing in a Relational Data Base System", pages = "169-180",
authoredBy = p_1, authoredBy = p_2, authoredBy = p_3, publishedIn = c_1}

a_2={title="Distributed query processing in a relational data base system", pages = "169-180",
authoredBy = p_4, authoredBy = p_5, authoredBy = p_6, publishedIn = c_2}

p_1={name = "Robert S. Epstein", coAuthor = p_2, coAuthor = p_3}

p_2={name = "Michael Stonebraker", coAuthor = p_1, coAuthor = p_3}

p_3={name = "Eugene Wong", coAuthor = p_1, coAuthor = p_2}

p_4={name = "Epstein, R.S.", coAuthor = p_5, coAuthor = p_6}

p_5={name = "Stonebraker, M.", coAuthor = p_4, coAuthor = p_6}

p_6={name = "Wong, E.", coAuthor = p_4, coAuthor = p_5}

p_7={name = "Eugene Wong", email = "eugene@berkeley.edu", emailContact = p_8}

p_8={email = "stonebraker@csail.mit.edu", emailContact = p_7}

p_9={name = "mike", email = "stonebraker@csail.mit.edu"}

c_1={name = "ACM Conference on Management of Data", year = "1978",
location = "Austin, Texas"}

c_2={name = "ACM SIGMOD", year = "1978"}

(a)

$$\{\{a_1, a_2\}, \{p_1, p_4\}, \{p_2, p_5, p_8, p_9\}, \{p_3, p_6, p_7\}, \{c_1, c_2\}\}$$

(b)

Figure 2.3: Example for reference reconciliation: (1) references extracted from personal data; (2) reconciliation results.

The reconciliation algorithm tries to partition the set of references in each class, such that each partition corresponds to a single unique real-world entity, and different partitions refer to different entities. We measure the quality of a reconciliation with *recall* and *precision*. The recall measures the percentage of correctly reconciled pairs of references over all pairs of references that refer to the same entity, and the precision measures the percentage of correctly reconciled pairs over all reconciled pairs of references.

As an example, Figure 2.3(b) shows the ideal reconciliation result for the references in Figure 2.3(a). According to this reconciliation results we can populate an association network as shown in Figure 2.2.

2.1.3 Overview of the approach

The goal of our approach is to address several challenges arising in dataspace applications such as PIM. First, it is often the case that each reference includes very limited information; that is, each reference contains values for only a few attribute properties. For example, a Person reference often has values for only one or two attribute properties. In Example 2.2, references p_5 and p_8 do not have any attribute in common. Second, some attributes are multi-valued, so the fact that two attribute values are different does not imply that the two references refer to different real-world objects. For example, two Person references with completely different email addresses may refer to the same person. This phenomenon is especially common in applications where the real-world entities (and hence, the data) evolve over time.

As we show in our experimental results, the above problems lead to unsatisfactory accuracy of existing reference reconciliation algorithms. The key behind our approach is that we exploit the richness of the information space at hand. We illustrate the main concepts of our algorithm with the example.

Exploiting context information: The main idea underlying our algorithm is to capture and leverage various forms of *context* we can glean about the references, which are not considered by traditional reference reconciliation approaches. For example, we can consider the co-author lists and email-contact lists of person references. In our example, we notice

that p_5 has co-authored articles with p_6, and p_8 has email correspondences with p_7. If we decide that p_6 and p_7 are the same person, we obtain additional evidence that may lead us to reconcile p_5 and p_8. Second, we compare values of *different* attributes. For example, the name "Stonebraker, M." and the email address "stonebraker@csail.mit.edu" are closely related: "stonebraker" corresponds to the last name of "Stonebraker, M.". This observation provides positive evidence for merging references p_5 and p_8.

Once we decide to merge two references, we have two mechanisms for leveraging this information: reconciliation propagation and reference enrichment.

Reconciliation propagation: When we reconcile two references, we reconsider reconciling references that are associated with them. For example, when we detect that the Article references a_1 and a_2 share the same title and similar authors and that they appeared in similar conferences and pages in the proceedings, we decide to reconcile them. Presumably an article has a unique set of authors, so the reconciliation of a_1 and a_2 implies that the authors p_1 and p_4, p_2 and p_5, and p_3 and p_6 should be reconciled respectively. Similarly, we reconcile the conference references c_1 and c_2.

Reference enrichment: When we decide to reconcile two references, we join their attribute values together, thereby enriching the reference. For example, consider the reconciliation of the person references p_5 and p_8. Although "stonebraker@csail.mit.edu" and "Stonebraker, M." are rather similar, this information is insufficient for reconciliation. Similarly, we lack evidence for reconciling p_5 and p_9. However, after we reconcile p_8 and p_9, we can aggregate their information and now know that "mike" and "Stonebraker, M." share the same first name initial, and contact the same person by email correspondence or coauthoring. This additional information will enable us to reconcile p_5, p_8, and p_9.

In summary, our approach obtains better reference reconciliation results by exploiting context information, propagating reconciliation decisions and enriching references. To facilitate these mechanisms, we define a graph, called the *similarity-dependency graph* (abbreviated as *dependency graph* in the rest of this chapter), whose nodes represent similarities between pairs of references and edges represent the dependencies between the reconciliation decisions. Using the dependency graph, we are able to capture the dependencies of reference

similarities and attribute-value similarities, and at the same time retain the flexibility of using established reference comparison techniques. In the next section, we describe how we construct the dependency graph and use it for reconciliation.

2.2 Reference Reconciliation Algorithm

Our reconciliation algorithm proceeds as follows. First, we construct the dependency graph that captures the relationships between different reconciliation decisions. Then, we iteratively recompute scores that are assigned to reconciliation-decision nodes in the graph until a fixed point is reached. Finally, we compute the transitive closure for the final reconciliation results. Section 2.2.1 describes the construction of the dependency graph. Section 2.2.2 describes how we use the graph for reconciliation. Section 2.2.3 describes how we enrich references during the reconciliation process. Section 2.2.4 describes how our algorithm exploits *negative* information to further improve the accuracy of reconciliation.

2.2.1 Dependency graph

To reconcile references, we need to compute the similarity for every pair of references of the same class. The similarity between a pair of references is based on the similarity of their attribute values and the similarities of their association properties. Meanwhile, the dependency is often bi-directional; that is, the similarity of the references also affects the similarity of the values of their attribute or association properties. In a dependency graph, a node represents the similarity between a pair of references or a pair of attribute values, and an edge represents the *dependency* between a pair of similarities. If we change the similarity value of a node, we may need to recompute the similarity of its neighbors. Formally, we define a dependency graph as follows:

Definition 2.4. *Let $\mathcal{R}$ be a set of references. The* dependency graph *for $\mathcal{R}$ is an undirected graph $G = (N, E)$, such that*

- *For each pair of references $r_1, r_2 \in \mathcal{R}$ of the same class, there is a node $m = (r_1, r_2)$ in G.*

- *For each pair of property values a_1 of r_1 and a_2 of r_2 (the two attributes may be of different types), there is a node $n = (a_1, a_2)$ in G and an edge between m and n in E.*

- *Each node has a real-valued* similarity score *(between 0 and 1), denoted interchangeably as $sim(r_1, r_2)$ or $sim(m)$.* □

In what follows, we refer to references and attribute values collectively as *elements*. Note that there is a unique node in the dependency graph for each pair of elements. This uniqueness is crucial for exploiting the dependencies between reconciliation decisions (as we will see in Section 2.2.2).

Pruning and refining the graph: In practice, building similarity nodes for *all* pairs of elements is unnecessarily wasteful. Hence, we only include in the graph nodes whose references *potentially* refer to the same real-world entity, or whose attribute values are *comparable* (*i.e.*, are of the same attribute, or according to the domain knowledge are of related attributes, such as a name and an email) and similar.

Rather than considering all edges in the graph to be the same, we refine the set of edges in the graph to be of several types, which we will later leverage to gain efficiency. This refinement can be obtained by applying domain knowledge, either specified by domain experts or learned from training data. The first refinement generates a subgraph of the original graph. In the subgraph, there is an edge from node n to m only if the similarity of m *truly* depends on the similarity of n. We call n an *incoming neighbor* of m, and m an *outgoing neighbor* of n. Note that the subgraph is directed.

The second refinement distinguishes several types of dependencies as follows. Consider nodes m and n, where there is an edge from m to n. First, we distinguish between *boolean-valued* dependencies and *real-valued* dependencies. If the similarity of a node n depends *only* on whether the references in the node m are reconciled, we call m a *boolean-valued* neighbor of n. In contrast, if the similarity of n depends on the actual similarity value of node m, we call m a *real-valued* neighbor of n. As an example, given two conference references, c_1 and c_2 in Figure 2.2(b), their similarity depends on the real similarity value of their names. Thus, a node for conference-name similarity is a real-valued incoming neighbor of a node for

the similarity of the conference references with the names. On the other hand, although it also depends on the similarity of the articles a_1 and a_2, what really matters is whether they are reconciled, not their actual similarity value (we assume that a single article cannot be published in two different conferences). So a node for article similarity is a boolean-valued incoming neighbor of a node for the similarity of the associated conference references.

We further divide *boolean-valued* neighbors into two categories. If the reconciliation of m's two references implies that the two references in n should also be reconciled, m is called n's *strong-boolean-valued* neighbor. The second case in the earlier example illustrates a strong-boolean-valued neighbor. If the reconciliation of m's references only increases the similarity score of n, but does not directly imply reconciliation, m is called n's *weak-boolean-valued* neighbor. For example, the similarity score of two persons will increase given that they have email correspondence with the same person.

Constructing the graph: We build the graph in two steps. In the first step we consider attribute properties, in the second step we consider association properties.

1. For every pair of references r_1 and r_2 of the same class, insert $m = (r_1, r_2)$ with similarity 0. Further,

   (1) For every pair of attribute values a_1 of r_1 and a_2 of r_2, if a_1 and a_2 are comparable, then proceed in two steps.

   Step 1. If $n = (a_1, a_2) \notin G$, we compute the similarity score of a_1 and a_2. If the score indicates that a_1 and a_2 are potentially similar, then insert n with the computed score.

   Step 2. Insert an edge from n to m and an edge from m to n when the corresponding dependency exists.

   (2) If m does not have any neighbors, remove m.

   Note that at this stage we are liberal in identifying potentially similar attribute pairs (we use a relatively low similarity threshold) in order not to lose important nodes in the graph.

2. We now consider references r_1 and r_2, where $m = (r_1, r_2) \in G$. For every pair of instances a_1 and a_2, where a_1 is an association property value of r_1 and a_2 is an association property value of r_2, if there exists dependence between $sim(r_1, r_2)$ and $sim(a_1, a_2)$, we do as follows:

   - if $a_1 = a_2$, we add the node $n = (a_1, a_1)$ and set $sim(n) = 1$ if n does not exist, and add an edge from n to m;
   - if $a_1 \neq a_2$ and the node $n = (a_1, a_2)$ exists, add an edge from n to m when the corresponding dependency exists, and add an edge from m to n when the corresponding dependency exists.

   The requirement $m = (r_1, r_2) \in G$ is based on the assumption that two references cannot refer to the same entity unless they have *some* similar attribute property values. However, this assumption is not germane to our algorithm.

We show the pseudo code for constructing a dependency graph in Figure 2.4.

Example 2.5. *Consider the dependency graph for the instances in Figure 2.2(b), as shown in Figure 2.5.*

Figure 2.5(a) shows the subgraph for references $a_1, a_2, p_1, p_2, p_3, p_4, p_5, p_6, c_1$ *and* c_2*. The similarity of papers* a_1 *and* a_2 *is represented by the node* m_1*; it is dependent on the similarity of the titles (represented by* n_1*), the pages (*n_2*), the authors (*m_2, m_3 *and* m_4*), and the conferences (*m_5*). Note that there does not exist an edge from* m_5 *to* n_7*, because the similarity of years is pre-determined and is independent of the similarity of any conferences. Also note that there is no node* (p_1, p_5)*, because their name attributes have very different values.*

Figure 2.5(b) shows the subgraph for references p_5 *and* p_8*. Since* p_5 *has coauthored with* p_6*, and* p_8 *has email correspondence with* p_7*, there is a node* $m_7 = (p_6, p_7)$*, and a bi-directional edge between* m_6 *and* m_7*. Note that* m_6 *does not have a neighbor* (p_4, p_7)*, because* p_4 *and* p_7 *do not have any similar attributes and so the node* (p_4, p_7) *does not exist.*

□

```
procedure GraphConstruction(R) return G
// R is a reference set;
// G = (N, E) is a dependency graph for references in R
  for each (class C)
    for each (references r1 and r2 of C)
      Insert into N a node m = (r1, r2) with similarity 0;
      for each (attribute a1 of r1 and a2 of r2)
        if (a1 and a2 are comparable)
          if (n = (a1, a2) ∉ N && a1 and a2 are potentially similar)
            Insert into N a node n with similarity sim(a1, a2);
          if (n = (a1, a2) ∈ N && sim(n) depends on sim(m))
            Insert into E an edge from m to n;
          if (n = (a1, a2) ∈ N && sim(m) depends on sim(n))
            Insert into E an edge from n to m;
      if (m does not have neighbors)
        Remove m from N;
  for each (m = (r1, r2) ∈ N)
    for each (association property value a1 of r1 and association property value a2 of r2)
      if (sim(r1, r2) depends on sim(a1, a2) or vice versa)
        if (a1 = a2)
          if ((a1, a2) ∉ N)
            Insert into N a node n = (a1, a1) with similarity 1;
          Insert into E an edge from n to m;
        if (a1 ≠ a2 && n = (a1, a2) ∈ N)
          if (sim(r1, r2) depends on sim(a1, a2))
            Insert into E an edge from n to m;
          if (sim(a1, a2) depends on sim(r1, r2))
            Insert into E an edge from m to n;
```

Figure 2.4: Algorithm for constructing the dependency graph.

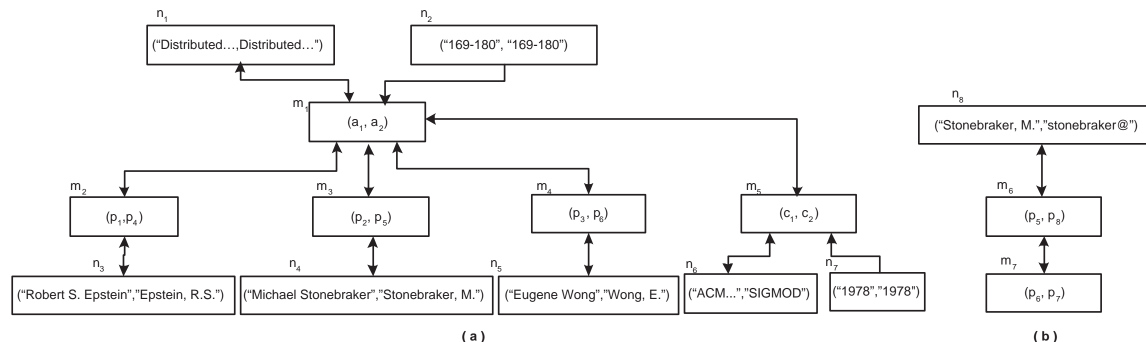


Figure 2.5: The dependency graph for references in Figure 2.2(b): (a) the subgraph for references $a_1, a_2, p_1, p_2, p_3, p_4, p_5, p_6, c_1$ and c_2. (b) the subgraph for references p_5 and p_8.

2.2.2 Exploiting the dependency graph

Our algorithm is based on propagating similarity decisions from node to node in the graph. For example, after we decide to reconcile articles a_1 with a_2, we should reconcile their associated conferences c_1 and c_2, and further trigger recomputation of the similarities of other papers that mention the conferences c_1 and c_2, etc. Given that the dependency graph has captured all the dependencies between similarities, it guides the recomputation process, as we now describe. We describe similarity-score functions in Section 2.3.

We mark the nodes in a dependency graph as *merged, active*, or *inactive*. A node is marked *merged* when its similarity score is above a pre-defined merge threshold, and it thus represents already reconciled references. A node is marked *active* if we need to reconsider its similarity. The rest are marked *inactive*. The algorithm proceeds as follows, until no node is marked *active*.

1. Initially, all nodes representing the similarity between different references are marked *active* and those representing the similarity between the same reference are marked *merged*. The nodes representing the similarity between attribute values are marked *merged* or *inactive* depending on their associated similarity score.

2. We select one active node at a time and recompute its similarity score. If the new similarity score is above a merge threshold, we mark the node as *merged*; otherwise,

we mark it as *inactive*. In addition, we mark as *active* all its outgoing neighbors with similarity scores below 1.

This process is guaranteed to terminate under the following two assumptions. First, we assume that the similarity-score functions for any node are monotonic in the similarity values of its incoming neighbors. Second, for a given node, we activate its neighbors only when its similarity increase is more than a small constant. It might appear that requiring monotonic similarity functions precludes fixing initial incorrect reconciliation decisions in the presence of later negative evidence; however, this is not the case and we will show in Section 2.2.4 how we account for negative evidence.

Implementing the propagation procedure: To improve the efficiency of the algorithm, we treat different types of neighbors (according to the categorization of similarity dependencies) differently in the second step, rather than activate all neighbors of a node. In this way, we can significantly reduce similarity recomputation. Specifically, given a node n whose similarity score increases, we do the following:

- we mark as *active* all of its outgoing real-valued inactive neighbors whose similarity scores are below 1.

- if we decide to reconcile the references in n, we mark as *active* all of its outgoing boolean-valued neighbors whose similarity scores are below 1.

In addition, a careful choice of the recomputation order can further reduce the number of recomputations and improve the algorithm's efficiency. In particular, we employ the following heuristics.

- We compute similarity for a node only if the scores of its incoming value-based neighbors have all been computed, unless there exist mutual real-valued dependencies. For example, we compare two articles only after comparing their associated authors and conferences (or journals).

- When a node is merged, we consider its outgoing strong-boolean-valued neighbors first for recomputation.

Our algorithm proceeds by maintaining a queue of active nodes. Initially, the queue contains all active similarity nodes (representing similarities of pairs of different references), and it satisfies the property that a node always precedes its outgoing real-valued neighbors if there does not exist mutual real-valued dependencies. At every iteration, we compute the similarity score for the top node in the queue. If we activate its outgoing real-valued or weak-boolean-valued neighbors, then we insert them at the end of the queue. If we activate its strong-boolean-valued neighbors, we insert them in front of the queue.

Figure 2.6 shows the algorithm for information propagation. Note that PROPAGATION invokes ENRICHMENT, which we will describe in the next sub-section.

Example 2.6. *Consider the dependency graph shown in Figure 2.5(a). Initially, the queue contains nodes* $\{m_5, m_4, m_3, m_2, m_1\}$*, and nodes* n_1, n_2*, and* n_7 *are marked* merged*. We then compute the similarity of* m_5, m_4, m_3, m_2*, and* m_1 *in succession. When we decide to merge papers* a_1 *and* a_2*, we insert* m_2, m_3, m_4*, and* m_5 *back to the front of the queue, so the queue becomes* $\{m_5, m_4, m_3, m_2\}$ *(the order among these nodes can be arbitrary). Note that* n_2 *is not added back both because it is not an outgoing neighbor of* m_1*, and because it already has similarity score 1. Next, we consider* m_5 *and decide to merge* c_1 *and* c_2*, so we insert its strong-boolean-valued neighbor,* n_6*, in the front of the queue and the queue becomes* $\{n_6, m_4, m_3, m_2\}$*. This process continues until the queue is empty.* □

2.2.3 Enriching the references

Another important aspect of our algorithm is that we *enrich* the references in the process of reconciliation. Specifically, after merging references r_1 and r_2, all the properties of r_2 can also be considered as those of r_1. For example, if r_1 has email address "stonebraker@csail.mit.edu" and r_2 has email address "stonebraker@mit.edu", the real-world person object actually has both email addresses. Now, when we compute the similarity between r_1 and another reference r_3, we compare both email addresses with the email address of r_3, and choose the one with a higher similarity. Note that for the purpose of reconciliation, we do not need to distinguish multiple email addresses and misspelled email addresses.

A naive way to enrich the references would be to run our propagation algorithm, then

```
procedure PROPAGATION(G)
// G = (N, E) is a dependency graph;
  Initialize queue q as empty;
  for each (node n = (r_1, r_2) ∈ N)
    if (r_1 and r_2 are references && r_1 ≠ r_2)
      Mark n as active;
      Insert n into q;
  while (there exists n = (r_1, r_2) ∈ q and m = (a_1, a_2) ∈ q such that (1) m is an incoming
      neighbor of n, (2) there is no loop between m and n, and (3) n is in front of m in q)
    switch m and n in q;
  while (q is not empty)
    Remove the first node n = (r_1, r_2) from q and mark it as inactive;
    oldSim = sim(n); Recompute sim(n);
    if (sim(n) > oldSim)
      for each (n's outgoing real-valued neighbor m)
        if (m is not active AND sim(m) < 1)
          Insert m to the end of q and mark m as active;
    if (sim(n) is above the merge threshold)
      Mark n as merged;
      for each (n's outgoing strong-boolean-valued neighbor m)
        if (m is not active AND sim(m) < 1)
          Insert m to the front of q and mark m as active;
      for each (n's outgoing weak-boolean-valued neighbor m)
        if (m is not active AND sim(m) < 1)
          Insert m to the end of q and mark m as active;
      ENRICHING(n, G);
```

Figure 2.6: Algorithm for information propagation in reference reconciliation.

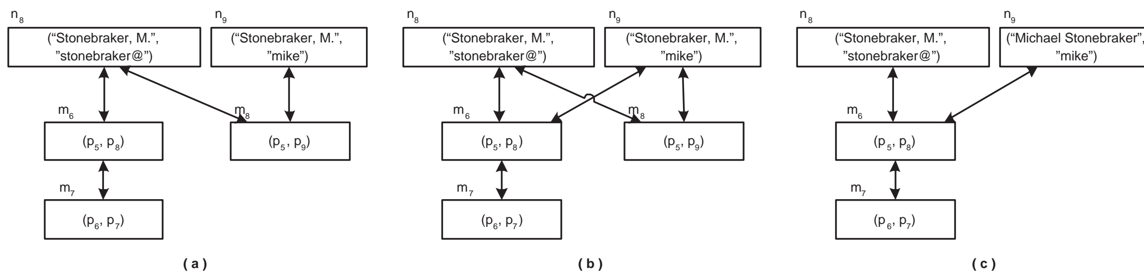


Figure 2.7: Enrichment of references in Figure 2.5(b): (a) the original subgraph representing the similarity of references p_5 and p_8 (node m_6), and the similarity of p_5 and p_9 (node m_8); (b) the subgraph after the first step of reference enrichment; (c) the subgraph after the second step of reference enrichment.

compute transitive closures and merge the references within the same cluster, and repeat the algorithm iteratively. However, with a little bit of care, we can implement enrichment with only local changes to the graph.

Specifically, after we decide to merge references r_1 and r_2, we search for all references r_3, such that there exist nodes $m = (r_1, r_3)$ and $n = (r_2, r_3)$. We proceed to remove n from the graph in the following steps: (1) connect all neighbors (incoming and outgoing neighbors) of n with m while preserving the direction of the edges, (2) remove node n and its associated edges from the dependency graph and from the queue, (3) if m gets new incoming neighbors and is not active in the queue, we insert m at the end of the queue; similarly, n's neighbors that get new incoming neighbors and are not active are inserted at the end of the queue.

We show in Figure 2.8 the algorithm for reference enrichment.

Example 2.7. *Consider enrichment of references in Figure 2.5(b). Figure 2.7(a) shows the original subgraph that represents the similarity of references* p_5 *and* p_8 *(node* m_6*), and the similarity of* p_5 *and* p_9 *(node* m_8*). When we decide to reconcile* p_8 *and* p_9*, we need to compute a similarity score for only one of* m_6 *and* m_8*.*

In the first step, whose result is shown in Figure 2.7(b), we connect all of m_8*'s neighbors with* m_6*, so* n_9 *is connected with* m_6*. Note that* n_8 *is already connected with* m_6 *so no change is needed. In the second step, whose result is shown in Figure 2.7(c), we remove node* m_8 *and all associated edges. We now have more evidence for reconciling* p_5 *and* p_8*.* □

```
procedure ENRICHING(n, G)
// n = (r1, r2) is the node that is just marked merged
//G = (N, E) is a dependency graph;
 for each (r3 where m = (r1, r3) and l = (r2, r3) exist)
  for each (edge e connecting l with a node j)
   if (j is not a neighbor of m)
    Insert into E an edge between j and m (with the same direction as e);
    Remove from E the edge e;
    if (j is not marked active)
     Mark j as active and insert j to the end of q;
  if (m's neighbors changed && m is not marked active)
   Mark m as active and insert m to the end of q;
  Remove l from N and q;
```

Figure 2.8: Algorithm for reference enrichment in reference reconciliation.

2.2.4 Enforcing constraints

Up to now, our algorithm has considered only *positive* evidence for similarity computation. In many cases, we may have negative evidence that can contribute to the reconciliation process. As a simple example, if in Figure 2.2(b), reference p_9 were p_9={name="Matt", email="stonebraker@csail.mit.edu"}, then we would not want to merge p_9 with p_2, which has "Michael Stonebraker" for the name attribute. However, as shown in Figure 2.7, the algorithm as described so far might reconcile p_8 and p_9 with p_5, and since p_2 is merged with p_5, they will all be merged together when we compute the transitive closure.

Indeed, this problem is a fundamental one when we generate partitions by computing transitive closures: if we decide to reconcile r_1 with r_2, and r_2 with r_3, then r_1, r_2 and r_3 will be clustered even if we have evidence showing that r_1 is *not* similar to r_3.

We begin to address this problem by considering *constraints*. A constraint is a rule enforcing that two references are guaranteed to be distinct. For example, a constraint may specify that the authors of one paper are distinct from each other. Constraints are typically domain dependent. They can be manually specified by domain experts, or learned from training data or a clean auxiliary source [41]. To enforce constraints, we add one more status to the nodes in the dependency graph: *non-merge*. The two elements in a *non-merge* node are guaranteed to be different and should never be reconciled. Note that a *non-merge* node is different from a non-existing node. The absence of the node (r_1, r_2) indicates that r_1 and r_2 do not have similar attributes; but r_1 can still be reconciled with r_2 if both of them are reconciled with another reference r_3.

To incorporate constraints in our algorithm, we make the following modifications.

1. When constructing the dependency graph we also include nodes whose elements are ensured to be distinct. Such nodes are marked *non-merge* and will never enter the processing queue.

2. A node that has a non-merge incoming real-valued neighbor might be marked as *non-merge* according to the constraints.

3. After the similarity computation reaches a fixed point, we examine every *non-merge*

procedure CONSTRAINT(G)
// $G = (N, E)$ *is a dependency graph;*
  for each ($l = (r_1, r_2)$ that is marked *non-merge*)
    for each (r_3 where $m = (r_1, r_3)$ and $n = (r_2, r_3)$ exist and $sim(m) \geq sim(n)$)
      if (m is marked *merged*)
        Mark n as *non-merge*;

Figure 2.9: Algorithm for enforcing constraints in reference reconciliation.

node $l = (r_1, r_2)$. Let r_3 be another reference such that there exist nodes $m = (r_1, r_3)$ and $n = (r_2, r_3)$, where $sim(m) \geq sim(n)$. If m is marked *merge*, we mark n as *non-merge*. This step propagates the negative evidence from l to n, so r_1 and r_2 will not be merged with a common reference. Figure 2.9 shows the algorithm for this step.

Note that with the above modification, the algorithm still converges under the assumptions described in Section 2.2.2.

Figure 4.7 shows the overall reconciliation algorithm. As a further optimization, the dependency graph can be pruned at the very beginning using inexpensive reference comparisons, such as merging Person references that have the same email address. This preprocessing can significantly reduce the size of the dependency graph and thus improve the efficiency.

2.3 Computing Similarity Scores

This section describes the similarity functions used in our algorithm. Given a node $m = (r_1, r_2)$, the similarity function for m takes the similarity scores of m's incoming neighbors as input and computes a score between 0 and 1.

Our dependency-graph based reconciliation algorithm has the flexibility that we can use different domain-specific similarity functions for different classes in the domain model. Training data, when available, can be used to learn or tune similarity functions for specific

procedure RECONCILIATION($\mathcal{R}$) **return** $\mathcal{P}$

// $\mathcal{R}$ is a reference set, $\mathcal{P}$ is a partitioning over $\mathcal{R}$;

G = GRAPHCONSTRUCTION($\mathcal{R}$);

PROPAGATION(G);

CONSTRAINT(G);

Compute transitive closure according to G and return $\mathcal{P}$;

Figure 2.10: Algorithm for reference reconciliation.

classes. As we explain in Section 2.5, this is an important advantage of our approach over those based on global detailed probabilistic modeling [103, 118].

In what follows we propose a template of similarity functions that proved effective in our experiments. The similarity functions are orthogonal to the dependency graph framework and alternate simple or complex models can also be used. There are tunable parameters in our functions and these can either be learned from training data, or manually set from experience. Note that as we propagate information between reconciliation decisions, we may also propagate incorrect information from an over-estimated similarity value and mislead later similarity computations. Hence, we choose conservative similarity functions and merge-thresholds in order to obtain high precision. We obtain improvement in recall mainly by employing a broad collection of evidence.

The components of the similarity function: the similarity score S of a pair of elements is the sum of three components and is always between 0 and 1: (1) S_{rv}, contributed by real-valued incoming neighbors, (2) S_{sb}, contributed by strong-boolean-valued incoming neighbors, and (3) S_{wb}, contributed by weak-boolean-valued incoming neighbors. When their sum exceeds 1, we trunk S to 1. Below we discuss each component in detail.

Real-valued neighbors ($\mathbf{S_{rv}}$): Given the similarity values of the real-valued incoming neighbors, we compute the value of S_{rv} between 0 and 1. For example, the similarity function of a person node combines the name similarities, email similarities, and name-and-email similarities.

Computing S_{rv} is similar in spirit to traditional record-linkage techniques, but with a few important differences. First, to account for the fact that most references do not have values for all attributes, we employ a *set* of similarity functions, rather than a single one. These functions account for cases in which some attributes are missing values (otherwise, we need to assign 0 to the similarity of those attributes, resulting in recall reduction). Second, our similarity functions account for some attributes serving as keys. For example, when two person instances have the same email address, they should be merged even if other attributes are different. Third, our similarity functions take into consideration any possible *non-merge* neighbors. We organize the set of similarity functions as a decision tree, where each branch node represents certain conditions, such as the existence of a similarity value or a *non-merge* neighbor, and each leaf node represents a function for S_{rv}.

Each function for S_{rv} combines the elementary similarity values using a linear combination. Formally, we define

$$S_{rv} = \sum_{i=1}^{n} \lambda_i x_i, \tag{2.1}$$

where n is the number of different types of real-valued neighbors (*e.g.*, email similarity, name similarity, etc.), x_i is the score for the elementary similarity of type T_i and λ_i is its corresponding weight.

The final wrinkle in computing S_{rv} is to account for attributes having multiple values (*e.g.*, a person's email). For the similarity node $m = (r_1, r_2)$, we consider all attribute values of r_1 and r_2, and compute the similarity for all pairs of attributes that are comparable. Let $N_i = \{n_1, \ldots, n_k\}$ be the set of incoming neighbors of type T_i. We use $MAX\{sim(n_j)|n_j \in N_i\}$ for the similarity value of type T_i when computing S_{rv} in Equation 2.1.[1]

Strong-boolean neighbors ($\mathbf{S_{sb}}$): Let m be a similarity node. In principle, if two references in one of m's strong-boolean-valued neighbors are merged, then the two references in m should also be merged, unless they have very different attribute values. For example, when we decide to merge two papers, we can merge their authors with similar names. However, to remain conservative and avoid possible errors caused by noisy information, we

[1]More complex cases can arise; for example, an attribute value might be a set with each element in the set being multi-valued. Strategies for such cases are beyond the scope of this dissertation.

only increase the similarity score with a constant β for each merged strong-boolean-valued incoming neighbor. Formally, S_{sb} is:

$$S_{sb} = \begin{cases} \beta \cdot |N_{sb}| & if \ S_{rv} \geq t_{rv} \\ 0 & otherwise \end{cases}$$

where β is a constant, $|N_{sb}|$ is the number of merged strong-boolean-valued incoming neighbors, and t_{rv} is a threshold indicating that two references may possibly refer to the same entity. Given this function, when the initial similarity value of a pair of references, S_{rv}, is not very high (but above the threshold t_{rv}), the two references will be reconciled only if there exist several strong-boolean-valued neighbors that are merged. A more sophisticated function can require stricter conditions; for example, we increase the similarity score of two person names only if both names are *full* names.

Weak-boolean neighbors ($\mathbf{S_{wb}}$): Intuitively, the reconciliation of two references in a weak-boolean-valued incoming neighbor of m increases the similarity score of m. For example, we will have higher confidence in reconciling two person references if they not only have similar attribute values, but also contact common people (by email or by co-authorship). However, people's contact lists may vary a lot in length, and it is quite possible that two references refer to the same real-world person but have very different contact lists. Hence, requiring two contact lists to be similar is unnecessarily strict. In our algorithm we count the number of common contacts and increase the similarity score with a constant γ for each common contact. More generally, we define S_{wb} as

$$S_{wb} = \begin{cases} \gamma \cdot |N_{wb}| & if \ S_{rv} \geq t_{rv} \\ 0 & otherwise \end{cases}$$

where γ is a constant, and $|N_{wb}|$ is the number of merged weak-boolean-valued incoming neighbors. Note that the functions S_{wb} and S_{sb} have the same form, but we assign a much higher value to β than to γ. Finally, a sophisticated function for S_{wb} can assign a higher reward for the first several merged neighbors and a lower reward for the rest, or consider the number of values for an association property.

Person (name, *coAuthor)

Article (title, pages, *authoredBy, *publishedIn)

Venue (name, year, location)

Figure 2.11: Domain Model for references from Cora.

Data Set	#(References)	#(Entities)	#Ref/#Entity
PIM *A*	27367	2731	10.0
PIM *B*	40516	3033	13.4
PIM *C*	18018	2586	7.0
PIM *D*	17534	1639	10.7
Cora	6107	338	18.1

Table 2.1: Properties of our data sets: the number of extracted references, the number of real-world entities and the reference-to-entity ratio.

2.4 Experimental Evaluation

We tested our algorithm on two domains: personal information management and publication portal. We now describe a set of experiments that validate the performance of our reconciliation algorithm. The experimental results show that our algorithm obtains high precision and recall in both contexts.

2.4.1 Data Sets

Our first experiment involved four personal data sets. To ensure that we got a variety of references, we chose the owners of the data sets to be in different areas of computer science (database and theory), in different positions (faculty and graduate students), and most importantly, from different countries (including China, India and the USA)[2]. The data sets span several years of computer usage (from 3 to 7), and include references obtained from

[2] Names of people from these countries have very different characteristics.

emails, Latex and Bibtex files, and PDF and Word documents. The references we extracted conform to the domain model shown in Figure 2.2(a), except that when we measure the reconciliation results, we consider both conferences and journals as venues and report precision and recall on them together. For each data set, we manually created the gold standard; that is, the perfect reconciliation result.

In order to demonstrate the applicability of our algorithm in a more conventional setting, our second experiment uses the subset of the Cora data set provided by McCallum [38] and used previously in [36, 17, 118]. This data set is a collection of 1295 different citations to 112 computer science research papers from the Cora Computer Science Research Paper Engine. We extracted references of types Person, Article and Venue from the citations, according to the domain model shown in Figure 2.11. The papers in the data set are already hand labeled, whereas we had to label the persons and venues.

The properties of our data sets are summarized in Table 2.1. The average reference-to-entity ratio, 11.8, underscores the importance of reference reconciliation.

2.4.2 Experimental methodology

We reported the overall performance of our algorithm using two sets of measures. The first set is *precision, recall*, and *F-measure*, which is also widely used in other reference reconciliation works. Precision computes the percentage of correctly reconciled reference pairs over all reconciled reference pairs; recall computes the percentage of correctly reconciled reference pairs over pairs of references that refer to the same real-world object; F-measure counts for both precision and recall, and is computed using the following formula:

$$\text{F-measure} = \frac{2 \cdot prec \cdot recall}{prec + recall}.$$

Precision, recall and F-measure all range from 0 to 1, and ideally should be 1.

When we compute precision, recall and F-measure, we can either consider all occurrences of references or consider only distinct references. We call the former *occurrence-based* measures and the latter *representation-based* measures. For example, consider three *Person* references, each containing a single name attribute, where the first two have value “Michael Stonebraker” and the third one has value “Mike”. Occurrence-based measures are com-

puted over all of the three references, whereas representation-based measures are computed by viewing the first and second references as the same one (so two references in total). On the one hand, occurrence-based measures penalize results more for incorrect reconciliation of *popular* entities; that is, entities with more occurrences. This is as required in the PIM context, where popular entities are browsed more often and errors in their reconciliation cause more inconvenience. In addition, it is possible that different occurrences of the same reference actually refer to different real-world entities; occurrence-based measures allow us to explore this possibility. On the other hand, since most different occurrences of the same reference will and should be merged, occurrence-based measures tend to result in very high values and hide the significance of the improvement obtained by our algorithm.

The second set of measures is *diversity* and *dispersion*. Diversity measures on average for every result partition, how many real-world entities are included. Dispersion measures on average for every real-world entity, how many result partitions include them. Diversity is related to precision and dispersion is related to recall. Both diversity and dispersion are no less than 1 and ideally should be 1. Note that considering all occurrences of references and considering only distinct references obtain the same diversity and dispersion, so we do not distinguish them.

These different measures indeed present the same trend. Unless we specify otherwise, we reported the performance of our algorithm using occurrence-based precision, recall, and F-measure. For some of the results we in addition reported representation-based measures and also diversity and dispersion for the purpose of comparison.

We employed the same set of similarity functions and thresholds for *all* data sets. We manually set the thresholds and parameters and we used the same ones in *all* our experiments. Specifically, we set the merge threshold to 0.85 for all reference similarities, and to 1 for all attribute similarities. We set $\beta = 0.1, \gamma = 0.05$ for all classes, except that we set $\beta = 0.2$ for Venue. We set $t_{rv} = 0.7$ for Person and Article references and $t_{rv} = 0.1$ for Venue references. We omit the detailed settings for S_{rv} since they vary from one class to another, and for each class there exists a set of functions forming a decision tree. We note that as we chose the thresholds and parameters to be conservative, the results were insensitive to small perturbations in their values.

Table 2.2: Average precision, recall and F-measure for each class of references.

Class	IndepDec		DepGraph	
	Prec/Recall	F-msre	Prec/Recall	F-msre
Person	0.967/0.926	0.946	0.995/0.976	0.986
Article	0.997/0.977	0.987	0.999/0.976	0.987
Venue	0.935/0.790	0.856	0.987/0.937	0.961

In our experiments, we refer to our algorithm as DepGraph. On the PIM data, we compared DepGraph with a candidate standard reference reconciliation approach, called IndepDec (it roughly corresponds to approaches such as [72, 96]). To compare the two algorithms on the class Person, IndepDec compares person names and emails independently and combines the results for reference similarity without exploiting the dependencies between individual reconciliation decisions. DepGraph, in addition, compares the names with the email accounts, considers the articles authored by the persons, counts the common people appearing in the coauthor or email-contact lists, applies reconciliation propagation and reference enrichment, and enforces constraints. We use the same similarity functions and thresholds for IndepDec and DepGraph.

For the Cora data set, we compared the results of DepGraph with the results reported in papers that proposed state-of-the-art reference reconciliation approaches.

2.4.3 Reference reconciliation for personal data

Table 2.2 shows the average precision and recall for articles, persons and venues (conferences and journals) over the four data sets. DepGraph obtained higher precision and recall for both person and venue references. Specifically, it improved the recall for venue references by 18.6%, and for person references by 5.4%. Note that this 5.4% is in fact substantial: as we shall see in Table 2.4, it corresponds to a decrease in hundreds of partitions. Our algorithm does not improve the results for articles. This is because the article references are obtained from a set of Bibtex files that are typically very well curated by the user. From

Table 2.3: Average precision, recall and F-measure for Person references when only the email or paper subsets are considered and when the full data sets are considered.

Data Set	IndepDec		DepGraph	
	Prec/Recall	F-msre	Prec/Recall	F-msre
Full	0.967/0.926	0.946	0.995/0.976	0.986
PArticle	0.999/0.761	0.864	0.997/0.994	0.996
PEmail	0.999/0.905	0.950	0.995/0.974	0.984

Table 2.2 we also observe that whereas IndepDec may obtain either a low precision or a low recall in some cases, DepGraph typically obtains both high precision and high recall. Furthermore, as we will explain shortly, the improvement of DepGraph over IndepDec is most pronounced on the data sets in which the references have little information, or there is a great variety in the references to the same real-world entity.

We now examine person references, which are associated with rich context information and therefore provide the most opportunities for performance improvement. We divided each data set into two subsets: one contains person references extracted only from emails (PEmail) and the other contains person references extracted from articles and other non-email sources (PArticle). Table 2.3 shows the average precision and recall of the two approaches on the whole data set and each subset of data. The DepGraph approach improved the recall by 30.7% on the article data sets, by 7.6% on the email data sets, and by 5.4% on the full data sets. It obtained significant recall increase on the article data sets by exploiting the associations between persons and articles, which compensate for the lack of information for each person reference in itself (each reference contains only a person name). The high precision and recall on the PEmail subset suggest that our algorithm has value also in information spaces that include a single class of references, but with rich associations between the references.

Table 2.4 shows the performance on each individual PIM data set. DepGraph obtained higher F-measures and generated many fewer person reference partitions on all data sets. It improved the recall by 34.8% on data set A, which has the highest variety in the pre-

Table 2.4: Performance for different PIM data sets measured in occurrence-based precision, recall and F-measure.

PIM data set	IndepDec			DepGraph		
#(Persons)/#(Refs)	Prec/Recall	F-measure	#(Par)	Prec/Recall	F-measure	#(Par)
A (1750/24076)	0.999/0.741	0.851	3159	0.999/0.999	0.999	1873
B (1989/36359)	0.974/0.998	0.986	2154	0.999/0.999	0.999	2068
C (1570/15160)	0.999/0.967	0.983	1660	0.982/0.987	0.985	1596
D (1518/17199)	0.894/0.998	0.943	1579	0.999/0.920	0.958	1546
Avg	0.967/0.926	0.946	-	0.995/0.976	0.986	-

Table 2.5: Performance for different PIM data sets measured in representation-based precision, recall and F-measure.

PIM data set	IndepDec		DepGraph	
#(Persons)/#(Refs)	Prec/Recall	F-measure	Prec/Recall	F-measure
A (1750/3114)	0.995/0.509	0.673	0.982/0.947	0.964
B (1989/3211)	0.81/0.803	0.806	0.958/0.891	0.923
C (1570/2430)	0.987/0.782	0.873	0.814/0.925	0.867
D (1518/2188)	0.694/0.837	0.759	0.942/0.737	0.827
Avg	0.872/0.733	0.778	0.924/0.875	0.895

sentations of individual person entities. Note that DepGraph reduced the recall on data set D. The main reason is that the owner of the data set changed her last name and also her email account (at the same email server) when she got married, so after enforcing the constraint, DepGraph divided her references into two partitions with similar sizes. Since the owner is typically the most popular entity in the data set, dividing her references into two partitions leads to large loss in recall. However, we observe that the other references in her data set were better reconciled: DepGraph obtained a much higher precision and reduced the number of partitions by 33. In contrast, two other data set owners also have name changing issues. DepGraph successfully merged their references because they continued to use the same email addresses after the name changes. One minor point is that

Table 2.6: Performance for different PIM data sets measured in diversity and dispersion.

PIM data set	IndepDec		DepGraph	
#(Persons)/#(Refs)	Diversity	Dispersion	Diversity	Dispersion
A (1750/24076)	1.003	1.18	1.003	1.047
B (1989/36359)	1.01	1.067	1.008	1.039
C (1570/15160)	1.003	1.053	1.017	1.03
D (1518/17199)	1.004	1.041	1.005	1.023
Avg	1.005	1.085	1.008	1.035

the precision on data set C is lower than others. The owner of the data set is Chinese and her Chinese friends typically have short names with significant overlap, which makes reconciliation more difficult. For such a data set, we did not find a set of similarity functions or heuristics that improve recall without sacrificing precision. Except for data set C, we did not find distinguishable performance difference between the data sets, which shows our algorithm is robust to variations in the nature of references.

Finally, we also reported the performance on each PIM data set using different measures. Table 2.5 shows the performance measured by representation-based precision, recall and F-measure. The average F-measure of DepGraph was 15% higher than that of IndepDec, showing the significant improvement achieved by our algorithm. Table 2.6 shows the performance measured by diversity and dispersion. Both average diversity and dispersion were close to 1, and the dispersion was improved over that of IndepDec.

Component Contributions

We now analyze the contribution of different components of the algorithm. We conducted experiments on data set A, which has the highest variety and most room for improvement (DepGraph improved the recall from 0.741 to 0.999). This data set contains 24076 Person references, and they refer to 1750 real-world persons.

Our analysis is along two orthogonal dimensions. Along one dimension, we analyzed the contribution of different types of evidence. We started with *Attr-wise* that compares person

Table 2.7: The number of Person reference partitions obtained by different variations of the algorithm on PIM data set A. For each mode, the last column shows the improvement in recall (measured as the percentage reduction in the difference between the number of result partitions and the number of real-world entities) from *Attr-wise* to *Contact* by considering all the additional available evidence. For each evidence variation, the last row shows the recall improvement by applying reconciliation propagation and reference enrichment. The bottom-right cell shows the overall recall improvement of DepGraph over IndepDec.

Mode	*Attr-wise*	*Name&Email*	*Article*	*Contact*	Reduction(%)
Traditional	3159	2169	2169	2096	75.4
Propagation	3159	2146	2135	2022	80.7
Merge	3169	2036	2036	1910	88.7
Full	3169	2002	1990	1873	91.3
Reduction(%)	-	39.9	42.7	64.6	91.3

references by their names and their emails respectively. Then, we considered *Name&Email* that compares names against email addresses. Next, we considered *Article* that exploits the associations between persons and their articles – persons with similar names and having authored the same paper are likely to be the same person. Finally, we considered *Contact* that exploits common email-contacts and co-authors – persons with similar names and also a similar set of email-contacts and co-authors are likely to be the same. Each of these variations considers new evidence in addition to that of the earlier variation. We considered *Contact* last because it is likely to perform the best when some person references have already been reconciled, and thus the contact lists can be merged and enriched.

Along another dimension, we examined the independent contributions of reconciliation propagation and reference enrichment. We considered four modes:

- Full: apply both reconciliation propagation and reference enrichment.

- Propagation: apply only reconciliation propagation.

- Merge: apply only reference enrichment.

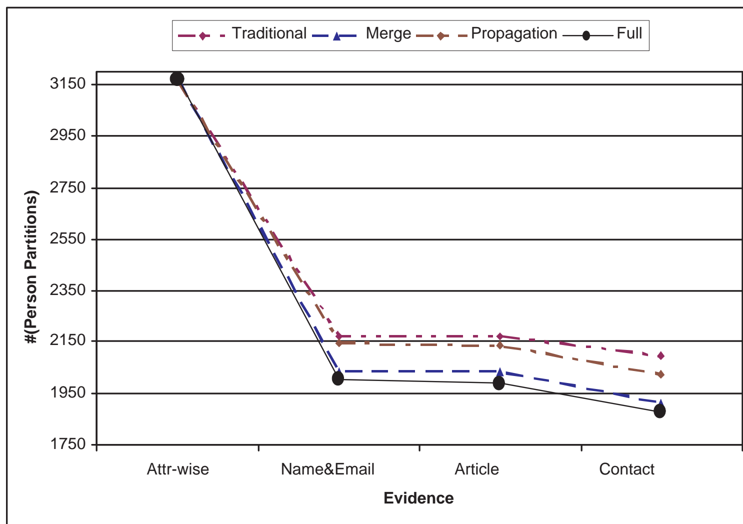


Figure 2.12: Contribution of each type of evidence and each algorithmic feature. The top-left most point represents IndepDec and the bottom-right most point represents DepGraph.

- Traditional: apply neither.

We observed very similar precision for the different variations and modes of the algorithm, so we focus on recall. To demonstrate the difference in recall, we counted the number of person entities returned by each approach (*i.e.*, the number of resulting partitions). Since the precision is about the same in all cases, this count is proportional to the recall. As shown in Figure 2.12, each type of evidence and each algorithm feature progressively contributes to more reconciliation on Person references. Table 2.7 shows the recall improvement in terms of the percentage of reduction in the difference between the number of person partitions returned by the reconciliation algorithm and the number of real-world person entities included in the data set. Note that *Attr-wise* in the Traditional mode is equivalent to the IndepDec approach, and *Contact* in the Full mode is equivalent to the DepGraph approach. The IndepDec approach reconciled the 24076 references into 3159 instances, whereas the DepGraph approach reconciled them into 1873 instances, reducing the difference between the result number of references and the real number of references by 91.3%.

Among the different evidence variations, *Name&Email* dramatically improved the recall.

It helped greatly in reconciling the references extracted from Latex and Bibtex files with the references extracted from emails. It also helped reconcile different email accounts of a person. *Contact* also significantly increased the recall. In the FULL mode, it successfully reduced an extra 117 person partitions.

Among the different modes, the FULL mode obtained the highest recall, and the TRADITIONAL mode obtained the lowest recall. Even when we considered all types of evidence, the TRADITIONAL mode generated 2096 instances. The gap between the resulting number of partitions and the real number of partitions is a factor of 2.81 times the corresponding gap to the FULL mode. We observed that MERGE performed much better than PROPAGATION independent of the types of evidence being used. One important reason is that when the data has a high variety (*i.e.*, one person has several different name presentations and email addresses), reference enrichment effectively accumulates the evidence for more informed reconciliation decisions. Another reason is that reference enrichment merges the contact lists that are originally scattered across different references of the same person, and thus significantly enhances *Contact*. However, the FULL mode does significantly better than either, thus demonstrating their combined utility.

Finally, we observed that reconciliation propagation and reference enrichment require abundant evidence to be effective. For *Attr-wise*, the four modes obtained very similar results. As we considered more evidence, the difference gradually grew. In particular, observe that *Article* enabled PROPAGATION to reconcile authors of the reconciled articles, although it did not provide any benefit for MERGE. On the other hand, *Contact* provided more benefit for MERGE than for PROPAGATION because of the consolidation of email-contact and co-author lists.

Effect of Constraints

We now examine the effect of constraint enforcement. We compared with the NON-CONSTRAINT approach, which does not consider any constraint or negative evidence. In contrast, DEPGRAPH enforces the following three conditions:

1. Authors of a paper are distinct persons.

Table 2.8: Effect of considering constraints on reconciliation: the precision, recall, the number of real-word entities that are involved in erroneous reconciliations (false-positives), and the graph size in terms of the number of nodes.

Method	Prec/Recall	#(Entities with false-positives	#(Nodes)
DepGraph	0.999/0.9994	13	692030
Non-Constraint	0.947/0.9996	61	590438

2. Two persons with the same first name but completely different last name, or with the same last name but completely different first name, are distinct persons unless they share the same email address.

3. A person has a unique account on an email server.

Table 2.8 shows the precision and recall of each approach, along with the number of real-world Person instances involved in false positives. The DepGraph approach obtained very high precision. Among the 1750 real-world instances, only 13 were incorrectly reconciled; among them 5 were mailing lists and hence only 4 pairs of real person instances were incorrectly reconciled. The Non-constraint approach had much lower precision, where 61 instances were incorrectly reconciled. We also observed that although considering constraints added more nodes in the dependency graph, a careful choice of constraints did not necessarily blow up the graph.

2.4.4 The Cora Data Set

Table 2.9 shows the precision, recall and F-measure for DepGraph and IndepDec on the Cora data set. We observed a large improvement of F-measure on Venue references and an improvement on Article and Person references. We note that the Cora data set is very noisy; for example, citations of the same paper may mention different venues. On such a data set, the effect of the propagation from article nodes to venue nodes was two-fold. On

Table 2.9: Precision, recall and F-measure for the Cora data set.

Class	IndepDec		DepGraph	
	Prec/Recall	F-msre	Prec/Recall	F-msre
Person	0.994/0.985	0.989	1/0.987	0.993
Article	0.985/0.913	0.948	0.985/0.924	0.954
Venue	0.982/0.362	0.529	0.837/0.714	0.771

the one hand, it helped reconcile a large number of venues and improve the recall on venue references, and in turn improved the recall on article references. On the other hand, it incorrectly reconciled the many different venues mentioned in citations to the same article and thus reduced the precision.

Finally, we compared our results with other reported experimental results on the same benchmark data set. Bilenko and Mooney [17] reported a 0.867 F-measure on their adaptive approach; Cohen and Richman [36] reported a 0.99/0.925 precision/recall on their approach; and Singla and Domingos [118] reported 0.845/0.949 (precision/recall) for papers, 0.802/0.997 for persons, and 0.720/0.965 for venues on their collective record linkage approach. Because our algorithms handle key attributes in a different way (two references are reconciled if they agree on key values), both IndepDec and DepGraph obtained high precision and recall. Nevertheless, the strength of dependency graph further improved the result and made it comparable to those of the adaptive approaches, even without using any training data.

2.5 Related Work

The problem of reference reconciliation, originally defined by Newcombe et al. [101], was first formalized by Fellegi and Sunter [51]. A large number of approaches that have been since proposed [17, 72, 96, 136, 125] use some variant of the original Felligi-Sunter model. Reconciliation typically proceeds in three steps: first, a vector of similarity scores is computed for individual reference pairs by comparing their attributes; second, based on this vector, each reference pair is compared as either a match or non-match; and finally, a transitive closure

is computed over matching pairs to determine the final partition of references. Attribute comparison is done by using either generic string similarity measures (see [35, 18] for a comprehensive comparison and [26, 125, 17] for recent adaptive approaches) or some domain-specific measures (*e.g.*, geographic locality in [6]). The classification of candidate reference pairs into match and non-match pairs is done through a variety of methods: (a) rule-based methods [72, 89, 59, 78] that allow human experts to specify matching rules declaratively; (b) unsupervised learning methods [136] that employ the Expectation-Maximization (EM) algorithm to estimate the importance of different components of the similarity vector; (c) supervised learning methods [105, 36, 125, 115, 18] that use labeled examples to train a probabilistic model, such as a decision tree, Bayesian network, or SVM, and later apply the model to identify matching pairs; and finally, (d) methods that prune erroneous matching pairs by validating their merged properties using knowledge in secondary sources [41, 46, 98].

Our approach departs from the basic three-step model in that our dependency graph effectively models relations between reconciliation decisions. There is a continuous feedback loop between computing similarities and matching decisions. Importantly, our framework retains the flexibility of employing the different established techniques described previously for computing attribute and reference similarities.

Most prior work has treated reference reconciliation as a single class problem, and it is typically assumed that attribute values, albeit noisy ones, are known for all the references. This model breaks down in dataspace applications such as PIM, where there are multiple classes and individual references can have not only missing attribute values, but also multiple valid attribute values. We are able to offset this complexity by exploiting associations between references to design new similarity measures, and to learn from matching decisions across multiple classes.

The idea of capturing the dependencies between reconciliation decisions has recently been explored in the Data Mining and Machine Learning community. In [103], a complex generative model is proposed to capture dependencies between various classes and attributes and also possible errors during reference extraction. In [118], a dependency model is proposed that propagates reconciliation decisions through shared attribute values. Both of the above approaches entail learning a global detailed probabilistic model from training

data and having the entire reconciliation process guided by that probabilistic model. In complex information spaces that contain multiple classes and complex associations between the classes, learning such a model is impractical. In contrast, our approach provides a mechanism for exploiting influences between reconciliation decisions, and it allows applying different domain-specific models (either heuristic or learned) for particular classes of references.

In [16, 81], associations are used to compute similarities and relate reconciliation decisions. Their proposed heuristics are just a subset of the many heuristics we use. Further, we consider a much more complex domain.

The use of negative information was proposed in [41] to validate individual reconciliation decisions. Our framework exploits the dependency graph to propagate such information for additional benefit.

Finally, several works [72, 96, 26, 78] have addressed the computational cost of reference reconciliation. We follow the spirit of the canopy mechanism [96] to reduce the size of our dependency graph. We insert into the graph only attribute-value pairs and reference pairs that have some potential to be similar.

2.6 Discussion

We now discuss the limitations and possible extensions of our reference reconciliation algorithm. We start with discussion of the performance, and then discuss incremental reconciliation, efficiency of our algorithm, and representation of the reconciliation results.

Performance: Our experimental results show that we obtained an average 0.986 occurrence-based F-measure and an average 0.895 representation-based F-measure on *Person* instances in the four personal data sets. Whereas the performance was good in general, there are several types of mistakes that our algorithm cannot avoid.

Regarding recall, our algorithm can miss reconciliations when there does not exist enough evidence. This error can happen if the attribute values in the two references are very different (such as "Luna Dong" and "Xin Dong") and extra evidence from associations do not exist. Such missed reconciliations often happen to *unpopular* instances, which occur rarely in the

data set and have very few associations; thus, not being able to correctly reconcile them typically is not too annoying.

In addition, constraints may inhibit correctly reconciling some references. Recall from Section 2.4 that the owner of the data set D changed her account in the same email server after marriage. However, we have the constraint that a person has a unique account on an email server, which is typically true. This constraint prevents reconciling the references to the owner and results in two similar-sized reference partitions. Therefore, choosing constraints that are generally applicable while not too strict to be useful is important.

Regarding precision, there are several factors that can affect performance. First, we may incorrectly reconcile references that contain exactly the same attribute values but actually refer to different real-world entities (*e.g.*, two persons with exactly the same name). To completely avoid such mistakes, we should start by considering each occurrence of a reference as an individual one, remove any rules that state references with exactly the same attribute values are highly similar (above the merge threshold), and reconcile two references very conservatively. However, such conservativeness can blow up the dependency graph and often sacrifice the recall. Our algorithm partly solves this problem by merging references that have *sufficient* attribute values that are the same; for example, two Person references with the same email address or with the same name that is fairly long are considered to have sufficient attribute values that are the same. However, how to distinguish individuals with exactly the same attribute values without sacrificing recall remains an open problem.

Second, recall that our algorithm obtained low precision on data set C, which contains a large number of Person references referring to Chinese people. The main reason is that there is a many-to-one relationship between Chinese names and their English translations. In particular, one reason for incorrect reconciliation is that the English translations are typically short and similar; for example, "Xin Deng" and "Xin Dong", which are significantly different in Chinese, look similar in English and our algorithm can consider one as mis-spelling of the other and incorrectly reconcile them. Another reason for incorrect reconciliation is that in English translation a Chinese name can occur both as a first name and as a last name; for example, "Dong Xin" and "Xin Dong" are indeed names of two different persons, but our algorithm may consider one has mis-ordered first name and last

name and so incorrectly merge them. Our algorithm can even aggravate these mistakes by gleaning extra evidence and enriching references. One solution to this problem is to use even more conservative similarity functions when comparing references to Chinese people; however, this requires detecting such references and defining a different set of similarity functions. Another solution is to learn the similarity functions from training data; however, it works better on data sets where all person references refer to Chinese people. It would be beneficial to study which characteristics of a data set can affect the performance, automatically detect such characteristics in the given data set, and adjust the similarity functions accordingly.

Third, our algorithm does not allow backtracking on a reconciliation decision. Sometimes the evidence we see later can fix an error we made early; however, once we decide to reconcile two references, our algorithm merges them through reference enrichment and cannot roll back. We currently solve this problem by introducing constraints to our framework. It is also possible to keep the history of the reconciliation process; however, this potentially can lead to a scalability problem and so we need to find the tradeoff.

Finally, the result of our algorithm is related to the order of the nodes for which we compute similarity scores. We solve this problem by running our algorithm iteratively and our experiments show that many computation orders indeed generate the same results.

User's feedback: Currently our algorithm relies on automatic reconciliation and does not ask for users' feedback; however, users' feedback can help fix errors made by our algorithm and guide further reconciliation. It would be beneficial to study how to provide an interface for user feedback, how to ask critical questions to fix possible errors made by the algorithm, and how to learn from users' feedback to improve the performance of later reconciliations [115].

Incremental reconciliation: Consider two reference sets $\bar{A}$ and $\bar{B}$. We can certainly reconcile references in the two sets by constructing a dependency graph for every pair of references in $\bar{A} \cup \bar{B}$. However, it often happens that we have already reconciled references in $\bar{A}$ and obtained a reference set $\bar{A}'$. In this case, we can reconcile references in $\bar{B}$ incrementally by constructing a dependency graph that compares (1) every pair of instances in $\bar{B}$ and

(2) every pair of instances where one instance is from $\bar{A}'$ and one is from $\bar{B}$. The latter method can significantly improve efficiency when $\bar{A}'$ is much larger than $\bar{B}$. However, our experiments showed that the first method exploits more evidence that is hidden in similarity of references in $\bar{A}'$ and tends to obtain slightly more accurate results. We plan to quantify the difference of the two methods and find the balance between performance and efficiency.

Efficiency: Our algorithm by nature is expensive, because it needs to keep the similarity of every pair of references or attribute values that are potentially similar. Indeed, building the dependency graph (along with comparing attribute values) dominates the execution time. Storing the dependency graph in memory is not scalable to large data sets. So far we solve this problem by pre-processing using some inexpensive reference comparisons, such as merging Person references with the same email address, and by incremental reconciliation. A more scalable solution is to store the dependency graph on disk such as in a database. The reconciliation algorithm requires frequent update of the graph, which can be well supported by the database management system (in contrast, storing the graph in files is not as appropriate).

Representation of the results: There are two representations of our reconciliation results. First, we can discard all similarity scores and generate an instance for each result reference partition. Second, we can keep the similarity score for each pair of input references. Results in the second representation can be used in systems that support management of probabilistic data [118] and the uncertainty we have about the reconciliation decisions can be reflected in answers to queries. However, the space complexity of the second representation is much higher than the first. Let m be the number of input references and n be the number of output partitions. Typically m is much larger than n. The space complexity of the first representation is $O(n)$ and that of the second is $O(m^2)$. Thus, we need to make a trade-off between the reserved information and the space complexity. For example, we can merge a pair of references when the similarity between them is high, and keep the similarity scores for reference pairs that are only marginally similar.

2.7 Summary

Many database applications require resolving heterogeneity at the instance level; that is, reconciling references of multiple classes where rich relationships exist between instances. Thus far, reference reconciliation has been mainly focused on the context of reconciling references of a single class, where each instance contains a fair number of attributes. This chapter fills in this gap by proposing a generic framework that effectively exploits the rich information present in the associations between references and reconciles references of multiple classes at one time. Specifically, to make more informed reconciliation decisions, our framework influences later similarity computation with early reconciliation decisions, and enriches references by instant merging. We apply our algorithm to PIM and Cora data sets, and our experimental results show that it obtains high precision and recall in both applications. Whereas we emphasize applications with multiple classes of objects, our experiments on email-address reconciliation show that our algorithm also benefits record-linkage tasks that match only a single class of objects.

Chapter 3

RESOLVING QUERY-LEVEL HETEROGENEITY I: INDEXING DATASPACES

A dataspace system aims to support queries over disparate data sources where users may not be aware of the structure of the data and can specify queries that have different structures (see Figure 3.1). In such a setting, the user's querying has two characteristics. First, much of the user interaction with the dataspace is of an exploratory nature—the user is getting to know the data and its structure. Second, since there are many disparate data sources, the user cannot query the data using a particular schema. To coordinate these two characteristics, it is important that users are able to use varying degrees of structure in their queries, ranging from keyword queries to structure-aware queries. Furthermore, it is beneficial that the system return *possibly related* data in answer to queries and not only the data that strictly satisfy the query.

To capture these novel querying needs, we propose a class of queries that use keywords to specify values and meanwhile are aware of the structure of the data. We consider the indexing support to efficiently answer such queries. Our index is based on extending inverted lists to capture structure in the data when it is present. This chapter begins by formally defining the queries we consider and introducing our indexing framework in Section 3.1. Then, Section 3.2 describes how to extend inverted lists to support attribute and association information. Section 3.3 shows further extensions for attribute hierarchies and synonyms. Section 3.4 presents experimental results and Section 3.5 discusses related work. Finally, Section 3.6 discusses possible extensions and Section 3.7 summarizes this chapter.

3.1 Problem Definition and Overview of Our Approach

A dataspace can contain data in various formats, such as relational data, XML data, RDF data, and even unstructured data, and we cannot assume the schema mappings between the data sources already known or specified. To provide uniform search and querying

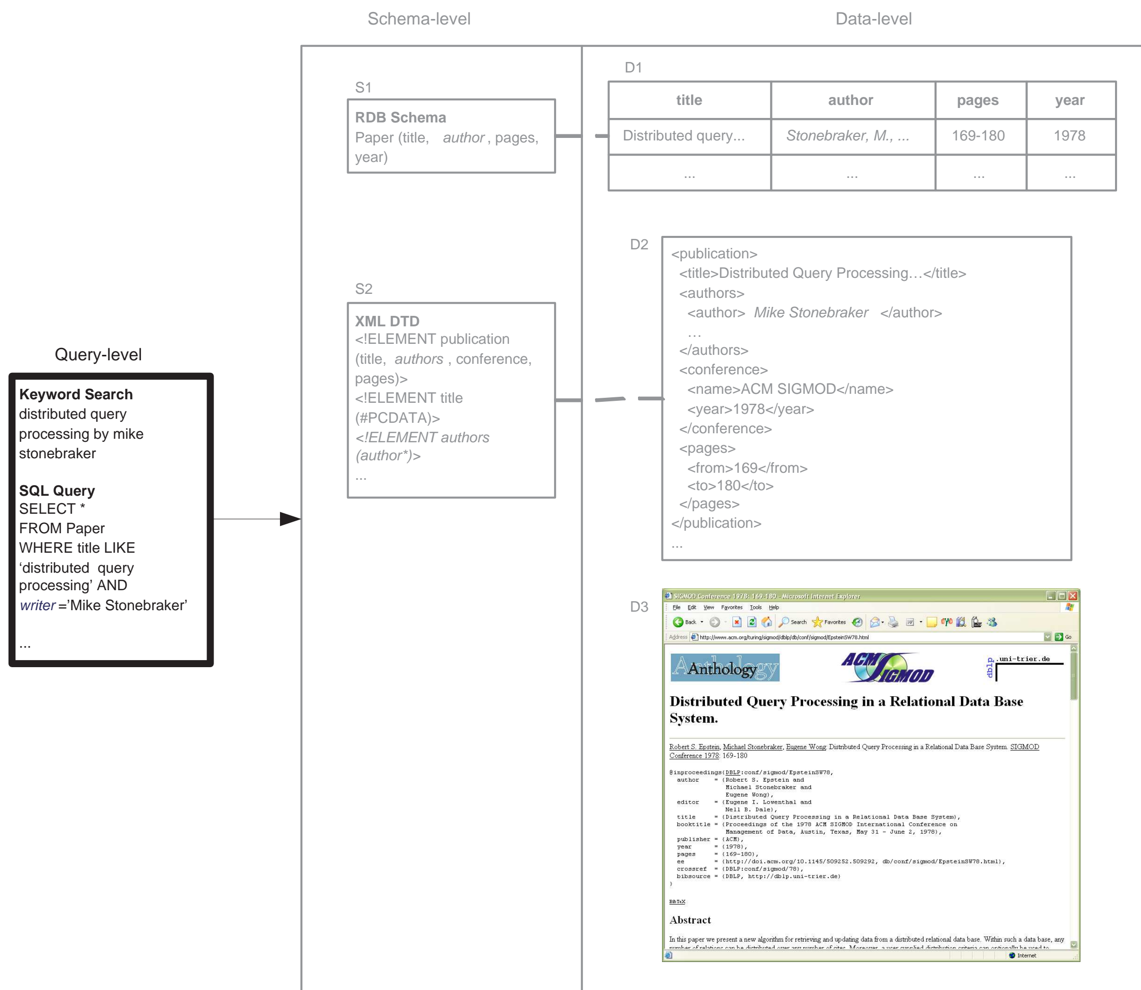


Figure 3.1: Heterogeneity at the query level in a dataspace.

over these heterogeneous data, and index the data to support efficient querying, we need a "normalized" view of the underlying data. The association network extracted from the data sources can serve for this purpose as it captures the essence of the schemas and provides a normalization of the data regardless of their particular formats. We can extract the instances and associations in the association network using a variety of methods. In Chapter 6 we describe several ways to extract instances and associations from personal data. As other examples, we can extract instances and associations from tuples in a relational database by trying to guess the E/R model that may lead to the schema. For example, if the key of a table consists of multiple attributes and each is a foreign key to another table, we consider tuples in the table as representing associations. Note that these extractions are imprecise in nature, so our querying mechanisms and indexing techniques need to allow more flexibility. This chapter omits the details of extraction and focuses on the indexing aspect.

We next formally define the queries we consider, and give an overview of our indexing framework to support these queries.

3.1.1 Querying Heterogeneous Data

Our goal is to support specifying queries with varying degrees of structure requirement over collections of heterogeneous data that are *not* necessarily semantically integrated as in data integration systems. Structured queries require detailed knowledge of the source schemas and precise values of the attributes and thus can be too strict in our context; keyword search, on the other hand, is forgiving but does not allow any specification of structural requirements. To fill in this gap, we consider a class of queries that use keywords to specify values and meanwhile are aware of the structure of the data.

We introduce two types of queries in this class: *predicate queries* and *neighborhood keyword queries*, which we will formally define next. When we define the queries, we use the following example for illustration.

Example 3.1. *Consider the association network depicted in Figure 3.2. It contains three* Person *instances* p_1, p_2, p_3*, one* Article *instance* a_1*, and one* Conference *instance* c_1*. For example,* Paper a_1 *has title "Birch:..."; it is associated with* Person *instances* p_1 *and* p_2*, and*

$p_1 = \{\mathsf{name} = \text{"Tian Zhang"}, \mathsf{authoredPaper} = a\}$

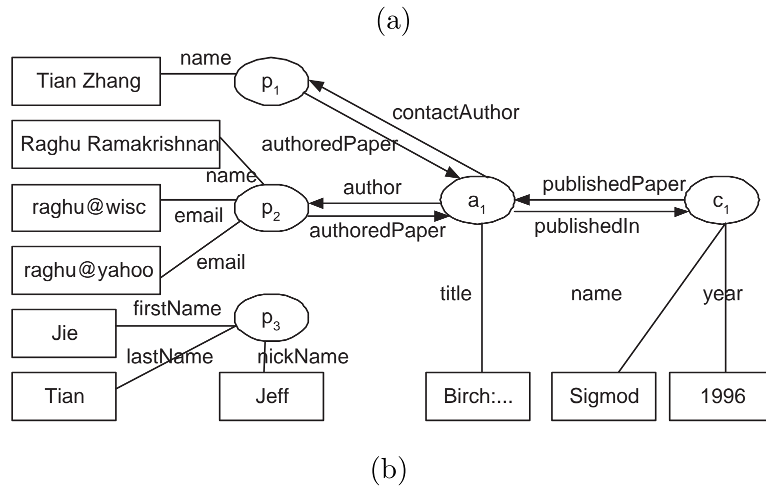
$p_2 = \{\mathsf{name} = \text{"Raghu Ramakrishnan"}, \mathsf{email} = \text{"raghu@wisc"}, \mathsf{email} = \text{"raghu@yahoo"},$
$\quad \mathsf{authoredPaper} = a\}$

$p_3 = \{\mathsf{firstName} = \text{"Jie"}, \mathsf{lastName} = \text{"Tian"}, \mathsf{nickName} = \text{"Jeff"}\}$

$a = \{\mathsf{title} = \text{"Birch:..."}, \mathsf{contactAuthor} = p_1, \mathsf{author} = p_2, \mathsf{publishedIn} = c\}$

$c = \{\mathsf{name} = \text{"Sigmod"}, \mathsf{year} = \text{"1996"}, \mathsf{publishedPaper} = a\}$

(a)



(b)

Figure 3.2: An example association network: (a) list representation; (b) graph representation.

Conference *instance* c_1*.*

Here we assume that the attributes firstName, lastName *and* nickName *are sub-attributes of* name*, and the association* contactAuthor *is a sub-association of* author*.* □

Predicate queries: The first type of queries, called *predicate queries*, describes the desired instances by a set of predicates, each specifying an attribute value or an associated instance.

Definition 3.2. *A* predicate query *contains a set of* predicates*. Each predicate is of the form* $(v, \{K_1, \ldots, K_n\})$*, where* v *is called a* verb *and is either an attribute name or an association name, and* $K_1, \ldots, K_n$ *are keywords.*

The predicate is called an attribute predicate *if* v *is an attribute, and an* association predicate *if* v *is an association.*

The semantics of predicate queries is as follows. The returned instances need to satisfy at least one predicate in the query. An instance satisfies an attribute predicate if it contains at least one of $\{K_1, \ldots, K_n\}$ *in the values of attribute* v *or sub-attributes of* v*. An instance* o *satisfies an association predicate if there exists* $i, 1 \leq i \leq n$*, such that* o *has an association* v *or sub-association of* v *with an instance* o' *that has an attribute value* K_i*.* □

We note that we can also express conjunctions of predicates in our language using boolean expressions with "AND", but the details are irrelevant to our discussion.

Example 3.3. *The query "Raghu's Birch paper in Sigmod 1996" can be described with the following three predicates. The query is satisfied by instance* a_1 *in our example association network.*

```
(title 'Birch'), (author 'Raghu'), (publishedIn '1996 Sigmod')
```

□

In practice, users can specify predicate queries in two ways. First, they can specify a query through a user interface featuring drop-down menus that show all existing attribute or association labels. Second, they can compose the query in a certain syntax (such as the one shown in Example 3.3), specifying attribute or association labels that they know (such

as those in data sources familiar to them). In general, our querying is aimed to be more forgiving in cases where users do not know the schema. For example, we support synonym terms, and we don't require knowledge of attribute hierarchies—users can specify terms anywhere in a hierarchy.

Neighborhood keyword queries: The second type of queries, called *neighborhood keyword queries*, extends keyword search by taking associations into account.

Definition 3.4. *A* neighborhood keyword query *is a set of keywords, $K_1, \ldots, K_n$. An instance satisfies a neighborhood keyword query if either of the following holds:*

- *The instance contains at least one of $\{K_1, \ldots, K_n\}$ in attribute values. In this case we call it a* relevant instance*.*
- *The instance is associated (in either direction) with a relevant instance. In this case we call it an* associated instance*.* □

Example 3.5. *Consider the query "Birch". Instance a_1 is a relevant instance as it contains "Birch" in the* title *attribute, and p_1, p_2, and c_1 are associated instances.* □

Predicate queries and neighborhood keyword queries are different from traditional structured queries in that the user can specify keywords instead of precise values, and provide only approximate structure information. For example, the query in Example 3.3 does not specify if "Raghu" should occur in an author attribute, or in an author sub-element, or in the attribute of another tuple that can be joined with the returned instance. These types of queries are also different from keyword search in that query answering explores the structure of the data to return associated relevant instances.

Clearly, a significant part of answering the above queries is intelligent ranking of the results. We can use a combination of methods, including ranking results that match *several* keywords in a predicate more highly, weighing associations differently when ranking associated instances, and applying PageRank on the association network. The rest of this chapter focuses on the indexing aspects of query answering.

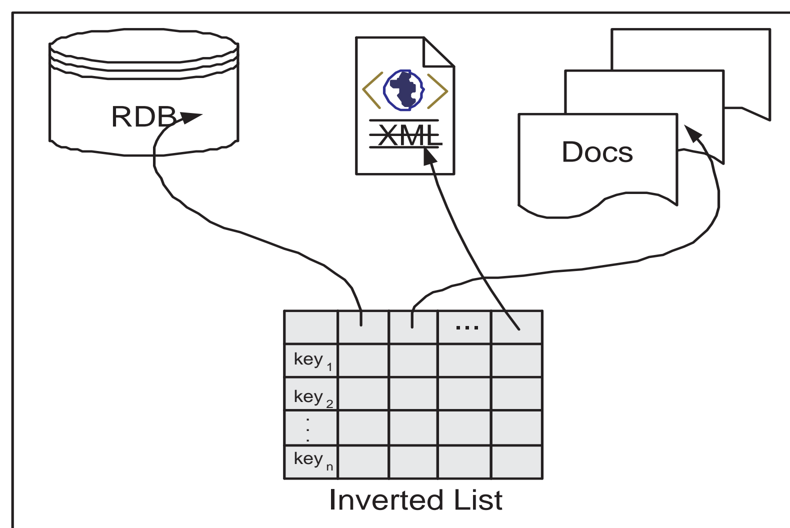


Figure 3.3: The framework of our index: an extended inverted list over a collection of heterogeneous data.

3.1.2 Extending Inverted Lists

Broadly speaking, existing indexing methods either build a separate index for each attribute in each data source to support structured queries on structured data, or create an *inverted list* to support keyword search on unstructured data. Consequently, as we shall show, they fall short in the context of queries that combine keywords and structure. The area in which indexing structure and keywords has received most attention is in the context of XML. However, the techniques proposed for XML indexing fall short in our context for two reasons. First, the XML techniques typically rely on encoding the parent-child and ancestor-descendant relationships in an XML tree; however, the relationships in a dataspace do not fit this model. Furthermore, most XML indexing methods build multiple indexes; as we show in our experiments, visiting multiple indexes to answer a predicate query or a neighborhood keyword query can be quite time-consuming.

Our index is based on extending inverted lists, a technique widely used in Information Retrieval. In our context, answers to queries are data items from the sources, such as files, rows in spreadsheets, tuples in relational databases, or elements in XML data. Hence, in

Table 3.1: The inverted list for the association network in Example 3.1.

	a_1	c_1	p_1	p_2	p_3
1996	0	1	0	0	0
birch	1	0	0	0	0
jeff	0	0	0	0	1
jie	0	0	0	0	1
raghu	0	0	0	3	0
ramakrishnan	0	0	0	1	0
sigmod	0	1	0	0	0
tian	0	0	1	0	1
wisc	0	0	0	1	0
yahoo	0	0	0	1	0
zhang	0	0	1	0	0

the inverted list each row represents a keyword and each column represents a data item from the data sources. (see Figure 3.3). We now review how an inverted list can index a set of instances in an association network and then briefly describe our extensions to the inverted list.

Inverted Lists

Conceptually, an inverted list is a two-dimensional table, where the i-th row represents *indexed keyword* K_i and the j-th column represents instance I_j. The cell at the i-th row and j-th column, denoted (K_i, I_j), records the number of occurrences, called *occurrence count*, of keyword K_i in the attributes of instance I_j. If the cell (K_i,I_j) is not zero, we say instance I_j is *indexed on* K_i. The keywords are ordered in alphabetic order, and the instances are ordered by their identifiers. Table 3.1 shows the inverted list for our example association network.

In practice, an inverted list is seldom stored as a matrix. There are multiple ways to store an inverted list, such as a sorted array, a prefix B-tree or a Patricia trie [9]. In addition,

[138] describes techniques for compression of inverted lists. The extensions we describe next are orthogonal to these physical implementations.

Overview of Extensions

Note that inverted lists, as described above, do not capture any structure information: in the example inverted list, we cannot tell that "tian" occurs as p_1's name (actually, first name) and p_3's lastName. To enable efficient answering of predicate queries and neighborhood keyword queries, we propose to capture both text values and structural information using an extended inverted list. We now briefly describe the several extensions we make to inverted lists.

Our extensions are based on augmenting the text terms in the inverted list with labels denoting the structural aspects of the data such as (but not limited to) attribute tags and associations between data items.

1. When an attribute tag is attached to a keyword, it means that this keyword appears as a value for that attribute; for example, "birch//title//" indexes the papers with title that includes the word "Birch". We assume // is a string reserved for indexing purposes only. Any other delimiter that never occurs in the indexed keywords works too.

2. When an association tag is attached to a keyword, it means that this keyword appears in an associated instance; for example, "birch//authoredPaper//" indexes the people who authored Birch papers.

3. Finally, when an attribute or association path is attached to a keyword, the path indicates the attribute or association hierarchy; for example, "Tian//name//firstName//" indexes people whose first name is "Tian", and indicates firstName is a sub-attribute of name.

In the rest of this chapter, we describe these extensions in detail and describe experiments that validate the utility of our extensions.

3.2 Indexing Structure

This section describes how we index attributes and associations along with keywords to support predicate queries. We consider hierarchies in the next section.

3.2.1 Indexing Attributes

Consider an attribute predicate $(A, \{K_1, \ldots, K_n\})$ in a predicate query. Instances satisfy the predicate if they contain some of the keywords $K_1, \ldots, K_n$ in their A attribute. To handle attribute predicates efficiently, our index should tell us which attributes contain a given keyword.

There are several ways to capture attribute types in indexing. One option is to build an index for each attribute, but as we shall show in the experiments, it can introduce a significant overhead to the index structure. Another option is to specify the attribute name in the cells of the inverted list. For example, the cell ("tian", p_1) in Table 3.1 could be modified to record "name:1". However, this method would considerably complicate query answering. The solution we propose captures attribute names with the indexed keywords to save both index space and lookup time.

Attribute inverted lists (ATIL): We create *an attribute inverted list* (see Table 3.2) as follows. Whenever the keyword k appears in a value of the a attribute, there is a row in the inverted list for $k//a//$. For each instance I, there is a column for I. The cell ($k//a//$, I) records the number of occurrences of k in I's a attributes.

To answer a predicate query with attribute predicate $(A, \{K_1, \ldots, K_n\})$, we only need to do keyword search for $\{K_1//A//, \ldots, K_n//A//\}$. For example, to answer the attribute predicate "lastName, 'Tian'", we transform it into a keyword query "tian//lastName//". In Table 3.2, the search will yield p_3 but not p_1.

3.2.2 Indexing Associations

Consider the association predicate $(R, \{K_1, \ldots, K_n\})$. Instances satisfy the predicate if they have associations of type R with instances that contain some of the keywords $K_1, \ldots, K_n$ in attribute values.

Table 3.2: The ATIL for the association network in Example 3.1.

	a_1	c_1	p_1	p_2	p_3
1996//year//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//nickName//	0	0	0	0	1
jie//firstName//	0	0	0	0	1
raghu//email//	0	0	0	2	0
raghu//name//	0	0	0	1	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
tian//lastName//	0	0	0	0	1
tian//name//	0	0	1	0	0
wisc//email//	0	0	0	1	0
yahoo//email//	0	0	0	1	0
zhang//name//	0	0	1	0	0

One naive solution here would be to perform a keyword search on keywords $\{K_1, \ldots, K_n\}$ and find a set of instances $\{I_1, \ldots, I_m\}$ that contain these keywords. Then, for each instance $I_k, k \in [1, m]$, we find all instances o, such that o is associated with I_k with association type R. This approach can be very expensive for two reasons. First, when m is large, iteratively finding associated instances for each I_k can be expensive. Second, a returned instance can be associated with one or more instances in $\{I_1, \ldots, I_m\}$. Ranking the returned results requires counting the number of associated instances for each result, which can be expensive. We offer a solution that extends inverted lists to also capture association information, thereby avoiding the expensive traversal of the association network.

Attribute-association inverted lists (AAIL): We index association information as follows. Suppose the instance I has an association r with instances $I_1, \ldots, I_n$ in the association network, and each of $I_1, \ldots, I_n$ has the keyword k in one of its attribute values. The inverted list will have a row for $k//r//$ and a column I. The cell ($k//r//$, I) has the value n.

An inverted list that captures both attribute and association information is called an *attribute-association inverted list (AAIL)* (see Table 3.3). Given an association predicate $(R, \{K_1, \ldots, K_n\})$, we can answer it by posing the keyword query $\{K_1//R//, \ldots, K_n//R//\}$ over the AAIL. For example, when searching for "Raghu's papers", the query contains an association predicate "author 'Raghu'" and so we look up keyword "raghu//author//". Based on the AAIL in Table 3.3, we return instance a_1.

Integrating association information in the inverted list increases the size of the index. However, in most applications when the size of the indexed data increases, the average number of associated instances for each instance increases only slightly or even remains the same, so the index typically grows linearly with the size of the data. As discussed in 3.4.4, our experiments show that adding association information into an ATIL (to obtain AAIL) slows down answering attribute predicates only slightly, but it speeds up answering association predicates by an order of magnitude compared with the naive method.

It is interesting to distinguish our association index from join indexes [130], where a precomputed join $R \bowtie S$ is materialized as a separate table and two copies of the table are maintained, one clustered on R's key columns and the other clustered on S's key columns.

Table 3.3: The AAIL for the association network in Example 3.1.

	a_1	c_1	p_1	p_2	p_3
1996//publishedIn//	1	0	0	0	0
1996//year//	0	1	0	0	0
birch//authoredPaper//	0	0	1	1	0
birch//publishedPaper//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//nickName//	0	0	0	0	1
jie//firstName//	0	0	0	0	1
raghu//author//	1	0	0	0	0
raghu//email//	0	0	0	2	0
raghu//name//	0	0	0	1	0
ramakrishnan//author//	1	0	0	0	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
sigmod//publishedIn//	1	0	0	0	0
tian//contactAuthor//	1	0	0	0	0
tian//lastName//	0	0	0	0	1
tian//name//	0	0	1	0	0
wisc//author//	1	0	0	0	0
wisc//email//	0	0	0	1	0
yahoo//author//	1	0	0	0	0
yahoo//email//	0	0	0	1	0
zhang//contactAuthor//	1	0	0	0	0
zhang//name//	0	0	1	0	0

```
procedure Index(𝒯) return L
//𝒯 is an association network;
//Return the AAIL stored in array L;
  Initialize each value of L[][] to 0;
  for each instance I in 𝒯
    for each attribute a of I
      for each keyword k in the values of attribute a
        L[k//a//, I] ++;
    for each association r of I
      for each associated instance J with association r
        for each distinct keyword k in the attribute values of J
          L[k//r//, I] ++;
  return L;
```

(a)

```
procedure Search(L, Q) return Ī
//L is an AAIL; Q is a predicate query;
//Return Ī as a set of relevant instances;
  K = “”; //the keyword query
  for each predicate (V, {K_1, ··· , K_n})
    for i = 1, n
      K += K_i//V// + “ ”;
  return KeywordSearch(L, K); //KeywordSearch() does keyword search
    by looking up the index L for keywords in K;
```

(b)

Figure 3.4: Algorithms for (a) constructing an AAIL for a given association network, and (b) answering a predicate query using an AAIL.

Our association index can be viewed as a union of the original data and multiple join results. This index structure enables us to count the occurrences of keywords and the numbers of associated instances with one scan of the index.

Finally, note that a k-ary association can be modeled as an instance that is related to the k instances involved in the association. Our indexing method can be easily extended to this case and we omit the details.

Figure 3.4 shows the algorithm for indexing an association network and answering a predicate query using the AAIL.

3.3 Indexing Hierarchies

We now consider answering predicate queries in the presence of hierarchies. For example, for the query "name 'Tian'", we wish to return instances p_1 and p_3, rather than only p_1.

A simple method to incorporate hierarchies would be first to find all descendants of the name attribute (in this example, they are firstName, lastName and nickName), and then to expand the keyword query by considering also descendant attributes (so search "tian//name// OR tian//firstName// OR tian//lastName// OR tian//nickName//"). However, this method requires multiple index lookups and thus can be expensive.

Our solution is based on integrating the hierarchy information into the index structure. We begin by describing two possible solutions and then combine their features and introduce a hybrid indexing scheme. For ease of explanation, we consider only attribute hierarchies, but the same principle applies to association hierarchies. We assume that each attribute has at most a single parent attribute. This covers most cases in practice and the approach can be easily extended to multiple-inheritance cases.

3.3.1 Index with Duplication

Our first solution duplicates a row that includes an attribute name for each of its ancestors in the hierarchy.

Attribute inverted lists with duplication (Dup-ATIL): We construct a Dup-ATIL as follows. If the keyword k appears in the value of attribute a_0, and a is an ancestor of a_0 in

Table 3.4: The Dup-ATIL for the association network in Example 3.1. The difference from Table 3.2 is highlighted using bold font.

	a_1	c_1	p_1	p_2	p_3
1996//year//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//name//	**0**	**0**	**0**	**0**	**1**
jeff//nickName//	0	0	0	0	1
jie//firstName//	0	0	0	0	1
jie//name//	**0**	**0**	**0**	**0**	**1**
raghu//email//	0	0	0	2	0
raghu//name//	0	0	0	1	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
tian//lastName//	0	0	0	0	1
tian//name//	0	0	1	0	**1**
wisc//email//	0	0	0	1	0
yahoo//email//	0	0	0	1	0
zhang//name//	0	0	1	0	0

the hierarchy (a could also be a_0), then there is a row $k//a//$. The cell ($k//a//$, I) records the number of occurrences of k in values of the a attribute and a's sub-attributes of I. We answer a predicate query with the Dup-ATIL in the same way as we use the ATIL.

Example 3.6. *Table 3.4 shows the Dup-ATIL for our example. Consider instance p_3. We index p_3 not only on* "jie//firstName//", "tian//lastName//", *and* "jeff//nickName//", *but also on* "jie//name//", "tian//name//", *and* "jeff//name//". *Thus, in the inverted list, row* "tian//name//" *also records one occurrence for instance p_3, and there are new rows* "jie//name//" *and* "jeff//name//", *each recording one occurrence for instance p_3.*

Now consider searching a person with name "Tian". We transform the query "name *'Tian'" into keyword search* "tian//name//" *and return instances p_1 and p_3.* □

Table 3.5: The Hier-ATIL for the association network in Example 3.1. The difference from Table 3.2 is highlighted using bold font.

	a_1	c_1	p_1	p_2	p_3
1996//year//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//name//nickName//	0	0	0	0	1
jie//name//firstName//	0	0	0	0	1
raghu//email//	0	0	0	2	0
raghu//name//	0	0	0	1	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
tian//name//	0	0	1	0	0
tian//name//lastName//	0	0	0	0	1
wisc//email//	0	0	0	1	0
yahoo//email//	0	0	0	1	0
zhang//name//	0	0	1	0	0

On the one hand, a Dup-ATIL has the benefit of simple query answering, but on the other hand, it may considerably expand the size of the index because of the duplication. The size of the Dup-ATIL will be especially affected if the attribute hierarchy contains long paths from the root attribute to the leaf attributes and most values in the association network belong to leaf attributes.

3.3.2 Index with Hierarchy Path

We now introduce a second solution, which does not affect the number of rows in the inverted list. Instead, the keyword in every row includes the entire hierarchy path.

Attribute inverted lists with hierarchies (Hier-ATIL): We construct a Hier-ATIL by extending the attribute inverted list as follows (see Table 3.5). Let $a_0, \ldots, a_n$ be attributes such that for each $i \in [0, n-1]$, attribute a_i is the super-attribute of a_{i+1}, and a_0 does

not have super-attribute. We call $a_0//\ldots//a_n//$ a *hierarchy path* for attribute a_n. For each keyword k in the value of attribute a_n, there is a row for $k//a_0//\ldots//a_n//$. For each instance I, there is a column for I. The cell ($k//a_0//\ldots//a_n//$, I) records the number of occurrences of k in I's a_n attributes.

A Hier-ATIL captures the hierarchy information using hierarchy paths, which have a nice feature: the hierarchy path of an attribute A is a prefix of the hierarchy paths of A's descendant attributes. Thus, we can transform an attribute predicate into a prefix search. Specifically, consider a query predicate $(A, \{K_1, \ldots, K_n\})$. We transform it into a prefix search: $K_1//A//*, \ldots, K_n//A//*$. For example, we can transform the query predicate "name 'Tian'" into a prefix search "tian//name//*" and so return both p_1 and p_3.

Since the indexed keywords in an inverted list are ordered, we can answer a prefix query easily. To look up prefix $P*$, we first locate the first row where the keyword is P or starts with P. We then scan the subsequent rows until we reach an indexed keyword that does not start with P, and we accumulate the occurrence counts in these rows for each instance.

Unlike Dup-ATIL, building a Hier-ATIL does *not* increase the number of indexed keywords. Although it can lengthen many of the indexed keywords, real indexing systems typically record a keyword only by the difference from its previous keyword (for example, given a keyword k_1 and a succeeding keyword k_2, where the maximal common prefix of k_1 and k_2 is p, an index can record k_2 by the length of p and k_2's suffix that differs from k_1). Thus, building a Hier-ATIL introduces only a small overhead. However, with Hier-ATIL we need to answer a predicate query by transforming it into a prefix search, which can be more expensive than a keyword search. Answering a prefix search $K//A//$ is especially expensive when K occurs in many different attributes that are descendants of A.

It is interesting to compare our approach with the one proposed in [37] in the context of indexing XML data, where the focus was on answering queries with path expressions. Whereas we index a keyword *followed by* the hierarchy path, [37] indexes an XPath with the keyword *in the end*. Our approach has two advantages in our context. First, attribute keywords have much higher variety than attribute names and thus are more selective. Second, in the presence of attribute hierarchies, using our index we can transform a query predicate into a prefix search (*e.g.*, "tian//name//*"), but using their index we need to transform it

Table 3.6: The Hybrid-ATIL with threshold t=1 for the association network in Example 3.1. The difference from Table 3.5 is the row for "tian//name////".

	a_1	c_1	p_1	p_2	p_3
1996//year//	0	1	0	0	0
birch//title//	1	0	0	0	0
jeff//name//nickName//	0	0	0	0	1
jie//name//firstName//	0	0	0	0	1
raghu//name//	0	0	0	1	0
raghu//email//	0	0	0	2	0
ramakrishnan//name//	0	0	0	1	0
sigmod//name//	0	1	0	0	0
tian//name////	**0**	**0**	**1**	**0**	**1**
tian//name//lastName//	0	0	0	0	1
wisc//email//	0	0	0	1	0
yahoo//email//	0	0	0	1	0
zhang//name//	0	0	1	0	0

into a general regular-expression query (*e.g.*, "name/*/tian//"), which can be much more expensive to answer.

3.3.3 Hybrid Index

The two solutions we have proposed have complementary benefits: Dup-ATIL is more suitable for the cases where a keyword occurs in many attributes with common ancestors, and Hier-ATIL is more suitable for the cases where a keyword occurs in only a few attributes with common ancestors. We now describe a hybrid indexing scheme that combines the strengths of both methods.

Hybrid attribute inverted list (Hybrid-ATIL): The goal of a Hybrid-ATIL is to build an inverted list that can answer any prefix search (ending with "//") by reading no more than t rows, where t is a threshold given as input to the algorithm.

procedure Lookup(L, P) **return** S

//L is a Hybrid-ATIL; P is a prefix to look up;

//Return S, an array summarizing for each instance the occurrences of keywords

//with prefix P;

Initialize each value of $S[]$ to 0;

Locate the first keyword $\bar{K} = K +'' //''$ with prefix P;

while P is the prefix of $\bar{K}$

  Update S according to the row for $\bar{K}$;

  if ($\bar{K}$ ends with "////")

    if $K = P$ **return** S;

    else Skip all succeeding keywords with prefix K;

  Read the next keyword $\bar{K} = K +'' //''$;

return S;

Figure 3.5: The algorithm for looking up a prefix in a Hybrid-ATIL.

We build the Hybrid-ATIL by starting with the Hier-ATIL and successively adding *summary rows*, using a strategy we shall describe shortly. The indexed keyword in a summary row is of the form $p//$, where $p = k//a_0//\dots//a_l//$, k is a keyword, and $a_0//\dots//a_l//$ is a hierarchy path for attribute a_l. Rows whose indexed keywords start with p are said to be *shadowed* by the summary row $p//$. Note that keywords in summary rows end with an additional // to be distinguished from ordinary rows. The cell ($p//$, I) has the sum of the occurrence counts of I in $p//$'s shadowed rows. The Hybrid-ATIL with threshold $t = 1$ for the example association network is shown in Table 3.6.

To answer a prefix query of the form $k//a_1//\dots//a_m//*$, we look at all the rows with prefix $k//a_1//\dots//a_m//$ *except* those shadowed by summary rows. Figure 3.5 shows the algorithm for prefix lookup in a Hybrid-ATIL. The key idea is that when we encounter a keyword of the form $K////$, we ignore its shadowed rows.

Example 3.7. *Consider the following two queries on the example association network.*

- *Q_1: **name** "Jeff"*

- *Q_2: **name** "Tian"*

Query Q_1 is transformed into prefix search "jeff//name//*"*. In Table 3.6, only keyword* "jeff//name//nickName//" *contains this prefix, so we return instance p_3.*

Query Q_2 is transformed into prefix search "tian//name//*"*. As the Hybrid-ATIL contains a summary row with indexed keyword* "tian//name////"*, we can directly return instances p_1 and p_3 without considering other keywords.*

In both cases, we read no more than one row (recall that $t = 1$ for the index) to answer a prefix search. □

Creating the Hybrid-ATIL: We begin with the Hier-ATIL and add summary rows until none can be added. We denote by $Ans(p)$ the number of rows we need to examine to answer a prefix query p. We create a summary row for a prefix p if $Ans(p) > t$ and there is no p', such that p is a prefix of p' and $Ans(p') > t$. If we add a summary row for $p//$, we remove the p row from the inverted list if one exists.

In Table 3.5, $Ans(\text{“}tian//name//\text{''}) = 2$. Therefore, with threshold $t = 1$, the row "tian//name//" would be replaced by a summary row, as shown in Table 3.6. As we show in Appendix A, we can construct the Hybrid-ATIL from the Hier-ATIL with a single pass over the keyword entries.

The construction of the Hybrid-ATIL guarantees that $Ans(p) \leq t$ for any prefix p. Note that adding summary rows can increase the size of the index. However, by choosing an appropriate threshold t we can make a tradeoff between index size (so the prefix-lookup time) and occurrence-accumulation time.

Note that the Information Retrieval community has proposed other types of indexes for regular-expression matching, such as *suffix tree* [8], which indexes all suffixes of each document, and *multigram index* [32], which creates k-gram indexes for reasonable k values ($e.g.$, $k = 2, 3, \ldots, 10$). In addition, HYB [11] and KISS [80] have recently been proposed for general prefix matching. Compared with these approaches, our index is oriented to prefix

matching where we know the exact prefix delimiters ("//" in our case), thus indexing and searching can be more efficient.

3.3.4 Schema-Level Synonyms

Accommodating different hierarchical structures is already an important step towards supporting data heterogeneity in our indexing mechanism. We now briefly describe how our techniques easily handle two other forms of heterogeneity.

The first form of heterogeneity is where an association in one source is an attribute in another. For example, author can be an attribute of a Paper instance with author names as attribute values, or an association between Paper instances and Person instances. Since our index does not distinguish attributes and associations in the indexed keywords, it naturally incorporates this kind of heterogeneity.

The second form of heterogeneity, *term heterogeneity*, is where different terms represent the same attribute or association. For example, author and authorship can describe the same association.

To accommodate term heterogeneity, we assume we have a synonym table for attribute and association names. If attribute a is referred to as $a_1, \ldots, a_n$ in different data sources, we choose the *canonical* name of a as one of $a_1, \ldots, a_n$. We note that the synonyms are either given to us or are derived using schema-matching techniques, and hence will typically be approximate.

In our index, when a keyword k appears in a value of the a_i attribute, there is a row in the inverted list for $k//a//$. For each instance I, there is a column for I. The cell $(k//a//, I)$ records the number of occurrences of k in I's $a_1, \ldots, a_n$ attributes.

To answer a predicate query with attribute predicate $(a_i, \{K_1, \ldots, K_n\}), i \in [1, n]$, we transform it into a keyword search for $\{K_1//a//, \ldots, K_n//a//\}$. In our example, if we consider author as a canonical name for author and authorship, the attribute predicate "authorship, 'Tian'" will be transformed into "tian//author//" instead of "tian//authorship//".

Table 3.7: The KIL with threshold $t = 1$ for the association network in Example 3.1. To save space, we only show the rows where the indexed keywords start with "birch".

	a_1	c_1	p_1	p_2	p_3
birch////	1	1	1	1	0
birch//authoredPaper//	0	0	1	1	0
birch//publishedPaper//	0	1	0	0	0
birch//title//	1	0	0	0	0

3.3.5 Neighborhood Keyword Queries

The indexing methods we described so far lend themselves almost immediately to answering neighborhood keyword queries. We build the *Keyword Inverted List (KIL)*, which is essentially a Hybrid-AAIL. In a KIL we summarize not only prefixes that end with hierarchy paths, but also prefixes that correspond directly to keywords. To answer a neighborhood keyword query with keywords $K_1, \ldots, K_n$, we transform it into a prefix search for $K_1//*, \ldots, K_n//*$.

Example 3.8. *Table 3.7 shows a fragment of the KIL with threshold $t = 1$. Given the neighborhood keyword query "Birch", we look up "birch//*" and return instances a_1, c_1, p_1 and p_2.* □

Note that if we wish to distinguish between the relevant instances (those for which the keywords occur in attribute values) and the associated instances (those for which the keywords occur in associated instances), we can add two special symbols as the root of all attributes and the root of all associations, and index accordingly.

3.4 Experimental Evaluation

We now describe a set of experiments that validate the efficiency of our indexing methods and compare them against several alternatives. Our main result is that by indexing both structure information and text values, we can considerably improve the performance of answering predicate queries and neighborhood keyword queries. In addition, we examine

the efficiency of updating the index and the scalability of our index.

3.4.1 Experimental Setup

The main data set we use is constructed from a collection of personal data on the desktop and a few external sources. We extract associations between disparate items on the desktop (*e.g.*, LATEX and BIBTEX files, Word documents, Powerpoint presentations, emails and contacts, and webpages in the web cache). The instances and associations are stored in an RDF file, managed by the Jena System [76]. The RDF file contains 105,320 object instances, 300,354 attribute values, 468,402 association instances, and the size of the file is 52.4MB. We describe additional data sets we used in our scale-up experiment in Section 3.4.4.

We considered four types of queries:

- PQAS (Predicate Queries with Attribute-clauses for Simple attributes): Predicate queries with only attribute clauses where the attributes do *not* have sub-attributes;

- PQAC (Predicate Queries with Attribute-clauses for Complex attributes): Predicate queries with only attribute clauses where the attributes *do* have sub-attributes;

- PQR (Predicate Queries with clauses for Relationships): Predicate queries with only association clauses;

- NKQ (Neighborhood keyword queries): Neighborhood keyword queries where we did not distinguish between relevant and associated instances.

We varied the number of clauses in the first three types of queries from one to five, and each clause had a single keyword. For NKQs, we varied the number of keywords from one to five. The keywords, attributes, and associations were randomly drawn from the data set.

For each query configuration, we randomly generated 100 queries and executed each three times. We report the average execution time. To further refine our measurements we also consider the index-lookup time, including the time to locate the entries in the inverted list, the time to retrieve the occurrence counts for each returned instance, and also the time to handle succeeding rows in case of prefix lookup.

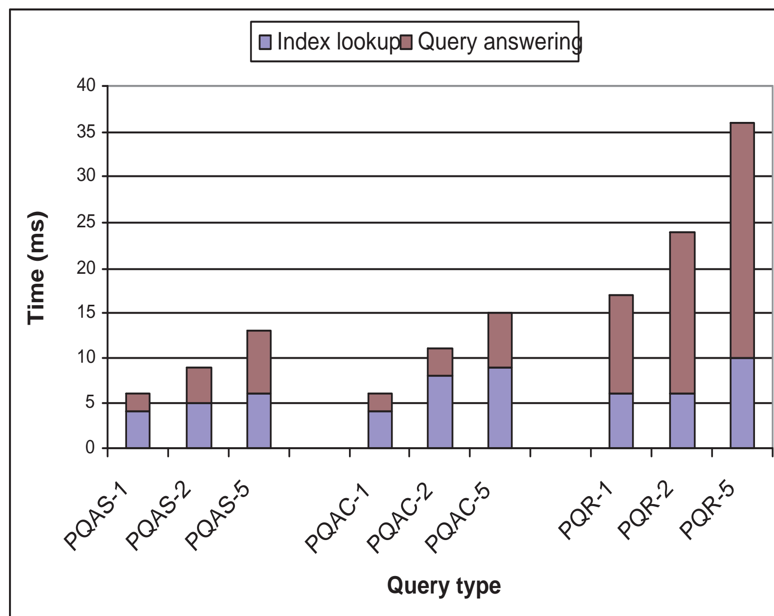


Figure 3.6: Efficiency of answering predicate queries. In each column, the longer bar shows the overall query-answering time and the shorter bar shows index-lookup time.

We implemented the indexing module using the Lucene indexing tool [92], which stores an inverted list as a sorted array on disk. We implemented our algorithm in Java, and conducted all the experiments on a machine with four 3.2GHz and 1024KB-cache CPUs, and 1GB memory.

3.4.2 Indexing and Searching

We tested the efficiency of the KIL, the hybrid hierarchical index (see Section 3.3.5). It took 11.6 minutes to build the KIL and its size is 15.2MB. The query-answering time of predicate queries is shown in Figure 3.6 and that of both predicate queries and neighborhood keyword queries is shown in Table 3.8.

We make three observations about the results. First, answering predicate queries and neighborhood keyword queries using the KIL was very efficient: on average it took 15.2 milliseconds to answer a predicate query with no more than 5 clauses, and took 224.3 milliseconds to answer a neighborhood keyword query with no more than 5 keywords. Second,

Table 3.8: Comparison of search efficiency using the KIL, using separate indexes as proposed in [85], and using a simple inverted list.

(ms)		1 clause		2 clauses		5 clauses	
		Index lookup	Query answer	Index lookup	Query answer	Index lookup	Query answer
PQAS	NAIVE	2	22	3	53	4	129
(Predicate queries	SEPIL	7	9	8	11	10	15
with simple attributes)	KIL	4	6	5	7	6	13
PQAC	NAIVE	3	43	3	119	4	583
(Predicate queries	SEPIL	7	11	23	28	31	38
with complex attributes)	KIL	4	6	8	11	9	15
PQR	NAIVE	3	88	7	147	12	368
(Predicate queries	SEPIL	301	415	559	749	1397	1871
with associations)	KIL	6	17	6	24	10	36
NKQ	NAIVE	18	4174	28	5244	50	8407
(Neighborhood	SEPIL	365	488	717	1052	1662	2376
keyword queries)	KIL	48	97	103	182	232	394

answering PQASs and PQACs (where attribute hierarchies were considered) consumed a similar amount of time, showing the effectiveness of our hybrid indexing scheme. Third, though answering PQRs (queries with associations) took longer than answering PQASs and PQACs, they spent a similar amount of time in index lookup. The difference was in the time to retrieve the answers, and there were much more of them for the PQRs than for the other two types of queries. For the same reason, it took much longer to answer NKQs.

Comparison of methods

Next, we compare our index with several alternative approaches. We first compare the efficiency of KIL with two other methods: NAIVE and SEPIL. The NAIVE method is based on the basic inverted list (alluded to in Section 3.1). Specifically, NAIVE begins by looking up the set of instances $\mathcal{I}$ that contain the given keywords in attribute values and then does the following:

- PQAS: Select from $\mathcal{I}$ the instances where the keywords appear in the specified attributes;

- PQAC: The same as PQAS, but also consider descendant attributes;

- PQR: Find the instances that are related to the ones in $\mathcal{I}$ with the specified associations;

- NKQ: Union $\mathcal{I}$ with all instances that are associated with those in $\mathcal{I}$.

The SepIL method is an adaptation of the approach proposed in [85] to our context (originally it was designed for complex XML queries). Specifically, it builds three separate indexes: the *inverted list* indexes each attribute value on its text, the *structured index* indexes each attribute value on the labels of the attribute and its ancestor attributes, and the *relationship index* indexes each instance on its associated instances. SepIL begins by looking up the inverted list for a set of attribute values $\mathcal{A}$ that contain the query keywords, and meanwhile getting their owner instance set $\mathcal{I}$. Then SepIL does the following:

- PQAS and PQAC: Look up the *structured index* for values of the specified attributes and intersect the results with $\mathcal{A}$, then return the owner instances;

- PQR: Look up the *relationship index* for the instances that are related to the ones in $\mathcal{I}$ with the specified associations;

- NKQ: Look up the *relationship index* for the instances associated with the ones in $\mathcal{I}$ and union the results with $\mathcal{I}$.

Note that unlike our approach, Naive and SepIL return the instances without counting keyword occurrences or the number of associated instances. Performing the count would add a significant overhead to both of these techniques.

Table 3.8 shows the query-answering time using these three different indexes. It took 1.7 minutes to build the Naive index, whose size was 10.6MB. Query answering was inefficient

using the Naive index, because we had to find the involved attributes and extract associated instances at run time. It was especially inefficient when we had to extract associated instances for a large number of instances that contain the given keywords. Indeed, compared with KIL, query-answering time on average increased by a factor of 15.9 and for 1-clause NKQs increased by a factor of 43. We also observed that although KIL spent more time on index lookup (because the index was 1.4 times as large and more instances were returned in each index lookup), the overall payoff in query-answering time significantly outweighed this additional cost.

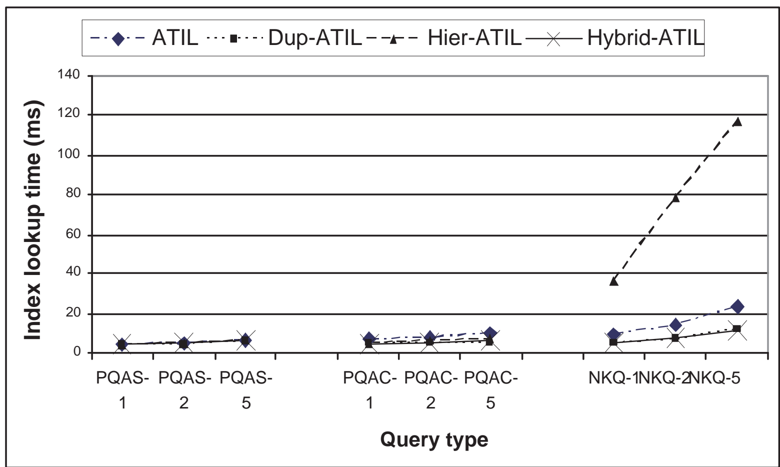
It took 5.7 minutes to build the SepIL index. The total size of the inverted list and the structured index was 28.1MB, and the size of the relationship index was 14.2MB. For PQAS and PQAC queries, query-answering time was reduced on average by a factor of 6.6 compared with Naive; however, because the inverted list is large, it still took about twice as much time as KIL. For other queries that require looking up the relationship index, query answering took much longer than KIL (by a factor of 20.7). This is because the index is large, and for each instance that contains the keyword we need to look up the index and accumulate its associated instances. Even when compared with Naive, there was only a benefit to building the relationship index for answering NKQ queries, which typically returned a large number of instances.

We performed several other experiments to validate different aspects of our indexing methods. For example, we considered only attributes and compared the efficiency of the ATIL with a technique that creates a separate index for each attribute. We observed that ATIL reduced indexing time by 63% and reduced keyword-lookup time by 33%.

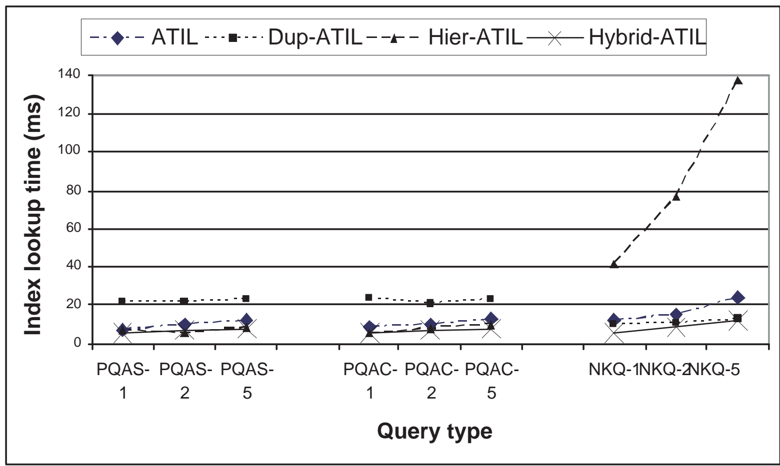
Indexing hierarchies

We now compare different methods for indexing attribute hierarchies that were described in Section 3.3:

- ATIL: expand a query by issuing a query for every descendant attribute (without accumulating keyword occurrences for result instances);
- Dup-ATIL: duplicate keywords for ancestors in the index;



(a)



(b)

Figure 3.7: Efficiency of looking up different types of indexes in answering predicate queries with attribute clauses and neighborhood keyword queries (a) on shallow-hierarchy association network, and (b) on deep-hierarchy association network.

Table 3.9: Comparison of indexing efficiency for different types of inverted lists.

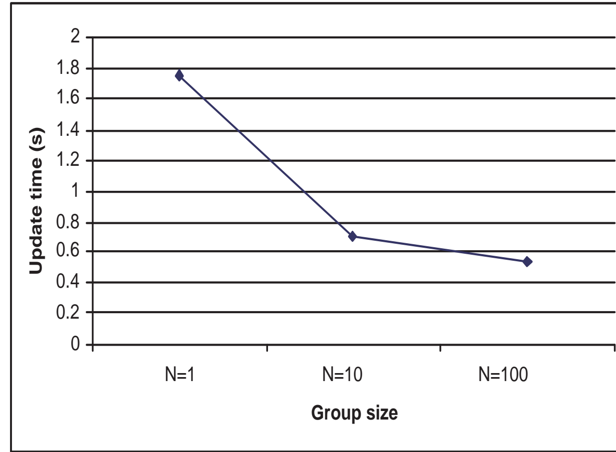
Index type	Indexing time for shallow-hierarchy association network (s)	Indexing time for deep-hierarchy association network (s)
ATIL	118	118
Dup-ATIL	125	418
Hier-ATIL	119	140
Hybrid-ATIL	125	144

- Hier-ATIL: attach the ancestor path in the index;
- Hybrid-ATIL: the hybrid index.

In the data set we experimented on, the depth of each attribute hierarchy is no more than 3. To examine the effect of hierarchy depth on search efficiency, we also experimented with an association network where depths of attribute hierarchies are all over 16. We call the former a *shallow-hierarchy* association network and the latter a *deep-hierarchy* association network. The two association networks have exactly the same data but different schemas: if an attribute does not have any parent attribute in the shallow-hierarchy association network, in the deep-hierarchy association network it has a parent attribute $attr_0$, a grand-parent attribute $attr_1$, and so on, till the upmost ancestor $attr_{15}$.

Table 3.9 shows the index-building time for these inverted lists on both association networks. Note that a higher hierarchy depth had significant effect for the construction of only the Dup-ATIL (increasing indexing time for a factor of 3.34 and the size of the index for a factor of 4); building a Hier-ATIL or a Hybrid-ATIL took a similar amount of time as building an ATIL on both association networks and the sizes of the indexes are similar. Also note that building a Hybrid-ATIL took only slightly longer than building a Hier-ATIL, which shows that our algorithm for adding summary rows in Hybrid-ATILs is efficient.

Figure 3.7 shows the index-lookup time (we omit the query-answering time as it adds the same amount of time over the index-lookup time for all alternative methods). As we index



(a)

Operation	Update time (s)
Rename an attribute	2.2
Insert a parent	2.4
Rename a parent	2.5
Delete a parent	2.0

(b)

Figure 3.8: Efficiency of index updates: (a) instance updates; (b) structure updates.

only attribute hierarchies, we illustrate the efficiency of our algorithm using only predicate queries with attribute clauses and neighborhood keyword queries. We observe that (1) Hier-ATIL performed poorly on NKQs, as prefix lookup became extremely expensive; (2) Dup-ATIL performed poorly on the deep-hierarchy association network, as the index size was increased a lot, and (3) Hybrid-ATIL performed better than or equal to any other inverted lists for all types of queries on both data sets.

3.4.3 Index Updates

Our next experiment was designed to measure the efficiency of updating the KIL, both for instance updates and for updates to the schema.

For instance updates, we randomly selected 100 instances, divided them into groups,

and interleaved insertion and deletion in each group. We updated a group of instances incrementally; that is, we inserted or deleted the instances in the group, and updated their associated instances in the index. We varied the size of the group: 1, 10 and 100, and compared the average time of updating an instance in KIL.

Figure 3.8(a) shows the time for inserting or deleting an instance in KIL. The results for insertion-only or deletion-only updates were similar. We observed that when the group size was increased, the update time per instance dramatically dropped. For example, when $N = 100$, updating an instance took on average only 0.5 seconds. In addition, when the size of the group was increased, the speedup of the updates slowed down.

We also observed that index updates in the SepIL method were slower by a factor of 2.25 compared to updates in KIL, but updates in Naive were considerably faster than in both methods. This is because most of the update cost arose from the need to update associated instances in the index. However, as previous experiments have shown, indexing associations significantly sped up query answering at run-time and thus was worthwhile.

For structure updates, we considered four types of operations: renaming an attribute, inserting, updating, and deleting a parent for an attribute. For each operation, we chose three attributes that occur with different frequencies in the data set, and reported the average time for updates. We performed each operation by scanning the inverted list and changing the indexed keywords appropriately. Structure updates were performed very efficiently. As shown in Figure 3.8(b), it took 2.28 seconds on average to perform each type of structure update.

3.4.4 Scalability

Finally, we tested the scalability of KIL with larger data sets. In the first experiment, we created a 250MB data set by adding to the original data set four copies of itself and then perturbing the result data set. Specifically, we chose a perturbation factor $f \in [0, 1]$. When we perturbed the keywords with factor f, we randomly selected a fraction f of the keywords in the data set, and for those words we added one of the suffixes in $\{!, @, \#, \%\}$ with equal probability (these signs do not occur in the original index). Hence, when $f = 0$,

Table 3.10: Index-lookup time for answering (a) the original queries and (b) suffix queries on 250MB data sets with perturbed keywords. For the purpose of comparison, we also list the index-lookup time on the 25MB data.

(ms)	25MB	f=0	f=0.2	f=0.4	f=0.6	f=0.8
PQAS	5	25	24	23	22	24
PQAC	8	26	27	27	26	26
PQR	6	32	30	35	37	49
NKQ	103	805	628	490	318	139

(a)

(ms)	f=0	f=0.2	f=0.4	f=0.6	f=0.8
PQAS	22	27	28	26	26
PQAC	24	27	28	28	27
PQR	29	43	48	49	46
NKQ	27	46	86	131	146

(b)

the keywords are the same as those in the original set; when $f = 0.8$, for any keyword k, the number of its occurrences is about the same as that for k! (or k with any other suffix). We perturbed attribute and association names in the same way.

We experimented on two sets of queries: the original queries and *suffix* queries, where "!" was added to each keyword. We considered randomly generated queries with two clauses.

We now describe our experimental results on data sets with perturbed keywords. We observed the same trend for data sets with perturbed attribute and association names. As f was increased, the indexing time went up gradually from 55.3 minutes to 58.2 minutes, and the size of the index went up gradually from 71.2MB to 76.4MB, all roughly 5 times as much as for the original data set. As shown in Table 3.10, index lookup was efficient: on average it took 30.3 milliseconds for predicate queries and 281.6 milliseconds for neighborhood keyword queries. In addition, we make three observations.

First, when the number of answers was small, index-lookup time was more related to the size of the index. For all predicate queries, index-lookup time was roughly 5 times as

Table 3.11: Indexing time and index-lookup time for 10GB XML data sets. Note that some index-lookup time is not reported as the type of queries does not apply.

(ms)	Wikipedia	XMark w/o asso	XMark with asso
Index	4.15hr	6.64hr	12.72hr
PQAS	156	94	116
PQAC	-	67	93
PQR	-	-	217
NKQ	1646	1838	13468

much as that for the original data set. We note that with Lucene, the index look-up time increases linearly with the index size.

Second, when the number of answers was large, index-lookup time was more related to the number of answers. Although the sizes of the indexes for all different data sets were similar, the index-lookup time for ordinary NKQ queries dropped significantly when f was increased (so the number of answers was decreased) and showed the opposite trend for suffix queries. In particular, when $f = 0.8$, the number of answers for ordinary NKQ queries and for suffix NKQ queries were similar, and also similar to that for NKQ queries on the original data set. We indeed observed similar index-lookup time for these three cases. This observation implies that our index scales especially well when the number of returned answers is large.

Third, for suffix queries on the non-perturbed data set, the answers are empty. Index lookup on average took 25 milliseconds and was still efficient.

In the second set of experiments, we considered two XML data sets, each of size 10GB. The first data set is from the INEX Wikipedia collection [40] (with duplicates). It contained 1.4 million instances, each with only two attributes. The second data set was generated by XMark [117]. It contained 11.4 million instances, 76.2 million attribute values, and 58.2 million association instances. We indexed the XMark data set in two ways: one indexed only attribute values and one indexed associations in addition. We used these indexes to answer randomly generated queries with two clauses. We reported only index-lookup time,

since both the indexes of the 25MB personal data set and the indexes of the 10GB XML data sets were stored on disk and read by Lucene and so were comparable. We omitted comparison of the query-answering time as we extracted XML elements using JDOM, which has different efficiency from extracting RDF instances using Jena.

The three indexes varied in size: the Wikipedia index was 1.13GB, the XMark index without associations was 3.04GB, and the XMark index with associations was 4.08GB. As shown in Table 3.11, our indexing technique scales well: on average it took 123.8 milliseconds to look up the index for predicate queries, only 4.1 times as much as for the 250MB data. We also observed that although indexing associations took about twice as much time as indexing only attributes, the increase in keyword-lookup time was not significant (except for neighborhood keyword queries, where considering associations considerably increased the number of returned instances); however, it significantly sped up query answering in the presence of association clauses.

3.5 Related Work

The two bodies of work most close to ours are indexing XML and on keyword queries in relational databases.

There have been many indexing algorithms proposed for answering XML queries. They can be categorized into three classes: indexing on structure, indexing on value, and indexing on both. The first class (*e.g.*, [62, 99, 33, 86, 27, 84, 71]) considers supporting schema-driven queries, such as "list all book authors", and does not index text values. The second class (*e.g.*, [5, 21, 31, 77, 133, 110, 141]) is mainly oriented to XML Twig queries [21]. It indexes text values and at the same time encodes parent-child and ancestor-descendant relationships by numbering the XML elements appropriately. The encoding methods are tuned for tree models and would not apply in our context where associations between instances form a graph.

The third class combines indexes on structure and on text. We have already compared our approach to that of [37] (in Section 3.3). Kaushik et al. [85] and Chen et al. [28] proposed building multiple indexes to capture different aspects of structural information and values of XML data. If we adapt their indexing methods for our context, to answer a query we

need to visit several indexes in sequence, looking up an index for each result returned from the previous index. As we show in Section 3.4.2, this process can be quite time-consuming. ViST (Virtual Suffix Tree), proposed in [132], encodes both XML documents and XML queries as suffix sequences and answers the queries by suffix matching. This strategy is again more suitable for tree models and falls short in our context.

Our approach is different from the ones described above in that it does not rely on any specific data model, and it uses one *single* index to capture both structure information and text values. In this way, our method is more oriented to keyword search, can more easily explore associations between data items, and can more efficiently answer keyword queries with simple structure specifications.

Several works have considered keyword queries on relational databases. The DISCOVER [73], DBXplorer [4], and BANKS [15] systems return minimal join-networks that contain all the keywords given in a query. These approaches require building the join-network at run-time and so query answering can be expensive. Queries of the form "SEARCH {instance-type} NEAR {keywords}" are proposed in [61, 25], where the distances between elements are precomputed and indexed, so the index can be quite large and hence costly. Su and Widom [122] proposed indexing *virtual documents*, generated by joining database tuples through foreign-keys. This method does not distinguish tuples that contain the keywords and tuples that only join with tuples containing the keywords, and does not distinguish through which type of foreign keys (associations) a join happens. Similar ideas were considered in the context of XML data, returning the least common ancestors (LCA) for the elements that contain the given keywords [139, 74].

In addition, SphereSearch [66] and Kite [116] studied search across heterogeneous data by first conducting data transformation or integration. In contrast, we take a "data-coexistence" approach and index heterogeneous data even if they are only loosely coupled.

Finally, the Information Retrieval Community has recently proposed faceted search [112], which searches webpages by characteristics. The predicate queries we propose are different from faceted queries in that they also allow specifications of associations between the data items.

3.6 Discussion

We now discuss limitations and several extensions to our indexing work.

Scalability: In our indexing mechanism, whereas considering associations in indexing can significantly improve the efficiency in answering predicate queries with associations and answering neighborhood keyword queries, it also increases the size of the index and in turn increases index-lookup time. Our experiments show that when the number of associations is linear in the number of instances, which is true for most real-world data sets, our indexing method scales well and the increase in index-lookup time is paid off at query answering. However, when the number of associations is non-linear (*e.g.*, square) in the number of instances, increasing the number of instances can quickly blow up the index and the benefit brought by indexing associations can diminish. Thus, our indexing method does not scale well for data sets that contain fully-connected association networks.

Heterogeneity: Whereas our index has started incorporating heterogeneity by indexing hierarchies and synonyms, there are still several forms of heterogeneity that are not captured by our index.

First, our current method can handle term heterogeneity by indexing synonyms. However, in many applications it is hard to establish exact schema mappings and be certain about attribute or association synonyms. Instead, it is more likely we consider two properties are synonyms with only a certain probability (*e.g.*, the probability that permanent-address and location are synonyms is 80%). We want our index to be able to encode this probability.

Second, it is often the case that multiple text values are semantically close. For example, “sublet’ is similar to “short-term rental”, and “polished pewter” is close to “metallic silver”. We want our index to correlate such values so query answering can take all into consideration. The challenge is that some of the values are phrases rather than single keywords.

Third, as discussed in Section 2.6, in reference reconciliation it is possible that we generate results with probabilities, such as saying instances i_a and i_b refer to the same real-world entity with probability 80%. We want our index to incorporate this probability too.

One initial thought of handling such heterogeneity in the index is to multiply the occur-

rence counts with certain probability number. For example, if we believe keywords k_1 and k_2 are similar with probability 70%, and k_1 occurs 10 times in the attributes of instance ins, then we can index ins on k_2 with occurrence count 10*70%=7. As another example, if we believe i_a and i_b refer to the same real-world entity with probability 80%, and keyword k occurs 10 times in the attributes of i_a, then we can index i_b on K with occurrence count 10*80%=8. However, how to handle non-integer occurrence counts and how to avoid blowing up the size of the index require further study.

3.7 Summary

We described a novel indexing method that is designed to support flexible querying over dataspaces. The querying mechanism allows users to specify structure when they can, but also to fall back on keywords otherwise. Answers to keyword queries also include objects associated with the ones that contain the keywords. Our methods extend inverted lists to capture structure when it is present, including attributes of instances, relationships between instances, synonyms on schema elements, and hierarchies of schema elements. We validated our techniques with a set of experiments and showed that incorporating structure into inverted lists can considerably speed up query answering.

Chapter 4

RESOLVING QUERY-LEVEL HETEROGENEITY II: ANSWERING STRUCTURED QUERIES ON UNSTRUCTURED DATA

In a dataspace the data vary from unstructured data (*e.g.*, documents and webpages), semi-structured data (*e.g.*, XML data and RDF data), to structured data (*e.g.*, relational databases), and well-defined mappings may not exist between disparate schemas. On the other hand, as we have explained in Chapter 3, a dataspace system should allow a spectrum of search strategies, ranging from simple keyword search to expressive but sophisticated querying (*e.g.*, SQL queries and XQuery queries). No matter which type of queries users pose, they would like the queries to apply to *all* the content in the dataspace, and retrieve information from both structured and unstructured data. Resolving heterogeneity at the query level requires providing a uniform search interface to the user such that they can search the database without being aware of the heterogeneity of the underlying data.

As shown in Figure 4.1, querying each kind of data in isolation has been the main subject of study for the fields of Databases and Information Retrieval. Recently the database community has studied the problem of answering keyword queries on structured data such as relational data or XML data [73, 4, 15, 139, 74]. The only combination that has not been fully explored is answering structured queries on unstructured data. Information extraction techniques [42, 23, 67] attempt to extract structure from unstructured data such that structured queries can be applied. However, such techniques rely on the *existence* of some underlying structure, so are limited especially in heterogeneous environments.

This chapter explores an approach in which we carefully construct a keyword query from a given structured query, and submit the query to the underlying engine (*e.g.*, a web-search engine) for querying unstructured data. We start by formally defining the problem and introducing our approach in Section 4.1. Section 4.2 and 4.3 describe our algorithm in selecting keywords from a given query. Section 4.4 presents experimental results and

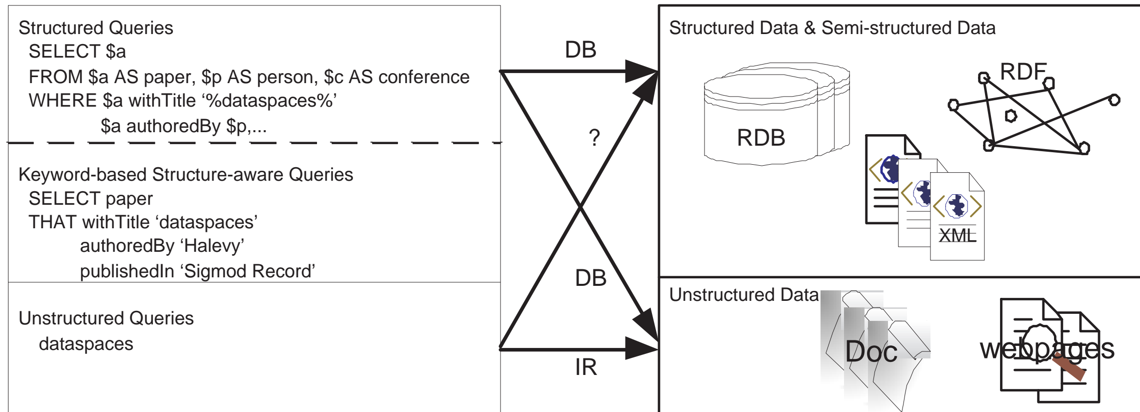


Figure 4.1: Searching and querying a dataspace. The left side of the graph shows the various models of queries and the right side shows the various models of data. The lines in the middle show the possible combination of data and query models and the communities that work on them.

Section 4.5 discusses related work. Finally, Section 4.6 discusses extensions to our approach and Section 4.7 summarizes this chapter.

4.1 Problem Definition and Overview of Our Approach

We study how to answer structured queries on unstructured data by extracting keywords from the given structured query, such that submitting a query with the keywords on the unstructured data repository obtains the most relevant answers. This technique brings two benefits. First, when the user poses a structured query on a dataspace, the system will be able to retrieve relevant information from unstructured data sources to supplement results from structured data. Second, if the schema the user uses to compose the query is different from those of the data sources and has not been seen beforehand, as an alternative to doing online schema mappings, the system can answer the extracted keyword query over the structured data sources by applying techniques for answering keyword search on databases. Our goal is to obtain reasonably precise answers even without domain knowledge, and improve the precision if knowledge of the schema and the data is available. We begin this section by formally defining our problem. We then give an example showing the challenges and introduce our approach.

4.1.1 Problem Definition

We define the *keyword extraction* problem as follows. Given a structured query (in SQL, XQuery, etc.), we extract a set of keywords from the query. These keywords are used to construct a keyword query that returns information potentially relevant to the structured query. A keyword search on a large volume of unstructured data often returns many results; thus, we measure the quality of the answers using top-k precision—the percentage of relevant results in the top-k results. We consider queries that do not contain disjunctions, comparison predicates (*e.g.*, $\neq$, $<$) or aggregation. Such queries are common in dataspace applications such as PIM.

The following example shows some of the challenges we face.

Example 4.1. *Consider a simple SQL query that asks for papers on Dataspaces published in 2005.*

```
SELECT title
FROM   paper
WHERE  title LIKE ‘%Dataspaces%’ AND year = ‘2005’
```

We have many options in keyword extraction. The following list gives a few:

1. *Use the whole query: “select title from paper where title LIKE ‘dataspaces’ and year = ‘2005’ ”.*

2. *Use the terms in the query excluding SQL syntactic symbols (*e.g.*, select, from, where): “paper title +dataspaces year +2005”. (Most search engines adopt the keyword-search syntax that requires the keyword following a “+” sign to occur in the returned documents or webpages.)*

3. *Use only the text values: “+dataspaces +2005”.*

4. *Use a subset of terms in the query: “+dataspaces +2005 paper title”.*

5. *Use another subset of terms in the query: “+dataspaces +2005 paper”.*

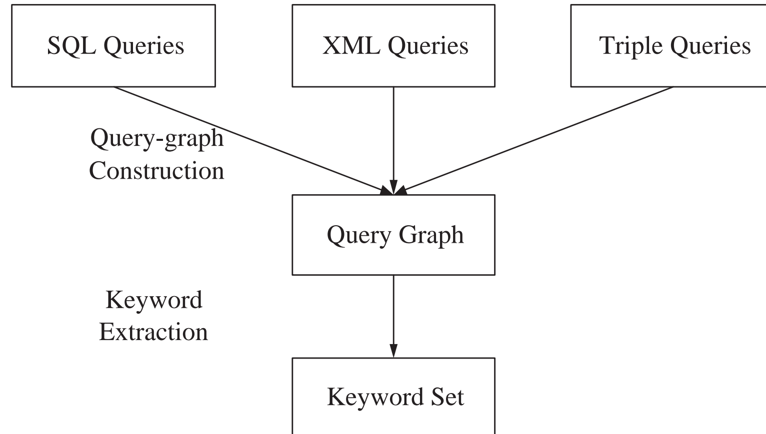


Figure 4.2: Framework of our approach to keyword extraction.

A human would most probably choose the last keyword set, which best summarizes the objects we are looking for. Indeed, at the time of the experiment, Google, Yahoo, and Windows Live all obtained the best results on the last keyword set (Google obtained 0.6 top-10 precision), and the top hits all mentioned the dataspaces paper authored by Franklin et al. in 2005 (two of the search engines returned exactly the paper as the first hit). In contrast, for the first two keyword sets, none of the three search engines returned relevant webpages in the top-10 results. For the third and fourth keyword sets, although some of the top-10 results were relevant, the top-10 precision was quite low (Google obtained 0.2 top-10 precision on both keyword sets). □

To explain the above results, let us consider the possible effects of a keyword. On the one hand, it may narrow down the search space by requiring the returned documents to contain the keyword. On the other hand, it may distract the search engine by bringing in irrelevant documents or webpages that by chance contain the keyword. Ideally, we should choose keywords that significantly narrow down the search space without distracting the search engine. Next, we introduce our algorithm for keyword extraction.

4.1.2 Overview

As depicted in Figure 4.2, the key element in our solution is to construct a *query graph* that captures the essence of the structured query, such as the object instances mentioned in the query, the attributes of these instances, and the associations between these instances. With this query graph, we can ignore syntactic aspects of the query and distinguish the query elements that convey different concepts. The keyword set is selected from the node and edge labels of the graph.

Our algorithm selects attribute values and schema elements that appear in the query and uses them as keywords to the search engine. Since these values or structure terms also appear as node labels and edge labels in the query graph, we also refer to them as *labels*. When selecting the labels, we wish to include only *necessary* ones, so keyword search returns exactly the query results and excludes irrelevant documents. We base our selection on the *informativeness* and *representativeness* of a label: the former measures the amount of information provided by the label, and the latter is the complement of the distraction that can be introduced by the label. Our keyword-extraction method is based on an important observation, that the informativeness of a label is dependent on the informativeness of the already selected labels; for example, once we have already selected keywords "dataspaces", "Alon Halevy", "Mike Franklin" and "2005", the keyword "paper" does not add much extra information. Given a query, we use its query graph to model the effect of a selected label on the informativeness of the rest of the labels. By applying a greedy algorithm, we select the labels with the highest informativeness and representativeness.

4.2 Constructing Query Graphs

To create a keyword query from a structured query Q, we first construct Q's query graph and then extract keywords from it. Intuitively, the query graph captures the essence of the query and already removes irrelevant syntactic symbols. In this section, we first define the query graph and then describe keyword extraction.

4.2.1 Query Graph

Recall that we model data from disparate data sources as a network of instances and associations (see Section 2.1.1); the same hold for queries. We can view a query as a subgraph pattern describing the queried instances with their attributes and directly or indirectly associated instances. We now formally define query graph.

Definition 4.2. *(Query Graph) A* query graph $G_Q = (V, E)$ *is an undirected graph describing the instances and associations mentioned in a query.*

- *Each* instance node *in V represents an instance mentioned in the query. The label of the node is the name of the instance class. If some attributes of the instance are queried, the node is in addition marked with "?".*

- *Each* association edge *in E represents an association mentioned in the query. The label of the edge is the association name. An association edge connects two instance nodes that are involved in the association.*

- *Each* value node *in V represents a ground value mentioned in the query. The label of the node is the value.*

- *Each* attribute edge *in E represents an attribute of an instance. The label of the edge is the attribute name. An attribute edge connects the value node and the node of the owner instance.*

- *Each* question node *in V represents an attribute being queried. The label of the node is "?".* □

As an example, Figure 4.3 shows the query graph for Example 4.1. The graph contains one instance node — paper, two value nodes — "Dataspaces" and "2005", and one question node. The nodes are connected by three attribute edges — two "title" edges and a "year" edge.

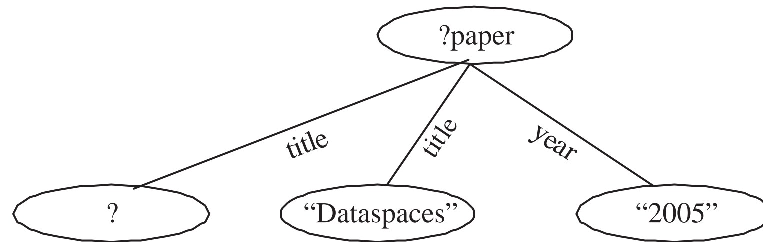


Figure 4.3: The query graph for the query in Example 4.1.

4.2.2 Query-graph Construction

Our goal is to construct a query graph from a structured query even if we do not know the schema of the query beforehand. We now describe the algorithm for SQL queries. The same intuition can be applied to XML queries and other structured queries.

Intuitively, attributes in a `SELECT`-clause correspond to question nodes. In the `WHERE`-clause, select predicates (of the form *attr* = *value* or *attr* LIKE *value*) correspond to value nodes, and join predicates correspond to association edges. The tricky part comes from tables in a `FROM`-clause: they can correspond to either instance nodes or association edges.

We construct a query graph for a SQL query in two steps. In the first step, we construct a preliminary graph by considering all tables in the query as object instances. In the second step, we compact the graph by updating certain instances to association edges. We now describe the algorithm in detail.

Step 1. Building preliminary graph: The first step builds the preliminary query graph as follows:

- For each table in the `FROM`-clause, there is an instance node labeled with the table name.

- For each attribute in the `SELECT`-clause, there is a question node connected with the corresponding instance node. The edge between the question node and the instance node is an attribute edge labeled with the attribute name.

- For each select predicate in the `WHERE`-clause, there is a value node labeled with the

given value, connected with the corresponding instance node. The edge between the value node and the instance node is an attribute edge labeled with the attribute name.

- For each join predicate in the `WHERE`-clause, there is an association edge connecting the two corresponding instance nodes, labeled with the two attribute names. We consider common attribute names such as "ID" and "key" as stopwords and omit them.

Step 2. Compacting the graph: The second step compacts the preliminary graph by removing unnecessary instance nodes. Suppose a sequence of instance nodes $N_0, \dots, N_t$ forms a chain; that is, each of $N_i, i \in [1, t-1]$, has only two neighbors N_{i-1} and N_{i+1}. We remove $N_1, \dots, N_{t-1}$ and connect N_0 and N_t with an association edge, labeled with all labels of the nodes and edges on the path from N_0 to N_t. Note that if there are multiple nodes with the same label, we either remove all of them or leave all as instance nodes to keep consistency.

The above algorithm for graph construction assumes no knowledge of the schema according to which the query is composed. However, in the presence of such knowledge, we can refine the compacting step. For example, if we know that all attributes of a table T are foreign keys to other tables, we consider T as describing associations and shortcut T's neighbors with an association edge.

Example 4.3. *Consider the following SQL query:*

```
SELECT p1.name
FROM   Paper AS a1, Paper AS a2, Cite, Person AS p1, Person AS p2,
       Author AS b1, Author AS b2
WHERE  b1.pid = p1.id AND b1.aid = a1.id
  AND  b2.pid = p2.id AND b2.aid = a2.id
  AND  Cite.pid = a1.id AND Cite.cid = a2.id
  AND  p2.name LIKE '%Halevy%' AND  a2.title LIKE '%Semex%'
```

In the first step, we construct a graph as shown in Figure 4.4(a). Note that the association edges do not have labels because all the involved attributes are various forms of "id". In the second step, we remove the cite *node and the two* author *nodes, obtaining the graph*

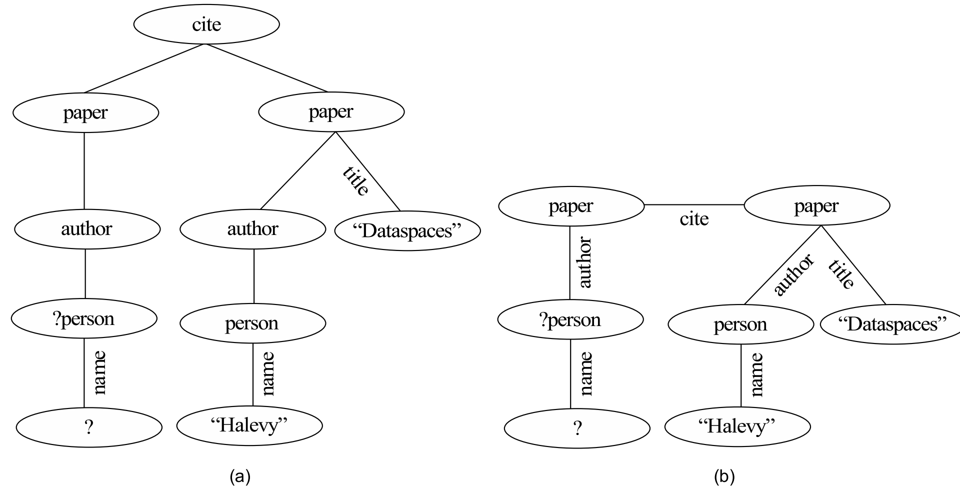


Figure 4.4: Constructing the query graph for the SQL query in Example 4.3: (a) the preliminary graph constructed in the first step; (b) the compact graph constructed in the second step

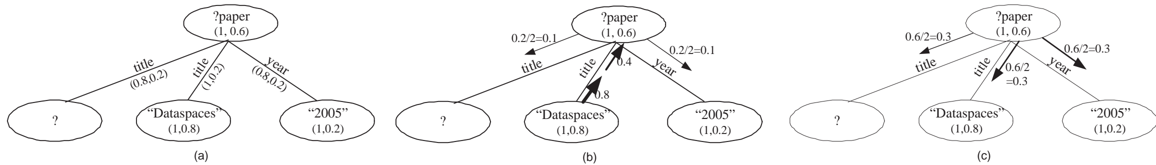


Figure 4.5: The query graph with i-scores and r-scores for the query in Example 4.1: (a) the initial (i-score, r-score) pairs; (b) the information flow representing the effect of the "Dataspaces" label; (c) the information flow representing the effect of the "Paper" label.

shown in Figure 4.4(b). We do not remove the paper *nodes because there exists a* paper *node with value-node neighbors.* □

4.3 Extracting Keywords

When we select keywords from the structured query, we wish to include only *necessary* keywords rather than adding all relevant ones. This principle is based on two observations. First, a keyword often introduces distraction, so unnecessary keywords often lower the search quality by returning irrelevant documents. Second, real-world documents are often crisp in describing instances. For example, rather than saying "a paper authored by a person

with name Halevy", we often say "Halevy's paper". Involving "authored by", "person" and "name" in the keyword set does not add much more information. We base our label selection on judging the informativeness and representativeness of labels. We first introduce measures for these two characteristics and then describe our algorithm.

4.3.1 Informativeness and representativeness

Intuitively, informativeness measures the amount of information provided by a label term. For example, attribute values are more informative than structure terms. Representativeness roughly corresponds to the probability that searching the given term returns documents or webpages in the queried domain. For example, the term "paper" is more representative than the term "title" for the publication domain. We use *i-score* to measure informativeness and *r-score* to measure representativeness. Given a node label or edge label l, we denote its i-score as i_l, and r-score as r_l. Both i_l and r_l range from 0 to 1. Note that the representativeness of label l is the complement of l's *distractiveness*, denoted as d_l, so $d_l = 1 - r_l$. Figure 4.5(a) shows the initial (i-score,r-score) pair for each label (we will discuss shortly how we initialize these scores).

We observe that the informativeness of a label also depends on the already selected keywords. For example, consider searching for a paper instance. The term "paper" is informative if we know nothing else about the paper, but its informativeness decreases if we know the paper is about "dataspaces", and further decreases if we also know the paper is by "Halevy". In other words, in a query graph, once we select a label for the keyword set, the informativeness of other labels is reduced.

We model the effect of a selected label s on the i-scores of other labels as an information flow, which has the following three characteristics:

- At the source node or edge, the flow has volume r_s. The reason is that the effect of s is limited to the search results that are related to the queried domain, and this percentage is r_s (by definition).

- The information flow first goes to the neighbor edges or nodes (not including the one from which the flow comes). If s is a label of an instance node, the flow value is

divided among the neighbor edges. Specifically, if n is the number of different labels of the neighbor edges, then the flow volume on each edge is r_s/n. The division follows the intuition that the more distinct edges are there, the more information each edge label provides even in presence of the s label, and thus the less effect s has on these labels.

- After a flow reaches a label, its volume decreases by half. It then continues flowing to the nodes or edges at the next hop and is divided again, until reaching value nodes or question nodes. In other words, s's effect dwindles exponentially in the number of hops. Note that the flow is only affected by the r-score of the source node, but not the r-scores of other nodes that it reaches.

When we add a new label to the keyword set, we compute the effect of the label on the rest of the labels and update their i-scores. Once a keyword set is fixed, the i-scores of the rest of the labels are fixed, independent of the order we select the keywords. Figure 4.6 gives the formal algorithm for i-score update.

Example 4.4. *Consider the query graph in Figure 4.5(a). Figure 4.5(b) shows the effect of the value label "Dataspaces" on the i-scores of the rest of the nodes, and Figure 4.5(c) shows the effect of the instance label "Paper". Note that in (c) we divide 0.6 by 2 rather than by 3, because the three edges are labeled by only two distinct labels.* □

4.3.2 Selecting labels

When we select node or edge labels, we wish to choose those that provide more information than distraction; that is, $i > d = 1 - r$, so $i + r > 1$. We select labels in a greedy fashion: in each step we choose the label with the highest $i + r$ and terminate the process when there are no more labels with $i + r > 1$. Specifically, we proceed in three steps.

1. We choose all labels of value nodes. After adding each label to the keyword set, we update the i-scores of the rest of the nodes.

2. If there are labels satisfying $i + r > 1$, we choose the one with the largest $i + r$. We add the label to the keyword set and update the i-scores of the rest of the nodes.

procedure $\textsc{UpdateIScore}(\mathcal{G}, S, I, r)$

//$\mathcal{G}$ is the input query graph;

//S is the node or edge whose label is just added to the keyword set;

//I is the array of i-scores for nodes or edges in $\mathcal{G}$; r is the r-score of S's label;

$queue = \{S\}$; // $queue$ contains all nodes or edges we need to consider;

$src[S] = null$; // src records the flow source for each node or edge;

$vol[S] = r * 2$; // vol records the volume of the flow for each node or edge;

while ($queue \neq \emptyset$)

$T = pop(queue)$;

if (T is an edge) $out = 1$;

else $out = \#$(distinct labels for T's neighbor edges excluding $src(T)$);

for each (T's neighbor node or edge L in $\mathcal{G}$)

if ($L \neq src(T)$)

$src[L] = T$;

$vol[L] = vol[T]/out/2$;

$I[L]$-=$vol[L]$;

if ($L \notin queue$) push($queue, L$);

Figure 4.6: Algorithm for i-score update.

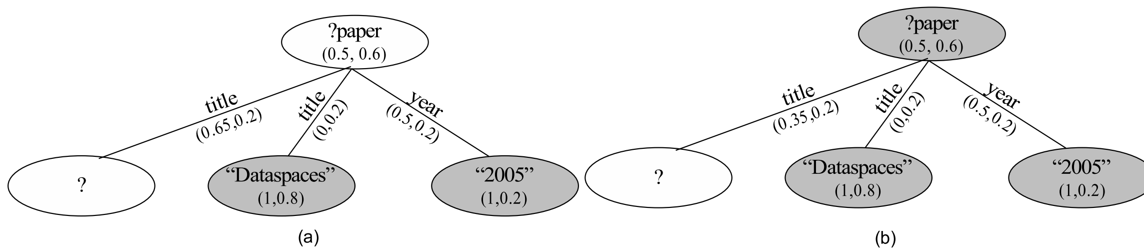


Figure 4.7: Extracting keywords from query graph in Figure 4.5(a): (a) the i-scores of the labels after selecting the labels "Dataspaces" and "2005"; (b) the i-scores of the labels after selecting the label "Paper".

procedure LABELSELECTION($\mathcal{G}, I, R$) return K
//$\mathcal{G}$ is the input query graph; I is the array of i-scores for labels in $\mathcal{G}$;
//R is the array of r-scores for labels in $\mathcal{G}$; K is the selected keyword set;
  $K = \emptyset$;
  for each (value node V in $\mathcal{G}$)
    Insert V's label to K;
    UpdateIScore($\mathcal{G}, V, I, R[V]$);
  while (true)
    Select the node or edge S whose label has the maximal $(I(S) + R(S))$;
    if $(I(S) + R(S) <= 1)$ **break;**
    Insert S's label to K;
    UPDATEISCORE($\mathcal{G}, S, I, R[S]$);
  return K;

Figure 4.8: Algorithm for label selection.

3. We iterate step 2 until no more labels satisfy $i + r > 1$.

Figure 4.8 gives the algorithm for label selection.

Example 4.5. *Consider the query graph in Figure 4.5(a). We select labels in two steps. In the first step, we select the labels of all value nodes, "Dataspaces" and "2005". The updated i-scores are shown in Figure 4.7(a). We then select label* Paper*, and the updated i-scores are shown in Figure 4.7(b). After this step no more labels satisfy the condition $i + r > 1$ so the algorithm terminates. The result keyword set is thus "Dataspaces 2005 paper".* □

4.3.3 Initializing i-scores and r-scores

We now discuss how to initialize the i-scores and r-scores. When we have no domain knowledge, we assign default values for different types of labels. We observe the web data for the representativeness of different types of nodes and assign r-scores accordingly. For

i-scores, we consider values and the class name of the queried instance as more informative and set the i-scores to 1, and consider other labels less informative. We will discuss the default score setting in our experiments in Section 4.4.

There are several ways to obtain more meaningful r-scores in the presence of domain knowledge. Here we suggest a few. The first method is to do keyword search on the labels. Specifically, for a label l, we search l using the unstructured data set on which we will perform keyword search. We manually examine the top-k (*e.g.*, $k = 10$) results and count how many are related to the queried domain. The percentage λ is considered as the r-score for the l label.

Another approach is to do Naive-Bayes learning on a corpus of schemas and structured data in the spirit of [93]. As an example, we discuss how to compute the r-scores of instance names. We divide the schemas into a set of domains, each containing a set of schemas. Suppose the corpus contains domains $D_1, \ldots, D_l$, schemas $S_1, \ldots, S_m$, and class names $C_1, \ldots, C_n$. We say $S_j \in D_i$ if the schema S_j belongs to the domain D_i, and we say $C_k \in S_j$ if the class name C_k occurs in the schema S_j. We now apply Naive Bayes learning to calculate the probability that the label C of an instance node represents a class in the queried domain D:

$$
\begin{aligned}
P(D|C) &= \frac{P(C|D) \cdot P(D)}{P(C)} \\
&= \frac{\frac{|\{S_j|S_j \in D, C \in S_j\}|}{|\{S_j|S_j \in D\}|} \cdot \frac{|\{S_j|S_j \in D\}|}{m}}{\sum_{i=1}^{l} \left(\frac{|\{S_j|S_j \in D_i, C \in S_j\}|}{|\{S_j|S_j \in D_i\}|} \cdot \frac{|\{S_j|S_j \in D_i\}|}{m} \right)} \\
&= \frac{|\{S_j|S_j \in D, C \in S_j\}|}{\sum_{i=1}^{l} |\{S_j|S_j \in D_i, C \in S_j\}|} \\
&= \frac{|\{S_j|S_j \in D, C \in S_j\}|}{\sum_{i=1}^{m} |\{S_j|C \in S_j\}|}
\end{aligned}
$$

The value of $P(D|C)$ can be considered as the r-score of the C label. Similarly, we can compute the r-scores for attribute names and association names. Finally, given an attribute a, to decide the r-score of its value labels, we randomly sample a number of values of the a attribute from the structured data and calculate the probability that the value belongs to the queried domain D. We use the average probability as the r-score.

Note that the second approach learns the scores from the corpus, and so performs well only if the corpus and the unstructured data follow the same pattern. Although this training phase is expensive, it is a one-time process and can significantly improve search performance.

4.4 Experimental Evaluation

This section describes a set of experiments that begin to validate our keyword-extraction algorithm. Our goal is to show that our algorithm performs well even without domain knowledge and that search quality improves when domain knowledge exists.

4.4.1 Experimental Setup

We used the web data as the unstructured data in our experiment. We chose web data for two reasons. First, web data are about anything, so for any queries we ask, we can expect a large number of relevant webpages. Second, we can submit the extracted keyword queries to the Google search engine, which typically provides high-quality search results for keyword queries.

We composed structured queries over six different schemas, selected from the UW XML repository [129] and the Niagara XML repository [102], including movie, geography, company profiles, bibliography, DBLP, and car profiles. These schemas vary in complexity, such as the number of elements and attributes, and the number of children of each element.

When we selected queries, we varied two parameters in the selected queries: *#values* and *length*. The former is the number of attribute values in the query, indicating the amount of value information given by the query. The latter is the longest path from a queried instance (the instance whose attributes are queried) to other instances in the query graph, corresponding to the complexity of the structure presented in the query. Finally, we randomly selected text values from the XML data for our queries. After generating the keyword set from the input queries, we used the Google Web API to search the web.

We measured the quality of our extracted keywords by top-k precision, which computes the percentage of the top k hits that provide information relevant to the query. We analyzed the results using top-2 and top-10 precision.

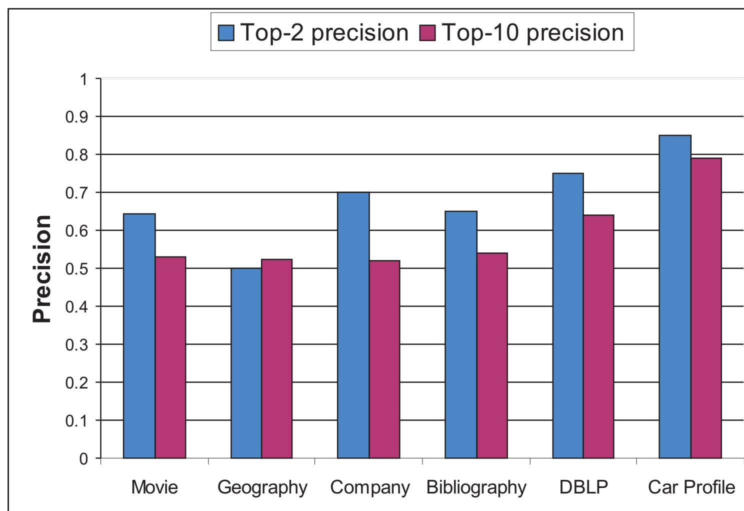


Figure 4.9: Top-2 and top-10 precision for queries over different schemas without applying domain knowledge.

We set the default values for i-scores and r-scores as follows (we used the same setting for all domains).

- **i-scores:** 1 for value labels and labels of queried instances, and 0.8 for other labels.
- **r-scores:** 0.8 for text-value labels and labels of associations between instances of the same type, 0.6 for instance labels, 0.4 for association labels, 0.2 for attribute labels, and 0 for number-value labels.

4.4.2 Experimental Results

We validated our algorithm on queries over the six schemas. Figure 4.9 shows the query-answering accuracy. We observed that our algorithm performed well in all domains. With our default settings for i-scores and r-scores, the top-2 and top-10 precisions in different domains were similar. The average top-2 precision was 0.68 and the average top-10 precision was 0.59.

We next compared QueryGraph with several other approaches that select terms directly from the query.

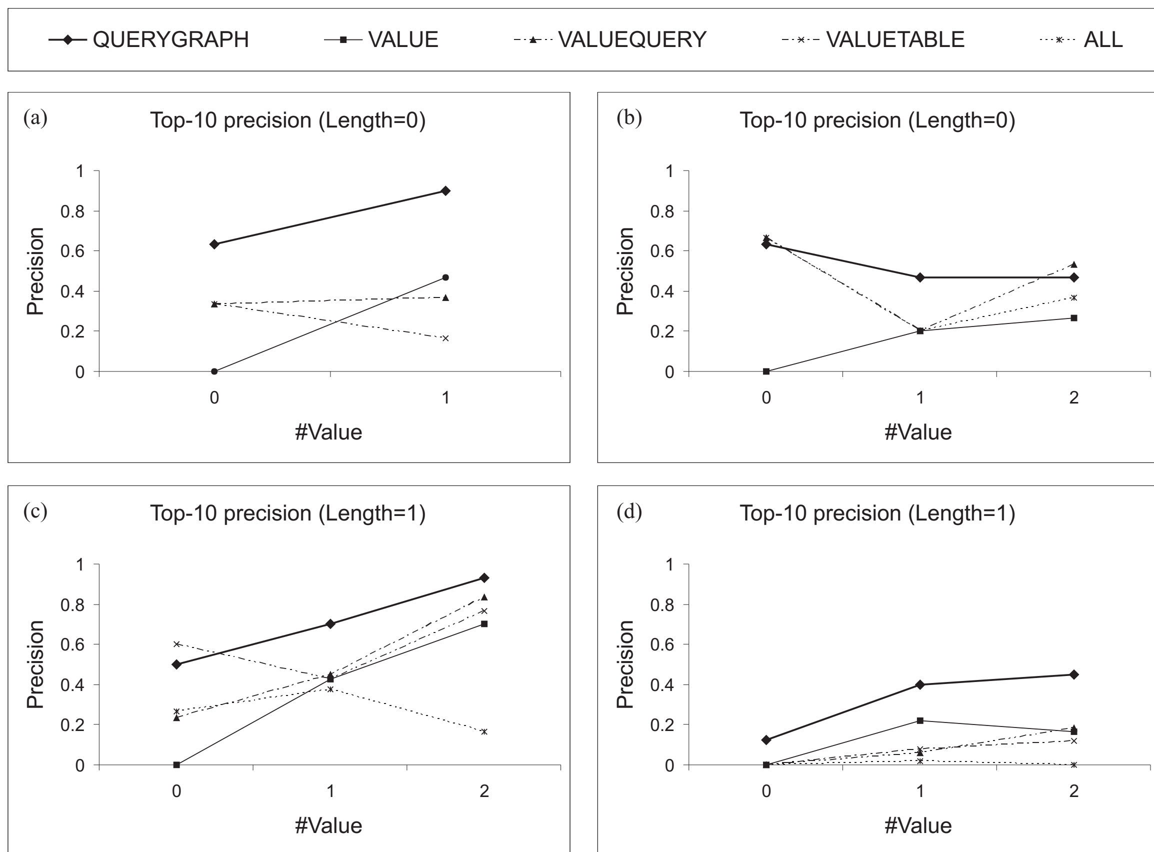


Figure 4.10: Top-10 precision for queries with length 0 in (a) the movie domain and (b) the geography domain, and with length 1 in (c) the movie domain, and (d) the geography domain. In (a) and (b) the ValueTable line and the QueryGraph line overlap, as the two methods extracted the same keywords.

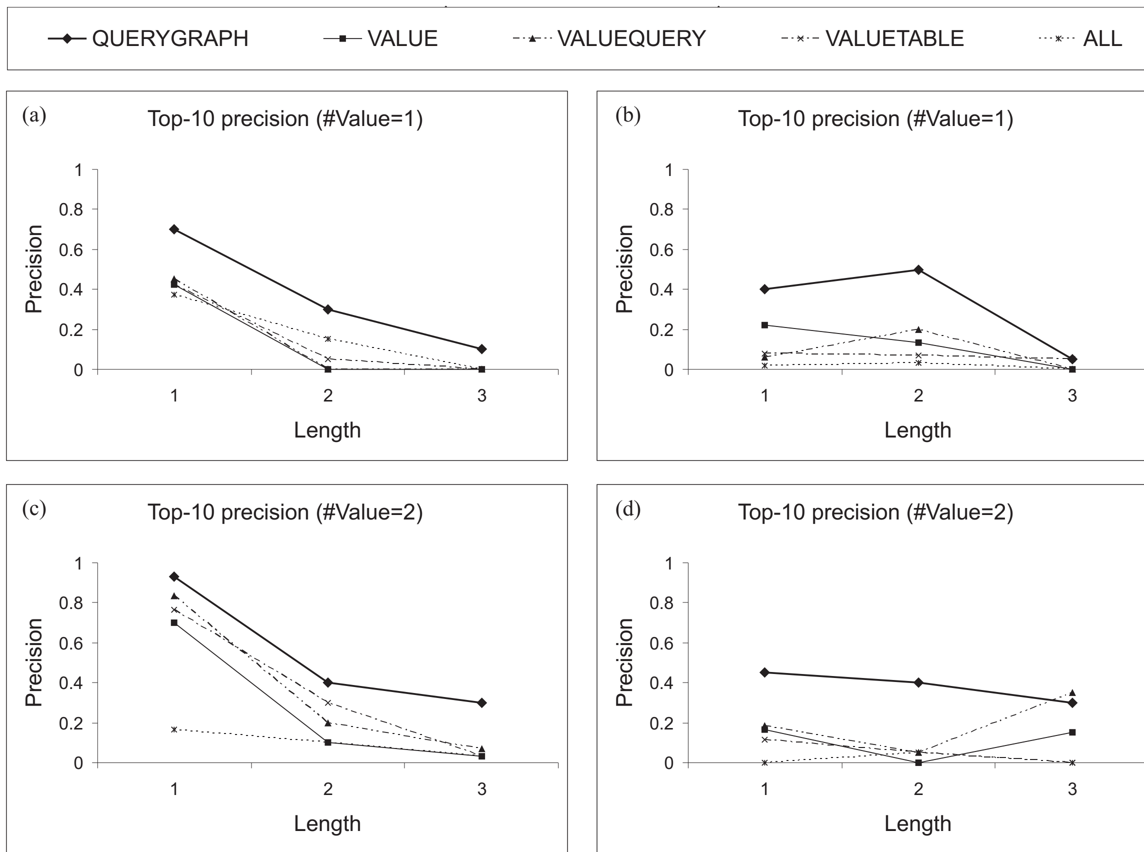


Figure 4.11: Top-10 precision of queries with one attribute value in (a) the movie domain and (b) the geography domain, and with two attribute values in (c) the movie domain and (d) the geography domain.

- All: Include all terms except syntactic symbols.
- Value: Include only attribute values.
- ValueQuery: Include attribute values and all table and attribute names in the `SELECT`-clause.
- ValueTable: Include attribute values and all table names in the `FROM`-clause.

We report the results on two domains: movie and geography. We observed similar trends on other domains.

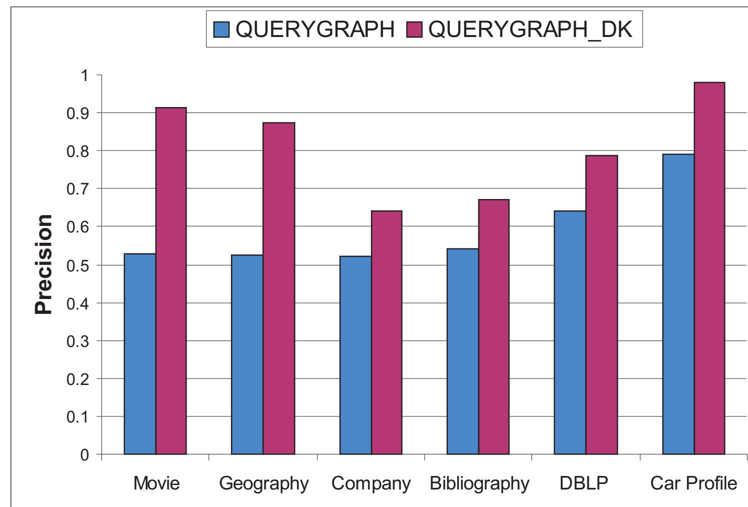


Figure 4.12: Top-10 precision for queries over different schemas without applying domain knowledge (QueryGraph) and with applying domain knowledge (QueryGraph_DK).

Varying the number of values: We first consider the effect of #values on keyword extraction. We considered queries with length 0 or 1 and varied #values from 0 to 2 when it applies. Figure 4.10 shows the top-10 precision.

We had two observations. First, in most cases QueryGraph obtained higher top-10 precision than the other approaches. It shows that including appropriate structure terms obtained much better results than searching only the text values. Second, when the number of attribute values increases, most approaches obtained better search results, but All performed even worse because it includes distractive keywords.

Varying query length: We now examine the effect of structure complexity on search performance. We considered queries with 1 or 2 attribute values, and varied the length from 1 to 3. Figure 4.11 shows the results. We observed that our algorithm again beat other methods in most cases. As the length grew, the top-10 precision dropped. This is not a surprise as complex query structure complicates the meaning of the query.

4.4.3 Applying Domain Knowledge

We finally examined how the domain knowledge helps in keyword extraction. For each structure term occurring in one of the six schemas, we searched for it on the web, examined how many top-10 results were relevant to the domain of the data set that the term belongs to, and set the r-score of the term accordingly. When we set the r-scores in this way, the average top-2 precision was 0.92 and the average top-10 precision was 0.81. Figure 4.12 shows a comparison of top-10 precisions with and without domain knowledge on various domains. The top-10 precisions were increased by 39% on average. It shows that our algorithm can further improve the search quality by applying domain knowledge.

4.5 Related Work

The Database community has recently considered how to answer keyword queries on relational data [73, 4, 15] and XML data [139, 74]. In this chapter, we consider the reverse direction, answering structured queries on unstructured data.

There are two bodies of research related to our work: the information-extraction approaches and the query-transformation approaches. The former approach extracts information from unstructured data and answers queries on the extracted data. Most information-extraction work [42, 56, 119, 120, 121, 95, 50, 12] uses supervised learning, which is hard to scale to data that involve a large number of domains and apply to the case where the query schema is unknown beforehand. Recently, extracting structure from the web was proposed in [50, 23, 67]. However, the extraction is typically restricted to certain domains, and no evidence has shown that the technique can smoothly scale to data in a large number of domains.

To the best of our knowledge, there is only one work, SCORE [113], that considers transforming structured queries into keyword search. SCORE extracts keywords from query results on structured data and uses them to submit keyword queries that retrieve supplementary information. Our approach extracts keywords from the query itself. It is generic in that we aim to provide reasonable results even without the presence of structured data and domain knowledge; however, the technique used in SCORE can serve as a

supplement to our approach.

4.6 Discussion

Although our experimental results already show that our algorithm obtains good results in various domains, there are multiple extensions we can make to our algorithm.

Improving search quality: Currently we obtained an average 0.68 top-2 precision and an 0.59 top-10 precision without domain knowledge, and an average 0.92 top-2 precision and an 0.81 top-10 precision by applying domain knowledge. The results are not 100% accurate partly because our algorithm may not be able to generate the best query, and partly because answering a keyword query on unstructured data is not 100% accurate either.

One possible improvement can be obtained by studying how to learn domain knowledge from existing structured and unstructured data, and how to apply domain knowledge to achieve better search results. There are also other ways in which we can improve the keyword query we generate. So far we extract keywords from the structured query. Ideally, we should be able to use the keywords that can generate the best results, even if the keywords do not occur in the structured query itself. There are two approaches to include keywords outside the structured query for improving the search results.

First, we can refine our extracted keyword set by considering the schema or maybe even a corpus of schemas. For example, we can replace an extracted keyword with a more domain-specific keyword in the corpus, such as replacing “paper” with “publication”; we can also add keywords selected from the corpus to further narrow down the search space. Second, we can use existing structured data to supplement the selected keyword set with values that are relevant to the given query, as proposed in SCORE [113].

Query-graph construction: In our algorithm the query graph plays a key role in keyword extraction; however, we construct the query graph from the structured query based on a set of heuristics and the resulting graph may not be accurate in capturing the real instances and associations described in the query. We would like to study the effect of inaccurate query graphs on the quality of the extracted keyword query and do experiments to quantify the robustness of our algorithm in terms of the accuracy of the query graph.

Ranking: Our current mechanism can answer structured queries on both structured and unstructured data. However, the results from structured and unstructured data sources are ranked separately. We would like to develop methods for ranking answers that are obtained from structured and unstructured data sources all together.

Comparing results of structured and unstructured queries: One interesting question one would ask is the follows: for a structured query and an unstructured query that express the same information needs, can we obtain better results on structured queries, which typically require more efforts for composing the query? Answering this question requires comparison of results on both structured and unstructured data. Our algorithm is mainly related to comparison on unstructured data. To obtain fair comparison results, we should experiment on structured and unstructured queries that are independently specified. Our experimental setting, described in Section 4.4, fixed the structured queries and generated unstructured queries by applying our algorithm, thus cannot serve as a comparison. Also, we cannot conduct comparison by fixing keyword queries, such as keyword queries over the TREC benchmark [127], and then guessing the corresponding structured queries for evaluation, because those guessed queries make an implicit assumption on the underlying query schema, which can highly influence the query-answering accuracy. A possible approach is to select from the query log structured and unstructured queries that ask for the same desired instances, and compare the query-answering results.

4.7 Summary

In this chapter we described an approach for extracting keyword queries from structured queries. The extracted keyword queries can be posed over a collection of unstructured data in order to obtain additional data that may be relevant to the structured query. The ability to widen queries in this way is an important capability in querying dataspaces that include heterogeneous collections of structured and unstructured data. Our experimental results show that our algorithm obtains good results for answering structured queries on unstructured data in various domains.

Chapter 5

RESOLVING SCHEMA-LEVEL HETEROGENEITY: PROBABILISTIC SCHEMA MAPPING

The key to resolving heterogeneity at the schema level (see Figure 5.1) is to specify schema mappings between data sources. These mappings describe the relationship between the contents of the different sources and are used to reformulate a query posed over one source (or a mediated schema) into queries over the sources that are deemed relevant. However, in a dataspace system we may not be able to provide all the schema mappings up front. This can be because the heterogeneity is at a large scale, because the users are not skilled enough, or because the data sources are evolving over time. Thus, providing best-effort querying even if we have only inaccurate mappings is crucial in building a dataspace system.

In this chapter, we propose *probabilistic schema mappings*, analyze their formal foundations, and study query answering in their presence. This chapter begins by illustrating *probabilistic schema mapping* with an example and presenting our results (Section 5.1). Then, Section 5.2 formally defines probabilistic schema mapping. Section 5.3 studies how to answer queries in their presence and Section 5.4 considers the effect of representations of probabilistic mappings on query answering. Section 5.5 discusses the extensions to more powerful mapping languages. Finally, Section 5.6 discusses related work, Section 5.7 discusses possible extensions, and Section 5.8 summarizes this chapter.

5.1 Overview of Our Results

Traditional data integration systems require generating schema mappings between heterogeneous data sources up front. Semi-automatic schema-mapping tools are often employed in such systems to generate candidate mappings. Refining the candidate mappings into a precise mapping requires database expertise and domain knowledge, and can be quite labor intensive. Thus, it would be beneficial if we can answer queries even in the presence of im-

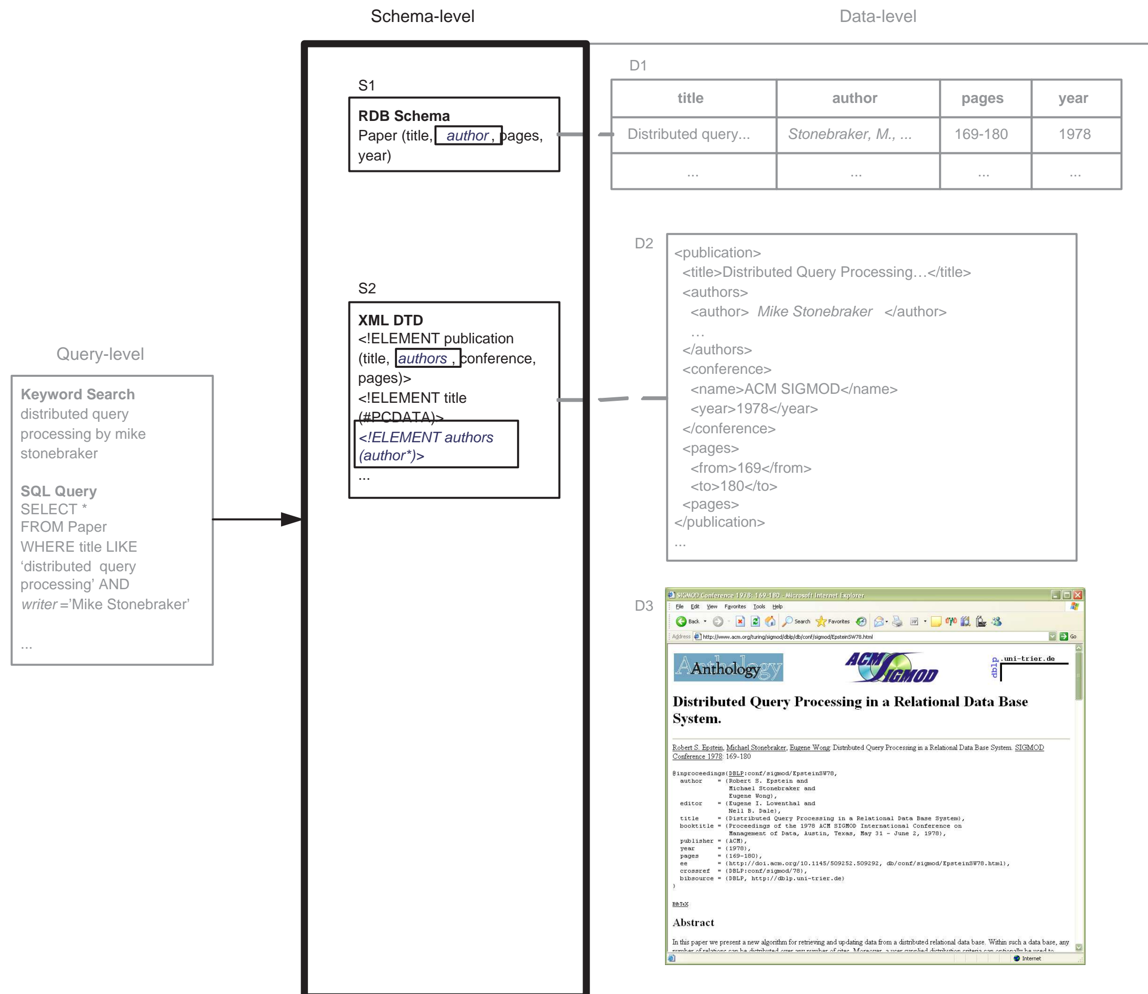


Figure 5.1: Heterogeneity at the schema level in a dataspace.

	Possible Mapping	Probability
$m_1 =$	{(pname, name), (email-addr, email), (current-addr, mailing-addr), (permanent-addr, home-address)}	0.5
$m_2 =$	{(pname, name), (email-addr, email), (permanent-addr, mailing-addr), (current-addr, home-address)}	0.4
$m_3 =$	{(pname, name), (email-addr, mailing-addr), (current-addr, home-addr)}	0.1

(a)

*p*name	*e*mail-addr	*c*urrent-addr	*p*ermanent-addr
Alice	alice@	Mountain View	Sunnyvale
Bob	bob@	Sunnyvale	Sunnyvale

(b)

Tuple	Prob
('Sunnyvale')	0.9
('Mountain View')	0.5
('alice@')	0.1
('bob@')	0.1

(c)

Figure 5.2: Example 5.1: (a) a probabilistic schema mapping between S and T; (b) a source instance D_S; (c) the answers of Q over D_S with respect to the probabilistic mapping.

precise mappings. For this purpose, we propose probabilistic schema mappings, with which we can answer a query by returning a set of tuples where each tuple is associated with a probability indicating how likely the tuple is an answer. Before the formal discussion, we illustrate the main ideas with an example.

Example 5.1. *Consider a data source S, which describes a person by her email address, current address, and permanent address, and a data target T, which describes a person by her name, email, mailing address, home address and office address:*

```
S=(pname, email-addr, current-addr, permanent-addr)
```

```
T=(name, email, mailing-addr, home-addr, office-addr)
```

A semi-automatic schema-mapping tool may generate three candidate mappings between S and T, assigning each a probability. Whereas the three mappings all map pname *to* name*, they map other attributes in the source and the target differently. Figure 5.2(a) describes the three mappings using sets of attribute correspondences. For example, mapping m_1 maps* pname *to* name*,* email-addr *to* email*,* current-addr *to* mailing-addr*, and* permanent-addr *to* home-addr*. Because of the uncertainty of which mapping is correct, we consider all of these mappings in query answering.*

Suppose the system receives a query Q on the target schema, asking for people's mailing addresses:

```
Q: SELECT mailing-addr FROM T
```

Using the possible mappings, we can reformulate Q into different queries:

```
Q1: SELECT current-addr FROM S
Q2: SELECT permanent-addr FROM S
Q3: SELECT email-addr FROM S
```

*To return all possible answers, the system sends Q_1, Q_2 and Q_3 to the data source, and then computes the probability of each returned tuple. Suppose the data source contains a table D_S as shown in Figure 5.2(b). The system will retrieve four answer tuples, each with a probability, as shown in Figure 5.2(c). If the data source supports aggregation queries (*e.g.*,* `SUM`*), the system can also generate a single aggregation query based on Q_1, Q_2 and Q_3 to compute the probability of each returned tuple directly, and send the query to the data source.* □

Formally, we define a probabilistic schema mapping as a set of possible (ordinary) mappings between a source schema and a target schema, where each possible mapping has an associated probability (Section 5.2). We begin by considering a simple class of mappings, where each mapping describes a set of correspondences between the attributes of a source table and the attributes of a target table. We introduce two possible semantics of probabilistic mappings. In the first, called *by-table* semantics, we assume there exists a single

correct mapping between the source and the target, but we don't know which one it is. In the second, called *by-tuple* semantics, the correct mapping may depend on the particular tuple in the source to which it is applied. In both cases, the semantics of query answers are a generalization of certain answers [2] for data integration systems.

We first study query answering with respect to probabilistic mappings (Section 5.3). We consider select-project-join queries, a core set of SQL queries, and we consider probabilistic mappings in both by-table and by-tuple semantics. We show that the data complexity of answering queries in the presence of probabilistic mappings is in PTIME for by-table semantics and #P-complete for by-tuple semantics. In addition, we identify a large subclass of real-world queries for which we can still obtain all the by-tuple answers in PTIME.

Next, we study whether we can compress the representations of probabilistic mappings to improve query answering (Section 5.4). Since a probabilistic schema mapping essentially enumerates a probability distribution by listing every combination of events in the probability space, its size can be quite large. In practice, we can often encode the same probability distribution much more concisely. We identify two concise representations of probabilistic mappings for which query answering can be done in PTIME in the size of the mapping. We also examine the possibility of representing a probabilistic mapping as a Bayes Net, but show that query answering may still be exponential in the size of a Bayes Net representation of a mapping.

Finally, we consider several more powerful mapping languages, such as complex mappings, where the correspondences are between sets of attributes, and conditional mappings, where the mapping is conditioned on a property of the tuple to which it is applied (Section 5.5). We show that our complexity results on query answering carry over to probabilistic mappings for these powerful mapping languages.

5.2 Definition

In this section we formally define the semantics of probabilistic schema mappings and the query answering problems we consider. Our discussion is in the context of the relational data model. A *schema* contains a finite set of relations. Each relation contains a finite set of *attributes* and is denoted by $R = \langle r_1, \ldots, r_n \rangle$. An *instance* D_R of R is a finite set of *tuples*,

where each tuple associates a value with each attribute in the schema.

We consider select-project-join (SPJ) queries in SQL. Note that answering such queries is in PTIME in the size of the data.

5.2.1 Schema Mappings

We begin by reviewing non-probabilistic schema mappings. The goal of a schema mapping is to specify the semantic relationships between a *source schema* and a *target schema*. We refer to the source schema as $\bar{S}$, and a relation in $\bar{S}$ as $S = \langle s_1, \ldots, s_m \rangle$. Similarly, we refer to the target schema as $\bar{T}$, and a relation in $\bar{T}$ as $T = \langle t_1, \ldots, t_n \rangle$.

The common formalism for schema mappings, GLAV, is based on expressions of the form

$$m : \forall \mathbf{x}(\phi(\mathbf{x}) \rightarrow \exists \mathbf{y} \psi(\mathbf{x}, \mathbf{y})).$$

In the expression, ϕ is the body of a conjunctive query over $\bar{S}$ and ψ is the body of a conjunctive query over $\bar{T}$. A pair of instances D_S and D_T *satisfies* a GLAV mapping m if for every assignment of $\mathbf{x}$ in D_S that satisfies ϕ there exists an assignment of $\mathbf{y}$ in D_T that satisfies ψ.

We consider a limited form of GLAV mappings where each side of the mapping involves only projection queries on a single table. These mappings have also been referred to as *schema matching* in the literature [109]. Specifically, we consider GLAV mappings where (1) ϕ (resp. ψ) is an atomic formula over S (resp. T), (2) the GLAV mapping does not include constants, and (3) each variable occurs at most once on each side of the mapping. We consider this class of mappings because they already expose many of the novel issues involved in probabilistic mappings and because they are quite common in practice. We also note that many of the concepts we define apply to a broader class of mappings, which we will discuss in detail in Section 5.5.

Given these restrictions, we can define our mappings in terms of *attribute correspondences*. An attribute correspondence is of the form $c_{ij} = (s_i, t_j)$, where s_i is a *source attribute* in the schema S and t_j is a *target attribute* in the schema T. Intuitively, c_{ij} specifies that there is a relationship between s_i and t_j. In practice, a correspondence also

involves a function that transforms the value of s_i to the value of t_j. For example, the correspondence (c-degree, temperature) can be specified as temperature=c-degree$*1.8+32$, describing a transformation from Celsius to Fahrenheit. These functions are irrelevant to our discussion, and therefore we omit them. Formally, we define relation mappings and schema mappings as follows.

Definition 5.2 (Schema Mapping)**.** *Let $\bar{S}$ and $\bar{T}$ be relational schemas. A* relation mapping *M is a triple (S,T,m), where S is a relation in $\bar{S}$, T is a relation in $\bar{T}$, and m is a set of attribute correspondences between S and T.*

When each source and target attribute occurs in at most one correspondence in m, we call M a one-to-one relation mapping.

A schema mapping *$\overline{M}$ is a set of one-to-one relation mappings between relations in $\bar{S}$ and in $\bar{T}$, where every relation in either $\bar{S}$ or $\bar{T}$ appears at most once.* □

Example 5.3. *Consider the mappings in Example 5.1. We can view m_1 as a GLAV mapping:*

$$\forall n,e,c,p(S(n,e,c,p) \rightarrow \exists o(T(n,e,c,p,o)))$$

The database in Figure 5.2(b) (repeated in Figure 5.3(a)) and the database in Figure 5.3(b) satisfy m_1. □

5.2.2 Probabilistic Schema Mappings

Intuitively, a probabilistic schema mapping describes a probability distribution of a set of *possible* schema mappings between a source schema and a target schema.

Definition 5.4 (Probabilistic Mapping)**.** *Let $\bar{S}$ and $\bar{T}$ be relational schemas. A* probabilistic mapping (p-mapping)*, pM, is a triple $(S,T,\mathbf{m})$, where $S \in \bar{S}$, $T \in \bar{T}$, and $\mathbf{m}$ is a set $\{(m_1, Pr(m_1)), \ldots, (m_l, Pr(m_l))\}$, such that*

- *for $i \in [1,l]$, m_i is a one-to-one mapping between S and T, and for every $i,j \in [1,l]$, $i \neq j \Rightarrow m_i \neq m_j$.*

- *$Pr(m_i) \in [0,1]$ and $\sum_{i=1}^{l} Pr(m_i) = 1$.*

*p*name	*e*mail-addr	*p*ermanent-addr	*c*urrent-addr
Alice	alice@	Mountain View	Sunnyvale
Bob	bob@	Sunnyvale	Sunnyvale

(a)

*n*ame	*e*mail	*m*ailing-addr	*h*ome-addr	*o*ffice-addr
Alice	alice@	Mountain View	Sunnyvale	office
Bob	bob@	Sunnyvale	Sunnyvale	office

(b)

*n*ame	*e*mail	*m*ailing-addr	*h*ome-addr	*o*ffice-addr
Alice	alice@	Sunnyvale	Mountain View	office
Bob	email	bob@	Sunnyvale	office

(c)

Tuple	Prob
('Sunnyvale')	0.9
('Mountain View')	0.5
('alice@')	0.1
('bob@')	0.1

(d)

Tuple	Prob
('Sunnyvale')	0.94
('Mountain View')	0.5
('alice@')	0.1
('bob@')	0.1

(e)

Figure 5.3: Example 5.11: (a) a source instance D_S; (b) a target instance that is by-table consistent with D_S; (c) a target instance that is by-tuple consistent with D_S; (d) $Q^{table}(D_S)$; (e) $Q^{tuple}(D_S)$.

A schema p-mapping, $\overline{pM}$*, is a set of p-mappings between relations in* $\bar{S}$ *and in* $\bar{T}$*, where every relation in either* $\bar{S}$ *or* $\bar{T}$ *appears in at most one p-mapping.* □

Note that we assume the possible mappings in a p-mapping are independent. We refer to a non-probabilistic mapping as an *ordinary mapping*. A schema p-mapping may contain both p-mappings and ordinary mappings. Example 5.1 shows a p-mapping (see Figure 5.2(a)) that contains three possible mappings.

5.2.3 Semantics of Probabilistic Mappings

Intuitively, a probabilistic schema mapping models the uncertainty about which of the mappings in pM is the correct one. When a schema matching system produces a set of candidate matches, there are two ways to interpret the uncertainty: (1) a single mapping in pM is the correct one and it applies to all the data in S, or (2) multiple mappings are correct and each suitable for a subset of tuples in S, though it is not known which mapping is the right one for a specific tuple. Example 5.1 illustrates the first interpretation. For the same example, the second interpretation is equally valid: some people may choose to use their current address as mailing address while others use their permanent address as mailing address; thus, for different tuples we may apply different mappings, so the correct mapping depends on the particular tuple.

This chapter analyzes query answering under both interpretations. We refer to the first interpretation as the *by-table* semantics and to the second one as the *by-tuple* semantics of probabilistic mappings. We are not trying to argue for one interpretation over the other. The needs of the application should dictate the appropriate semantics. Furthermore, our complexity results, which will show advantages to by-table semantics, should not be taken as an argument in the favor of by-table semantics.

We next define the semantics of p-mappings in detail and the definitions for schema p-mappings are the obvious extensions. The semantics of p-mappings is defined as a natural extension of that of ordinary mappings, which we review now. A mapping defines a relationship between instances of S and instances of T that are *consistent* with the mapping.

Definition 5.5 (Consistent Target Instance)**.** *Let* $M = (S, T, m)$ *be a relation mapping and*

D_S be an instance of S.

An instance D_T of T is said to be consistent with D_S and M, *if for each tuple $t_s \in D_S$, there exists a tuple $t_t \in D_T$, such that for every attribute correspondence $(a_s, a_t) \in m$, the value of a_s in t_s is the same as the value of a_t in t_t.* □

For a relation mapping M and a source instance D_S, there can be an infinite number of target instances that are consistent with D_S and M. We denote by $Tar_M(D_S)$ the set of all such target instances. The set of answers to a query Q is the intersection of the answers on all instances in $Tar_M(D_S)$. The following definition is from [2].

Definition 5.6 (Certain Answer)**.** *Let $M = (S, T, m)$ be a relation mapping. Let Q be a query over T and let D_S be an instance of S.*

A tuple t is said to be a certain answer of Q with respect to D_S and M, *if for every instance $D_T \in Tar_M(D_S)$, $t \in Q(D_T)$.* □

By-table semantics: We now generalize these notions to the probabilistic setting, beginning with the by-table semantics. Intuitively, a p-mapping pM describes a set of possible worlds, each with a possible mapping $m \in pM$. In by-table semantics, a source table can fall in one of the possible worlds; that is, the possible mapping associated with that possible world applies to the whole source table. Following this intuition, we define target instances that are *consistent with* the source instance.

Definition 5.7 (By-table Consistent Instance)**.** *Let $pM = (S, T, \mathbf{m})$ be a p-mapping and D_S be an instance of S.*

An instance D_T of T is said to be by-table consistent with D_S and pM, *if there exists a mapping $m \in \mathbf{m}$ such that D_S and D_T satisfy m.* □

Given a source instance D_S and a possible mapping $m \in \mathbf{m}$, there can be an infinite number of target instances that are consistent with D_S and m. We denote by $Tar_m(D_S)$ the set of all such instances.

In the probabilistic context, we assign a probability to every answer. Intuitively, we consider the certain answers with respect to each possible mapping in isolation. The prob-

ability of an answer t is the sum of the probabilities of the mappings for which t is deemed to be a certain answer. We define by-table answers as follows:

Definition 5.8 (By-table Answer)**.** *Let $pM = (S, T, \mathbf{m})$ be a p-mapping. Let Q be a query over T and let D_S be an instance of S.*

Let t be a tuple. Let $\bar{m}(t)$ be the subset of $\mathbf{m}$, such that for each $m \in \bar{m}(t)$ and for each $D_T \in Tar_m(D_S)$, $t \in Q(D_T)$.

Let $p = \sum_{m\in\bar{m}(t)} Pr(m)$. If $p > 0$, then we say (t, p) is a by-table answer of Q with respect to D_S and pM. □

By-tuple semantics: If we follow the possible-world notions, in by-tuple semantics, different tuples in a source table can fall in different possible worlds; that is, different possible mappings associated with those possible worlds can apply to the different source tuples.

Formally, the key difference in the definition of by-tuple semantics from that of by-table semantics is that a consistent target instance is defined by a mapping *sequence* that assigns a (possibly different) mapping in $\mathbf{m}$ to each tuple in D_S. (Without losing generality, in order to compare between such sequences, we assign some order to the tuples in the instance).

Definition 5.9 (By-tuple Consistent Instance)**.** *Let $pM = (S, T, \mathbf{m})$ be a p-mapping and let D_S be an instance of S with d tuples.*

An instance D_T of T is said to be by-tuple consistent with D_S and pM, *if there is a sequence $\langle m^1, \ldots, m^d \rangle$ such that for every $1 \le i \le d$,*

- *$m^i \in \mathbf{m}$, and*
- *for the ith tuple of D_S, t_i, there exists a target tuple $t'_i \in D_T$ such that for each attribute correspondence $(a_s, a_t) \in m^i$, the value of a_s in t_i is the same as the value of a_t in t'_i .* □

Given a mapping sequence $seq = \langle m^1, \ldots, m^d \rangle$, we denote by $Tar_{seq}(D_S)$ the set of all target instances that are consistent with D_S and seq. Note that if D_T is by-table consistent with D_S and m, then D_T is also by-tuple consistent with D_S and a mapping sequence in which each mapping is m.

We can think of every sequence of mappings $seq = \langle m^1, \ldots, m^d \rangle$ as a separate event whose probability is $Pr(seq) = \Pi_{i=1}^{d} Pr(m^i)$. (In Section 5.5 we relax this independence assumption and introduce *conditional mappings*.) If there are l mappings in pM, then there are l^d sequences of length d, and their probabilities add up to 1. We denote by $\mathbf{seq}_d(pM)$ the set of mapping sequences of length d generated from pM.

Definition 5.10 (By-tuple Answer)**.** *Let* $pM = (S, T, \mathbf{m})$ *be a p-mapping. Let* Q *be a query over* T *and* D_S *be an instance of* S *with* d *tuples.*

Let t *be a tuple. Let* $\overline{seq}(t)$ *be the subset of* $\mathbf{seq}_d(pM)$*, such that for each* $seq \in \overline{seq}(t)$ *and for each* $D_T \in Tar_{seq}(D_S)$*,* $t \in Q(D_T)$*.*

Let $p = \sum_{seq \in \overline{seq}(t)} Pr(seq)$*. If* $p > 0$*, we call* (t, p) *a* by-tuple answer of Q with respect to D_S and pM. □

The set of by-table answers for Q with respect to D_S is denoted by $Q^{table}(D_S)$ and the set of by-tuple answers for Q with respect to D_S is denoted by $Q^{tuple}(D_S)$.

Example 5.11. *Consider the p-mapping* pM*, the source instance* D_S*, and the query* Q *in the motivating example.*

In by-table semantics, Figure 5.3(b) shows a target instance that is consistent with D_S *(repeated in Figure 5.3(a)) and possible mapping* m_1*. Figure 5.3(d) shows the by-table answers of* Q *with respect to* D_S *and* pM*. As an example, for tuple* $t =$*('Sunnyvale'), we have* $\bar{m}(t) = \{m_1, m_2\}$*, so the possible tuple ('Sunnyvale', 0.9) is an answer.*

In by-tuple semantics, Figure 5.3(c) shows a target instance that is by-tuple consistent with D_S *and the mapping sequence* $< m_2, m_3 >$*. Figure 5.3(e) shows the by-tuple answers of* Q *with respect to* D_S *and* pM*.* □

5.3 Complexity of Query Answering

This section considers query answering in the presence of probabilistic mappings. We describe algorithms for query answering and study the complexity of query answering in terms of the size of the data (*data complexity*) and the size of the mapping (*mapping complexity*). We also consider cases in which we are not interested in the actual probability of an answer, just whether or not a tuple is a possible answer.

We show that when the schema is fixed, returning all by-table answers is in PTIME for both complexity measures, whereas returning all by-tuple answers in general is #P-complete with respect to the data complexity. Recall that #P is the complexity class of some hard counting problems (*e.g.*, counting the number of variable assignments that satisfy a boolean formula). It is believed that a #P-complete problem cannot be solved in polynomial time, unless $P = NP$. We show that computing the probabilities is the culprit here: even deciding the probability of a *single* answer tuple under by-tuple semantics is already #P-complete, whereas computing all by-tuple answers without returning the probabilities is in PTIME. Finally, we identify a large subclass of common queries where returning all by-tuple answers with their probabilities is still in PTIME.

5.3.1 By-table Query Answering

In the case of by-table semantics, answering queries is conceptually simple. Given a p-mapping $pM = (S, T, \mathbf{m})$ and an SPJ query Q, we can compute the certain answers of Q under each of the mappings $m \in \mathbf{m}$. We attach the probability $Pr(m)$ to every certain answer under m. If a tuple is an answer to Q under multiple mappings in $\mathbf{m}$, then we add up the probabilities of the different mappings.

Algorithm ByTable takes as input an SPJ query Q that mentions the relations $T_1, \ldots, T_l$ in the `FROM` clause. Assume that we have the p-mapping pM_i associated with the table T_i. The algorithm proceeds as follows.

Step 1: We generate the possible reformulations of Q (a reformulation query computes all certain answers when executed on the source data) by considering every combination of the form $(m^1, \ldots, m^l)$, where m^i is one of the possible mappings in pM_i. Denote the set of reformulations by $Q'_1, \ldots, Q'_k$. The probability of a reformulation $Q' = (m^1, \ldots, m^l)$ is $\Pi_{i=1}^{l} Pr(m^i)$.

Step 2: For each reformulation Q', retrieve each of the unique answers from the sources. For each answer obtained by $Q'_1 \cup \ldots \cup Q'_k$, its probability is computed by summing the probabilities of the Q''s in which it is returned.

Importantly, note that it is possible to express both steps as a SQL query with grouping

Tuple	Prob
('Sunnyvale')	0.94
('Mountain View')	0.5
('alice@')	0.1
('bob@')	0.1

(a)

Tuple	Prob
('Sunnyvale')	0.8
('Mountain View')	0.8

(b)

Figure 5.4: Example 5.14: (a) $Q_1^{tuple}(D)$ and (b) $Q_2^{tuple}(D)$.

and aggregation. Therefore, if the underlying sources support SQL, we can leverage their optimizations to compute the answers.

With our restricted form of schema mapping, the algorithm takes time polynomial in the size of the data and the mappings. We thus have the following complexity result. We give full proofs for results in this chapter in Appendix B.2.

Theorem 5.12. *Let $\overline{pM}$ be a schema p-mapping and let Q be an SPJ query.*

Answering Q with respect to $\overline{pM}$ in by-table semantics is in PTIME in the size of the data and the mapping. □

GLAV mappings: It is rather straightforward to extend the above results to arbitrary GLAV mappings. We define *general p-mappings* to be triples of the form $pGM = (\bar{S}, \bar{T}, \mathbf{gm})$, where $\mathbf{gm}$ is a set $\{(gm_i, Pr(gm_i)) \mid i \in [1, n]\}$, such that for each $i \in [1, n]$, gm_i is a general GLAV mapping. The definition of by-table semantics for such mappings is a simple generalization of Definition 5.8. The following result holds for general p-mappings.

Theorem 5.13. *Let pGM be a general p-mapping between a source schema $\bar{S}$ and a target schema $\bar{T}$. Let D_S be an instance of $\bar{S}$. Let Q be an SPJ query with only equality conditions over $\bar{T}$. The problem of computing $Q^{table}(D_S)$ with respect to pGM is in PTIME in the size of the data and the mapping.* □

5.3.2 By-tuple Query Answering

To extend the by-table query-answering strategy to by-tuple semantics, we would need to compute the certain answers for every *mapping sequence* generated by pM. However, the number of such mapping sequences is exponential in the size of the input data. The following example shows that for certain queries this exponential time complexity is not avoidable.

Example 5.14. *Suppose that in addition to the tables in Example 5.1, we also have* U(city) *in the source and* V(hightech) *in the target. The p-mapping for V contains two possible mappings:* ({(city, hightech)}, *.8) and* ($\emptyset$, *.2).*

Consider the following query Q, which decides if there are any people living in a high-tech city.

```
Q: SELECT ‘true’
   FROM T, V
   WHERE T.mailing-addr = V.hightech
```

One may conjecture that we can answer the query by first executing the following two sub-queries Q_1 and Q_2, then joining the answers of Q_1 and Q_2 and summing up the probabilities.

```
Q1: SELECT mailing-addr FROM T
Q2: SELECT hightech FROM V
```

Now consider the source instance D, where D_S is shown in Figure 5.2(a), and D_U has two tuples (‘Mountain View’) and (‘Sunnyvale’). Figure 5.4(a) and (b) show $Q_1^{tuple}(D)$ and $Q_2^{tuple}(D)$. If we join the results of Q_1 and Q_2, we obtain for the true *tuple the following probability:* $0.94 * 0.8 + 0.5 * 0.8 = 1.152$. *However, this is incorrect. By enumerating all consistent target tables, we in fact compute 0.864 as the probability. The reason for this error is that on some target instance that is by-tuple consistent with the source instance, the answers to both Q_1 and Q_2 contain tuple (‘Sunnyvale’) and tuple (‘Mountain View’). Thus, generating the tuple (‘Sunnyvale’) as an answer for both Q_1 and Q_2 and generating the tuple (‘Mountain View’) for both queries are not independent events, so simply adding up their probabilities leads to incorrect results.*

Indeed, we cannot answer Q by dividing it into several sub-queries and then joining the results in some way, but have to answer the query by enumerating all by-tuple consistent target instances. □

In fact, we show that in general, answering SPJ queries in by-tuple semantics with respect to schema p-mappings is hard.

Theorem 5.15. *Let Q be an SPJ query and let $\overline{pM}$ be a schema p-mapping. The problem of finding the probability for a by-tuple answer to Q with respect to $\overline{pM}$ is #P-complete with respect to data complexity and is in PTIME with respect to mapping complexity.* □

The lower bound in Theorem 5.15 is proved by reducing the problem of counting the number of variable assignments that satisfy a bipartite monotone 2DNF boolean formula to the problem of finding the answers to Q. We give the full proof of this theorem in Appendix B.1.

In fact, the reason for the high complexity is exactly that we are asking for the probability of the answer. The following theorem shows that if we only want to know the possible by-tuple answers, we can do so in polynomial time.

Theorem 5.16. *Given an SPJ query and a schema p-mapping, returning all by-tuple answers without probabilities is in PTIME with respect to data complexity.* □

The key to proving the PTIME complexity is that we can find all by-tuple answer tuples (without knowing the probability) by answering the query on the *mirror target* of the source data. Formally, let D_S be the source data and $\overline{pM}$ be the schema p-mapping. The mirror target of D_S with respect to $\overline{pM}$ is defined as follows. If R is not involved in any mapping, the mirror target contains R itself; if R is the target of $pM = (S, T, \mathbf{m}) \in \overline{pM}$, the mirror target contains a relation R' where for each source tuple t_S of S and each $m \in \mathbf{m}$, there is a tuple t_T in R' that (1) is consistent with t_S and m and contains null value for each attribute that is not involved in m, (2) contains an id column with the value of the id column in t_S (we assume the existence of identifier attribute id for S and in practice we can use S's key attributes in place of id), and (3) contains a mapping column with the identifier of m. Meanwhile, we slightly modify a query Q into a *mirror query* Q_m with respect to

$\overline{pM}$ as follows: Q_m is the same as Q except that for each relation R that is the target of a p-mapping in $\overline{pM}$ and occurs multiple times in Q's FROM clause, and for any of R's two aliases R_1 and R_2 in the FROM clause, Q' contains in addition the following predicates: (R_1.id <> R_2.id OR R_1.mapping=R_2.mapping).

Lemma 5.17. *Let $\overline{pM}$ be a schema p-mapping. Let Q be an SPJ query and Q_m be Q's mirror query with respect to $\overline{pM}$. Let D_S be the source database and D_T be the mirror target of D_S with respect to $\overline{pM}$.*

Then, $t \in Q^{tuple}(D_S)$ if and only if $t \in Q_m(D_T)$ and t does not contain null *value.* □

The size of the mirror target is polynomial in the size of the data and the p-mapping. The PTIME complexity bound follows from the fact that answering the mirror query on the mirror target takes only polynomial time.

GLAV mappings: Extending by-tuple semantics to arbitrary GLAV mappings is much trickier than by-table semantics. It would involve considering mapping sequences whose length is the product of the number of tuples in each source table, and the results are much less intuitive. Hence, we postpone by-tuple semantics to future work.

5.3.3 Two Restricted Cases

In this section we identify two restricted but common classes of queries for which by-tuple query answering takes polynomial time. We conjecture that they are the only cases where it is possible to answer a query in polynomial time.

In our discussion we refer to *subgoals* of a query. The subgoals are tables that occur in the FROM clause of a query. Hence, even if the same table occurs twice in the FROM clause, each occurrence is a different subgoal.

Queries with a single p-mapping subgoal

The first class of queries we consider are those that include only a single subgoal being the target of a p-mapping. Relations in the other subgoals are either involved in ordinary mappings or do not require a mapping. Hence, if we only have uncertainty with respect to one part of the domain, our queries will typically fall in this class. We call such queries *non-p-join queries*. The query Q in the motivating example is an example non-p-join query.

Definition 5.18 (non-p-join queries)**.** *Let $\overline{pM}$ be a schema p-mapping and let Q be an SPJ query.*

If at most one subgoal in the body of Q is the target of a p-mapping in $\overline{pM}$, then we say Q is a non-p-join query with respect to *$\overline{pM}$.* □

For a non-p-join query Q, the by-tuple answers of Q can be generated from the by-table answers of Q over a set of databases, each containing a single tuple in the source table. Specifically, let $pM = (S, T, \mathbf{m})$ be the single p-mapping whose target is a relation in Q, and let D_S be an instance of S with d tuples. Consider the set of *tuple databases* $\mathbf{T}(D_S) = \{D_1, \ldots, D_d\}$, where for each $i \in [1, d]$, D_i is an instance of S and contains only the i-th tuple in D_S. The following lemma shows that $Q^{tuple}(D_S)$ can be derived from $Q^{table}(D_1), \ldots, Q^{table}(D_d)$.

Lemma 5.19. *Let $\overline{pM}$ be a schema p-mapping between $\bar{S}$ and $\bar{T}$. Let Q be a non-p-join query over $\bar{T}$ and let D_S be an instance of $\bar{S}$. Let $(t, Pr(t))$ be a by-tuple answer with respect to D_S and $\overline{pM}$. Let $\bar{T}(t)$ be the subset of $\mathbf{T}(D_S)$ such that for each $D \in \bar{T}(t)$, $t \in Q^{table}(D)$. The following two conditions hold:*

1. *$\bar{T}(t) \neq \emptyset$;*

2. *$Pr(t) = 1 - \Pi_{D \in \bar{T}(t), (t,p) \in Q^{table}(D)}(1 - p)$.* □

In practice, answering the query for each tuple database can be expensive. We next describe Algorithm NonPJoin, which computes the answers for all tuple databases in one step. The key of the algorithm is to distinguish answers generated by different source tuples. To do this, we assume there is an identifier attribute id for the source relation whose values are concatenations of values of the key columns. We now describe the algorithm in detail.

Algorithm NonPJoin takes as input a non-p-join query Q, a schema p-mapping $\overline{pM}$, and a source instance D_S, and proceeds in three steps to compute all by-tuple answers.

Step 1: Rewrite Q to Q' such that it returns T.id in addition. Revise the p-mapping such that each possible mapping contains the correspondence between S.id and T.id.

Step 2: Invoke ByTable with Q', $\overline{pM}$ and D_S. Note that each generated result tuple contains the id column in addition to the attributes returned by Q.

Step 3: Project the answers returned in Step 2 on Q's returned attributes. Suppose projecting $t_1, \ldots, t_n$ obtains the answer tuple t, then the probability of t is $1 - \Pi_{i=1}^{n}(1 - Pr(t_i))$.

Example 5.20. *Consider rewriting Q in the motivating example, repeated as follows:*

```
Q: SELECT mailing-addr FROM T
```

Step 1 rewrites Q into query Q' by adding the id *column:*

```
Q': SELECT id, mailing-addr FROM T
```

In Step 2, ByTable *may generate the following SQL query to compute by-table answers for Q':*

```
Qa: SELECT id, mailing-addr, SUM(pr)
    FROM (
      SELECT DISTINCT id, current-addr AS mailing-addr, 0.5 AS pr
      FROM S
      UNION ALL
      SELECT DISTINCT id, permanent-addr AS mailing-addr, 0.4 AS pr
      FROM S
      UNION ALL
      SELECT DISTINCT id, email-addr AS mailing-addr, 0.1 AS pr
      FROM S)
    GROUP BY id, mailing-addr
```

Step 3 then generates the results using the following query.

```
Qu:  SELECT mailing-addr, NOR(pr) AS pr
     FROM Qa
     GROUP BY mailing-addr
```

where for a set of probabilities $pr_1, \ldots, pr_n$, NOR computes $1 - \Pi_{i=1}^{n} pr_i$. □

An analysis of Algorithm NonPJoin leads to the following complexity result for non-p-join queries.

Theorem 5.21. *Let $\overline{pM}$ be a schema p-mapping and let Q be a non-p-join query with respect to $\overline{pM}$.*

Answering Q with respect to $\overline{pM}$ in by-tuple semantics is in PTIME in the size of the data and the mapping. □

Projected p-join queries

We now show that query answering can be done in polynomial time for a class of queries, called *projected p-join queries*, that include multiple subgoals involved in p-mappings. In such a query, we say that a join predicate is a *p-join predicate* with respect to a schema p-mapping $\overline{pM}$, if at least one of the involved relations is the target of a p-mapping in $\overline{pM}$. We define projected p-join queries as follows.

Definition 5.22 (projected p-join query)**.** *Let $\overline{pM}$ be a schema p-mapping and Q be an SPJ query over the target of $\overline{pM}$. If the following conditions hold, we say Q is a* projected p-join query with respect to $\overline{pM}$*:*

- *at least two subgoals in the body of Q are targets of p-mappings in $\overline{pM}$.*
- *for every p-join predicate, the join attribute (or an equivalent attribute implied by the predicates in Q) is returned in the* `SELECT` *clause.* □

Example 5.23. *Consider the schema p-mapping in Example 5.14. A slight revision of Q, shown as follows, is a non-p-join query.*

```
Q': SELECT V.hightech
    FROM T, V
    WHERE T.mailing-addr = V.hightech
```

□

Note that in practice, when joining data from multiple tables in a data integration scenario, we typically project the join attributes, thereby leading to projected p-join queries.

The key to answering a projected-p-join query Q is to divide Q into multiple subqueries, each of which is a non-p-join query, and compute the answer to Q from the answers to the subqueries. We proceed by considering partitions of the subgoals in Q. We say that a partitioning $\bar{J}$ is a *refinement* of a partitioning $\bar{J}'$, denoted $\bar{J} \preceq \bar{J}'$, if for each partition $J \in \bar{J}$, there is a partition $J' \in \bar{J}'$, such that $J \subseteq J'$. We consider the following partitioning of Q, the generation of which will be described in detail in the algorithm.

Definition 5.24 (Maximal P-Join Partitioning)**.** *Let $\overline{pM}$ be a schema p-mapping. Let Q be an SPJ query and $\bar{J}$ be a partitioning of the subgoals in Q.*

We say that $\bar{J}$ is a p-join partitioning of Q*, if (1) each partition $J \in \bar{J}$ contains at most one subgoal that is the target of a p-mapping in $\overline{pM}$, and (2) if neither subgoal in a join predicate is involved in p-mappings in $\overline{pM}$, the two subgoals belong to the same partition.*

We say that $\bar{J}$ is a maximal p-join partitioning of Q*, if there does not exist a p-join partitioning $\bar{J}'$, such that $\bar{J} \preceq \bar{J}'$.* □

For each partition $J \in \bar{J}$, we can define a query Q_J as follows. The `FROM` clause includes the subgoals in J. The `SELECT` clause includes J's attributes that occur in (1) Q's `SELECT` clause or (2) Q's join predicates that join subgoals in J with subgoals in other partitions. The `WHERE` clause includes Q's predicates that contain only subgoals in J. When J is a partition in a maximal p-join partitioning of Q, we say that Q_J is a *p-join component* of Q.

The following is the main lemma underlying our algorithm. It shows that we can compute the answers of Q from the answers to its p-join components.

Lemma 5.25. *Let $\overline{pM}$ be a schema p-mapping. Let Q be a projected p-join query with respect to $\overline{pM}$ and let $\bar{J}$ be a maximal p-join partitioning of Q. Let $Q_{J1}, \ldots, Q_{Jn}$ be the p-join components of Q with respect to $\bar{J}$.*

For any instance D_S of the source schema of $\overline{pM}$ and result tuple $t \in Q^{tuple}(D_S)$, the following two conditions hold:

1. *For each $i \in [1, n]$, there exists a single tuple $t_i \in Q^{tuple}_{Ji}(D_S)$, such that $t_1, \ldots, t_n$ generate t when joined together.*

2. Let $t_1, \ldots, t_n$ be the above tuples. Then $Pr(t) = \Pi_{i=1}^{n} Pr(t_i)$. □

Lemma 5.25 leads naturally to the query-rewriting algorithm PROJECTEDPJOIN, which takes as input a projected-p-join query Q, a schema p-mapping $\overline{pM}$, and a source instance D_S, outputs all by-tuple answers, and proceeds in three steps.

Step 1: Generate maximum p-join partitions $J_1, \ldots, J_n$ as follows. First, initialize each partition to contain one subgoal in Q. Then, for each join predicate with subgoals S_1 and S_2 that are not involved in p-mappings in $\overline{pM}$, merge the partitions that S_1 and S_2 belong to. Finally, for each partition that contains no subgoal involved in $\overline{pM}$, merge it with another partition.

Step 2: For each p-join partition $J_i, i \in [1, n]$, generate the p-join component Q_{Ji} and invoke Algorithm NONPJOIN with Q_{Ji}, $\overline{pM}$ and D_S to compute answers for Q_{Ji}.

Step 3: Join the results of $Q_{J1}, \ldots, Q_{Jn}$. If an answer tuple t is obtained by joining $t_1, \ldots, t_n$, then the probability of t is computed by $\Pi_{i=1}^{n} Pr(t_i)$.

We illustrate the algorithm using the following example.

Example 5.26. *Consider query Q' in Example 5.23. Its two p-join components are Q_1 and Q_2 shown in Example 5.14. Suppose we compute Q_1 with query Q_u (shown in Example 5.20) and compute Q_2 with query Q'_u. We can compute by-tuple answers of Q' as follows:*

```
SELECT Qu'.hightech, Qu.pr*Qu'.pr
FROM Qu, Qu'
WHERE Qu.mailing-addr = Qu'.hightect
```

□

Since the number of p-join components is bounded by the number of subgoals in a query, and for each of them we invoke Algorithm NONPJOIN, query answering for projected p-join queries takes polynomial time.

Theorem 5.27. *Let $\overline{pM}$ be a schema p-mapping and let Q be a projected-p-join query with respect to $\overline{pM}$.*

Answering Q with respect to $\overline{pM}$ in by-tuple semantics is in PTIME in the size of the data and the mapping. □

Other SPJ queries

A natural question is whether the two classes of queries we have identified are the only ones for which query answering is in PTIME for by-tuple semantics. As Example 5.14 shows, if Q contains multiple subgoals that are involved in a schema p-mapping, but Q is not a projected-p-join query, then Condition 1 in Lemma 5.25 does not hold and query answering needs to proceed by enumerating all mapping sequences.

We believe that the complexity of the border case, where a query joins two relations involved in p-mappings but does not return the join attribute, is #P-hard, but currently it remains an open problem.

5.4 Representation of Probabilistic Mappings

Thus far, a p-mapping was represented by listing each of its possible mappings, and the complexity of query answering was polynomial in the size of that representation. Such a representation can be quite lengthy since it essentially enumerates a probability distribution by listing every combination of events in the probability space. Hence, an interesting question is whether there are more concise representations of p-mappings and whether our algorithms can leverage them.

We consider three representations that can reduce the size of the p-mapping exponentially. In Section 5.4.1 we consider a representation in which the attributes of the source and target tables are partitioned into groups and p-mappings are specified for each group separately. We show that query answering can be done in time polynomial in the size of the representation. In Section 5.4.2 we consider probabilistic correspondences, where we specify the marginal probability of each attribute correspondence. However, we show that such a representation can only be leveraged in limited cases. Finally, we consider Bayes Nets, the most common method for concisely representing probability distributions, in Section 5.4.3, and show that even though some p-mappings can be represented by them, query answering does not necessarily benefit from the representation.

Mapping	Prob
{(a,a'), (b,b'), (c,c')}	0.72
{(a,b'), (c,c')}	0.18
{(a,a'), (b,b')}	0.08
{(a,b')}	0.02

(a)

Mapping	Prob
{(a,a'), (b,b')}	0.8
{(a,b')}	0.2

(b)

Mapping	Prob
{(c,c')}	0.9
$\emptyset$	0.1

(c)

Figure 5.5: Example 5.29: the p-mapping in (a) is equivalent to the 2-group p-mapping in (b) and (c).

5.4.1 Group Probabilistic Mapping

In practice, the uncertainty we have about a p-mapping can often be represented as a few localized choices, especially when schema mappings are created by semi-automatic methods. To represent such p-mappings more concisely, we can partition the source and target attributes and specify p-mappings for each partition.

Definition 5.28 (Group P-Mapping). *An n-group p-mapping gpM is a triple $(S, T, \overline{pM})$, where*

- *S is a source relation schema and $S_1, \ldots, S_n$ is a set of disjoint subsets of attributes in S;*
- *T is a target relation schema and $T_1, \ldots, T_n$ is a set of disjoint subsets of attributes in T;*
- *$\overline{pM}$ is a set of p-mappings $\{pM_1, \ldots, pM_n\}$, where for each $1 \leq i \leq n$, pM_i is a p-mapping between S_i and T_i.* □

The semantics of an n-group p-mapping $gpM = (S, T, \overline{pM})$ is a p-mapping that includes the Cartesian product of the mappings in each of the pM_i's. The probability of the mapping composed of $m_1 \in pM_1, \ldots, m_n \in pM_n$ is $\Pi_{i=1}^{n} Pr(m_i)$.

Example 5.29. *Figure 5.5(a) shows p-mapping pM between the schemas $S(a, b, c)$ and $T(a', b', c')$. Figure 5.5(b) and (c) show two independent mappings that together form a 2-group p-mapping equivalent to pM.* □

Note that a group p-mapping can be considerably more compact than an equivalent p-mapping. Specifically, if each pM_i includes l_i mappings, then a group p-mapping can describe $\Pi_{i=1}^{n} l_i$ possible mappings with $\sum_{i=1}^{n} l_i$ sub-mappings. The important feature of n-group p-mappings is that query answering can be done in time polynomial in their size.

Theorem 5.30. *Let $\overline{gpM}$ be a schema group p-mapping and let Q be an SPJ query. The mapping complexity of answering Q with respect to $\overline{gpM}$ in both by-table semantics and by-tuple semantics is in PTIME.* □

Note that as n grows, fewer p-mappings can be represented with n-group p-mappings. Formally, suppose we denote by $\mathcal{M}_{ST}^{n}$ the set of all n-group p-mappings between S and T, then:

Proposition 5.31. *For each $n \geq 1$, $\mathcal{M}_{ST}^{n+1} \subset \mathcal{M}_{ST}^{n}$.* □

We typically expect that when possible, a mapping would be given as a group p-mapping. The following theorem shows that we can find the best group p-mapping for a given p-mapping in polynomial time.

Theorem 5.32. *Given a p-mapping pM, we can find in polynomial time in the size of pM the maximal n and an n-group p-mapping gpM, such that gpM is equivalent to pM.* □

5.4.2 Probabilistic Correspondences

The second representation we consider, *probabilistic correspondences*, represents a p-mapping with the marginal probabilities of attribute correspondences. This representation is the most compact one as its size is proportional to the product of the schema sizes of S and T.

Definition 5.33 (Probabilistic Correspondences)**.** *A* probabilistic correspondence mapping (p-correspondence) *is a triple $pC = (S, T, \mathbf{c})$, where $S = \langle s_1, \ldots, s_m \rangle$ is a source relation schema, $T = \langle t_1, \ldots, t_n \rangle$ is a target relation schema, and*

Mapping	Prob
{(a,a'), (b,b'), (c,c')}	0.8
{(a,b'), (c,c')}	0.1
{(a,b')}	0.1

(a)

Corr	Prob
{(a,a')}	0.8
{(a,b')}	0.2
{(b,b')}	0.8
{(c,c')}	0.9

(b)

Figure 5.6: Example 5.34: the p-mapping in (a) corresponds to the p-correspondence in (b).

- **c** *is a set* $\{(c_{ij}, Pr(c_{ij}))|i \in [1,m], j \in [1,n]\}$, *where* $c_{ij} = (s_i, t_j)$ *is an attribute correspondence, and* $Pr(c_{ij}) \in [0,1]$;
- *for each* $i \in [1,m], \sum_{j=1}^{n} Pr(c_{ij}) \leq 1$;
- *for each* $j \in [1,n], \sum_{i=1}^{m} Pr(c_{ij}) \leq 1$. □

Note that for a source attribute s_i, we allow $\sum_{j=1}^{n} Pr(c_{ij}) < 1$. This is because in some of the possible mappings, s_i may not be mapped to any target attribute. The same is true for target attributes.

From each p-mapping, we can infer a p-correspondence by calculating the marginal probabilities of each attribute correspondence. Specifically, for a p-mapping $pM = (S, T, \mathbf{m})$, we denote by $pC(pM)$ the p-correspondence where each marginal probability is computed as follows:

$$Pr(c_{ij}) = \sum_{c_{ij} \in m, m \in \mathbf{m}} Pr(m)$$

However, as the following example shows, the relationship between p-mappings and p-correspondences is many-to-one.

Example 5.34. *The p-correspondence in Figure 5.6(b) is the one computed for both the p-mapping in Figure 5.6(a) and the p-mapping in Figure 5.5(a).* □

Given the many-to-one relationship, the question is when it is possible to compute the correct answer to a query based only on the p-correspondence. That is, we are looking for a class of queries $\bar{Q}$, called *p-mapping independent queries*, such that for every $Q \in \bar{Q}$ and every database instance D_S, if $pC(pM_1) = pC(pM_2)$, then the answer of Q with respect to pM_1 and D_S is the same as the answer of Q with respect to pM_2 and D_S. Unfortunately, this property holds for a very restricted class of queries, defined as follows:

Definition 5.35 (Single-Attribute Query)**.** *Let $pC = (S, T, \mathbf{c})$ be a p-correspondence. An SPJ query Q is said to be a* single-attribute *query with respect to pC if T has one single attribute occurring in the* `SELECT` *and* `WHERE` *clauses of Q. This attribute of T is said to be a* critical attribute. □

Theorem 5.36. *Let $\overline{pC}$ be a schema p-correspondence, and Q be an SPJ query. Then, Q is p-mapping independent with respect to $\overline{pC}$ if and only if for each $pC \subseteq \overline{pC}$, Q is a single-attribute query with respect to pC.* □

Example 5.37. *Continuing with Example 5.34, consider the p-correspondence pC in Figure 5.6(b) and the following two queries Q_1 and Q_2. Query Q_1 is mapping independent with respect to pC, but Q_2 is not.*

```
Q1: SELECT T.a FROM T,U WHERE T.a=U.a'
Q2: SELECT T.a, T.c FROM T
```

□

Theorem 5.36 simplifies query answering for p-mapping independent queries. Wherever we needed to consider every possible mapping in previous algorithms, we consider only every attribute correspondence for the critical attribute.

Corollary 5.38. *Let $\overline{pC}$ be a schema p-correspondence, and Q be a p-mapping independent SPJ query with respect to $\overline{pC}$. The mapping complexity of answering Q with respect to $\overline{pC}$ in both by-table semantics and by-tuple semantics is in PTIME.* □

The result in Theorem 5.36 can be generalized to cases where we know the p-mapping is an n-group p-mapping. Specifically, as long as Q includes at most a single attribute in

each of the groups in the n-group p-mapping, query answering can still be done with the correspondence mapping. We omit the details of this generalization.

5.4.3 Bayes Nets

Bayes Nets are a powerful mechanism for concisely representing probability distributions and reasoning about probabilistic events [104]. The following example shows how Bayes Nets can be used in our context.

Example 5.39. *Consider two schemas* $S = (s_1, \ldots, s_n, s'_1, \ldots, s'_n)$ *and* $T = (t_1, \ldots, t_n)$. *Consider the p-mapping* $pM = (S, T, \mathbf{m})$, *which describes the following probability distribution: if* s_1 *maps to* t_1 *then it is more likely that* $\{s_2, \ldots, s_n\}$ *maps to* $\{t_2, \ldots, t_n\}$, *whereas if* s'_1 *maps to* t_1 *then it is more likely that* $\{s'_2, \ldots, s'_n\}$ *maps to* $\{t_2, \ldots, t_n\}$.

We can represent the p-mapping using a Bayes Net as follows. Let c *be an integer constant. Then,*

1. $Pr((s_1, t_1)) = Pr((s'_1, t_1)) = 1/2$;

2. *for each* $i \in [1, n]$, $Pr((s_i, t_i)|(s_1, t_1)) = 1 - \frac{1}{c}$ *and* $Pr((s'_i, t_i)|(s_1, t_1)) = \frac{1}{c}$;

3. *for each* $i \in [1, n]$, $Pr((s_i, t_i)|(s'_1, t_1)) = \frac{1}{c}$ *and* $Pr((s'_i, t_i)|(s'_1, t_1)) = 1 - \frac{1}{c}$.

Since the p-mapping contains 2^n *possible mappings, the original representation would take space* $O(2^n)$; *however, the Bayes-Net representation takes only space* $O(n)$. □

Although the Bayes-Net representation can reduce the size exponentially for some p-mappings, this conciseness may not help reduce the complexity of query answering. We formalize this result in the following theorem.

Theorem 5.40. *There exists a schema p-mapping* $\overline{pM}$ *and a query* Q, *such that answering* Q *with respect to* $\overline{pM}$ *in by-table semantics takes exponential time in the size of* $\overline{pM}$*'s Bayes-Net representation.* □

5.5 Broader Classes of Mappings

In this section we briefly show how our results can be extended to capture two common practical extensions to our mapping language.

Complex mappings: Complex mappings map a set of attributes in the source to a set of attributes in the target. For example, we can map the attribute address to the concatenation of street, city, and state.

Formally, a *set correspondence* between S and T is a relationship between a subset of attributes in S and a subset of attributes in T. Here, the function associated with the relationship specifies a single value for each of the target attributes given a value for each of the source attributes. Again, the actual functions are irrelevant to our discussion. A *complex mapping* is a triple (S, T, cm), where cm is a set of set correspondences, such that each attribute in S or T is involved in at most one set correspondence. A *probabilistic complex mapping* is of the form $pCM = \{(cm_i, Pr(cm_i)) \mid i \in [1, n]\}$, where $\sum_{i=1}^{n} Pr(cm_i) = 1$.

Theorem 5.41. *Let $\overline{pCM}$ be a schema probabilistic complex mapping between schemas $\bar{S}$ and $\bar{T}$. Let D_S be an instance of $\bar{S}$. Let Q be an SPJ query over $\bar{T}$. The data complexity and mapping complexity of computing $Q^{table}(D_S)$ with respect to $\overline{pCM}$ are PTIME. The data complexity of computing $Q^{tuple}(D_S)$ with respect to $\overline{pCM}$ is #P-complete. The mapping complexity of computing $Q^{tuple}(D_S)$ with respect to $\overline{pCM}$ is in PTIME.* □

Conditional mappings: In practice, our uncertainty is often conditioned. For example, we may want to state that daytime-phone maps to work-phone with probability 60% if age ≤ 65, and maps to home-phone with probability 90% if age > 65.

We define a *conditional p-mapping* as a set $cpM = \{(pM_1, C_1), \ldots, (pM_n, C_n)\}$, where $pM_1, \ldots, pM_n$ are p-mappings, and $C_1, \ldots, C_n$ are pairwise disjoint conditions. Intuitively, for each $i \in [1, n]$, pM_i describes the probability distribution of possible mappings when condition C_i holds. Conditional mappings make more sense for by-tuple semantics. The following theorem shows that our results carry over to such mappings.

Theorem 5.42. *Let $\overline{cpM}$ be a schema conditional p-mapping between $\bar{S}$ and $\bar{T}$. Let D_S be an instance of $\bar{S}$. Let Q be an SPJ query over $\bar{T}$. The problem of computing $Q^{tuple}(D_S)$*

with respect to $\overline{cpM}$ is in PTIME in the size of the mapping and #P-complete in the size of the data. □

5.6 Related Work

We are not aware of any previous work studying the semantics and properties of probabilistic schema mappings. Gal [58] used the top-K schema mappings obtained by a semi-automatic mapper to improve the precision of the top mapping, but did not address any of the issues we consider. Florescu et al. [54] were the first to advocate the use of probabilities in data integration. Their work used probabilities to model (1) a mediated schema with overlapping classes (*e.g.*, DatabasePapers and AIPapers), (2) source descriptions stating the probability of a tuple being present in a source, and (3) overlap between data sources. While these are important aspects of many domains and should be incorporated into a data integration system, our focus here is different. De Rougement and Vieilleribiere [39] considered approximate data exchange in that they relaxed the constraints on the target schema, which is a different approach from ours.

There has been a flurry of activity around probabilistic and uncertain databases lately [14, 118, 29, 7]. Our intention is that a data integration system will be based on a probabilistic data model, and we leverage concepts from that work as much as possible. We also believe that uncertainty and lineage are closely related, in the spirit of [14], and that relationship will play a key role in data integration. We leave exploring this topic to future work.

5.7 Discussion

We now discuss several extensions to our study of probabilistic schema mapping.

Top-k query answering: Answering queries in the presence of probabilistic schema mappings in by-tuple semantics is #-P complete, and so is quite expensive. In practice, users often want to see only the answers with the top-k probabilities and are satisfied even if the probabilities of these tuples are not returned. Rather than first computing all answers and then returning the top-k answer tuples, we can improve the efficiency by performing only the necessary query reformulations and executions at every step and halt when the top-k

answers are found.

Generating probabilities: To employ probabilistic mappings in resolving heterogeneity at the schema level, we must have a good method of generating probabilities for the mappings. This is possible as techniques for semi-automatic schema mapping are often based on Machine Learning techniques that at their core compute the confidence of correspondences they generate. However, such confidence is meant more as a ranking mechanism than true probabilities between candidates and is associated with attribute correspondences rather than candidate mappings. We plan to study how to generate from them probabilities for candidate mappings by pursuing maximum entropy.

Reasoning uncertainties: Another direction we would like to explore is to reason about the uncertainty on schema mappings between data sources and its effect on query answering. By analyzing the probabilities of the candidate mappings, we would like to find the critical parts (*i.e.*, attribute correspondences) where it is most beneficial to expand more resources (human or otherwise) to improve schema mapping.

Probabilistic data integration: One of our future goals is to build a data integration system that supports uncertainty about mappings, data extracted from sources, and the exact meaning of keyword queries. Studying the theoretical underpinning of probabilistic mappings is the first step towards building such a system. In addition, we need to extend the current work in the community on probabilistic databases [118] to study how to efficiently answer queries in the presence of uncertainties in schemas and in data, and study how to translate a keyword query into structured queries by exploiting evidence obtained from the existing data and users' search and querying patterns.

5.8 Summary

In this chapter we introduced probabilistic schema mappings, with which we are able to answer queries on heterogeneous data sources even if we have only a set of candidate mappings that may not be precise. We presented query answering algorithms for by-table and by-tuple semantics and studied the complexity of query answering. We also considered concise encoding of probabilistic mappings, with which we are able to improve the efficiency

of query answering. Finally, we extended our definition to more powerful schema mapping languages and showed the extensibility of our approach.

Chapter 6

AN APPLICATION: THE SEMEX PERSONAL INFORMATION MANAGEMENT SYSTEM

The explosion of information available in digital form has made search a hot research topic for the Information Management Community. Whereas most of the research on search is focused on the WWW, individual computer users have developed their own vast collections of data on their desktops, and these collections are in critical need of good search and querying tools. The problem is exacerbated by the proliferation of varied electronic devices (laptops, PDAs, cell phones) that are at our disposal, which often hold subsets or variations of our data. In fact, several recent venues have noted Personal Information Management (PIM) as an area of growing interest to the data management community [1, 87, 30, 79].

Personal information contains data created and managed by different applications, with extension to organizational data and even Web data that the user often accesses. Whereas personal data can be highly heterogeneous, typical users are not skilled enough to provide semantic mappings; thus, personal information management is an example scenario of dataspaces. To offer users a flexible platform for personal information management, we built the Semex System (short for SEMantic EXplorer). Semex provides a logical view of one's personal information and leverages this logical view for best-effort browsing, search and querying across disparate data sources.

This chapter begins by describing the browsing and search services provided by Semex. Section 6.2 describes the architecture of Semex. Finally, Section 6.3 discusses related work and Section 6.4 summarizes this chapter.

6.1 Browsing and Querying in Semex

The key idea of Semex is to provide a logical view of *meaningful* objects and the relations between them to support associative browsing of one's personal information. For example, a user can go from an email to the information on the sender, then browse the publications of

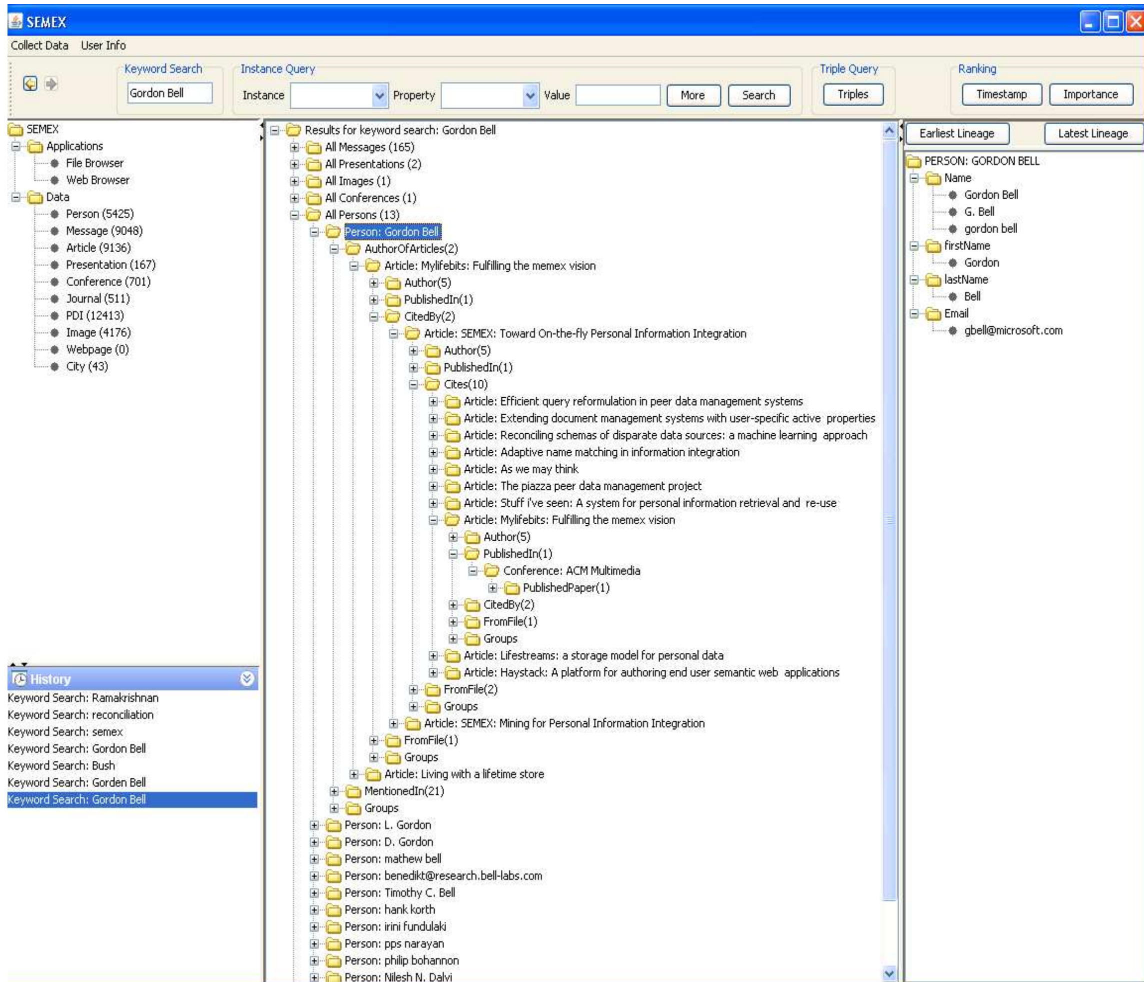


Figure 6.1: A sample screenshot of the SEMEX interface. The browsing trace in the middle pane answers the query elaborated in Example 6.1.

the sender, and then browse the cited papers. This logical view essentially is an *association network*, observing a default *domain model* (formally defined in Chapter 2) that describes the personal information domain. SEMEX creates such a view by *automatically* populating object and association instances from the unstructured information on one's desktop (*e.g.*, LATEX and BIBTEX files, Word documents, Powerpoint presentations, emails and contacts, and webpages in the web cache) and also disparate structured information (*e.g.*, spreadsheets, relational databases, and XML data).

The first and most obvious benefit of such an association network is the ability to *find* information in one's personal data. SEMEX offers its users an interface that combines intuitive browsing and a range of query facilities (see Figure 6.1). We next describe the browsing and search services provided by SEMEX in detail.

Browsing: SEMEX lists all classes of instances extracted from the personal data (see left pane of Figure 6.1). When the user poses a query (or selects one class from the list), SEMEX classifies the returned objects into their classes. The user can select a particular object instance to see detailed information, including its attribute values (see right pane of Figure 6.1) and its associated instances, which have been grouped by associations (see middle pane of Figure 6.1). The user can then browse the data by following association links, much like web browsing.

Keyword search: The easiest way to search the personal data is by keyword search. SEMEX supports neighborhood keyword search (the formal definition was given in Chapter 3), a search mechanism that is more intelligent than simple keyword search. For example, when searching for the keyword "schema matching", SEMEX returns not only the instances representing papers and presentations that contain "schema matching" in the text, but also instances representing people working on this area, conferences and journals that have published papers on this topic, etc. In addition, SEMEX goes to the Web and returns webpages that are relevant to the keyword search.

Predicate query: As a more complex search mechanism, the user can use the interface to compose a predicate query (the formal definition was given in Chapter 3), which combines keywords and structural requirements (see top middle of Figure 6.1). A predicate query

describes a desired instance by a set of predicates, each of which describes either an attribute value or an associated instance. For example, the user can specify a predicate query asking for Gordon Bell's MyLifeBits paper published in ACM Multimedia.

```
Paper (title ‘MyLifeBits’),
      (author ‘Gordon Bell’),
      (publishedIn ‘ACM Multimedia’)
```

Semex returns all instances in the association network that satisfy all or some of the predicates in the query. In addition, Semex searches unstructured data in the personal information space and searches the Web to return text documents and webpages that are relevant to the query.

Triple query: Semex also supports a more sophisticated query interface, through which the user can create *triple queries* (the interface is not shown in Figure 6.1 to avoid clutter). A triple query is a conjunctive query over triplets, each triplet describing an association between a pair of objects or an attribute of an object. It is more powerful than a predicate query in that it can describe a chain of associations, where the length of the chain is larger than one. Semex searches over personal data and the Web to answer triple queries. The following example triple query asks for a paper that is cited together with Gordon Bell's paper in one of the user's papers (suppose the user is named ‘Alice’):

```
select $a3
where  ($a1 cites $a2) AND
       ($a1 cites $a3) AND
       ($a1 author $p1) AND
       ($p1 name ‘Alice’) AND
       ($a2 author $p2) AND
       ($p2 name ‘Gordon Bell’)
```

In Chapter 3 we described the index that supports efficiently answering neighborhood keyword queries and predicate queries. The techniques we described in Chapter 4 can be used to answer predicate queries and triple queries on unstructured texts and the Web.

Ranking: Now consider the ranking of the returned instances. SEMEX provides three options for ranking: by *relevance score*, by *importance score*, and by *timeline* (see top right of Figure 6.1). By default, the returned instances are ranked by their relevance to the query, computed in a way close to the TF/IDF measure [114]. The relevance score is based on the number of times the keywords occur in the attributes of the instance or the number of associated instances that contain the keywords. A user can choose to rank the returned instances according to how important they are in the personal information space. We compute the importance score in a way similar to the PageRank algorithm [20], considering nodes as pages and associations as links between pages. The difference is that we weigh associations differently based on their types (*e.g.*, authorOf is weighted more than mentionedIn). These weights can be assigned manually by domain experts or learned from training data. Finally, a user can also choose to rank certain instances such as Articles and Emails by their latest modification time.

Finding association chains: In daily life a user often tries to remember how she gets to know a person, an article, etc. SEMEX attempts to answer such questions by finding the *association chains* between the instance of interest and the instance representing the user herself. An example association chain is as follows: a person is *mentioned in* an email that is *sent to* the user. Note that the shortest path between two instances is not necessarily the desired one; instead, the path that reflects the first time or the most recent time that the two instances interact can be more interesting. We define *earliest lineage* as the initial association chain in terms of chronological order, and *latest lineage* as the most recent association chain. For example, consider the association chain from a Person instance to the instance representing the user. The earliest lineage may indicate that the user gets to know the person by citing one of her papers, and the latest lineage may show that the latest contact between the user and the person is through an email.

Ranking and finding association chains are not focus of this dissertation and we skip the technical details.

Finally, we illustrate the above browsing and search facilities using an example.

Example 6.1. *Suppose the user wants to search for all the publications authored by Gordon*

Bell in the user's personal data. She first types the keyword "Gordon Bell". Semex *returns a set of persons, messages, and documents related to Gordon Bell. When the user selects a particular* Person *object,* Semex *presents all objects associated with this object, categorized into papers authored by him, messages sent to him, etc. The user can simply browse the category* AuthorOfArticles *to find Gordon Bell's publications.*

As an example of a more complex search, suppose the user is trying to find a specific reference to insert in a paper. She does not remember the title or authors of the paper, but does remember that she used this reference in a previous paper that also cited a paper by Gordon Bell. Having found the papers by Gordon Bell, she can find which ones were cited in her papers by following the CitedBy *association. Then, she can determine which was the previous paper of hers that cited both, and follow the* Cites *association to find the reference in question. Figure 6.1 indeed shows such a browsing trace. Finally, a more sophisticated user may choose not to follow this browsing chain but formulate a triple query as the one shown in the previous example.* □

6.2 System Architecture

Figure 6.2 depicts the components of the Semex system. Semex has three modules: the *domain management module* plays the central role by providing and managing the domain model; the *data collection module* is responsible for data extraction, integration, cleaning and indexing; the *query processing module* analyzes data for search and browsing. We now briefly explain each of these modules.

Domain management module

The domain management module provides the domain model for the other two modules. Currently Semex provides a default domain model to the users; ideally, the *domain model manager* should offer several ways of manually personalizing it. For example, the user can extend the domain model by example. The user begins by recording a specific pattern of browsing through instances in Semex. The browsing pattern can, in itself, already define a new class or association in the domain model. Alternatively, the user may refine, modify or generalize the pattern or combine it with other patterns to create the desired class.

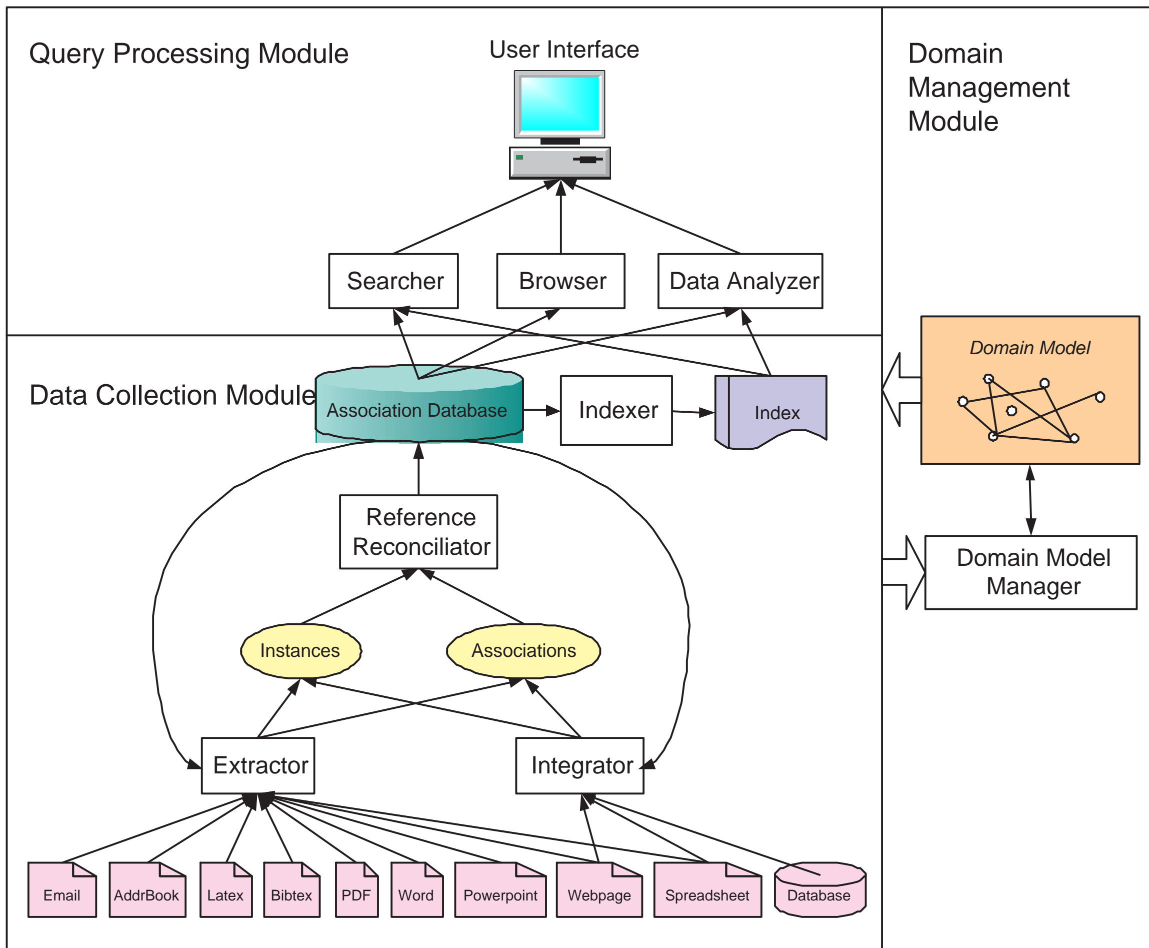


Figure 6.2: SEMEX architecture.

Ultimately, we would like the system to identify interesting clusters of information, and propose extensions based on them.

Data collection module

Semex provides access to data stored in multiple applications and sources, such as emails and address book contacts, pages in the user's web cache, documents (*e.g.*, Latex and Bibtex, PDF, Word, and Powerpoint) in the user's personal or shared file directory, and data in more structured sources (*e.g.*, spreadsheets and databases).

Semex starts data collection by using a set of object-and-association *extractors*. The extracted objects are processed by the *reference reconciliator* such that multiple references to the same real-world object are reconciled (Chapter 2). Reconciliation results are stored in the *association database*. Instances in the association database enable more data extraction (such as the association mentionedIn) and facilitate the *integrator* to integrate external data sources. Finally, the *indexer* indexes object instances in the association database for fast lookup of the keywords (Chapter 3).

We now describe the different components in detail.

Data extractor: The key architectural premise in Semex is that it should support a variety of mechanisms for obtaining object and association instances. We describe the main ones as follows.

1. *Simple:* In many cases, objects and associations are already stored conveniently in the data sources and they only need to be extracted according to the domain model. For example, a contact list already contains several important attributes of persons.

2. *Extracted:* A rich set of objects and associations can be extracted by analyzing specific file formats. For example, authors and titles can be extracted from Word documents and Powerpoint presentations. In more complex cases, Semex needs to examine data sources of multiple types. For example, citations can be computed from Latex and Bibtex files.

3. *Computed:* Finally, new associations can be computed from existing ones, such as one's co-authors.

In each of these cases, associations are obtained by some code. The architecture of Semex enables adding new extractors in a plug-and-play fashion; thus, users are able to incorporate association extractors as they see fit. Ideally, the system should also provide an interface for manually creating and revising instances and associations.

Reference reconciliation: Since the data we manage in PIM is very heterogeneous and we need to support multiple sources of associations, it is crucial that the data instances mesh together seamlessly. This leads to one of the key technical challenges in PIM: reference reconciliation. As seen in the right pane of Figure 6.1, the same real-world individual can be referred to using many different attribute values. To truly follow chains of associations and find all the information about a particular individual (or publication, conference, etc.), Semex needs to be able to reconcile the many references to the same real-world object. Chapter 2 described our reference reconciliation algorithm.

Association database: The result of the reconciliation algorithm is a high-quality reference list of a set of objects (*e.g.*, people, publications). Semex stores the objects and associations in a separate database, referred to as the *association database*. Except for enabling seamless querying and browsing, Semex also leverages this database to extract additional associations. For example, we can search for occurrences of person names in email bodies, spreadsheets, Word and PDF files, and so on, to create associations such as MentionedIn.

Indexer: The *indexer* indexes instances and associations in the association database. The index captures both the text values and the structural aspects of the data, such that it can efficiently support queries that combine keywords and structural requirements. In Chapter 3 we described our indexing techniques in detail.

Data integrator: As the users often need to access organizational data and the Web data, they can choose to integrate the external data into the personal information space. Integrating external data requires reformulating the external schema into the domain model. The extracted instances and associations can help in this process. For example, if it is detected that a column in a spreadsheet includes person names in the association database, it is often the case that the rest of the values in the column also represent person names. Since typical users may not be skilled enough to help in generating mappings, we may need to rely on

the candidate mappings generated by automatic schema mapping tools. In Chapter 5, we introduced the concept of *probabilistic schema mapping* to facilitate the use of candidate mappings that may not be precise.

Query processing module

SEMEX offers its users an interface that combines intuitive browsing and a variety of query facilities, ranging from simple keyword search to sophisticated triple query (see Section 6.1). The *browser* component supports associative browsing by exploring the associative network. The *searcher* component answers keyword search, predicate queries, and triple queries on both structured data and unstructured data in the personal information space and on the Web. The *data analyzer* performs offline analysis of the association network, such as computing earliest lineage and latest lineage for the instances to support finding association chains, and applying PageRank to compute the importance score of the instances to support ranking. Further, the *data analyzer* can trigger certain notifications and alarms when an event occurs; for example, when a user opens a webpage, SEMEX will list all instances that occur both on the webpage and in the association database, so the user can easily find her acquaintances on the webpage.

SEMEX is implemented in Java. The association database is stored as RDF data and queried using the Jena RDF Engine [76]. The indexing component is implemented by extending the Lucene Indexing Tool [92].

6.3 Related Work

A number of PIM projects [83, 49, 60, 48, 55, 57, 75, 100, 82, 13] studied how to effectively organize and search personal information. They all attempt to go beyond the traditional hierarchical directory model and present a unified user interface for personal data. We now survey several main ones and discuss where their data models fall short.

The Stuff I've Seen (SIS) project [49] and the Google Desktop Search toolkit [63] consider personal information as a collection of documents with *indexes* on the text of the documents. They index all types of information (in files, emails, webpages, etc.) as unstructured data and emphasize access through a unique full-text keyword search, which is independent of

the applications that store the data.

MyLifeBits [60] views personal data as a *network* of documents. Nodes in the graph represent documents and annotation meta-data; edges represent the `annotate` relationship. MyLifeBits focuses on integrating text and multimedia objects, allowing to annotate a file by another file, or by manually adding text annotation or audio annotation.

Placeless Documents [48] models personal information as *overlapping collections* of documents. It annotates documents with property/value pairs and groups documents into overlapping collections according to the property value. It also enables annotation with executable code as active properties. The Haystack project [107] once explored the same intuition and modeled data as *dynamic hierarchies* of documents. The hierarchy is called *dynamic* because users can use the properties in an arbitrary order to narrow down the search space (in contrast to following a fixed order in the traditional directory model).

The LifeStreams project [55] views personal information as a *sequence* of documents. It organizes documents in chronological order and allows the user to view the documents from different viewpoints in terms of time.

Although the above PIM systems all organize personal information in a way different from traditional directory hierarchies and provide a uniform access to all of one's personal information, they are different from SEMEX in the following aspects. First, SEMEX provides a logical view on one's personal data and supports a spectrum of search facilities ranging from keyword search to structured queries; however, none of these PIM systems provides search and browsing through such a logical view. In fact, among these PIM systems, only MyLifeBits manages to explicitly capture associations; even so, it does not distinguish different classes of instances or different types of associations and so only captures associations at a coarse granularity. Second, SEMEX models personal information at the instance level whereas the above PIM projects all consider information at the document level. As a result, it is hard for these systems to seamlessly integrate data in a semantically meaningful way. Third, SEMEX automatically populates the association network by extracting instances and associations from personal data; in contrast, the above PIM systems more heavily rely on manual work (for example, in MyLifeBits manual annotations are required and in LifeStreams properties are mainly set by hand).

The Haystack project [83] models personal information as objects and associations between objects, and has successfully demonstrated how this model enables personalized information presentation. Associations in Haystack can be automatically extracted from certain fields of documents, set up by observing the user's behaviors such as browsing trails, or added manually by the user [108]. However, Haystack mainly focuses on visualization of the information, rather than providing better search and querying services.

Finally, the Information Retrieval Community and the Human Computer Interaction Community have conducted research that study how people organize personal files [94, 88, 10, 52], emails [134], and web information (bookmarks) [106, 3, 124, 123]. Boardman and Sasse [19] compared the long-term organization behavior for the above three types of information. The design of our PIM system conforms to these research results.

6.4 Summary and Overarching PIM Themes

This chapter described SEMEX, a system that offers best-effort search, querying and browsing of personal information. The first key idea in SEMEX is to automatically construct an association network consisting of instances and associations from the information on one's desktop. The immediate benefit of this association network is to enable browsing personal information by association in the spirit of the Memex vision [22]. In addition, this database can support tasks such as coordination between multiple personal devices and context-aware behaviors. The second key idea in SEMEX is that the association network can be used as an anchor for importing external information sources, thereby offering seamless search and querying across different types of personal data, with extension to organizational data and Web data.

Beyond the specific technical challenges, our experience with Semex has highlighted several higher-level themes that we believe will pervade many of the challenges in PIM. First, many of the challenges arise because PIM manages *long-lived* and *evolving* data. In contrast, most data management is used to model database states that capture snapshots of the world. The evolution occurs at the instance level as well as the schema level. So far, the evolution has manifested itself in challenges to querying, reference reconciliation and schema mapping. The second theme is to find the right *granularity* for modeling personal

data. It is often possible to model the data at a very fine level. However, since PIM tools are geared toward users who are not necessarily technically savvy, it is important to keep the models as simple as possible. As we continue to investigate this tradeoff, we may find an interesting middle point between the models traditionally used for structured data and those for unstructured data. Third, when designing PIM systems it is important to think from the perspective of the user and her interactions with data in her daily routine, rather than from the perspective of the database. We need to build systems to support users in their *own* habitat, rather than trying to fit their activities into traditional data management. Finally, there has been a lot of interest in systems that combine structured and unstructured data in a seamless fashion. We believe that PIM is an excellent application to drive the development of such systems, raising challenges concerning storing, modeling and querying hybrid data.

Chapter 7
CONCLUSIONS AND FUTURE DIRECTIONS

In many applications we need to manage dataspaces, which contain heterogeneous data and partially unstructured data. Semantic mappings between the data sources may not exist either because users are not skilled enough to provide the mappings, or because the scale of the data makes it impossible to specify precise mappings. This dissertation studied how we can provide best-effort browsing, search and querying on dataspaces when the system evolves, even if we do not have mappings or have only imprecise mappings between the data sources. This is the first step towards pay-as-you-go data management: provide some services from the outset and improve the semantic mappings on an as-needed basis.

This chapter recaps the key contributions of this dissertation and discusses directions for future research.

7.1 Key Contributions

This dissertation has studied how to resolve heterogeneity at three levels in a dataspace system: the instance level, the schema level, and the query level. In particular, with the technical contributions this dissertation has made, we are able to build a dataspace system with the following functionalities.

- The system can reconcile references that refer to the same real-world entity and thus mesh the instances from various data sources seamlessly (Chapter 2). This is achieved by exploiting the association network extracted from the data sources. Specifically, we consider associated instances in comparison of object instances, propagate information from one reconciliation decision to another, and enrich references at the time of reconciliation.

- The system can index heterogeneous data to efficiently answer queries that combine

keywords and structural specification (Chapter 3). The index is based on extending inverted lists to capture both text values and structure of the data when it is present. In particular, we augment each keyword in the index by concatenating it with an attribute or association label to indicate the attribute that the keyword belongs to or the association that the keyword is involved in.

- The system can answer structured queries on unstructured data and hence support seamless search and querying on both structured and unstructured data (Chapter 4). We proposed answering structured queries on unstructured data by first translating a given structured query into a keyword query and then answering the keyword query over unstructured data. The key to this translation is building a query graph to capture the essence of the query and find the node or edge labels that best summarize the query graph.

- The system can employ probabilistic schema mappings to answer queries even when perfect mappings do not exist (Chapter 5). We have studied various aspects of query answering with respect to probabilistic schema mappings, including the complexity of query answering, the effect of different representations of probabilistic mappings on query answering, and the effect of more complex mapping languages on query answering.

Whereas our technical contributions apply to dataspace applications in general, we have grounded them to a particular system, the Semex Personal Information Management System. Semex provides a logical view of one's personal information on the desktop, on personal electronic devices, and on the Intranet or the Web. Semex leverages this logical view to allow associative browsing of one's personal data, and provides seamless search and querying over all types of personal data.

7.2 Future Work

This dissertation is the first step towards the pay-as-you-go dataspace management. There are still many open problems to solve in building dataspace systems and we next list a few.

Reuse of Human Attention: One of the key properties of a dataspace is that semantic integration evolves over time and only where needed. For reducing the burden of mapping creation, we would like to weave data integration into the fabric of people's daily searching and browsing and leverage experiences from previous integration tasks. Thus, one important direction is to apply Machine Learning techniques to study the reuse of human attention from several aspects: how to leverage existing structured data to improve the extraction of structure from text (*e.g.*, based on existing person instances, discover new person instances occurring in a table or a list of a web page), how to record previously composed queries and the refinement on these queries to better understand the source structure, how to leverage operations on the data (*e.g.*, cutting and pasting values from one column in a spreadsheet into a different column in a different spreadsheet) to discover relationships between data, and how to find the hot spots where specification of structure can bring the most benefits.

Probabilistic Data Integration: When we manage a large number of data sources, especially a dataspace at the web scale, semantic mappings will be approximate at best: not only is it hard to create exact mappings and maintain them, but it is not even clear what a correct mapping would mean in many cases. To model *all* domains of possible interest to users and to provide access to *data about anything*, the dataspace platform needs to handle *uncertainty* at its core. Specifically, we need to handle uncertainty from several sources: 1) uncertainty about mappings between disparate schemas, 2) uncertainty about the structure extracted from unstructured texts (typically using automatic techniques), and 3) uncertainty about the structural information encoded in a query (often appearing as keyword queries).

We envision a system that supports probabilistic data integration. First, based on the work on probabilistic schema mapping (Chapter 5) and recent work in the community on probabilistic databases [118], we can investigate the foundation of generating probabilistic answers to queries and design algorithms that efficiently compute top-K answers in presence of uncertainties in schemas and in data. Second, we can extend our work on indexing mechanism (Chapter 3) to incorporate heterogeneity in form of probabilistic matching and fuzzy values and study the ranking schemes for queries where keywords meet approximate

structure. Finally, it is interesting to study how to quantify probabilities from schema matchings generated by semi-automatic tools and how to use these probabilities as feedback to obtain better matching results.

Universal Search: Keyword search has the property that it is more forgiving than a query, based on similarity and providing ranked results to end users. It is especially suitable in a dataspace environment, as users typically do not know all the disparate underlying structure. A dataspace platform should enable a user to specify a keyword query and retrieve data from all relevant data sources and iteratively refine the query to a structured query when appropriate.

Towards providing a universal search service, it is interesting to study the following immediate problems: 1) *query routing*: given a keyword query, detect the user's intention and find the sources that are most relevant to the query; 2) *query reformulation*: given a keyword query and a relevant structured source, reformulate the query according to the schema of the source; 3) *result ranking*: given a set of answers obtained from both structured and unstructured sources, rank them according to multiple criteria such as the relevance of the answers, the details of the information, and the authority of the sources; 4) *query refinement*: given a keyword query or a reformulated structured query, help the user refine it to obtain better query results.

Information Redundancy: With new data sources quickly added to a dataspace, some of which might duplicate or derive from existing ones, query answering often returns overwhelming volume of data that are hard for users to digest. Whereas a good ranking method can bring up the most relevant results first, it would still be beneficial to detect overlaps in results and return only distinct ones. To obtain this goal, the dataspace platform should be able to model the relationships between data participants, such as one is a view or replica of another, one overlaps with another with a particular percentage, or one covers a certain percentage of existing information in a specific domain [54]. It is interesting to examine how to formalize such relationships between data sources, how to leverage them to provide distinct search results, how to use them to optimize query answering on multiple sources, and how to retrieve relevant data sources and allow users to inquire about their completeness,

correctness, and freshness.

BIBLIOGRAPHY

[1] Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia-Molina, Dieter Galwick, Jim Gray, Laura Haas, Alon Y. Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker, Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stand Zdonik. The Lowell Database research self assessment. *Communications of the ACM (CACM)*, 48(5):111–118, 2005.

[2] Serge Abiteboul and Oliver M. Duschka. Complexity of answering queries using materialized views. In *Proc. of PODS*, pages 254–263, 1998.

[3] David Abrams, Ronald Baecker, and Mark H. Chignell. Information archiving with bookmarks: Personal Web space constrction and organization. In *Proc. of CHI*, pages 41–48, 1998.

[4] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. of ICDE*, pages 5–16, 2002.

[5] Shurug Al-Khalifa, H.V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. of ICDE*, page 141, 2002.

[6] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. of VLDB*, pages 586–597, 2002.

[7] Lyublena Antova, Christoph Koch, and Dan Olteanu. World-set decompositions: Expressiveness and efficient algorithms. In *Proc. of ICDT*, pages 194–208, 2007.

[8] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Fast text searching for regular expressions or automaton simulation over tires. *Journal of the ACM*, 43(6):915–936, 1996.

[9] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, New York, 1999.

[10] Deborah Barreau and Bonnie A. Nardi. 'Finding and reminding' file organization from the desktop. *SIGCHI Bulletin*, 27(3):39–43, 1995.

[11] Holger Bast and Ingmar Weber. Type less, find more: Fast autocompletion search with a succinct index. In *Proc. of SIGIR*, pages 364–371, 2006.

[12] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual Web information extraction with lixto. In *Proc. of VLDB*, pages 119–128, 2001.

[13] Victoria Bellotti, Nicolas Ducheneaut, Mark Howard, and Ian Smith. Taking email to task: The design and evaluation of a task management centered email tool. In *Proc. of CHI*, pages 345–352, 2003.

[14] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *Proc. of VLDB*, pages 953–964, 2006.

[15] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proc. of ICDE*, pages 431–440, 2002.

[16] Indrajit Bhattacharya and Lise Getoor. Iterative record linkage for cleaning and integration. In *Proc. of DMKD*, pages 11–18, 2004.

[17] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proc. of SIGKDD*, pages 39–48, 2003.

[18] Mikhail Bilenko, Raymond J. Mooney, William Cohen, Pradeep Ravikumar, and Stephen Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems Special Issue on Information Integration on the Web*, 18(5):16–23, 2003.

[19] Richard Boardman and Martina Angela Sasse. 'Stuff goes into the computer and doesn't come out': A cross-tool study of personal information management. In *Proc. of CHI*, pages 583–590, 2004.

[20] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[21] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig joins: Optimal XML pattern matching. In *Proc. of SIGMOD*, pages 310–321, 2002.

[22] Vannevar Bush. As we may think. *The Atlantic Monthly*, July 1945.

[23] Michael J. Cafarella, Christopher Re, Dan Suciu, Oren Etzioni, and Michele Banko. Structured querying of web text data: a technical challenge. In *Proc. of CIDR*, pages 225–234, 2007.

[24] Yuhan Cai, Xin Dong, Alon Y. Halevy, Jing Liu, and Jayant Madhavan. Personal information management with SEMEX. In *Proc. of SIGMOD*, pages 921–923, 2005.

[25] Soumen Chakrabarti, Kriti Puniyani, and Sujatha Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *Proc. of the Int. WWW Conf.*, pages 169–172, 2006.

[26] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. of SIGMOD*, pages 313–324, 2003.

[27] Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proc. of SIGMOD*, pages 134–144, 2003.

[28] Zhiyuan Chen, Johannes Gehrke, Flip Korn, Nick Koudas, Jayavel Shanmugasundaram, and Divesh Srivastava. Index structures for matching XML Twigs using relational query processors. *Data and Knowledge Engineering*, 60(2):283–302, 2005.

[29] Reynold Cheng, Sunil Prabhakar, and Dmitri V. Kalashnikov. Querying imprecise data in moving object environments. In *Proc. of ICDE*, pages 723–725, 2003.

[30] Mitch Cherniack, Michael J. Franklin, and Stanley B. Zdonik. Data management for pervasive computing: A tutorial. Proc. of VLDB, 2001.

[31] Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In *Proc. of VLDB*, pages 263–274, 2002.

[32] Junghoo Cho and Sridhar Rajagopalan. A fast regular expression indexing engine. In *Proc. of ICDE*, pages 419–430, 2002.

[33] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An adaptive path index for XML data. In *Proc. of SIGMOD*, pages 121–132, 2002.

[34] William W. Cohen, Henry Kautz, and David McAllester. Hardening soft information sources. In *Proc. of SIGKDD*, pages 255–259, 2000.

[35] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proc. of IIWEB*, pages 73–78, 2003.

[36] William W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proc. of SIGKDD*, pages 475–480, 2002.

[37] Brian F. Cooper, Neal Sample, Michael J.Franklin, Gisli R.Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proc. of VLDB*, 2001.

[38] Cora. http://www.cs.umass.edu/~mccallum/data/cora-refs.tar.gz.

[39] Michel de Rougemont and Adrien Vieilleribiere. Approximate data exchange. In *Proc. of ICDT*, pages 44–58, 2007.

[40] Ludovic Denoyer and Patrick Gallinari. The Wikipedia XML corpus. *SIGIR Forum*, 40(1):64–69, 2006.

[41] AnHai Doan, Ying Lu, Yoonkyong Lee, and Jiawei Han. Object matching for information integration: A profiler-based approach. In *Proc. of IIWEB*, pages 53–58, 2003.

[42] AnHai Doan, Raghu Ramakrishnan, and Shivakumar Vaithyanathan. Managing information extraction: state of the art and research directions. In *Proc. of SIGMOD*, pages 799–800, 2006.

[43] Xin Dong and Alon Y. Halevy. A platform for personal information management and integration. In *Proc. of CIDR*, pages 119–130, 2005.

[44] Xin Dong and Alon Y. Halevy. Indexing dataspaces. In *Proc. of SIGMOD*, pages 43–54, 2007.

[45] Xin Dong, Alon Y. Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *Proc. of SIGMOD*, pages 85–96, 2005.

[46] Xin Dong, Alon Y. Halevy, Ema Nemes, Stephan Sigurdsson, and Pedro Domingos. Semex: Toward on-the-fly personal information integration. In *Proc. of IIWEB*, 2004.

[47] Xin Dong, Alon Y. Halevy, and Cong Yu. Data integration with uncertainties. In *Proc. of VLDB*, 2007.

[48] Paul Dourish, W. Keith Edwards, Anthony LaMarca, John Lamping, Karin Petersen, Michael Salisbury, Douglas B. Terry, and James Thornton. Extending document management systems with user-specific active properties. *ACM Transactions on Information Systems (TOIS)*, 18(2):140–170, 2000.

[49] Susan Dumais, Edward Cutrell, Jonathan J. Cadiz, Gavin Jancke, Raman Sarin, and Daniel C. Robbins. Stuff I've Seen: A system for personal information retrieval and re-use. In *SIGIR*, pages 72–79, 2003.

[50] Oren Etzioni, Michael J. Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Web-scale information extraction in KnowItAll (preliminary results). In *Proc. of the Int. WWW Conf.*, pages 100–110, 2004.

[51] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[52] Scott Fertig, Eric Freeman, and David Gelernter. 'Finding and reminding' reconsidered. *SIGCHI Bulletin*, 28(1), 1996.

[53] Flickr. http://www.flickr.com/, 2006.

[54] Daniela Florescu, Daphne Koller, and Alon Y. Levy. Using probabilistic information in data integration. In *Proc. of VLDB*, pages 216–225, 1997.

[55] Eric Freeman and David Gelernter. Lifestreams: A storage model for personal data. *SIGMOD Record*, 25(1):80–86, 1996.

[56] Dayne Freitag and Andrew McCallum. Information extraction with HMMs and shrinkage. In *Proceedings of the AAAI-99 Workshop on Machine Learning for Information Extraction*, 1999.

[57] Doug Gage. Lifelog. http://www.darpa.mil/ipto/Programs/lifelog/.

[58] Avigdor Gal. Managing uncertainty in schema matching with Top-K schema mappings. *Journal on Data Semantics*, 6, 2006.

[59] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *Proc. of VLDB*, pages 371–380, 2001.

[60] Jim Gemmell, Gordon Bell, Roger Lueder, Steven Drucker, and Curtis Wong. MyLifeBits: Fulfilling the Memex vision. In *ACM Multimedia*, pages 235–238, 2002.

[61] Roy Goldman, Narayanan Shivakumar, Suresh Venkatasubramanian, and Hector Garcia-Molina. Proximity search in databases. In *Proc. of VLDB*, pages 26–37, 1998.

[62] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. of VLDB*, pages 436–445, 1997.

[63] Google desktop search toolkit. http://desktop.google.com/, 2004.

[64] Google co-op. http://www.google.com/coop, 2006.

[65] GoogleBase. http://base.google.com/, 2005.

[66] Jens Graupmann, Ralf Schenkel, and Gerhard Weikum. The SphereSearch engine for unified ranked retrieval of heterogeneous XML and Web documents. In *Proc. of VLDB*, pages 529–540, 2005.

[67] Michael Gubanov and Philip A. Bernstein. Structural text search and comparison using automatically extracted schema. In *Proc. of the WebDB Workshop*, 2006.

[68] Alon Y. Halevy, Michael J. Franklin, and David Maier. Principles of dataspace systems. In *Proc. of PODS*, pages 1–9, 2006.

[69] Alon Y. Halevy, Anand Rajaraman, and Joann J. Ordille. Data integration: The teenage years. In *Proc. of VLDB*, pages 9–16, 2006.

[70] Laura M. Hass. Beauty and the beast: The theory and practice of information integration. In *Proc. of ICDT*, pages 28–43, 2007.

[71] Hao He and Jun Yang. Multiresolution indexing of XML for frequent queries. In *Proc. of ICDE*, pages 683–694, 2004.

[72] Mauricio A. Hernandez and Salvatore J. Stolfo. The merge/purge problem for large databases. In *Proc. of SIGMOD*, pages 127–138, 1995.

[73] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: Keyword search in relational databases. In *Proc. of VLDB*, pages 670–681, 2002.

[74] Vagelis Hristidis, Yannis Papakonstantinou, and Andrey Balmin. Keyword proximity search on XML graphs. In *Proc. of ICDE*, pages 367–378, 2003.

[75] Frank M. Shipman III, Haowei Hsieh, Robert Airhart, Preetam Maloor, J. Michael Moore, and Divya Shah. Emergent structure in analytic workspaces: Design and use of the visual knowledge builder. In *Proceedings of Interact*, pages 132–139, 2001.

[76] Jena. http://jena.sourceforge.net/, 2005.

[77] Haifeng Jiang, Hongjun Lu, Wei Wang, and Beng Chin Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *Proc. of ICDE*, pages 253–263, 2003.

[78] Liang Jin, Chen Li, and Sharad Mehrotra. Efficient record linkage in large data sets. In *Proc. of DASFAA*, pages 137–146, 2003.

[79] William Jones and Jaime Teevan, editors. *Personal information management.* University of Washington Press, Seattle, WA, 2007.

[80] Yuh-Jzer Joung and Li-Wei Yang. KISS: A simple prefix search scheme in P2P networks. In *Proc. of the WebDB Workshop*, pages 56–61, 2006.

[81] Dmitri V. Kalashnikov, Sharad Mehrotra, and Zhaoqi Chen. Exploiting relationships for domain-independent data cleaning. In *SIAM Data Mining (SDM)*, 2005.

[82] Victor Kaptelinin. UMEA: Translating interaction histories into project contexts. In *Proc. of CHI*, pages 353–360, 2003.

[83] David Karger, Karun Bakshi, David Huynh, Dennis Quan, and Vineet Sinha. Haystack: A general-purpose information management tool for end users based on semistructured data. In *Proc. of CIDR*, pages 13–26, 2005.

[84] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *Proc. of SIGMOD*, pages 133–144, 2002.

[85] Raghav Kaushik, Rajasekar Krishnamurthy, Jeffrey F. Naughton, and Raghu Ramakrishnan. On the integration of structure indexes and inverted lists. In *Proc. of SIGMOD*, pages 779–790, 2004.

[86] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proc. of ICDE*, pages 129–140, 2002.

[87] Martin Kersten, Gerhard Weikum, Michael Franklin, Daniel Keim, Alejandro Buchmann, and Surajit Chaudhuri. A database striptease or how to manage your personal databases. In *Proc. of VLDB*, pages 1043–1044, 2003.

[88] M. Lansdale. The psychology of personal information management. *Applied Ergonomics*, 19(1):458–465, 1988.

[89] Mong Li Lee, Tok Wang Ling, and Wai Lup Low. IntelliClean: A knowledge-based intelligent data cleaner. In *Proc. of SIGKDD*, pages 290–294, 2000.

[90] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proc. of PODS*, pages 233–246, 2002.

[91] Jing Liu, Xin Dong, and Alon Y. Halevy. Answering structured queries on unstructured data. In *WebDB*, pages 25–30, 2006.

[92] Lucene. http://jakarta.apache.org/lucene/docs/index.html, 2005.

[93] Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Y. Halevy. Corpus-based schema matching. In *Proc. of ICDE*, pages 57–68, 2005.

[94] Thomas W. Malone. How do people organize their desks? Implications for the design of office information system. *ACM Transactions on Office Information Systems (TOIS)*, 4:42–63, 1986.

[95] Andrew McCallum. Efficiently inducing features or conditional random fields. In *Proceedings of the Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 403–410, 2003.

[96] Andrew K. McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of SIGKDD*, pages 169–178, 2000.

[97] Andrew K. McCallum and Ben Wellner. Toward conditional models of identity uncertainty with application to proper noun coreference. In *Proc. of IIWEB*, pages 79–84, 2003.

[98] Martin Michalowski, Snehal Thakkar, and Craig A. Knoblock. Exploiting secondary sources for unsupervised record linkage. In *Proc. of IIWEB*, 2004.

[99] Tova Milo and Dan Suciu. Index structures for path expressions. In *Proc. of ICDT*, pages 277–295, 1999.

[100] Bonnie A. Nardi, Steve Whittaker, Ellen Isaacs, Mike Creech, Jeff Johnson, and John Hainsworth. Integrating communication and information through contactmap. *Communications of the ACM (CACM)*, 45(4):89–95, 2002.

[101] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.

[102] Niagara XML repository. http://www.cs.wisc.edu/ niagara/data.html, 2004.

[103] Hanna Pasula, Bhaskara Marthi, Brian Milch, Stuart Russell, and Ilya Shpitser. Identity uncertainty and citation matching. In *Proc. of NIPS*, pages 1401–1408, 2002.

[104] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, Inc., California, 1988.

[105] Jose C. Pinheiro and Don X. Sun. Methods for linking and mining massive heterogeneous databases. In *Proc. of SIGKDD*, pages 309–313, 1998.

[106] James E. Pitkow and Colleen M. Kehoe. Emerging trends in the WWW user population. *Communications of the ACM (CACM)*, 39(6):106–108, 1996.

[107] Dennis Quan, Karun Bakshi, David Huynh, and David R. Karger. User interfaces for supporting multiple categorization. In *Proceedings of Interact*, 2003.

[108] Dennis Quan, David Huynh, and David R. Karger. Haystack: A platform for authoring end user semantic Web applications. In *Proc. of ISWC*, pages 738–753, 2003.

[109] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[110] Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using Prufer sequences. In *Proc. of ICDE*, pages 288–300, 2004.

[111] Christopher Re, Dan Suciu, and Nilesh N. Dalvi. Query evaluation on probabilistic databases. *IEEE Data Eng. Bull.*, 29(1), 2006.

[112] Kemmetj A. Ross and Angel Janevski. Querying facted databases. In *Proceedings of the International Workshop on Semantic Web and Databases (SWDB)*, 2004.

[113] Prasan Roy, Mukesh K. Mohania, Bhuvan Bamba, and Shree Raman. Towards automatic association of relevant unstructured content with structured query results. In *Proc of CIKM*, pages 405–412, 2005.

[114] Gerard Salton, editor. *The SMART Retrieval System—Experiments in Automatic Document Retrieval*. Prentice Hall Inc., Englewood Cliffs, NJ, 1971.

[115] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *Proc. of SIGKDD*, pages 269–278, 2002.

[116] Mayssam Sayyadian, Hieu Lekhac, AnHai Doan, and Luis Gravano. Efficient keyword search across heterogeneous relational databases. In *Proc. of ICDE*, pages 346–355, 2007.

[117] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A benchmark for XML data management. In *Proc. of VLDB*, pages 974–985, 2002.

[118] Parag Singla and Pedro Domingos. Object identification with attribute-mediated dependences. In *Ninth European Conference on Principles and Practice of Knowledge Discovery in Databaes (PKDD)*, pages 297–308, 2005.

[119] Marios Skounakis, Mark Craven, and Soumya Ray. Hierarchical hidden Markov models for information extraction. In *Proc. of IJCAI*, pages 427–433, 2003.

[120] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[121] Stephen Soderland, David Fisher, Jonathan Aseltine, and Wendy G. Lehnert. Crystal: Inducing a conceptual dictionary. In *Proc. of IJCAI*, pages 1314–1321, 1995.

[122] Qi Su and Jennifer Widom. Indexing relational database content offline for efficient keyword-based search. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 297–306, 2005.

[123] Linda Tauscher and Saul Greenberg. How people revisit Web pages: Empirical findings and implications for the design of history systems. *International Journal of Human-Computedr Studies*, 47:97–137, 1997.

[124] Linda Tauscher and Saul Greenberg. Revisitation patterns in World Wide Web navigation. In *Proc. of CHI*, pages 399–406, 1997.

[125] Sheila Tejada, Craig A. Knoblock, and Steven Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *Proc. of SIGKDD*, 2002.

[126] Thodoros Topaloglou. Biological data management: Research, practice, and opportunities. In *Proc. of VLDB*, 2004.

[127] Text REtrieval Conference (TREC). http://trec.nist.gov//, 2007.

[128] Jeffrey D. Ullman. Information integration using logical views. In *Proc. of ICDT*, pages 19–40, Delphi, Greece, 1997.

[129] UW XML data repository. http://www.cs.washington.edu/ research/xmldatasets/, 2002.

[130] Patrick Valduriez. Join indices. *ACM transactions on Database Systems*, 12(2), 1987.

[131] Luio von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proc. of CHI*, pages 319–326, 2004.

[132] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: A dynamic index method for querying XML data by tree structures. In *Proc. of SIGMOD*, pages 110–121, 2003.

[133] Wei Wang, Haifeng Jiang, Hongjun Lu, and Jeffrey Xu Yu. PBiTree coding and efficient processing of containment joins. In *Proc. of ICDE*, page 391, 2003.

[134] Steve Whittaker and Candace L. Sidner. Email overload: Exploring personal information management of email. In *Proc. of CHI*, pages 276–283, 1996.

[135] Windows Desktop Search. http://www.microsoft.com/windows/desktopsearch/default.mspx, 2007.

[136] William E. Winkler. Using the EM algorithm for weight computation in the fellegi-sunter model of record linkage. In *Section on Survey Research Methods*, 1988.

[137] William E. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, D.C., 1999.

[138] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and indexing documents and images.* Morgan Kaufmann Publishers, San Francisco, 1999.

[139] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *Proc. of SIGMOD*, pages 537–538, 2005.

[140] Yahoo! Desktop Search. http://desktop.yahoo.com/, 2007.

[141] Ning Zhang, Tamer Ozsu, Ihab F. Ilyas, and Ashraf Aboulnaga. Fix: Feature-based indexing technique for XML documents. In *Proc. of VLDB*, pages 259–270, 2006.

Appendix A

ALGORITHM OF BUILDING A HYBRID-ATIL

In this appendix, we consider how to build a hybrid-ATIL from a Hier-ATIL. The key in the algorithm is to identify the prefixes for which we need to add summary rows. We now describe an algorithm that finds such prefixes with a single scan of the keywords in the Hier-ATIL.

As shown in Figure A.1, our algorithm has two main components. The first component, the procedure SCAN, takes a Hier-ATIL and a threshold t as input, and generates the Hybrid-ATIL by adding summary rows to the given Hier-ATIL. SCAN maintains a stack of prefixes, where if the top prefix in the stack is of the form $a_1//\dots//a_n//$, then the i-th ($i \in [1, n]$) element in the stack is of the form $a_1//\dots//a_i//$; in other words, except the last prefix, each prefix is a prefix of its subsequent prefix and has only one less "//" than the subsequent prefix. In addition, SCAN maintains a *counter* array that records for each prefix in the stack the number of its occurrences in the already scanned keywords.

SCAN proceeds by scanning the indexed keywords in the Hier-ATIL. When it reaches a keyword k, if k starts with the prefix at the top of the stack, it pushes into the stack each prefix of k that ends with "//" but is not in the stack yet; otherwise, SCAN pops up prefixes in the stack that are not prefixes of k, and invokes procedure JUDGE for each popped prefix to decide whether a summary row should be added to the inverted list. Finally, after scanning all keywords in the Hier-ATIL, SCAN pops up all prefixes in the stack and invokes JUDGE for each of them to decide whether a summary row needs to be added.

Another component, the procedure JUDGE, pops up a prefix p from the stack and decides if a summary row should be added for p. If the counter for p (*i.e.*, the number of indexed keywords with prefix p) is above the threshold t, JUDGE adds a summary row for p and increases the counter for the current top prefix in the stack by 1. Otherwise, JUDGE does not add a summary row but instead increases the counter for the current top prefix in the

procedure SCAN(L, t)

//L is the Hier-ATIL of an association network and t is the threshold;

Initialize *stack* with a single element "";

Initialize each element of array *counter*[] to 0;

//counter summarizes the occurrences of each prefix;

for each (keyword k in L)

while (top(*stack*) is not a prefix of k)

JUDGE(*stack*, *counter*, t);

while (there is a prefix *pre* of k ending with "//" and having one more "//" than *top*(*stack*))

Push *pre* into *stack*;

counter[*pre*] = 0;

counter[k] = 1;

while (top(*stack*)<> "")

JUDGE(*stack*, *counter*, t);

procedure JUDGE(*stack*, *counter*, t)

str_0 = pop(*stack*);

str = top(*stack*);

if ($counter[str_0] > t$)

Add a summary row for str_0//;

Remove the row with keyword str_0;

counter[*str*]++;

else

$counter[str]$+=$counter[str_0]$;

Figure A.1: The algorithm for building a hybrid-ATIL.

stack by the counter for p.

We now illustrate the algorithm with an example.

Example A.1. *Consider building the Hybrid-ATIL with $t = 2$ for a Hier-ATIL with the following indexed keywords:*

- *"tian//desc//",*
- *"tian//desc//name//firstName//",*
- *"tian//desc//name//lastName//",*
- *"tian//desc//name//nickName//",*
- *"tian//email//".*

After reading the first keyword, SCAN *pushes into the stack prefixes "tian//" and "tian//desc//". Since the second keyword starts with "tian//desc//",* SCAN *then pushes into the stack prefixes "tian//desc//name//" and "tian//desc//name//firstName//". Figure A.2(a) shows the stack and the counter array after processing the first two keywords.*

When SCAN *reaches keyword "tian//desc//name//lastName//", which does not start with the prefix at the top of the stack ("tian//desc//name//firstName//"), it invokes procedure* JUDGE*. Since the counter for "tian//desc//name//firstName//" is 1, not above the threshold,* JUDGE *does not need to add a summary row for "tian//desc//name//firstName//" and so only increases the counter for "tian//desc//name//" by the counter for "tian//desc//name//firstName//". It handles the keyword "tian//desc//name//nickName//" in a similar way. Figure A.2(b)(c) shows the stack and the counter array after processing these two keywords respectively.*

When SCAN *reads keyword "tian//email//", it needs to pop up prefixes "tian//desc//name//nickName//", "tian//desc//name//" and "tian//desc//". When* JUDGE *pops up "tian//desc//name//", it finds that its counter is 3, above the threshold.* JUDGE *then adds a summary row for "tian//desc//name//", and increases the counter of "tian//desc//" by 1. After popping up "tian//desc//",* JUDGE *increases*

stack	counter
tian//desc//name//firstName//	1
tian//desc//name//	0
tian//desc//	1
tian//	0

(a)

stack	counter
tian//desc//name//lastName//	*1*
tian//desc//name//	*1*
tian//desc//	1
tian//	0

(b)

stack	counter
tian//desc//name//nickName//	*1*
tian//desc//name//	*2*
tian//desc//	1
tian//	0

(c)

stack	counter
tian//desc//name//	*3*
tian//desc//	1
tian//	0

(d)

stack	counter
tian//desc//	*2*
tian//	0

(e)

stack	counter
tian//	*2*

(f)

Figure A.2: The *stack* and *counter* array after (a) processing keyword "tian//desc//" and "tian//desc//name//firstName//"; (b) processing keyword "tian//desc//name// lastName//"; (c) processing keyword "tian//desc// name//nickName//"; (d) popping up prefix "tian// desc//name//nickName//"; (e) popping up prefix "tian//desc//name//"; (f) popping up prefix "tian//desc//". The difference of each sub-graph from the former one is highlighted using the italic font.

the counter for "tian//" by the counter for "tian//disc//". Note that JUDGE *updates the counters for "tian//desc//" and "tian//" differently, because it adds a summary row for "tian//desc//name//" but not for "tian//desc//". Figure A.2(d)(e)(f) shows the stack and the counter array after popping up "tian//desc//name//nickName//", "tian//desc//name//" and "tian//desc" respectively.*

Finally, after processing all keywords, JUDGE *is again invoked and it adds a summary row for prefix "tian//".* □

Appendix B

PROOFS FROM CHAPTER 5

B.1 Proof for Theorem 5.15

Theorem 5.15. Let Q be an SPJ query and let $\overline{pM}$ be a schema p-mapping.

The problem of finding the probability for a by-tuple answer to Q with respect to $\overline{pM}$ is #P-complete with respect to data complexity and is in PTIME with respect to mapping complexity. □

Proof. We prove the theorem by proving three lemmas, stating that (1) the problem is in PTIME in the size of the mapping; (2) the problem is in #P in the size of the data; (3) the problem is #P-hard in the size of the data.

Lemma B.1. *Let Q be an SPJ query and let $\overline{pM}$ be a schema p-mapping.*

The problem of finding the probability for a by-tuple answer to Q with respect to $\overline{pM}$ is in PTIME in the size of the mapping. □

Proof. We can generate all answers in three steps. Let $T_1, \ldots, T_l$ be the relations mentioned in Q's `FROM` clause. Let pM_i be the p-mapping associated with table T_i. Let d_i be the number of tuples in the source table of pM_i.

1. For each $seq^1 \in \mathbf{seq}_{d1}(pM_1), \ldots, seq^l \in \mathbf{seq}_{dl}(pM_l)$, generate a target instance that is consistent with the source instance and $\overline{pM}$ as follows. For each $i \in [1, l]$, the target relation T_i contains d_i tuples, where the j-th tuple (1) is consistent with the j-th source tuple and the j-th mapping m^j in seq^i, and (2) contains null as the value of each attribute that is not involved in m^j.

2. For each target instance, answer Q on the instance. Consider only the answer tuples that do not contain the null value and assign probability $\Pi_{i=1}^{l} Pr(seq^i)$ to the tuple.

3. For each distinct answer tuple, sum up its probabilities.

According to the definition of by-tuple answers, the algorithm generates all by-tuple answers. We now prove it takes polynomial time in the size of the mapping. Assume each p-mapping pM_i contains l_i mappings. Then, the number of instances generated in step 1 is $\Pi_{i=1}^{l} l_i^{d_i}$, polynomial in the size of $\overline{pM}$. In addition, the size of each generated target instance is linear in the size of the source instance. So the algorithm takes polynomial time in the size of the mapping. □

Lemma B.2. *Let Q be an SPJ query and let $\overline{pM}$ be a schema p-mapping.*

The problem of finding the probability for a by-tuple answer to Q with respect to $\overline{pM}$ is in #P in the size of the data. □

Proof. We prove the claim by reducing the problem to answering queries on *disjunctive probabilistic databases*, which is proved to be in #P [111]. Before we describe the reduction, we first introduce probabilistic databases.

Definition B.3 (Probabilistic Database)**.** *A* probabilistic database (p-database) *pD over a schema $\bar{R}$ is a set $\{(D_1, Pr(D_1)), \ldots, (D_n, Pr(D_n))\}$, such that*

- *for $i \in [1, n]$, D_i is an instance of $\bar{R}$, and for $i \neq j, D_i \neq D_j$;*
- *$Pr(D_i) \in [0, 1]$ and $\sum_{i=1}^{n} Pr(D_i) = 1$.* □

Answers to queries over p-databases have probabilities associated with them. Specifically, let Q be a query over pD, and let t be a tuple. We denote by $\bar{D}(t)$ the subset of pD such that for each $D \in \bar{D}(t)$, $t \in Q(D)$. Let $p = \sum_{D \in \overline{D}(t)} Pr(D)$. If $p > 0$, we call (t, p) a *possible tuple in the answer of Q on pD*.

Given a SPJ query Q and a p-database pD, we denote by $Q(pD)$ the set of all possible tuples in the answer of Q on pD. Computing $Q(pD)$ takes polynomial time in the size of pD.

We next define a compact representation of p-databases, called *disjunctive p-database*, over which query answering is #P-complete in the size of the representation.

Definition B.4 (Disjunctive P-Database)**.** *Let R be a relation schema where there exists a set of attributes that together form the* key *of the relation. Let $pD_R^\vee$ be a set of tuples of R, each has a probability.*

We say that $pD_R^\vee$ is a disjunctive p-database *if for each key value that occurs in $pD_R^\vee$, the probabilities of the tuples with this key value sum up to 1.* □

In a disjunctive p-database, we consider tuples with the same key value as disjoint. Formally, let $key_1, \ldots, key_n$ be the set of all distinct key values in $pD_R^\vee$. For each $i \in [1, n]$, we denote by d_i the number of tuples whose key value is key_i. Then, $pD_R^\vee$ defines a set of $\Pi_{i=1}^n d_i$ possible databases. Each possible database $(D, Pr(D))$ contains n tuples $t_1, \ldots, t_n$, such that (1) for each $i \in [1, n]$, the key value of t_i is key_i; and (2) $Pr(D) = \Pi_{i=1}^n Pr(t_i)$.

We now describe the reduction. We reduce the problem of query answering with respect to probabilistic mappings to the problem of query answering on disjunctive p-databases. The reduction proceeds as follows.

For each relation T that occurs in Q and is involved in a p-mapping $pM = (S, T, \mathbf{m})$, generate the target instance as follows. The target instance is a disjunctive p-database with attributes in T and a key column that is the key of the relation. For the i-th tuple t_s in S and each $m \in \mathbf{m}$, generate a target tuple t_t, such that (1) for each attribute correspondence $(a_s, a_t) \in m$, the value of a_t is the same as the value of a_s in t_s; (2) for each attribute a_t in T that is not involved in any attribute correspondence in m, the value of a_t is null; and (3) the value of the key attribute is i. The probability of the tuple is $Pr(m)$. Let n be the number of tuples in T and l be the number of mappings in pM. Generating the target instance takes time $O(l \cdot n)$, polynomial in the size of the data and the mapping.

Let D_S be a source instance and pD_T be the generated target instance. We now prove $Q^{tuple}(D_S) = Q(pD_T^\vee)$, where we assume $Q(pD_T)$ does not return answers containing null values. We prove by showing that for each possible database D_T of $pD_T^\vee$, there exists a mapping sequence seq, such that $Pr(D_T) = Pr(seq)$ and the set of tuples in $Q(D_T)$ is the same as the set of certain answers with respect to seq, and vice versa.

I. Suppose D_T contains tuples $t_1, \ldots, t_n$, where $t_i, i \in [1, n]$, has i as the value of key. Then, t_i is consistent with the i-th source tuple in S and some mapping in $\mathbf{m}$. Let m^i

be this mapping. We then have a mapping sequence $< m^1, \dots, m^n >$. Here, $Pr(D_T) = \Pi_{i=1}^n Pr(t_i) = \Pi_{i=1}^n Pr(m^i) = Pr(seq)$.

Because D_T is consistent with D_S and seq, the certain answer a must also be an answer tuple in $Q(D_T)$. We now prove for each tuple $a \in Q(D_T)$ and database D'_T that is consistent with D_S and seq, $a \in Q(D'_T)$ (so a is a certain answer with respect to seq). Suppose the i-th tuple $t_i \in D_T$ is involved in generating a. Because $Q(pD_T^{\vee})$ does not return null values, t_i's attributes that are not involved in m^i do not contribute to generating a. Tuple t'_i has the same value with t_i on all attributes that are involved in m^i. Thus, we can also generate a with t'_i and $a \in Q(D'_T)$.

II. Consider a mapping sequence $< m^1, \dots, m^n >$. Consider the possible database D_T where the i-th tuple has i as the value of key and is consistent with m^i and the i-th source tuple. Obviously, $Pr(seq) = Pr(D_T)$. We can prove tuples in $Q(D_T)$ are certain answers with respect to seq in the same way as in I. □

Lemma B.5. *Consider the following query*

```
Q: SELECT 'true'
   FROM T, J, T'
   WHERE T.a = J.a AND J.b = T'.b
```

Answering Q with respect to $\overline{pM}$ is #P-hard in the size of the data. □

Proof. We prove the lemma by reducing the *bipartite monotone 2-DNF problem* to the above problem.

Consider a bipartite monotone 2-DNF problem where variables can be partitioned into $X = \{x_1, \dots, x_m\}$ and $Y = \{y_1, \dots, y_n\}$, and $\varphi = C_1 \vee \cdots \vee C_l$, where each clause C_i has the form $x_j \wedge y_k, x_j \in X, y_k \in Y$. We construct the following query-answering problem.

P-mapping: Let $\overline{pM}$ be a schema p-mapping containing pM and pM'. Let $pM = (S, T, \mathbf{m})$ be a p-mapping where $S =< a >, T =< a' >$ and

$$\mathbf{m} = \{(\{(a, a')\}, .5), (\emptyset, .5)\}.$$

Let $pM' = (S', T', \mathbf{m}')$ be a p-mapping where $S' =< b >$, $T' =< b' >$ and

$$\mathbf{m}' = \{(\{(b, b')\}, .5), (\emptyset, .5)\}.$$

Source data: The source relation S contains m tuples: $x_1, \ldots, x_m$. The source relation S' contains n tuples: $y_1, \ldots, y_n$. The relation J contains l tuples. For each clause $C_i = x_j \wedge y_k$, there is a tuple (x_j, y_k) in J.

Obviously the construction takes polynomial time. We now prove the answer to the query is tuple true with probability $\frac{\#\varphi}{2^{m+n}}$, where $\#\varphi$ is the number of variable assignments that satisfy φ. We prove by showing that for each variable assignment $v_{x1}, \ldots, v_{xm}, v_{y1}, \ldots, v_{yn}$ that satisfies φ, there exists a mapping sequence *seq* such that true is a certain answer with respect to *seq* and the source instance, and vice versa.

For each variable assignment $v_{x1}, \ldots, v_{xm}, v_{y1}, \ldots, v_{yn}$ that satisfies φ, there must exist j and k such that $v_{xj} =$true, $v_{yk} =$true, and there exists $C_i = x_j \wedge y_k$ in φ. We construct the mapping sequence for pM such that for each $j \in [1, m]$, if $v_{xj} =$true, $m^j = (\{(a, a')\}, .5)$, and if $v_{xk} =$false, $m^j = (\emptyset, .5)$. We construct the mapping sequence for pM' such that for each $k \in [1, n]$, if $v_{yk} =$true, $m'^k = (\{(b, b')\}, .5)$, and if $v_{yk} =$false, $m'^k = (\emptyset, .5)$. Any target instance that is consistent with the source instance and $\{seq, seq'\}$ contains x_j in T and y_k in T'. Since $C_i \in \varphi$, J contains tuple (x_j, y_k) and so true is a certain answer.

For each mapping sequence *seq* for pM and *seq'* for pM', if true is a certain answer, there must exist $j \in [1, m]$ and $k \in [1, n]$, such that x_j is in any target instance that is consistent with S and *seq*, y_k is in any target instance that is consistent with S' and *seq'*, and there exists a tuple (x_j, y_k) in J. Thus, $m^j \in seq$ must be $(\{(a, a')\}, .5)$ and $m'^k \in seq'$ must be $(\{(b, b')\}, .5)$. We construct the assignments $v_{x1}, \ldots, v_{xm}, v_{y1}, \ldots, v_{yn}$ as follows. For each $j \in [1, m]$, if we have $m^j = (\{(a, a')\}, .5)$ in *seq*, $x_j =$true; otherwise, $x_j =$false. For each $k \in [1, n]$, if $m^k = (\{(b, b')\}, .5)$ in *seq*, $y_k =$true; otherwise, $y_k =$false. Obviously, the values of x_j and y_k are true, φ contains a term $x_j \wedge y_k$, and so φ is satisfied.

Counting the number of variable assignments that satisfy a bipartite monotone 2DNF boolean formula is #P-complete. Thus, answering query Q is #P-hard. □

Note that in Lemma B.5 Q contains two joins. Indeed, as stated in the following conjecture, we suspect that even for a query that contains a single join, query answering is also

#P-complete. The proof is still an open problem.

Conjecture B.6. *Let $\overline{pM}$ be a schema p-mapping containing pM and pM'. Let $pM = (S, T, \mathbf{m})$ be a p-mapping where $S =< a, b >, T =< c >$ and*

$$\mathbf{m} = \{(\{(a, c)\}, .5), (\{(b, c)\}, .5)\}.$$

Let $pM' = (S', T', \mathbf{m}')$ be a p-mapping where $S' =< d >$, $T' =< e >$ and

$$\mathbf{m}' = \{(\{(d, e)\}, .5), (\emptyset, .5)\}.$$

Consider the following query

```
Q: SELECT 'true'
   FROM T1, T2
   WHERE T1.c=T2.e
```

Answering Q with respect to $\overline{pM}$ is #P-hard in the size of the data.

□

B.2 Proofs for Other Results in Chapter 5

Theorem 5.12. Let $\overline{pM}$ be a schema p-mapping and let Q be an SPJ query.

Answering Q with respect to $\overline{pM}$ in by-table semantics is in PTIME in the size of the data and the mapping. □

Proof. It is trivial that Algorithm ByTable computes all by-table answers. We now consider its time complexity by examining the time complexity of each step.

Step 1: Assume for each target relation $T_i, i \in [1, l]$, the involved p-mapping contains n_i possible mappings. Then, the number of reformulated queries is $\Pi_{i=1}^{l} n_i$, polynomial in the size of the mapping.

Given the restricted class of mappings we consider, we can reformulate the query as follows. For each of T_i's attributes t, if there exists an attribute correspondence $(S.s, T.t)$ in m^i, we replace t everywhere with s; otherwise, the reformulated query returns empty result.

Let $|Q|$ be the size of Q. Thus, reformulating a query takes time $O(|Q|)$, and the size of the reformulated query does not exceed the size of Q.

Therefore, Step 1 takes time $O(\Pi_{i=1}^{l} n_i \cdot |Q|)$, which is polynomial in the size of the p-mapping and does not depend on the size of the data.

Step 2: Answering each reformulated query takes polynomial time in the size of the data and the number of answer tuples is polynomial in the size of the data. Because there are polynomial number of answer tuples and each occurs in the answers of no more than $\Pi_{i=1}^{l} n_i$ queries, summing up the probabilities for each answer tuple takes time $O(\Pi_{i=1}^{l} n_i)$. Thus, Step 2 takes polynomial time in the size of the mapping and the data. □

Theorem 5.13. Let pGM be a general p-mapping between a source schema $\bar{S}$ and a target schema $\bar{T}$. Let D_S be an instance of $\bar{S}$. Let Q be an SPJ query with only equality conditions over $\bar{T}$.

The problem of computing $Q^{table}(D_S)$ with respect to pGM is in PTIME in the size of the data and the mapping. □

Proof. We proceed in two steps to return all by-table answers. In the first step, for each $gm_i, i \in [1, n]$, we answer Q according to gm_i on D_S. The certain answer with regard to gm_i has probability $Pr(gm_i)$. SPJ queries with only equality conditions are conjunctive queries. According to [2], we can return all certain answers in polynomial time in the size of the data, and the number of certain answers is polynomial in the size of the data. Thus, the first step takes polynomial time in the size of the data and the mapping.

In the second step, we sum up the probabilities of each answer tuple. Because there are a polynomial number of answer tuples and each occurs in the answers of no more than n reformulated queries, this step takes polynomial time in the size of the data and the mapping. □

Lemma 5.17. Let $\overline{pM}$ be a schema p-mapping. Let Q be an SPJ query and Q_m be Q's mirror query with respect to $\overline{pM}$. Let D_S be the source database and D_T be the mirror target of D_S with respect to $\overline{pM}$.

Then, $t \in Q^{tuple}(D_S)$ if and only if $t \in Q_m(D_T)$ and t does not contain null value. □

Proof. If: We prove $t \in Q^{tuple}(D_S)$ by showing that we can construct a mapping sequence seq such that for each target instance D'_T that is consistent with D_S and seq, $t \in Q(D'_T)$.

Assume query Q (and so Q_m) contains n subgoals (*i.e.*, occurrences of tables in the FROM clause). Assume we obtain t by joining n tuples $t_1, \dots, t_n \in D_T$, each in the relation of a subgoal. Consider a relation R that occurs in Q. Assume $t_{k_1}, \dots, t_{k_l}, (k_1, \dots, k_l \in [1, n])$ are tuples of R (for different subgoals). Let $pM \in \overline{pM}$ be the p-mapping where R is the target and let S be the source relation of pM. For each $j \in [1, l]$, we denote the id value of t_{k_j} by $t_{k_j}.\mathsf{id}$, and the mapping value of t_{k_j} by $t_{k_j}.\mathsf{mapping}$. Then, t_{k_j} is consistent with the $t_{k_j}.\mathsf{id}$-th source tuple in S and the mapping $t_{k_j}.\mathsf{mapping}$.

We construct the mapping sequence of R for seq as follows: (1) for each $j \in [1, l]$, the mapping for the $t_{k_j}.\mathsf{id}$-th tuple is $t_{k_j}.\mathsf{mapping}$; (2) the rest of the mappings are arbitrary mappings in pM. To ensure the construction is valid, we need to prove that all tuples with the same id value have the same mapping value. Indeed, for every $j, h \in [1, l], j \neq h$, because t_{k_j} and t_{k_h} satisfy the predicate ($R_1.\mathsf{id} <> R_2.\mathsf{id}$ OR $R_1.\mathsf{mapping}{=}R_2.\mathsf{mapping}$) in Q_m, if $t_{k_j}.\mathsf{id}{=}t_{k_h}.\mathsf{id}$ then $t_{k_j}.\mathsf{mapping}{=}t_{k_h}.\mathsf{mapping}$.

We now prove for each target instance D'_T that is consistent with D_S and seq, $t \in Q(D'_T)$. For each $t_i, i \in [1, n]$, we denote by t'_i the tuple in D'_T that is consistent with the $t_i.\mathsf{id}$-th source tuple and the $t_i.\mathsf{mapping}$ mapping. We denote by $R(t_i), i \in [1, n]$, the subgoal that t_i belongs to. By the definition of mirror target and also because t does not contain null value, for each attribute of $R(t_i)$ that is involved in Q, t_i has non-null value, and so they are involved in the mapping $t_i.\mathsf{mapping}$. Thus, t'_i has the same value for these attributes. So t can be obtained by joining $t'_1, \dots, t'_n$ and $t \in Q(D'_T)$.

Only if: $t \in Q^{tuple}(D_S)$, so there exists a mapping sequence seq, such that for each D'_T that is consistent with D_S and seq, $t \in Q(D'_T)$. Consider such a D'_T. Assume t is obtained by joining tuples $t_1, \dots, t_n \in D'_T$, and for each $i \in [1, n]$, t_i is a tuple of subgoal R_i. Assume t_i is consistent with source tuple s_i and m_i. We denote by t'_i the instance in D_T whose id value refers to s_i and mapping value refers to m_i. Let $\bar{A}_i$ be the set of attributes of the subgoal R_i that are involved in the query. Since t is a "certain answer", all attributes in $\bar{A}_i$ must be involved in m_i. Thus, t_i and t'_i have the same value for these attributes, and all predicates in Q hold on $t'_1, \dots, t'_n$.

Because D'_T is consistent with D_S, for every pair of tuples t_i and $t_j, i, j \in [1, n]$, of the same relation, t_i and t_j are either consistent with different source tuples in D_S, or are consistent with the same source tuple and the same possible mapping. Thus, predicate R_1.id <> R_2.id OR R_1.mapping=R_2.mapping in the mirror query must hold on t'_i and t'_j. Thus, $t \in Q_m(D_T)$. □

Theorem 5.16: Given an SPJ query and a schema p-mapping, returning all by-tuple answers without probabilities is in PTIME with respect to data complexity. □

Proof. According to the previous lemma, we can generate all by-tuple answers by answering the mirror query on the mirror target. The size of the mirror target is polynomial in the size of the data and the size of the p-mapping, so answering the mirror query on the mirror target takes polynomial time. □

Lemma 5.19. Let $\overline{pM}$ be a schema p-mapping between $\bar{S}$ and $\bar{T}$. Let Q be a non-p-join query over $\bar{T}$ and let D_S be an instance of $\bar{S}$. Let $(t, Pr(t))$ be a by-tuple answer with respect to D_S and $\overline{pM}$. Let $\bar{T}(t)$ be the subset of $\mathbf{T}(D_S)$ such that for each $D \in \bar{T}(t)$, $t \in Q^{table}(D)$. The following two conditions hold:

1. $\bar{T}(t) \neq \emptyset$;

2. $Pr(t) = 1 - \Pi_{D \in \bar{T}(t), (t,p) \in Q^{table}(D)}(1 - p)$. □

Proof. We first prove (1). Let T be the relation in Q that is the target of a p-mapping and let pM be the p-mapping. Let *seq* be the mapping sequence for pM with respect to which t is a by-tuple answer. Because Q is a non-p-join query, there is no self join over T. So there must exist a target tuple, denoted by t_t, that is involved in generating t. Assume this target tuple is consistent with the i-th source tuple and a possible mapping $m \in pM$. We now consider the i-th tuple database D_i in $\mathbf{T}(D_S)$. There is a target database that is consistent with D_i and m, and the database also contains the tuple t_t. Thus, t is a by-table answer with respect to D_i and m, so $D_i \in \bar{T}(t)$ and $\bar{T}(t) \neq \emptyset$.

We next prove (2). We denote by $\bar{m}(D_i)$ the set of mappings in $\mathbf{m}$, such that for each $m \in \bar{m}(D_i)$, t is a certain answer with respect to D_i and m. For the by-table answer (t, p_i) with respect to D_i, obviously $p_i = \sum_{m \in \bar{m}(D_i)} Pr(m)$.

Let d be the number of tuples in D_S. Now consider a sequence $seq = < m^1, \ldots, m^d >$. As far as there exists $i \in [1, d]$, such that $m^i \in \bar{m}(D_i)$, t is a certain answer with respect to D_S and seq. The probability of all sequences that satisfy the above condition is $1 - \Pi_{i=1}^{d}(1 - \sum_{m \in \bar{m}(D_i)} Pr(m)) = 1 - \Pi_{D \in \bar{T}(t),(t,p) \in Q^{table}(D)}(1 - p)$. Thus, $Pr(t) = 1 - \Pi_{D \in \bar{T}(t),(t,p) \in Q^{table}(D)}(1 - p)$. □

Theorem 5.21. Let $\overline{pM}$ be a schema p-mapping and let Q be a non-p-join query with respect to $\overline{pM}$.

Answering Q with respect to $\overline{pM}$ in by-tuple semantics is in PTIME in the size of the data and the mapping. □

Proof. We first prove Algorithm NonPJoin generates all by-tuple answers. According to Lemma 5.19, we should first answer Q on each tuple database, and then compute the probabilities for each answer tuple. In Algorithm NonPJoin, since we introduce the id attribute and return its values, Step 2 indeed generates by-tuple answers for all tuple databases. Finally, Step 3 computes the probability according to (2) in the lemma.

We next prove Algorithm NonPJoin takes polynomial time in the size of the data and the size of the mapping. Step 1 goes through each possible mapping to add one more correspondence and thus takes linear time in the size of the mapping. In addition, the size of the revised mapping is linear in the size of the original mapping. Since Algorithm ByTable takes polynomial time in the size of the data and the mapping, so does Step 2 in Algorithm NonPJoin; in addition, the size of the result is polynomial in the size of the data and the mapping. Step 3 of the algorithm goes over each result tuple generated from Step 2, doing the projection and computing the probabilities according to the formula, so takes linear time in the size of the result generated from Step 2, and so takes also polynomial time in the size of the data and the mapping. □

Lemma 5.25. Let $\overline{pM}$ be a schema p-mapping. Let Q be a projected p-join query with

respect to $\overline{pM}$ and let $\bar{J}$ be a maximal p-join partitioning of Q. Let $Q_{J1}, \ldots, Q_{Jn}$ be the p-join components of Q with respect to $\bar{J}$.

For any instance D_S of the source schema of $\overline{pM}$ and result tuple $t \in Q^{tuple}(D_S)$, the following two conditions hold:

1. For each $i \in [1, n]$, there exists a single tuple $t_i \in Q_{Ji}^{tuple}(D_S)$, such that $t_1, \ldots, t_n$ generate t when joined together.

2. Let $t_1, \ldots, t_n$ be the above tuples. Then $Pr(t) = \Pi_{i=1}^{n} Pr(t_i)$. □

Proof. We first prove (1). The existence of the tuple is obvious. We now prove there exists a *single* such tuple for each $i \in [1, n]$. A join component returns all attributes that occur in Q and the join attributes that join partitions. The definition of maximal p-join partitioning guarantees for every two partitions, they are joined only on attributes that belong to relations involved in p-mappings. A projected-p-join query returns all such join attributes, so all attributes returned by the join component are also returned by Q. Thus, every two different tuples in the result of the join component lead to different query results.

We now prove (2). Since a partition in a join component contains at most one subgoal that is the target of a p-mapping in $\overline{pM}$, each p-join component is a non-p-join query. For each $i \in [1, n]$, let $\overline{seq}_i$ be the mapping sequences with respect to which t_i is a by-tuple answer. Obviously, $Pr(t_i) = \sum_{seq \in \overline{seq}_i} Pr(seq)$.

Consider choosing a set of mapping sequences $\bar{S} = \{seq_1, \ldots, seq_n\}$, where $seq_i \in \overline{seq}_i$ for each $i \in [1, n]$. Obviously, t is a certain answer with respect to $\bar{S}$. Because choosing different mapping sequences for different p-mappings are independent, the probability of $\bar{S}$ is $\Pi_{i=1}^{n} Pr(seq_i)$. Thus, we have

$$\begin{aligned} Pr(t) &= \sum_{seq_1 \in \overline{seq}_1, \ldots, seq_n \in \overline{seq}_n} \Pi_{i=1}^{n} Pr(seq_i) \\ &= \Pi_{i=1}^{n} \sum_{seq_i \in \overline{seq}_i} Pr(seq_i) \\ &= \Pi_{i=1}^{n} Pr(t_i) \end{aligned}$$

This proves the claim. □

Theorem 5.27. Let $\overline{pM}$ be a schema p-mapping and let Q be a projected-p-join query with respect to $\overline{pM}$.

Answering Q with respect to $\overline{pM}$ in by-tuple semantics is in PTIME in the size of the data and the mapping. □

Proof. We first prove Algorithm ProjectedPJoin generates all by-tuple answers for projected-p-join queries. First, it is trivial to verify that the partitioning generated by step 1 satisfies the two conditions of a p-join partitioning and is maximal. Then, step 2 and step 3 compute the probability for each by-tuple answer according to Lemma 5.25.

We next prove it takes polynomial time in the size of the mapping and in the size of the data. Step 1 takes time polynomial in the size of the query, and is independent of the size of the mapping and the data. The number of p-join components is linear in the size of the query and each is smaller than the original query. Since Algorithm NonPJoin takes polynomial time in the size of the data and the size of the mapping, Step 2 takes polynomial time in the size of the mapping and the size of the data too, and the size of each result is polynomial in size of the data and the mapping. Finally, joining the results from Step 2 takes polynomial time in the size of the results, and so also polynomial in the size of the data and the mapping. □

Theorem 5.30. Let $\overline{gpM}$ be a schema group p-mapping and let Q be an SPJ query. The mapping complexity of answering Q with respect to $\overline{gpM}$ in both by-table semantics and by-tuple semantics is in PTIME. □

Proof. We first consider by-table semantics and then consider by-tuple semantics. For each semantics, we prove the theorem by first describing the query-answering algorithm, then proving the algorithm generates the correct answer, and next analyzing the complexity of the algorithm.

By-table semantics: I. First, we describe the algorithm that we answer query Q with respect to the group p-mapping $\overline{gpM}$. Assume Q's `FROM` clause contains relations $T_1, \ldots, T_l$. For each $i \in [1, l]$, assume T_i is involved in group p-mapping gpM_i, which contains g_i groups

(if T_i is not involved in any group p-mapping, we assume it is involved in an identity p-mapping that corresponds each attribute with itself). The algorithm proceeds in five steps.

Step 1. We first partition all target attributes for $T_1, \ldots, T_l$ as follows. First, initialize each partition to contain attributes in one group (there are $\sum_{i=1}^{l} g_i$ groups). Then, for each pair of attributes a_1 and a_2 that occur in the same predicate in Q, we merge the two groups that t_1 and t_2 belong to. We call the result partitioning an *independence partitioning* with respect to Q and $\overline{gpM}$.

Step 2. For each partition p in an independence partitioning, if p contains attributes that occur in Q, we generate a sub-query of Q as follows. (1) The `SELECT` clause contains all variables in Q that are included in p, and an id column for each relation that is involved in p (we assume each tuple contains an identifier column id; in practice, we can use the key attribute of the tuple in place of id); (2) The `FROM` clause contains all relations that are involved in p; and (3) The `WHERE` clause contains only predicates that involve attributes in p. The query is called the *independence query* of p and is denoted by $Q(p)$.

Step 3. For each partition p, let $pM_1, \ldots, pM_n$ be the p-mappings for the group of attributes involved in p. For each $m^1 \in pM_1, \ldots, m^n \in pM_n$, rewrite $Q(p)$ regarding $m^1, \ldots, m^n$ and answer the rewritten query on the source data. For each returned tuple, assign $\Pi_{i=1}^{n} m^i$ as the probability and add n columns $\mathsf{mapping}_1, \ldots, \mathsf{mapping}_n$, where the column $\mathsf{mapping}_i, i \in [1, n]$, has the identifier for m_i as the value. Union all result tuples.

Step 4. Join the results of the sub-queries on the id attributes. Assume the result tuple t is obtained by joining $t_1, \ldots, t_k$, then $Pr(t) = \Pi_{i=1}^{k} Pr(t_k)$.

Step 5. For tuples that have the same values, assuming to be tuple t, for attributes on Q's returned attributes but different values for the mapping attributes, sum up their probabilities as the probability for the result tuple t.

II. We now prove the algorithm returns the correct by-table answers. For each result answer tuple a, we should add up the probabilities of the possible mappings with respect to which a is generated. This is done in Step 5. So we only need to show that given a specific combination of mappings, the first four steps generate the same answer tuples as with normal p-mappings. The partitioning in Step 1 guarantees that different independence queries involve different p-mappings and so Step 2 and 3 generate the correct answer for

each independence query. Step 4 joins results of the sub-queries on the id attributes; thus, for each source tuple, the first four steps generate the same answer tuple as with normal p-mappings. This proves the claim.

III. We next analyze the time complexity of the algorithm. The first two steps take polynomial time in the size of the mapping and the number of sub-queries generated by Step 2 is polynomial in the size of the mapping. Step 3 answers each sub-query in polynomial time in the size of the mapping and the result is polynomial in the size of the mapping. Step 4 joins a set of results from Step 3, where the number of the results and the size of each result is polynomial in the size of the mapping, so it takes polynomial time in the size of the mapping too and the size of the generated result is also polynomial in the size of the mapping. Finally, Step 5 takes polynomial time in the size of the result generated in Step 4 and so takes polynomial time in the size of the mapping. This proves the claim.

By-tuple semantics: First, we describe the algorithm that we answer query Q with respect to the group p-mapping $\overline{gpM}$. The algorithm proceeds in five steps and the first two steps are the same as in by-table semantics.

Step 3. For each partition p, let $pM_1, \ldots, pM_n$ be the p-mappings for the group of attributes involved in p. For each mapping sequence seq over $pM_1, \ldots, pM_n$, answer $Q(p)$ with respect to seq in by-tuple semantics. For each returned tuple, assign $Pr(seq)$ as the probability and add a column seq with an identifier of seq as the value.

Step 4. Join the results of the sub-queries on the id attributes. Assume the result tuple t is obtained by joining $t_1, \ldots, t_k$, then $Pr(t) = \Pi_{i=1}^{k} Pr(t_k)$.

Step 5. Let $t_1, \ldots, t_n$ be the tuples that have the same values, tuple t, for attributes on Q's returned attributes but different values for the seq attributes, sum up their probabilities as the probability for the result tuple t.

We can verify the correctness of the algorithm and analyze the time complexity in the same way as in by-table semantics. □

Proposition 5.31. For each $n \geq 1$, $\mathcal{M}_{ST}^{n+1} \subset \mathcal{M}_{ST}^{n}$. □

Proof. We first prove for each $n \geq 1$, $\mathcal{M}_{ST}^{n+1} \subseteq \mathcal{M}_{ST}^{n}$, and then prove there exists an instance in $\mathcal{M}_{ST}^{n}$ that does not have an equivalent instance in $\mathcal{M}_{ST}^{n+1}$.

(1) We prove $\mathcal{M}_{ST}^{n+1} \subseteq \mathcal{M}_{ST}^{n}$ by showing for each $(n+1)$-group p-mapping we can find a n-group p-mapping equivalent to it. Consider an instance $gpM = (S, T, \overline{pM}) \in \mathcal{M}_{ST}^{n+1}$, where $\overline{pM} = \{pM_1, \ldots, pM_{n+1}\}$. We show how we can construct an instance $gpM' \in \mathcal{M}_{ST}^{n}$ that is equivalent to gpM. Consider merging $pM_1 = (S_1, T_1, \mathbf{m}_1)$ and $pM_2 = (S_2, T_2, \mathbf{m}_2)$ and generating a probabilistic mapping $pM_{1-2} = (S_1 \cup S_2, T_1 \cup T_2, \mathbf{m}_{1-2})$, where $\mathbf{m}_{1-2}$ includes the Cartesian product of the mappings in $\mathbf{m}_1$ and $\mathbf{m}_2$. Consider the n-group p-mapping $gpM' = (S, T, \overline{pM'})$, where $\overline{pM'} = \{pM_{1-2}, pM_3, \ldots, pM_{n+1}\}$. Then, gpM and gpM' describe the same mapping.

(2) We now show how we can construct an instance in $\mathcal{M}_{ST}^{n}$ that does not have an equivalent instance in $\mathcal{M}_{ST}^{n+1}$. If S and T contain less than n attributes, $\mathcal{M}_{ST}^{n} = \emptyset$ and the claim holds. Otherwise, we partition attributes in S and T into $\{\{s_1\}, \ldots, \{s_{n-1}\}, \{s_n, \ldots, s_m\}\}$ and $\{\{t_1\}, \ldots, \{t_{n-1}\}, \{t_n, \ldots, t_l\}\}$. Without losing generality, we assume $m \leq l$. For each $i \in [1, n-1]$, we define

$$\mathbf{m}_i = \{(\{(s_i, t_i)\}, 0.8), (\emptyset, 0.2)\}.$$

In addition, we define

$$\mathbf{m}_n = \{(\{(s_n, t_n)\}, \frac{1}{(m-n+1)}), \ldots, (\{(s_m, t_n)\}, \frac{1}{(m-n+1)})\}.$$

We cannot further partition S into $n+1$ subsets such that attributes in different subsets correspond to different attributes in T. Thus, we cannot find a $(n+1)$-group p-mapping equivalent to it. □

Theorem 5.32. Given a p-mapping $pM = (S, T, \mathbf{m})$, we can find in polynomial time in the size of pM the maximal n and an n-group p-mapping gpM, such that gpM is equivalent to pM. □

Proof. We prove the theorem by first presenting an algorithm that finds the maximal n and the equivalent n-group p-mapping gpM, then proving the correctness of the algorithm, and next analyzing its time complexity.

I. We first present the algorithm that takes a p-mapping $pM = (S, T, \mathbf{m})$, finds the maximal n and the n-group p-mapping that is equivalent to pM.

Step 1. First, partition attributes in S and T. Initialize the partitions such that each contains a single attribute in S or T. Then for each attribute correspondence (s, t) occurring in a possible mapping, if s and t are in different partitions, merge the two partitions. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the result partitioning.
Step 2. For each partition $p_i, i \in [1, n]$, and each $m \in \mathbf{m}$, select the correspondences in m that involve only attributes in p_i, use them to construct a sub-mapping, and assign $Pr(m)$ to the sub-mapping. We compute the marginal probability of each sub-mapping.
Step 3. For each partition $p_i, i \in [1, n]$, examine if its possible mappings are independent of the possible mappings for the rest of the partitions. Specifically, for each partition $p_j, j > i$, if there exists a possible mapping m for p_i and a possible mapping m' for p_j, such that $Pr(m|m') \neq Pr(m)$, merge p_i into p_j. For the new partition p_j, update its possible sub-mappings and their marginal probabilities. Step 3 generates a set of partitions, each with a set of sub-mappings and their probabilities.
Step 4. Each partition generated in Step 3 is associated with a p-mapping. The set of all p-mappings forms the group p-mapping gpM that is equivalent to pM.

II. We now prove the correctness of the algorithm. It is easy to prove gpM is equivalent to pM. Assume gpM is an n-group p-mapping. We next prove n is maximal. Consider another group p-mapping gpM'. We now prove for each p-mapping in gpM', it either contains all attributes in a partition generated in Step 3 or contains none of them. According to the definition of group p-mapping, each p-mapping in gpM' must contain either all attributes or none of the attributes in a partition in $\mathcal{P}$. In addition, every two partitions in $\mathcal{P}$ that are merged in Step 3 are not independent and have to be in the same p-mapping in gpM' too. This proves the claim.

III. We next consider the time complexity of the algorithm. Let m be the number of mappings in pM, and a be the minimum number of attributes in R and in S. Step 1 considers each attribute correspondence in each possible mapping. A mapping contains no more than a attribute correspondences, so Step 1 takes time $O(ma)$. Step 2 considers each possible mapping for each partition to generate sub-mappings. The number of partitions cannot exceed a, so Step 2 also takes time $O(ma)$. Step 3 considers each pair of partitions. and takes time $O(ma^2)$. Finally, Step 4 outputs the results and takes time $O(ma)$. Overall,

the algorithm takes time $O(ma^2)$, which is polynomial in the size of the full-distribution instance. □

Theorem 5.36. Let $\overline{pC}$ be a schema p-correspondence, and Q be an SPJ query. Then, Q is p-mapping independent with respect to $\overline{pC}$ if and only if for each $pC \subseteq \overline{pC}$, Q is a single-attribute query with respect to pC. □

Proof. We prove for the case when there is a single p-correspondence in $\overline{pC}$ and it is easy to generalize our proof to the case when there are multiple p-correspondences in $\overline{pC}$.

If: Let pM_1 and pM_2 be two p-mappings over S and T where $pC(pM_1) = pC(pM_2)$. Let D_S be a database of schema S. Consider a query Q over T. Let t_j be the only attribute involved in query Q. We prove $Q(D_S)$ is the same with respect to pM_1 and pM_2 in both by-table and by-tuple semantics.

We first consider by-table semantics. Assume S has n attributes $s_1, \ldots, s_n$. We partition all possible mappings in pM_1 into $\bar{m}_0, \ldots, \bar{m}_n$, such that for any $m \in \bar{m}_i, i \in [1, n]$, m maps attribute s_i to t_j, and for any $m \in \bar{m}_0$, m does not map any attribute in S to t_j. Thus, for each $i \in [1, n], Pr(\bar{m}_i) = Pr(c_{ij})$.

Consider a tuple t. Assume t is an answer tuple with respect to a subset of possible mappings $\bar{m} \subseteq \mathbf{m}$. Because Q contains only attribute t_j, for each $i \in [0, n]$, either $\bar{m}_i \subseteq \bar{m}$ or $\bar{m}_i \cap \bar{m} = \emptyset$. Let $\bar{m}_{k_1}, \ldots, \bar{m}_{k_l}, k_1, \ldots, k_l \in [0, n]$, be the subsets of $\bar{m}$ such that $\bar{m}_{k_j} \subseteq \bar{m}$ for any $j \in [1, l]$. We have

$$Pr(t) = \sum_{i=1}^{l} Pr(\bar{m}_{k_i}) = \sum_{i=1}^{l} Pr(c_{k_i j}).$$

Now consider pM_2. We partition its possible mappings in the same way and obtain $\bar{m}'_0, \ldots, \bar{m}'_n$. Since Q contains only attribute t_j, for each $i \in [0, n]$, the result of Q with respect to $m' \in \bar{m}'_i$ is the same as the result with respect to $m \in \bar{m}_i$. Therefore, the probability of t with respect to pM_2 is

$$Pr(t)' = \sum_{i=1}^{l} Pr(\bar{m}'_{k_i}) = \sum_{i=1}^{l} Pr(c_{k_i j}).$$

Thus, $Pr(t) = Pr(t)'$ and this proves the claim.

We can prove the claim for by-tuple semantics in a similar way where we partition mapping sequences. We omit the proof here.

Only if: We prove by showing that for every query Q that contains more than one attribute in a relation being involved in a p-correspondence, there exist p-mappings pM_1 and pM_2 and source instance D_S, such that $Q(D_S)$ obtains different results with respect to pM_1 and pM_2.

Assume query Q contains attributes a' and b' of T. Consider two p-mappings pM_1 and pM_2, where

$$\begin{aligned} pM_1 &= \{(\{(a,a'),(b,b')\},.5),(\{(a,a')\},.3),(\{(b,b')\},.2)\} \\ pM_2 &= \{(\{(a,a'),(b,b')\},.6),(\{(a,a')\},.2),(\{(b,b')\},.1),(\emptyset,.1)\} \end{aligned}$$

One can verify that $pC(pM_1) = pC(pM_2)$.

Consider a database D_S, such that for each tuple of the source relation in pM_1 and pM_2, the values for attributes a and b satisfy the predicates in Q. Since only when the possible mapping $\{(a,a'),(b,b')\}$ is applied can we generate valid answer tuples, but the possible mapping $\{(a,a'),(b,b')\}$ has different probabilities in pM_1 and pM_2, $Q(D_S)$ obtains different results with respect to pM_1 and pM_2 in both semantics. □

Corollary 5.38. Let $\overline{pC}$ be a schema p-correspondence, and Q be a p-mapping independent SPJ query with respect to $\overline{pC}$. The mapping complexity of answering Q with respect to $\overline{pC}$ in both by-table semantics and by-tuple semantics is in PTIME. □

Proof. **By-table:** We revise algorithm BY-TABLE, which takes polynomial time in the size of the schema p-mapping, to compute answers with respect to schema p-correspondences. At the place where we consider a possible mapping in the algorithm, we revise to consider a possible attribute correspondence. Obviously the revised algorithm generates the correct by-table answers and takes polynomial time in the size of the mapping.

By-tuple: We revise the algorithm in the proof of Theorem 5.15, which takes polynomial time in the size of the schema p-mapping, to compute answers with respect to schema p-correspondences. Everywhere we consider a possible mapping in the algorithm, we revise

to consider a possible attribute correspondence. Obviously the revised algorithm generates the correct by-tuple answers and takes polynomial time in the size of the mapping. □

Theorem 5.40. There exists a schema p-mapping $\overline{pM}$ and a query Q, such that answering Q with respect to $\overline{pM}$ in by-table semantics takes exponential time in size of $\overline{pM}$'s Bayes-Net representation. □

Proof. Consider pM in Example 5.39. Consider the following query:

```
SELECT t1, ..., tn
FROM T
```

Consider a source instance D_S with one tuple, where each attribute value in the tuple is distinct. There are 2^n tuples in $Q^{table}(D_S)$. To enumerate all these answers takes time $O(2^n)$, which is exponential in the size of pM's Bayes-Net representation. □

Theorem 5.41. Let $\overline{pCM}$ be a schema probabilistic complex mapping between schemas $\bar{S}$ and $\bar{T}$. Let D_S be an instance of $\bar{S}$. Let Q be an SPJ query over $\bar{T}$. The data complexity and mapping complexity of computing $Q^{table}(D_S)$ with respect to $\overline{pCM}$ are PTIME. The data complexity of computing $Q^{tuple}(D_S)$ with respect to $\overline{pCM}$ is #P-complete. The mapping complexity of computing $Q^{tuple}(D_S)$ with respect to $\overline{pCM}$ is in PTIME. □

Proof. We prove the theorem by showing that we can construct a normal schema p-mapping from $\overline{pCM}$ and answer a query with respect to the normal p-mapping. For each $pCM \in \overline{pCM}$ between source $S(s_1, \ldots, s_m)$ and target $T(t_1, \ldots, t_n)$, we construct a normal p-mapping $pM = (S', T', m)$ as follows. The source S' contains all elements of the power set of $\{s_1, \ldots, s_m\}$ and the target T' contains all elements of the power set of $\{t_1, \ldots, t_n\}$. For each complex mapping $cm \in pCM$, we construct a mapping m such that for each set correspondence between S and T in cm, m contains an attribute correspondence between the corresponding sets in S' and T'. Because each attribute occurs in one correspondence in cm, m is a one-to-one mapping. The result pM contains the same number of possible mappings and each mapping contains the same number of correspondences as pCM. We denote the result schema p-mapping by $\overline{pM}$. Query answering with respect to $\overline{pCM}$ gets

the same result as with respect to $\overline{pM}$ and so the complexity results for normal schema p-mappings carry over. □

Theorem 5.42. Let $\overline{cpM}$ be a schema conditional p-mapping between $\bar{S}$ and $\bar{T}$. Let D_S be an instance of $\bar{S}$. Let Q be an SPJ query over $\bar{T}$. The problem of computing $Q^{tuple}(D_S)$ with respect to $\overline{cpM}$ is in PTIME in the size of the mapping and #P-complete in the size of the data. □

Proof. By-tuple query answering with respect to schema conditional p-mappings is essentially the same as that with respect to normal p-mappings, where for each source tuple, we first decide which condition it satisfies and then consider applying possible mappings associated with that condition. Thus, the complexity of by-tuple query-answering with respect to normal schema p-mappings carries over. □

VITA

Xin Dong was born and raised in Tianjin, the fourth largest city in China. She was fascinated by computers and wrote her first "Hello World" program at eight, when personal computers were first introduced to China. In the following 23 years, she learned programming, attended various programming contests, received a Bachelor' degree in Computer Science from Nankai University in 1998, a Master's degree in Computer Science from Peking University in 2001, and a Ph.D. in Computer Science from University of Washington in 2007.

Beginning October 2007, she will join AT&T Research Lab. Till the point of this dissertation, three fourths of her life were spent with computers, and this percentage is likely to grow in the future.