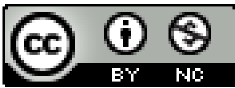
DATA ANALYTICS: INTEGRATION AND PRIVACY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Steven E. Whang
June 2012

© 2012 by Steven Euijong Whang. All Rights Reserved.
Re-distributed by Stanford University under license with the author.



This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/us/

This dissertation is online at: http://purl.stanford.edu/ch599mk0589

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jurij Leskovec

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Jennifer Widom

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Data analytics has become an extremely important and challenging problem in disciplines like computer science, biology, medicine, finance, and homeland security. As massive amounts of data are available for analysis, scalable integration techniques become important. At the same time, new privacy issues arise where one's sensitive information can easily be inferred from the large amounts of data.

In this thesis, we first cover the problem of *entity resolution* (ER), which identifies database records that refer to the same real-world entity. The recent explosion of data has now made ER a challenging problem in a wide range of applications. We propose scalable ER techniques and new ER functionalities that have not been studied in the past. We also view ER as a black-box operation and provide general techniques that can be used across applications.

Next, we introduce the problem of managing *information leakage*, where one must try to prevent important bits of information from being resolved by ER, to guard against loss of data privacy. As more of our sensitive data gets exposed to a variety of merchants, health care providers, employers, social sites and so on, there is a higher chance that an adversary can "connect the dots" and piece together our information, leading to even more loss of privacy. We propose a measure for quantifying information leakage and use "disinformation" as a tool for containing information leakage.

Acknowledgements

I would first like to thank my advisor Prof. Hector Garcia-Molina for the wonderful experience I had at Stanford. You are the best advisor I can ever imagine. I started without much experience in research, but you have trained me to become a professional researcher that is ready to tackle the challenging and exciting problems in the real world. I have learned how to be unbiased when solving problems and provide original solutions with impact. I have trained to become persistent enough to navigate through rough waves. I have also learned the art of communicating my research with colleagues. I have always appreciated the deep respect you have on your students. Throughout my PhD, I was truly self-motivated in my research while getting just the right advice for each pitfall I encountered. I probably made a lot of mistakes, but you never discouraged me. I am grateful to have trained under a true master of education. When I become a professor, I hope to be half as good as you are both in terms of research and personality.

I thank Prof. Gio Wiederhold for caring about me since my father was your student. I really appreciate the genuine advice you gave me before and during my graduate studies. I loved our occasional InfoLab retreats to San Francisco for various classical performances. Those were the times I could broaden my scope and have a more balanced view of life. I also enjoyed assisting you in maintaining the Gates Computer Science Museum. I will always remember the first Google server that used to be displayed in the basement. I also thank Voy Wiederhold for always being such a warm presence in the lab and also throughout my life.

I thank Prof. Jennifer Widom for being the gold standard in research. You have set the bars high for writing papers and giving talks, but it was well worth the effort to meet those standards. I really appreciate all the guidance you gave me. I thank Prof. Jeffrey

Ullman for being the perfect computer scientist who taught us how exactly we should be doing theoretical and practical computer science. It was an honor to be working in an office right next to yours. I thank Prof. Jure Leskovec for kindly agreeing to be in my reading committee. I really admire your hard-working ethic and would like to be as diligent as you are. I thank Prof. Kincho Law for chairing my defense. I thank Andreas Paepcke for giving me fresh perspectives in research with his vast creativity. I thank Marianne Siroker for her impeccable management of the InfoLab and cannot imagine the lab running without her. I thank Andy Kacsmar for solving all my technical problems.

I thank my wonderful InfoLab colleagues. I first thank David Menestrina for being my long-time office mate. I especially thank you for being such a kind and encouraging TA when I took CS245 as a Master's student and was trying to make a good impression in class. I thank Petros Venetis for being a great office mate and appreciate your help in research and the encouragements you gave me. I wish the best in your thesis work. In addition, there are many names I would like to thank and mention: Parag Aggrawal, Yannis Antonellis, Omar Benjelloun, Anish Das Sarma, Heng Gong, Stephen Guo, Zoltán Gyöngyi, Paul Heymann, Robert Ikeda, Jawed Karim, Hideki Kawai, Myunghwan Kim, Georgia Koutrika, Bobji Mungamuru, Raghotham Murthy, Aditya Parameswaran, Hyunjung Park, Eldar Sadikov, Semih Salihoglu, Qi Su, Sutthipong Thavisomboon, Martin Theobald, Gary Wesley, and Jaewon Yang. Many of you have already graduated and are having awesome careers. I would like to follow your footsteps.

I also thank my other friends within the Gates building: James Chen, Philip Guo, Sungpack Hong, Eric Kao, Minsung Kim, Sangkyun Kim, Seungbeom Kim, Aleksandra Korolova, Jinsung Kwon, Honglak Lee, HyoukJoong Lee, Jungwoo Lee, Suin Lee, Austen McDonald, Sung Hee Park, Jiwon Seo, Dongjun Shin, Qiqi Yan, and Richard Yoo.

I thank my KAIST Alumni buddies, who have been a great support throughout my PhD. I also thank my numerous bible study friends for making me grow spiritually in God and teaching me what is really important in life. I especially thank my Korean Christian Fellowship and Menlo Park Presbyterian Church friends.

I thank my parents who have sacrificed so much for my education. Without them, I would not have been able to study at Stanford. I love them from the bottom of my heart.

Contents

Abstract **iv**

Acknowledgements **v**

1 Introduction **1**

1.1 Basic ER Model 4

1.2 ER Measure 5

1.3 Structure of This Thesis 5

1.3.1 Data Integration 6

1.3.2 Data Privacy 7

1.4 Related Work 8

2 Pay-As-You-Go Entity Resolution **10**

2.1 Framework 13

2.1.1 ER Model 13

2.1.2 Pay-As-You-Go Model 14

2.2 Sorted List of Record Pairs 16

2.2.1 Generation 17

2.2.2 Use 19

2.3 Hierarchy of Record Partitions 21

2.3.1 Generation 22

2.3.2 Use 25

2.3.3 General Incremental Property 28

2.4 Ordered List of Records 31

2.4.1 Generation 32
2.4.2 Use 37
2.5 Using Multiple Hints 38
2.6 Determining Which Hint to Use 39
2.7 Experimental Results 39
2.7.1 Experimental setting 41
2.7.2 Hint Benefit 44
2.7.3 Hint Overhead 47
2.7.4 Choosing the Number of Levels 52
2.7.5 Sampling Performance 53
2.7.6 Using Weights on Partition Hierarchy Levels 54
2.7.7 Non-incremental ER algorithms 55
2.7.8 Early Termination on Large Datasets 56
2.7.9 Scalability of Generating Hints 58
2.8 Related Work 59
2.9 Conclusion 60

3 Evolving Rules **61**
3.1 Match-based Evolution 64
3.1.1 Match-based Clustering Model 64
3.1.2 Properties 66
3.1.3 Materialization 79
3.1.4 Rule Evolution 79
3.2 Materialization Strategies 86
3.3 Distance-based Evolution 86
3.3.1 Distance-based Clustering Model 87
3.3.2 Properties 88
3.3.3 Rule Evolution 93
3.4 Experimental Evaluation 94
3.4.1 Experimental Setting 95
3.4.2 Evaluating IO costs 97

3.4.3 Rule Evolution Efficiency . . . 98
3.4.4 Common Rule Strictness . . . 100
3.4.5 Materialization Overhead . . . 101
3.4.6 Total Runtime . . . 103
3.4.7 Without the Properties . . . 104
3.5 Related Work . . . 106
3.6 Conclusion . . . 107

4 Joint Entity Resolution **109**
4.1 Framework . . . 112
4.1.1 ER Model . . . 113
4.1.2 Joint ER Model . . . 115
4.1.3 Physical Execution . . . 115
4.2 Scheduler . . . 119
4.2.1 Influence Graph . . . 119
4.2.2 Execution Plan . . . 121
4.2.3 Exploiting the Influence Graph . . . 128
4.3 Joint ER Processor . . . 129
4.3.1 Concurrent Set . . . 129
4.3.2 Fixed-Point Set . . . 130
4.3.3 Joint ER Algorithm . . . 132
4.3.4 Expander Function . . . 132
4.4 ER Algorithm Training . . . 136
4.4.1 Match Rule . . . 136
4.4.2 Training . . . 137
4.4.3 State-based Training . . . 138
4.5 Experimental Results . . . 139
4.5.1 Synthetic Data Experiments . . . 139
4.5.2 Real Data Experiments . . . 147
4.5.3 Training Accuracy . . . 152
4.6 Related Work . . . 156

4.7 Conclusion 157

5 Entity Resolution with Negative Rules **159**

5.1 ER-N Model 162

5.1.1 ER 162

5.1.2 Negative Rules 165

5.1.3 ER-N 165

5.1.4 Resolving Inconsistencies 166

5.2 The GNR Algorithm 168

5.3 Properties for the Rules 171

5.3.1 Match and Merge Functions 172

5.3.2 Negative Rules 174

5.4 The ENR Algorithm 175

5.5 Precision and Recall 178

5.5.1 Experimental Setting 178

5.5.2 Rules 179

5.5.3 Strategies 180

5.5.4 Other Negative Rules 183

5.5.5 Choosing Negative Rules 185

5.6 Performance 187

5.6.1 Human Effort 187

5.6.2 System Runtime 189

5.6.3 Without the Properties 191

5.7 Related Work 192

5.8 Conclusion 194

6 A Model for Quantifying Information Leakage **195**

6.1 Information Leakage Measure 197

6.1.1 Correctness 199

6.1.2 Completeness 200

6.1.3 Adversary Confidence 201

6.1.4 Adversary Effort on Data Analysis 203

6.2 Relationship to Other Measures . . . 205
6.2.1 k-anonymity . . . 206
6.2.2 l-diversity . . . 208
6.3 Applications . . . 209
6.3.1 Releasing Critical Information . . . 209
6.3.2 Releasing Disinformation . . . 211
6.3.3 Enhancing a Composite Record . . . 212
6.4 Computation . . . 213
6.4.1 Exact Solution using Constant Weights . . . 214
6.4.2 Approximation using Arbitrary Weights . . . 217
6.5 Experiments . . . 218
6.5.1 Trends . . . 219
6.5.2 Accuracy of Approximate Algorithm . . . 220
6.5.3 Runtime Performance . . . 220
6.6 Related Work . . . 221
6.7 Conclusion . . . 222

7 Disinformation Techniques for Entity Resolution 223
7.1 Framework . . . 227
7.1.1 ER Model . . . 227
7.1.2 Pairwise Approach for Merging Clusters . . . 228
7.1.3 Disinformation Problem . . . 229
7.1.4 Monotonicity . . . 234
7.2 Planning Algorithms . . . 235
7.2.1 Exact Algorithm for 1-Level Plans . . . 235
7.2.2 Approximate Algorithm for 1-Level Plans . . . 237
7.2.3 Heuristics for General Plans . . . 238
7.3 Creating New Records . . . 241
7.3.1 Euclidean Space . . . 241
7.3.2 Non-Euclidean Space . . . 242
7.4 Experiments . . . 244

7.4.1 Synthetic Data Experiments 244
7.4.2 Real Data Experiments 257
7.5 Related Work 262
7.6 Conclusion 263

8 Conclusion **264**
8.1 Summary 264
8.2 Future Work 266

Bibliography **269**

List of Tables

1.1 A set of records representing persons 4

3.1 Match Rules 96
3.2 ER and rule evolution algorithms tested 97
3.3 Basic operations in blocking ER framework 98
3.4 Decomposition of ER processes for one rule evolution 98
3.5 ER algorithm and rule evolution runtimes 99
3.6 Time overhead (ratio to old ER algorithm runtime) and space overhead (ratio to old ER result) of rule materialization, 3K records 102
3.7 Runtime and accuracy results for ER algorithms without the properties . . . 106

4.1 Papers and Venues 111
4.2 Execution Plan Syntax 122
4.3 Parameters for generating synthetic data 140
4.4 Physical Execution Summary on Spock Data 152
4.5 Cora data match rules 153
4.6 Trained weights 154
4.7 Trained weights without states 155
4.8 Accuracy results of training with states 155
4.9 Accuracy results of training without states 156

6.1 Database of Patients (R) 206
6.2 3-Anonymous Version of Table 6.1 (R_a) 207
6.3 Background Information (R_b) 208

6.4 Parameters for Data Generation 218
6.5 Information Leakage Comparison 220

7.1 Camera Rumor Records 225
7.2 Parameters for generating synthetic data 245
7.3 Disinformation Plan Algorithms 247
7.4 Decrease in confusion (%) with sampling 253
7.5 Decrease in confusion (%) with restrictions 254

List of Figures

2.1 Pay-as-you-go approach of ER . . . 11
2.2 Pay-As-You-Go ER Framework . . . 16
2.3 A partition hierarchy hint for resolving R . . . 21
2.4 Hints to generate and ER algorithms to run . . . 40
2.5 Recall of ER algorithms using hints against work or runtime, 3K shopping/hotel records . . . 48
2.6 Time and space hint construction overhead depending on the type of hint, 3K shopping/hotel records . . . 49
2.7 Construction time versus time to obtain 0.8 recall, 3K shopping records . . . 52
2.8 Construction time impact on hint payoff point, 3K shopping records . . . 53
2.9 Number of levels impact on recall, 3K shopping records . . . 54
2.10 List of pairs using sampling, 3K shopping records . . . 55
2.11 Weights impact on accuracy, 3K shopping records . . . 56
2.12 Non-incremental algorithm accuracy, 3K shopping records . . . 57
2.13 Runtime and recall for different schemes, 2M shopping records . . . 58
2.14 Hint generation time (secs), 2M shopping records . . . 59

3.1 Records to resolve . . . 62
3.2 Evolving from rule B_1 to rule B_3 . . . 62
3.3 ER Algorithms satisfying properties . . . 74
3.4 Degree of change impact on runtime, 3K shopping records . . . 101
3.5 Scalability, 1M shopping records . . . 104

4.1 System Architecture . . . 113

4.2 The Physical Execution (((R), (S)),((),(S, T))) 116
4.3 An Influence Graph 120
4.4 Linear structure results 142
4.5 Random structure results 143
4.6 Value similarity weights versus iterations 144
4.7 Threshold versus expander function performance 145
4.8 Number of duplicates versus expander function performance 146
4.9 Number of processors versus record comparisons 146
4.10 Blocking scenario versus record comparisons 148
4.11 Scalability results on the Spock dataset 151
4.12 Scalability results when running R-Swoosh on P 152

5.1 A list of people 160
5.2 Package formation 176
5.3 Precision and recall for different strategies 181
5.4 Distribution of base records per output record 181
5.5 Precision and recall for various negative rules 184
5.6 Human effort versus recall 185
5.7 List of negative rules 186
5.8 Results for various combinations of negative rules 186
5.9 Human effort 188
5.10 Binary density impact on human effort 189
5.11 Runtime decomposition 190
5.12 Scalability 190
5.13 Result sizes and similarities 192

6.1 Information Leakage with Entity Resolution 197
6.2 Self and Linkage Disinformation 212
6.3 Trends and Scalability 219

7.1 Cost Graph 230
7.2 Robustness of the HC_S and SN algorithms 248

7.3 Comparison of disinformation algorithms 249
7.4 Window size impact on confusion for SN 250
7.5 Robustness of the HC_S algorithm 250
7.6 Entity distance impact on confusion 251
7.7 Universal disinformation 252
7.8 Number of dimensions impact on confusion 255
7.9 Entities with fewer duplicates 256
7.10 Scalability of disinformation generation 257
7.11 Hotel data confusion results 260
7.12 Shopping data confusion results 262

Chapter 1

Introduction

Data analytics has become an extremely important and challenging problem in disciplines like computer science, biology, medicine, finance, and homeland security. This problem involves several aspects. First, large volumes of data must be imported and stored relying on cleansing and filtering techniques. Next, sophisticated algorithms are used to analyze the data and extract "useful" information. Finally, various user interfaces can be used to visualize and understand the data.

Analyzing large amounts of data has become an extremely hard task. Everywhere you look around, the quantity of information in the world is large and increasing exponentially. For example, various social networks like Facebook generate terabytes of data per day in the form of photos, videos, wall posts, etc., and will generate significantly more data in the near future. The size of today's data is unprecedented and cannot simply be analyzed with conventional data management techniques.

Nevertheless, being able to efficiently "make sense" out of big data is becoming even more important than ever in various areas. In computer science, Web-scale data needs to be analyzed in order to understand global trends and user behavior. In biology, interpreting massive amounts of DNA and RNA sequencing data is essential for understanding complex biological systems. Already, the explosive growth of sequencing data has exceeded the growth rate of storage capacity. In medicine, health devices generate huge amounts of data that reflect the condition of patients by monitoring their sleep, heart rate, and other health conditions. In finance, the stock market generates immense quantities of transaction

data that can help companies maximize profit. In homeland security, the U.S. government receives more terabytes each day than the amount of text in the Library of Congress. This data may then be analyzed for identifying potential threats to the country. While we have only listed a few examples, there are many other areas that are starting to exploit large amounts of information as well.

One of the main challenges in data analytics is to collect data from multiple sources and combine them together so that data analysts can access and manipulate the information in a unified way. When combining different data, a fundamental problem is identifying which pieces of information describe the same real-world entity. In this thesis, we thus focus on Entity Resolution (ER) (sometimes referred to as deduplication), which is the process of "matching" and "merging" database records judged to represent the same real-world entity. To illustrate ER, mailing lists may contain multiple entries representing the same physical address, but each record may be slightly different, e.g., containing different spellings or missing some information. As a second example, a comparative shopping website may aggregate product catalogs from multiple merchants.

Records that refer to the same real-world entity are said to *match* with each other. Identifying records that match poses challenging problems because typically there are no unique identifiers across sources. For example, when we are trying to identify the records that represent the same product, merchant catalogs that contain product information may use different product codes. A given record may appear in a different way in each source, and there is a fair amount of guesswork in determining which records match. Deciding if records match is often computationally expensive, e.g., may involve finding maximal common subsequences in two strings.

An ER process sometimes combines the matching records by merging them together. How to actually merge records is often application dependent. For example, let us suppose different prices appear in two records to be merged; in some cases we may wish to keep both prices while in others we may want to pick just one as the "consolidated" price.

The first contribution of this thesis is a set of scalable and general solutions for ER. Scaling ER on very large datasets is essential for applications using big data. For example, a people search engine may have to quickly resolve hundreds of millions of people records collected from the Web to stay in business. We view either the entire ER process or the

matching and merging of records as black-box operations and provide general techniques that can be used in a wide range of applications.

One way to solve the scalability problem is to produce approximate ER results using significantly less time. In practice, applications may need to resolve large data sets efficiently, but do not require the ER result that would be produced from running ER exhaustively on the entire data. For example, people data from the Web may simply be too large to completely resolve with a reasonable amount of work. As another example, real-time applications may not be able to tolerate any ER processing that takes longer than a certain amount of time. We thus study a pay-as-you-go approach for ER where the goal is to maximize the ER progress using a limited amount of work.

In addition, there are scalability challenges for specific functionalities of ER. First, ER is not a one-time process, but may be constantly improved as the data, schema and application are better understood. Hence we study the problem of incrementally updating an ER result where the logic of the ER algorithm is improved frequently. Second, ER may involve different types of records (e.g., authors, publications, institutions, venues), and resolving records of one type can impact the resolution of other types of records. We thus study the problem of jointly resolving multiple types of records together using efficient scheduling. Finally, ER results may contain "inconsistencies," which are patterns that should not appear in a final ER result. Inconsistencies may occur either due to mistakes by the match and merge function writers or changes in the application semantics. We thus study the problem of ER with integrity constraints that can be used to produce ER results without the inconsistencies. All of these functionalities of ER must be performed in a scalable fashion as well.

The second contribution of this thesis is a study of data analytics from a privacy point of view. The flip side of data integration is that there is now a danger of one's personal information being more exposed to the public. For example, life insurers are exploring ways to predict the life spans of their customers by piecing together health-related personal information on the Web [100]. As another example, people search engines like Spock.com [92] resolve hundreds of millions of person records crawled from the Web to create one profile per person and thus reveal more personal information.

The first step for solving the data privacy problem is to quantify *information leakage*.

	Name	**Phone**	**E-mail**
r_1	John Doe	235-2635	jdoe@yahoo
r_2	J. Doe	234-4358	
r_3	John D.	234-4358	jdoe@yahoo

Table 1.1: A set of records representing persons

As more of our sensitive data gets exposed to a variety of merchants, health care providers, employers, social sites and so on, there is a higher chance that an *adversary* can use ER to "connect the dots" and piece together our information, leading to even more loss of privacy. The more complete the integrated information, the more our privacy is compromised.

Once we are able to quantify information leakage, the next step is to develop techniques for managing information leakage. We assume that an *agent* has some sensitive information that the adversary is trying to obtain. For example, a camera company (the agent) may secretly be developing its new camera model, and a user (the adversary) may want to know in advance the detailed specs of the model. The agent's goal is to disseminate false information to "dilute" what is known by the adversary.

1.1 Basic ER Model

We define a basic model for ER. This model, or its variants, are used throughout this thesis. We start with a set of records $R = \{r_1, r_2, \ldots, r_n\}$. We do not assume any particular form or data model for representing records. An ER algorithm takes R as its input and clusters together records that the algorithm decides are likely to represent the same real-world entity. The ER output can thus be viewed as a simple partition of the input set.

To illustrate an ER algorithm, consider the records of Table 1.1. Note that the structure of these records and the way we determine if records refer to the same real-world entity (via a pairwise comparison) are part of the example. For this example, the ER algorithm works as follows: A function compares the name, phone, and email values of pairs of records. If the names are very similar (above some threshold), the records are said to match. The records also match if the phone and email are identical.

For our sample data, the ER algorithm may determine that r_1 and r_2 match, but r_3 does

not match either r_1 or r_2. For instance, the function may determine that "John Doe" and "J. Doe" are similar to each other, but "John D." is not similar to anything. Thus, r_1 and r_2 are clustered together to form the partition $\{\{r_1, r_2\}, \{r_3\}\}$ where the inner curly brackets denote the clusters in the output partition.

Since records in an output cluster are meant to represent a single real-world entity, the cluster can be considered a "composite" new record. In some cases we may apply a merge operation to actually generate the composite record. In our example above, suppose that the ER algorithm combines the names into a "normalized" representative, and performs a set-union on the emails and phone numbers. Then the records r_1 and r_2 will merge into the new record $\langle r_1, r_2\rangle$ as shown below. (Here we denote a merged record with angle brackets.)

$\langle r_1, r_2\rangle$	John Doe	234-4358, 235-2635	jdoe@yahoo

In this case, the ER result is the set of records $\{\langle r_1, r_2\rangle, r_3\}$. Notice that the ER algorithm can now iteratively match $\langle r_1, r_2\rangle$ with r_3 and merge them together because they have an identical phone and email pair.

1.2 ER Measure

Our basic method for evaluating an ER algorithm is to compare the ER result with a "Gold Standard". Suppose that the Gold Standard G contains the set of record pairs that correctly refer to the same entity while set S contains the matching pairs produced by our ER algorithm. Then the precision Pr is $\frac{|G \cap S|}{|S|}$ while the recall Re is $\frac{|G \cap S|}{|G|}$. For example, if $G = \{\{r, s, t\}, \{u\}, \{v\}\}$ and $S = \{\{r, s\}, \{t\}, \{u, v\}\}$, then $Pr = \frac{1}{2}$ and $Re = \frac{1}{3}$. Using Pr and Re, we compute the F_1-measure, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, and use it as our accuracy measure. Throughout this thesis, we use the F_1-measure, or some of its variants, for evaluating ER.

1.3 Structure of This Thesis

In this thesis, we study two problems: data integration and data privacy. For data integration (Chapters 2–5), we propose scalable techniques and new functionalities for ER. For data

privacy (Chapters 6 and 7), we propose a measure for quantifying information leakage using ER and techniques for managing information leakage, respectively. In Chapter 8, we summarize the results of this thesis and discuss future ER extensions. We now elaborate on each chapter.

1.3.1 Data Integration

In Chapter 2, we propose a "pay-as-you-go" approach for ER [108] where we investigate how to maximize the progress of ER with a limited amount of work. The key idea is to exploit *hints*, which give information on records that are likely to refer to the same real-world entity. A hint can be represented in various formats (e.g., a grouping of records based on their likelihood of matching), and ER can use this information as a guideline for which records to compare first. We introduce a family of techniques for constructing hints efficiently and techniques for using the hints to maximize the number of matching records identified using a limited amount of work. Using real data sets, we illustrate the potential gains of our pay-as-you-go approach compared to running ER without using hints.

In Chapter 3, we study the problem of incremental ER [103]. ER may not be a one-time process, but is constantly improved as the data, schema, and application are better understood. We say that ER logic *evolves* whenever it is modified. Our goal is to keep the ER result up-to-date when the ER logic used to compare records evolves frequently. A naïve approach that re-runs ER from scratch may not be tolerable for resolving large datasets. This chapter investigates when and how we can instead exploit previous materialized ER results to save redundant work with evolved logic. We introduce algorithm properties that facilitate evolution, and we propose efficient rule evolution techniques for two clustering ER models: match-based clustering and distance-based clustering. Using real data sets, we illustrate the cost of materializations and the potential gains over the naïve approach.

In Chapter 4, we study the problem of *joint ER* where multiple datasets of different entity types are resolved together [107]. We propose a flexible, modular resolution framework where existing ER algorithms developed for a given record type can be plugged in and used in concert with other ER algorithms. Our approach also makes it possible to run ER on subsets of similar records at a time, important when the full data is too large to

resolve together. We study the problem of joint ER where individual ER algorithms are scheduled and coordinated in order to resolve the full data set. We then evaluate our joint ER techniques on synthetic and real data and show the scalability of our approach. We also introduce a *state-based* training technique where each ER algorithm is trained using machine learning for the particular execution context (relative to other types of records) where it will be used.

In Chapter 5, we study the problem of ER with inconsistencies [102]. To remove ER inconsistencies, we introduce *negative rules* that disallow inconsistencies in the ER solution. A consistent solution is then derived based on guidance from a domain expert, leading to accurate solutions. We formalize the problem of ER with negative rules (ER-N), treating the match, merge, and negative rules as black boxes, which permits expressive and extensible ER-N solutions. We identify important properties for the rules that, if satisfied, enable much more efficient ER-N. We develop and evaluate two algorithms that find an ER-N solution based on guidance from the domain expert: the GNR algorithm that does not assume the properties and the ENR algorithm that exploits the properties.

1.3.2 Data Privacy

In Chapter 6, we address the problem of quantifying *information leakage* [105, 106]. We present a model that captures the privacy loss (information leakage) relative to a target person, on a continuous scale from 0 (no information about the target is known by the adversary) to 1 (adversary knows everything about the target). The model takes into account the confidence the adversary has for the gathered information (leakage is less if the adversary is not confident), as well as incorrect information (leakage is less if the gathered information does not match the target's). We compare our information leakage model with existing privacy models, and we propose several interesting problems that can be formulated with our model. We also propose efficient algorithms for computing information leakage and evaluate their performance and scalability.

In Chapter 7, we propose *disinformation* techniques for Entity Resolution in order to manage information leakage [104]. We model the adversary as an ER process that pieces together available information. We formalize the problem of finding the disinformation

with the highest benefit given a limited budget for creating the disinformation and propose efficient algorithms for solving the problem. We then evaluate our disinformation planning algorithms on synthetic and real data and compare the robustness of existing ER algorithms. In general, our disinformation techniques can be used as a framework for testing ER robustness.

1.4 Related Work

In this section, we cover the related work for ER in general. The other chapters in this thesis cover different aspects of ER and will have their own specific related work.

Originally introduced by Newcombe et al. [78] as "record linkage", entity resolution was then studied under various names, such as merge/purge [54], deduplication [87], reference reconciliation [31], object identification [97], and others.

Several works have addressed the issue of performance for ER algorithms. However, most of them make strong assumptions on the data and/or the match functions to make their algorithms efficient. For example, references [54, 55] assume that records can be represented by one or multiple alphanumeric keys, and that most matches occur between records whose keys are lexicographically close. A "blocking key" can be used to split records into buckets [60] or canopies [71]. Reference [62] proposed mapping the records' values into a multi-dimensional Euclidean space, then performing a similarity join. An overview of such "blocking" methods can be found in [8]. Since they do not compare all records, such techniques make ER algorithms produce approximate results. More recently, reference [6] proposed efficient algorithms for set similarity joins using string similarity functions. In comparison, we view the entire ER process as a black-box operation and provide general and scalable techniques that can be used in various applications. (In some cases we only view the match and merge functions of ER as black-box operations.)

While our work focuses on performance, there has also been a significant amount of work on enhancing the precision and recall of the ER process. The first formalization, by Fellegi and Sunter [37], optimizes the relative importance of numerical similarity functions between records, in a probabilistic setting. In this paper and most follow-ups (see [111, 48]

for recent surveys), the assessment of ER is in terms of precision and recall of the obtained classification. Many string comparison techniques based on edit-distances [91], TF-IDF [25], or adaptive techniques such as q-grams [20, 47] are used for matching records. Reference [75] removes attribute level conflicts of matching records by comparing the quality of their data sources. Reference [88] provides user-defined grouping as part of an SQL extension. As domain-independent techniques may not be suitable for some domains, one may need domain-specific value comparison functions [5]. Any of these techniques can fill in the black boxes of either the entire ER process or the match and merge functions. We can thus decouple the techniques above from our ER algorithms that call the black boxes.

Chapter 2

Pay-As-You-Go Entity Resolution

An ER process is often extremely expensive due to very large data sets and compute-intensive record comparisons. For example, collecting people profiles on social websites can yield hundreds of millions of records that need to be resolved. Comparing each pair of records to estimate their "similarity" can be expensive as many of their fields may need to be compared and substantial application logic must be invoked.

At the same time, it may be very important to run ER within a limited amount of time. For example, anti-terrorism applications may require almost real-time analysis (where streaming data is processed in small batches using operations like ER) to capture a suspect who is on the brink of escaping. Although the analysis may not be as complete as when the full data is available, the fast processing can increase the chance of the suspect being captured. As another example, a newsfeed entity matching algorithm may have very limited time for resolving company names and individuals in a stock market trading scenario where financial data is generated with high frequency.

In this chapter we explore a "pay-as-you-go" approach to entity resolution, where we obtain partial results gradually as we perform resolution, so we can at least get some results faster. As we will see, the partial results may not identify all the records that correspond to the same real-world entity. Our goal will be to obtain as much of the overall result as possible, as quickly as possible.

Figure 2.1 is a simple cartoon sketch to illustrate our approach. The horizontal axis is the amount of work performed, say the number of record pairs that are compared. The

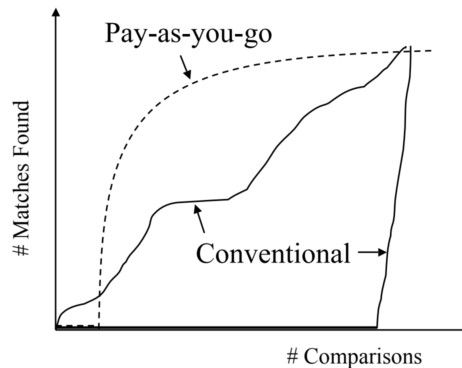


Figure 2.1: Pay-as-you-go approach of ER

vertical axis shows the "quality" of the result, say the number of pairs that have been found to match (i.e., to represent the same entity). The bottom curve in the figure (running mostly along the horizontal axis) illustrates the behavior of a typical non-incremental ER algorithm: it only yields its final answer after it has done all the work. If we do not have time to wait to the end, we get no results. The center solid line represents a typical incremental ER algorithm that reports results as it goes along. This algorithm is preferable when we do not have time for the full resolution.

The dotted line in Figure 2.1 shows the type of algorithm we want to develop here: instead of comparing records in random order, it looks for matches in the pairs that are most likely to match, hence it gets good quality results very fast. To identify the most profitable work to do early on, the algorithm performs some pre-analysis (the initial flat part of the curve). The pre-analysis yields what we call *hints* that are then used by the subsequent resolution phase to identify profitable work. If we have limited time, in our example say half of the time taken by the full resolution, our approach is able to give us a much better result than the traditional algorithms. Of course, in other cases our approach may be counterproductive (e.g., if the pre-analysis takes too long relative to the available time). Furthermore, not all ER approaches are amenable to the pay-as-you-go approach.

In this chapter we address three important questions. First, how do we construct the hints? Our hints rely on an approximate and inexpensive way to compare records, e.g.,

two records are more likely to represent the same person if they have similar zip codes. However, there are several ways in which the hint can be encoded. For instance, a hint can be an ordered list of record pairs, sorted by likelihood of matching. A hint can also be an ordering of the records such that the number of matching records identified is maximized when the list is resolved sequentially.

Second, how do we use the hints? The answer to this question depends on the ER strategy one is utilizing, and as stated earlier, some algorithms are not amenable to using hints. Since there are so many ER strategies available, clearly we cannot give a comprehensive answer to this second question, but we do illustrate the use of different types of hints in several representative instances.

Third, in what cases does pay-as-you-go pay off? Again, we cannot give a comprehensive answer but we do illustrate performance on several real scenarios and we identify the key factors that determine the desirability of pay-as-you-go.

It is important to note that our work is empirical by nature. Hints are heuristics. We will show they work well in representative cases, but they provide no formal guarantees. Also, our goal here is to provide a unifying framework for hints and to evaluate the potential gains. Certain types of hints have been used before (see Section 2.8), and we do not claim to cover all possible types of hints.

In summary, our contributions in this chapter are as follows:

- We formalize pay-as-you-go ER where our goal is to improve the partial ER result (Section 2.1). Our techniques build on top of blocking [77], which is a standard technique for scaling ER.
- We propose three types of hints:
  - *Sorted List of Record Pairs*: The most informative (but least compact) type of hint (Section 2.2).
  - *Hierarchy of Partitions*: A moderately informative and compact type of hint (Section 2.3).
  - *Sorted List of Records*: The most compact (but least informative) type of hint (Section 2.4).

For each hint type, we propose techniques for efficiently generating hints and investigate how ER algorithms can utilize hints to maximize the quality of ER while minimizing the number of record comparisons.

- We extend our approach to using multiple hints (Section 2.5).
- We experimentally evaluate how applying hints can help ER produce good quality results fast (Section 2.7). We use actual comparison shopping data from Yahoo! Shopping and hotel information from Yahoo! Travel. Our results show scenarios where hints improve the ER processing to find the majority of matching records within a fraction of the total runtime.

2.1 Framework

In this section, we define our framework for pay-as-you-go ER. We first define a general model for entity resolution, and then we explain how pay-as-you-go fits in.

2.1.1 ER Model

An ER algorithm E takes as input a set of records R that describe real-world entities. The ER output is a partition of the input that groups together records that describe the same real-world entity. For example, the output $F = \{\{r_1, r_3\}, \{r_2\}, \{r_4, r_5, r_6\}\}$ indicates that records r_1 and r_3 represent one entity, r_2 by itself represents a different entity, and so on. Since sometimes we wish to run ER on the output of a previous resolution, we actually define the input as a partition. Initially, each record is in its own partition, e.g., $\{\{r_1\}$, $\{r_2\}$,$\{r_3\}$, $\{r_4\}$, $\{r_5\}$, $\{r_6\}\}$.

We denote the ER result of E on R at time t as $E(R)[t]$. In the above example, if E has grouped $\{r_1\}$ and $\{r_3\}$ after 5 seconds, then $E(R)[5] = \{\{r_1, r_3\}$, $\{r_2\}$, $\{r_4\}$, $\{r_5\}$, $\{r_6\}\}$. We denote the total runtime of $E(R)$ as $T(E, R)$. A quality metric M can be used to evaluate an ER result against the correct partitioning of R. For example, suppose that M computes the fraction of clustered record pairs that are also clustered according to

the correct ER answer. Then if $E(R) = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ and the correct clustering is $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$, $M(E(R)) = \frac{1}{3}$.

We focus on ER algorithms that work by repeatedly comparing pairs of records to determine their semantic similarity or difference. Although ER algorithms use different strategies, the general principle is that if a pair of records appear "similar," then they are candidates for the same output partition. Since there are many potential records pairs to compare ($\frac{n\times(n-1)}{2}$ pairs for n records), most algorithms use some type of pruning strategy, where many pairs are ruled out based on a very coarse computation.

The most popular pruning strategy uses *blocking* or *indexing* [77, 54, 71, 43]. Input records are placed in blocks according to one or more of their fields, e.g., for product records, cameras are placed in one block, cell phones in another, and so on. LSH (locality sensitive hashing) [43] can also be used to place each record in one or more blocks. Then only pairs of records within the same block are compared. The number of record comparisons is substantially reduced, although of course matches may be missed. For instance, one store may call a camera-phone a cell phone while another may (mistakenly) call it a camera, so the two records from different stores will not be matched up even though they represent the same product.

Conceptually then we can think of blocking as defining a *set of candidate pairs* that will be carefully compared. The set may not be materialized, i.e., may only be implicitly defined. For instance, the placement of records in blocks defines the candidate set to be all pairs of records residing within a single block.

2.1.2 Pay-As-You-Go Model

With the pay-as-you-go model, we *conceptually* order the candidate pairs by the likelihood of a match. Then the ER algorithm performs its record comparisons considering first the more-likely-to-match pairs. The key of course is to determine the ordering of pairs very efficiently, even if the order is approximate.

To illustrate, say we have placed six records into two blocks: the first block contains records r_1, r_2, and r_3, while the second block contains r_4, r_5, and r_6. The implicit set of candidate pairs is $\{r_1 - r_2, r_1 - r_3, r_2 - r_3, r_4 - r_5 \ldots\}$. A traditional ER algorithm

would then compare these pairs, probably by considering all pairs in the first block in some arbitrary order, and then the pairs in the second block. With pay-as-you-go, we instead first compare the most likely pair from either bucket, say $r_5 - r_6$. Then we compare the next most likely, say $r_2 - r_3$. However, if only one block at a time fits in memory, we may prefer to order each block independently. That is, we first compare the pairs in the first block by descending match likelihood, then we do the same for the second block. Either way, the goal is to discover matching pairs faster than by considering the candidate pairs in an arbitrary order. The ER algorithm can then incrementally construct an output partition that will more quickly approximate the final result. (As noted earlier, not all ER algorithms can be changed to compute the output incrementally and to consider candidate pairs by increasing match likelihood.)

More formally, we define a pay-as-you-go version of an ER algorithm as follows.

Definition 2.1.0.1. *Given a quality metric M, a pay-as-you-go algorithm E' of the ER algorithm E satisfies the following conditions.*

- Improved Early Quality*: For some given target time(s) $t_g < T(E,R)$, $M(E'(R)[t_g]) > M(E(R)[t_g])$. Target time t_g may be substantially smaller than $T(E,R)$ and represent the time at which early results are needed.*
- Same Eventual Quality*: $M(E'(R)[t]) = M(E(R)[t])$ for some time $t \geq T(E,R)$.*

The first condition captures our goal of producing higher-quality ER results early on. The second condition guarantees that the pay-as-you-go algorithm will eventually produce an ER result that has the same quality as the ER result produced without hints. In comparison, blocking techniques may return an approximate ER result where the quality has decreased.

To efficiently generate candidate pairs in (approximate) order by match likelihood, we use an auxiliary data structure we call the *hints*. As illustrated in Figure 2.2, we discuss three types of hints. The most general form is a sorted list of record pairs, although as we will see, the list need not be fully materialized. A less general but more compact structure is a hierarchy where each level represents a partition of the records grouped by their likelihood of matching. A partition on a higher level is always coarser (see Definition 2.3.0.2) than a

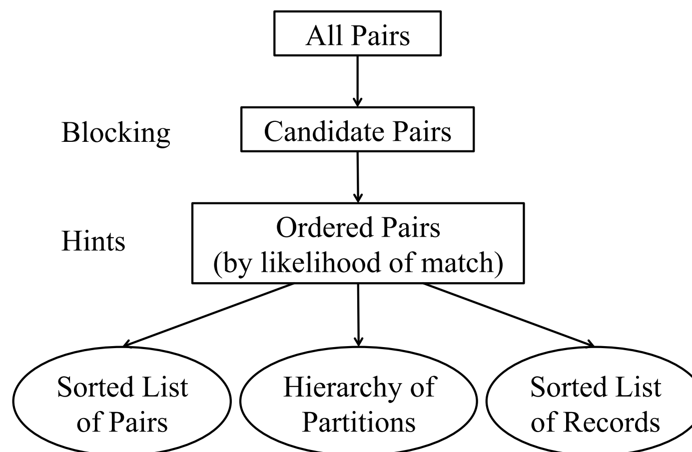


Figure 2.2: Pay-As-You-Go ER Framework

partition on a lower level of the hierarchy. The third structure is a sorted list of records (not pairs) where records that appear early in the list are more likely to match with each other than records far down the list.

Note that a hint is not an interchangeable "module" than can simply be plugged into any ER algorithm. Each hint is a tool that may or may not be applicable for a given ER algorithm. In the following three sections we describe each hint type in more detail, and show how it can be used by some ER algorithms. For simplicity we will focus on processing a single block of records (although as noted earlier a single hint could span multiple blocks). In Section 2.5, we discuss how to use multiple hints for resolving records.

2.2 Sorted List of Record Pairs

In this section we explore a hint that consists of a list of record pairs, ranked by the likelihood that the pairs match. We assume that the ER algorithm uses either a distance or a match function. The distance function $d(r, s)$ quantifies the differences between records r and s: the smaller the distance the more likely it is that r and s represent the same real-world entity. A match function $m(r, s)$ evaluates to true if it is deemed that r and s represent the same real-world entity. Note that a match function may use a distance function. For instance, the match function may be of the form "if $d(r, s) < T$ and other conditions then

true," where T is a threshold.

We also assume the existence of an estimator function $e(r, s)$ that is much less expensive to compute than both $m(r, s)$ and $d(r, s)$. The value of $e(r, s)$ approximates the value of $d(r, s)$, and if the ER algorithm uses a match function, then the smaller the value of $e(r, s)$, the more likely it is that $m(r, s)$ evaluates to true.

Conceptually, our hint will be the list of all record pairs, ordered by increasing e value. In practice, the list may not be explicitly and fully generated. For instance, the list may be truncated after a fixed number of pairs, or after the estimates reach a given threshold. As we will see, another alternative is to generate the pairs "on demand": the ER algorithm can request the next pair on the list, at which point that pair is computed. As a result, we can avoid an $O(N^2)$ complexity for generating the hint.

We now discuss how to generate the pair-list hint, and then how an ER algorithm can use such a list.

2.2.1 Generation

We first discuss how we can generate pair-list hints using cheaper estimation techniques. We then discuss a more general technique that does not require application estimates.

Using Application Estimates

In some cases, it is possible to construct an application-specific estimate function that is cheap to compute. For example, if the distance function computes the geographic distance between people records, we may estimate the distance using zip codes: if two records have the same zip code, we say they are close, else we say they are far. If the distance function computes and combines the similarity between many of the record's attributes, the estimate can only consider the similarity of one or two attributes, perhaps the most significant.

To generate the hint, we can compute $e(r, s)$ for all record pairs, and insert each pair and its estimate into a heap data structure, with the pair with smallest estimate at the top. After we have inserted all pairs, if we want the full list we can remove all pairs by increasing estimate. However, if we only want the top estimates, we can remove entries until we reach a threshold distance, a limited number of pairs, or until the ER algorithm stops requesting

pairs from the hint.

In other cases, the estimates map into distances along a single dimension, in which case the amount of data in the heap can be reduced substantially. For example, say $e(r, s)$ is the difference in the price attribute of records. (Say that records that are close in price are likely to match.) In such a case, we can sort the records by price. Then, for each record, we enter into the heap its closest neighbor on the price dimension (and the corresponding price difference). To get the smallest estimate pair, we retrieve from the heap the record r with the closest neighbor. We immediately look for r's next closest neighbor (by consulting the sorted list) and re-insert r into the heap with that new estimate. The space requirement in this case is proportional to $|R|$, the number of records. On the other hand, if we store all pairs of records in the heap, the space requirement is order of $O(|R|^2)$.

Application Estimate Not Available

In some cases, there may be no known inexpensive application specific estimate function $e(r, s)$. In such scenarios, we can actually construct a "generic but rough" estimate based on sampling. This technique may not always give good results, but as we show in Section 2.7, it can yield surprisingly good estimates in some cases.

The basic idea is to use the expensive function d to compute the distances for a small subset of record pairs, and then use the computed distances to estimate the rest of the distances. We do not assume the records to be in any space (e.g., Euclidean), so d does not have to compute an absolute distance. The main advantage of this sampling technique is its generality where we can estimate distances by only using the given distance function. Suppose we have a sample S, which is a subset of the set of records R. We first measure the actual distances between all the records within S and between records in S and records in $R - S$. Assuming that the sample size $|S|$ is significantly smaller than the total number of records $|R|$, the number of real distances measured is much smaller than the total number of pairwise distances. For example, if $|R|$ = 1000 and $|S|$ = 10, then the fraction of real distances we compute is $\frac{\binom{10}{2}+990\times10}{\binom{1000}{2}} = \frac{9945}{499500} \approx 2\%$.

Given a fraction of the real distances, we can estimate the other distances. One possible scheme captures the distance between two records r and s as the sum of squares of the difference of $d(r, t)$ and $d(t, s)$ for each $t \in S$. Formally, the estimate $e(r, s) =$

$\Sigma_{t \in S}(d(r,t) - d(t,s))^2$. The intuition is that, if r and s are very close, then they will be almost the same distance from any sample point t. For example, if $d(r,t_1)$ = 8, $d(r,t_2)$ = 10, $d(t_1,s)$ = 5, and $d(t_2,s)$ = 4, then $e(r,s) = (8-5)^2 + (10-4)^2 = 45$. While 45 is not a "real" distance, we only need to compare the *relative* sizes of estimates of different record pairs to construct hints. The estimated distances among records within S and between records in S and $R - S$ must also be computed the same way as above. Our techniques resemble triangulation techniques where a point is located by measuring angles to it from known reference points.

The sample set may affect the quality of estimation. In the worst case, the sample can be $|S|$ duplicate records, and all estimates turn out to be the same for any pair of records. Hence it is desirable for the sample records to be evenly dispersed within R as much as possible. In practice, selecting a small random subset of $|S|$ records works reasonably well (see Section 2.7.5).

2.2.2 Use

The details on how to use a pair-list hint depend on the actual ER algorithm used. However, there are two general principles that can be employed:

- If there is flexibility on the order in which functions $m(r,s)$ or $d(r,s)$ are called, evaluate these functions first on r, s pairs that are higher in the pair-list. This approach will hopefully let the algorithm identify matching pairs (or pairs that are clustered together) earlier than if pairs are evaluated in random order.

- Do not call the d or m functions on pairs of records that are low on the pair-list, assuming instead that the pair is "far" (pick some large distance as default) or does not match.

Note that in some cases the ER algorithm with hints will return the same final answer (call it F') as the unmodified algorithm (call it F), but matches or clusters will be found faster. In other cases, the ER algorithm will return an answer F' that is different from the unmodified answer F, but hopefully F' will have a high quality compared to F.

We now illustrate how the Sorted Neighbor algorithm [54] (called SN) can benefit from a pair-list hint. Say a block contains the records $R = \{r_1, r_2, r_3\}$. The SN algorithm first sorts the records in R using a certain key assuming that closer records in the sorted list are more likely to match. For example, suppose that we sort the records in R by their names (which are not visible in this example) in alphabetical order to obtain the list [r_3, r_2, r_1]. The SN algorithm then slides a window of size w on the sorted record list and compares all the pairs of clusters that are inside the same window at any point. If the window size is 2 in our example, then we compare r_3 with r_2 and then r_2 with r_1, but not r_3 with r_1 because they are never in the same window. We thus produce pairs of records that match with each other. We can repeat this process using different keys (e.g., we could also sort the person records by their address values). While collecting all the pairs of records that match, we can perform a transitive closure on all the matching pairs of records to produce a partition S of records. For example, if r_3 matches with r_2 and r_2 matches with r_1, then we merge r_1, r_2, r_3 together into the output $S = \{\{r_1, r_2, r_3\}\}$.

To use a pair list as a hint, we define the cheap distance function $e(r, s)$ to be the difference in rank between records according to the sorted list. That is, given two records r and s, $e(r, s) = |Rank(r) - Rank(s)|$ where $Rank(r)$ indicates the index of r in the sorted list of the records in R. Intuitively, the closer records are according to the sorted list, the more they are likely to match. In our example above, our sorted list is $[r_3, r_2, r_1]$, so $Rank(r_3) = 1$, $Rank(r_2) = 2$, and $Rank(r_1) = 3$. Hence, the distance between r_1 and r_2 is 1 while the distance between r_1 and r_3 is 2. The modified ER algorithm SN compares the records with the shortest estimated distances first, we are effectively comparing records within the smallest sliding window, and repeating the process of increasing the size of the window by 1 and comparing the records that are within the new sliding window, but have not been compared before. Notice that once the next shortest distance of records exceeds the window size w, we have done the exact same record comparisons as the SN algorithm. In addition, we can also stop comparing records in the middle of ER once we have exceeded the work limit W. For instance, if we set W to only allow one record comparison, then we only compare either $\langle r_3, r_2 \rangle$ or $\langle r_2, r_1 \rangle$ and terminate the ER algorithm.

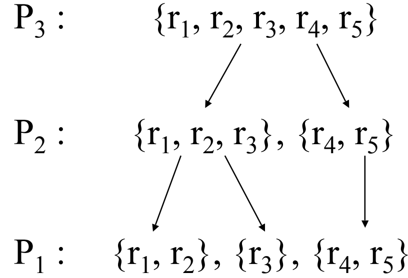


Figure 2.3: A partition hierarchy hint for resolving R

2.3 Hierarchy of Record Partitions

In this section, we propose the partition hierarchy as a possible format for hints. A partition hierarchy gives information on likely matching records in the form of partitions with different levels of granularity where each partition represents a "possible world" of an ER result. The partition of the bottom-most level is the most fine-grained clustering of the input records. Higher partitions in the hierarchy are more coarse grained with larger clusters. That is, instead of storing arbitrary partitions, we require the partitions to have an order of granularity where coarser partitions are higher up in the hierarchy.

Definition 2.3.0.2. *A partition P is* coarser *than another partition P' (denoted as $P' \leq P$) when the following condition holds:*

- $\forall c' \in P', \exists c \in P$ *s.t.* $c' \subseteq c$

Figure 2.3 shows a hierarchy hint for the set of records $\{r_1, r_2, r_3, r_4, r_5\}$. Suppose that the most likely matching pairs of the records are $\langle r_1, r_2 \rangle$ and $\langle r_4, r_5 \rangle$. We can express this information as the bottom-level partition $\{\{r_1, r_2\}, \{r_3\}, \{r_4, r_5\}\}$ of the hierarchy. Among the clusters in the bottom level, suppose that $\{r_1, r_2\}$ is more likely to be the same entity as $\{r_3\}$ than $\{r_4, r_5\}$. The next level of the hint can then be a coarser partition of the bottom level partition where the clusters $\{r_1, r_2\}$ and $\{r_3\}$ from the bottom-level partition have merged.

We now formally define a partition hierarchy hint.

Definition 2.3.0.3. *A valid partition hierarchy hint H with L levels is a list of partitions $P_1, \ldots, P_L$ of R where $P_j \leq P_{j+1}$ for any $1 \leq j < L$.*

For example, Figure 2.3 is a valid partition hierarchy hint where P_1 = $\{\{r_1, r_2\}$, $\{r_3\}$, $\{r_4, r_5\}\}$ and P_2 = $\{\{r_1, r_2, r_3\}$, $\{r_4, r_5\}\}$ (i.e., $P_1 \leq P_2$). However, if P_2 were $\{\{r_1\}$, $\{r_2\}$, $\{r_3\}$, $\{r_4, r_5\}\}$, then H would not be valid because $P_1 \not\leq P_2$.

Within the hierarchy of a partition hierarchy hint, a cluster c in a higher level is connected to the clusters in the lower level that were combined to construct c. We call these clusters the children of c.

Definition 2.3.0.4. *The* children *of a cluster c (denoted as $c.ch$) in the ith level ($i > 1$) of a partition hierarchy hint H is the largest set of clusters S in the $(i-1)$st level of H such that $\forall c' \in S, c' \leq c$.*

For example, in Figure 2.3, the children of cluster $\{r_1, r_2, r_3\}$ in P_2 is the set $\{\{r_1, r_2\}$, $\{r_3\}\}$, and the children of cluster $\{r_4, r_5\}$ in P_2 is the set $\{\{r_4, r_5\}\}$.

A significant advantage of the partition hierarchy structure is that the storage space is linear in the number of records regardless of the height L. A compact way to store the information of a partition hierarchy is to keep track of the clusters splitting into their children in lower levels. For example, in Figure 2.3, there are two cluster splits: one that splits the cluster $\{r_1, r_2, r_3, r_4, r_5\}$ in P_3 into $\{r_1, r_2, r_3\}$ and $\{r_4, r_5\}$ and another that splits $\{r_1, r_2, r_3\}$ in P_2 into $\{r_1, r_2\}$ and $\{r_3\}$. Hence we only need to save the information of two cluster splits. Since a partition hierarchy can have at most $|R| - 1$ splits, the maximum space required to store the splits information is linear in the number of records.

2.3.1 Generation

We propose various methods for efficiently constructing a partition hierarchy. In the following section, we construct hints based on sorted records, which are application estimates. Next, we discuss how partition hierarchies can also be generated using hash functions (which are also application estimates) and inexpensive distance functions (which are not application estimates).

ALGORITHM 1: Generating a partition hierarchy hint from sorted records

```
1: Input: a list of sorted records Sorted = [r_1, r_2, ...] and a list of thresholds T = [T_1, ..., T_L]
2: Output: a hint H = {P_1, ..., P_L}
3: Initialize partitions P_1, ..., P_L
4: for r ∈ Sorted do
5:     for T_j ∈ T do
6:         if r.prev.exists() ∧ KeyDistance(r.key, r.prev.key) ≤ T_j then
7:             Add r into the newest cluster in P_j
8:         else
9:             Create new cluster in P_j containing r
10: return {P_1, ..., P_L}
```

Using Sorted Records

We explore how a partition hierarchy can be generated when the estimated distances between records can map into distances along a single dimension according to a certain attribute key.

Algorithm 1 shows how we can construct a partition hierarchy hint H using different thresholds $T_1, \ldots, T_L$ for partitioning records based on their key value distances. (The thresholds values are pre-specified based on the number of levels L in H.) For example, say we have a list of three records [Bob, $Bobby$, $Bobji$] (the records are represented and sorted by their names). Suppose that we set two thresholds T_1 = 1 and T_2 = 2, and use edit distance (i.e., the number of character inserts and deletes required to convert one string to another) for measuring the key distance between records. Algorithm 1 first reads Bob and adds it into a new cluster both for P_1 and P_2 (Step 9). Then we read $Bobby$ and compare it with the previous record Bob (Step 6). The edit distance between Bob and $Bobby$ is 2. Since this value is larger than T_1, we create a new cluster in P_1 and add $Bobby$ (Step 9). Since the edit distance does not exceed T_2, we add $Bobby$ into the first cluster in P_2 (Step 7). For the last record $Bobji$, the edit distance with the previous record $Bobby$ is 4, which exceeds both thresholds. As a result, a new cluster with $Bobji$ is created for both P_1 and P_2. The resulting hint thus contains two partitions: P_1 = $\{\{Bob\}$, $\{Bobby\}$, $\{Bobji\}\}$ and P_2 = $\{\{Bob, Bobby\}$, $\{Bobji\}\}$.

The following result shows the correctness of Algorithm 1.

Proposition 2.3.1. *Algorithm 1 returns a valid hint.*

Proof. A higher level partition P is always coarser than a lower level partition P' because a higher threshold is used to split records in the sorted list. Hence, $P' \leq P$. Since higher level partitions is always coarser than lower level partitions, H is a valid hint according to Definition 2.3.0.3. $\square$

Given that the input $Sorted$ is already sorted, Algorithm 1 runs in $O(L \times |R|)$ time by iterating all records in $Sorted$ and, for each record, iterating through all thresholds.

Using Hash Functions

We can also generate a partition hierarchy based on hash functions with different probabilities of collision. For example, minhash signatures [57] can be used to estimate set similarity. Or if an attribute of records contains categorical values, then each record can be hashed as its category. To generate L partitions for a hint, we can use a family of hash functions $H_1, \ldots, H_L$ where for any $1 \leq i < L$, H_i has a lower probability of collision than H_{i+1} and any collision that occurs in H_i also occurs in H_{i+1}. The algorithm for constructing the hint is similar to Algorithm 1, except that records are now assigned to clusters based on their hash values. For example, suppose that we have a set of three records $\{Bobbie$, $Bobby$, $Bobji\}$. Given H_1 that uses the first four characters of a name as a record's hash value while H_2 uses the first three characters, then P_1 = $\{\{Bobbie, Bobby\}$, $\{Bobji\}\}$ while P_2 = $\{\{Bobbie$, $Bobby$, $Bobji\}\}$. The complexity of the algorithm is $O(L \times |R|)$ because for each level, we iterate all the records and assign each record to its bucket in each level.

Using Distance Estimation Functions

We can also use an inexpensive distance estimator function $e(r, s)$ to generate a partition hierarchy. The $e(r, s)$ function can be application specific or generated by a sampling technique (see Section 2.2.1).

Algorithm 2 shows how we can construct a partition hierarchy hint given the distance estimates. For each level L_j in H, we can use a union-find algorithm [95] to generate a transitive closure of records that have estimated distances less than a given threshold T_j. For example, suppose we have three records r_1, r_2, r_3 whose estimated distances are set as $e(r_1, r_2)$ = 1, $e(r_1, r_3)$ = 2, $e(r_2, r_3)$ = 3. Also, suppose that we set T_1 = 1 and T_2 = 2.

ALGORITHM 2: Generating a partition hierarchy hint from pairwise distance estimates

1: **Input:** a list of pairs with their estimated distances $Pairs$ = [$\langle r_1, s_1\rangle$, $\langle r_2, s_2\rangle$, $\ldots$], and a list of thresholds T = [$T_1, \ldots, T_L$]
2: **Output:** a hint H = $\{P_1, \ldots, P_L\}$
3: Initialize partitions $P_1, \ldots, P_L$
4: **for** $\langle r, s\rangle \in Pairs$ **do**
5:     **for** $T_j \in T$ **do**
6:         **if** $e(r, s) \leq T_j$ **then**
7:             TransitiveClosure($P_j, \langle r, s\rangle$)
8: **return** $\{P_1, \ldots, P_L\}$

Algorithm 2 first initializes all partitions $P_1, \ldots, P_L$ into empty sets (Step 3). For the first pair $\langle r_1, r_2\rangle$, we compare its estimated distance 1 with T_1 = 1 (Step 6). Since r_1 and r_2 are close enough, we connect r_1 and r_2 in P_1 (Step 7). Next, we compare the estimated distance 1 with T_2 = 2. Again, r_1 and r_2 are connected in P_2. We then read the next pair of records $\langle r_1, r_3\rangle$. Since the estimated distance is 2, r_1 and r_3 are connected in P_2, but not in P_1. For the last pair $\langle r_2, r_3\rangle$, the estimated distance 3 exceeds both thresholds. As a result, the resulting hint contains two partitions: P_1 = $\{\{r_1, r_2\}, \{r_3\}\}$ and P_2 = $\{\{r_1, r_2, r_3\}\}$. In Step 7, one can use a more sophisticated clustering algorithm (instead of a transitive closure) provided that the clustering results $\{P_1, \ldots, P_L\}$ satisfy Definition 2.3.0.3.

Notice that when estimating the distances between records, we do not have to actually store the estimates for each pair of records, which would require a space quadratic in the number of records. Instead, we can construct the partition hierarchy *while* generating the estimates. Hence, the space complexity of construction based on sampling is $O(L \times |R|)$. The time complexity for constructing the partition hierarchy is $O(|R|^2 + L \times C(|R|))$ where $|R|^2$ is needed for the sampling and $C(|R|)$ is the complexity of the clustering algorithm used to generate the partitions of R in the hierarchy.

2.3.2 Use

Given a partition hierarchy, the next question is how an ER algorithm can actually exploit this information to maximize the ER quality with a limited amount of work. We assume the ER algorithm is given based on what works best for the application or what developers

have experience with. In general, there are two principles that can be employed to use a partition hierarchy:

- If there is flexibility on the order of which records are resolved, compare the records that are in the same cluster in the bottom-most level of the hierarchy hint.
- If there is more time, start comparing records in the same cluster in higher levels of the hierarchy hint.

Algorithm 3 shows how a partition hierarchy hint can be used by an ER algorithm. Given a set of records R, an ER algorithm E, a partition hierarchy hint H, and a work limit W, we intuitively resolve the records in the bottom-level clusters first and progressively resolve more records in higher-level clusters in the hierarchy until there are no more records to resolve or the amount of work done exceeds W (e.g., the number of record comparisons should not exceed 1 million).

We illustrate Algorithm 3 using the Single-link Hierarchical Clustering algorithm [45, 70] (which we call HC_S). The HC_S algorithm merges the closest pair of clusters (i.e., the two clusters that have the smallest distance) into a single cluster until the smallest distance among all pairs of clusters exceeds a certain threshold T. The distance between two records is measured using a commutative distance function D that returns a non-negative distance between two records. When measuring the distance between two clusters, the algorithm takes the smallest possible distance between records within the two clusters. Now suppose we have $R = \{r_1, r_2, r_3\}$ (which can also be viewed as a list of three singleton clusters) where the pairwise distances are $D(r_1, r_2) = 2$, $D(r_2, r_3) = 4$, and $D(r_1, r_3) = 5$ with a given threshold $T = 2$. The HC_S algorithm first merges r_1 and r_2, which are the closest records and have a distance smaller or equal to T, into $\{r_1, r_2\}$. The cluster distance between $\{r_1, r_2\}$ and $\{r_3\}$ is the minimum of $D(r_1, r_3)$ and $D(r_2, r_3)$, which is 4. Since the distance exceeds T, $\{r_1, r_2\}$ and $\{r_3\}$ do not merge, and the final ER result is $\{\{r_1, r_2\}, \{r_3\}\}$.

We can use Algorithm 3 to run the HC_S algorithm with a hint that is a partition hierarchy. Continuing our example above where $R = \{r_1, r_2, r_3\}$, suppose that we are given the hint $P_1 = \{\{r_1, r_2\}, \{r_3\}\}$ and $P_2 = \{\{r_1, r_2, r_3\}\}$. Also say that W is set to three record comparisons. According to Algorithm 3, we first resolve the clusters in P_1 of the

ALGORITHM 3: Using a partition hierarchy in ER

1: **Input:** a set of records R, an ER algorithm E, a hint $H = \{P_1, \ldots, P_L\}$, and a work limit W
2: **Output:** an intermediate ER result F of $E(R)$
3: $F \leftarrow \emptyset$, $h \leftarrow \emptyset$
4: **for** $i = 1 \ldots L$ **do**
5:     **for** $c \in P_i$ **do**
6:         **for** $child \in c.ch$ **do**
7:             $F \leftarrow F - \{clus | clus \in F \wedge clus \subseteq child\}$
8:         $h(c) \leftarrow Resolve(E, c, h)$
9:         $F \leftarrow F \cup h(c)$
10:        **if** total work $\geq W$ **then**
11:            **return** F
12: **return** F

hint. Thus we compare r_1 with r_2 by invoking $Resolve(E, \{r_1, r_2\}, h)$ in Step 8. Since r_1 and r_2 match, F becomes $\{\{r_1, r_2\}, \{r_3\}\}$. We also store the ER results of $\{r_1, r_2\}$ and $\{r_3\}$ in h. Next, we start resolving records in the cluster $\{r_1, r_2, r_3\}$ in P_2. When resolving $\{r_1, r_2, r_3\}$, we first subtract from F the clusters that are subsets of $\{r_1, r_2, r_3\}$, leaving us with $F = \{\}$ (Step 7). We now run $Resolve(E, \{r_1, r_2, r_3\}, h)$ in Step 8. Again, only r_1 and r_2 match and we union F with $\{\{r_1, r_2\}, \{r_3\}\}$ (Step 9). Assuming $Resolve$ used at least two more record comparisons to resolve $\{r_1, r_2, r_3\}$, the total work is larger or equal to the work limit W, and we return the ER result $F = \{\{r_1, r_2\}, \{r_3\}\}$ (Step 12), which is the correct answer. Notice that, if W was set to 1 instead of 3, the same ER result $F = \{\{r_1, r_2\}, \{r_3\}\}$ would have been returned using only one record comparison.

Proposition 2.3.2. *Given a valid ER algorithm E, Algorithm 3 returns a correct ER result when $P_L = \{R\}$ and W is unlimited.*

Proof. Given that W is never satisfied, E can always run to the end. Since $P_L = \{R\}$, the final result $F = E(R_p)$, which is the correct result by definition. □

The complexity of Algorithm 3 is at least the complexity of the ER algorithm E because we can always use a hierarchy with one level having $\{R\}$ as its partition. The actual efficiency of the algorithm largely depends on the implementation of $Resolve(E, c, h)$ in Step 8. In the worst case, E can simply ignore the information of resolved records h and run $E(c)$ from scratch. However, an ER algorithm can exploit the information in h to

produce $Resolve(E, c, h)$ more efficiently. For example, if $c = \{r_1, r_2, r_3\}$ and we know by h that r_1 and r_2 are the same entity. Then the ER algorithm can avoid a redundant record comparison between r_1 and r_2.

2.3.3 General Incremental Property

We explore an interesting property of an ER algorithm (called "general incremental") that, when satisfied, can enable efficient computation given information of previously resolved records. That is, given an input set of records R and ER results of previously resolved records, we would like $E(R)$ to run faster than resolving R from scratch. An ER algorithm is incremental [59] if it can resolve one record at a time. We use a more generalized version of the incremental property [103] for our ER model where subsets of R can be resolved in any order.

In order to precisely define the general incremental property, we need to formalize our general ER definition further. First, we assume that an ER algorithm receives a partition of R (called R_p) and returns a new partition of R. This view does not change our original ER model (where ER partitions a set of records) because a set of records $R = \{r_1, \ldots, r_n\}$ can also be viewed as a set of singleton clusters $R_p = \{\{r_1\}, \ldots, \{r_n\}\}$. We denote all the possible partitions that can be produced by the ER algorithm E as $\bar{E}(R_p)$, which is a set of partitions of R. That is, we assume that ER is non-deterministic in a sense that different partitions of R may be produced depending on the order of records processed or by some random factor (e.g., the ER algorithm could be a random algorithm). Hence, $E(R_p)$ is always one of the partitions in $\bar{E}(R_p)$. For example, given $R_p = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, $\bar{E}(R_p)$ could be $\{\{\{r_1, r_2\}, \{r_3\}\}, \{\{r_1\}, \{r_2, r_3\}\}\}$ while $E(R_p) = \{\{r_1, r_2\}, \{r_3\}\}$.

Definition 2.3.2.1. *An ER algorithm is* generally incremental *[103] if for any four partitions P_1, P_2, F_1, and F_2 such that*

- $P_1 \subseteq P_2$ *and*
- $F_1 \in \bar{E}(P_1)$ *and*
- $F_2 \in \bar{E}(F_1 \cup (P_2 - P_1))$

then $F_2 \in \bar{E}(P_2)$.

For example, suppose we have $P_1 = \{\{r_1\}, \{r_2\}\}$, $P_2 = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, and $F_1 = \{\{r_1, r_2\}\}$. That is, we have already resolved P_1 into the result F_1. We can then add to F_1 the remaining cluster $\{r_3\}$, and resolve all the clusters together (i.e., we run $E(\{\{r_1, r_2\}, \{r_3\}\})$). The result is as if we had resolved everything from scratch (i.e., from P_2). Presumably, the former way (incremental) will be more efficient than the latter by exploiting the information on records have already been resolved.

Proposition 2.3.3. *Suppose we have a partition* $S = \{s_1, \ldots, s_n\}$ *of* R_p*. That is,* $\bigcup s_i = R_p$ *and for any* $1 \leq i, j \leq n$ *where* $i \neq j$*,* $s_i \cap s_j = \emptyset$*. Then given a general incremental ER algorithm* E*,* $F = E(\bigcup_{i=1\ldots n} E(s_i)) \in \bar{E}(R_p)$*.*

Proof. We show that, if E satisfies the general incremental property, then $E(\bigcup_{i=1\ldots n} E(s_i)) \in \bar{E}(R_p)$. We define the following two notations: $\alpha(k) = \bigcup_{s \in R_p - \{s_1,\ldots,s_k\}} E(s)$ and $\beta(k) = \bigcup_{s \in \{s_1,\ldots,s_k\}} s$. To prove that $F = E(\alpha(0)) \in \bar{E}(R_p) = \bar{E}(\beta(|S|))$, we prove the more general statement that $F \in \bar{E}(\alpha(k) \cup \beta(k))$ for $k \in \{0, \ldots, |S|\}$. Clearly, if our general statement holds, we can show that $F \in \bar{E}(\beta(|S|)) = \bar{E}(R_p)$ by setting $k = |S|$.

Base case: We set $k = 0$. Then $F = E(\alpha(0)) \in \bar{E}(\alpha(0)) = \bar{E}(\alpha(0) \cup \beta(0))$.

Induction: Suppose that our statement holds for $k = n$, i.e., $F = E(\alpha(0)) \in \bar{E}(\alpha(n) \cup \beta(n))$. We want to show that the same expression holds for $k = n + 1$ where $n + 1 \leq |S|$. We use the general incremental property by setting $P_1 = s_{n+1}$ and $P_2 = \alpha(n+1) \cup \beta(n+1)$. The first condition $P_1 \subseteq P_2$ is satisfied because $\beta(n + 1)$ contains P_1. We then set $F_1 = E(P_1) = E(s_{n+1})$ and $F_2 = E(F_1 \cup (P_2 - P_1)) = E(E(s_{n+1}) \cup \alpha(n+1) \cup \beta(n)) = E(\alpha(n) \cup \beta(n))$. The general incremental property tells us that $F_2 \in \bar{E}(P_2) = \bar{E}(\alpha(n+1) \cup \beta(n+1))$. Thus, any $E(\alpha(n) \cup \beta(n)) \in \bar{E}(\alpha(n + 1) \cup \beta(n + 1))$. Using our induction hypothesis, we conclude that $F = E(\alpha(0)) \in \bar{E}(\alpha(n) \cup \beta(n)) \subseteq \bar{E}(\alpha(n + 1) \cup \beta(n + 1))$. □

We now propose Algorithm 4 that runs ER on previously resolved records and can be used as the $Resolve$ function in Algorithm 3. For example, suppose that $c = \{r_1, r_2, r_3, r_4, r_5\}$ and c's children $c.ch = \{\{r_1, r_2, r_3\}, \{r_4, r_5\}\}$. Also say that $h(\{r_1, r_2, r_3\}) = \{\{r_1, r_3\}, \{r_2\}\}$, and $h(\{r_4, r_5\}) = \{\{r_4, r_5\}\}$. We thus construct R_p as $\{\{r_1, r_3\}, \{r_2\}, \{r_4, r_5\}\}$ in Steps 4–5. Alternatively, if h did not contain any ER result, then at Step 7

ALGORITHM 4: Efficient $Resolve$ function using information on previously resolved records

1: **Input:** an ER algorithm E, a set c of records to resolve and a hash table h containing ER results of sets of records
2: **Output:** $F \in \bar{E}(\{\{r\}|r \in c\})$
3: $R_p \leftarrow \emptyset$
4: **for** $s \in c.ch$ **do**
5: $\quad R_p \leftarrow R_p \cup h(s)$
6: **if** $R_p = \emptyset$ **then**
7: $\quad R_p \leftarrow \{\{r\}|r \in c\}$
8: **return** $E(R_p)$

R_p would have been set to the singleton partition of c, i.e., $\{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}, \{r_5\}\}$. The algorithm then returns $E(R_p)$. In the former case where h does contain ER results of previously resolved records, Algorithm 4 is presumably faster than simply running ER from the singleton partition of c by avoiding redundant record comparisons.

The following result shows the correctness of Algorithm 4.

Proposition 2.3.4. *If E is general incremental (satisfying Definition 2.3.2.1), Algorithm 4 correctly returns an ER result $F \in \bar{E}(\{\{r\}|r \in c\})$.*

Proof. In the case where $R_p \neq \emptyset$ in Step 5, we have $R_p = \bigcup_{s \in c.ch} E(s)$. By Proposition 2.3.3, the final ER result $E(R_p) = E(\bigcup_{s \in c.ch} E(s)) \in \bar{E}(c)$. Otherwise, if $R_p = \emptyset$ in Step 5, then again by Proposition 2.3.3, $E(R_p) = E(\{\{r\}|r \in c\} = E(\bigcup_{r \in c} E(\{r\})) \in \bar{E}(c)$. □

We now show that the HC_S algorithm is general incremental and can thus be used in Algorithm 4.

Proposition 2.3.5. *The HC_S algorithm is general incremental.*

Proof. We first define the notation of connectedness for HC_S. Two records r and s are connected under D, T, and R_p if there exists a sequence of records [r_1 (= r), ..., r_n (= s)] where for each pair (r_i, r_{i+1}) in the path, either $D(r_i, r_{i+1}) \leq T$ or $\exists c \in R_p$ s.t. $r_i \in c, r_{i+1} \in c$.

We now prove the following Lemma.

Lemma 2.3.6. *Two records r and s are connected under D, T, and R_p if and only if r and s are in the same cluster in $E(R_p)$ using the HC_S algorithm.*

Proof. Suppose that r and s are in the same cluster in $E(R_p)$. If r and s are in the same cluster in R_p, then r and s are trivially connected under D, T, and R_p. Otherwise, there exists a sequence of merges of the clusters in R_p that grouped r and s together. When two clusters c_i and c_j in R_p merge, all the records in c_i and c_j are connected by transitivity because two records within c_i or c_j are trivially connected and there exists at least one pair of records from c_i and c_j whose distance according to D does not exceed T. Furthermore, for any two clusters (not necessarily in R_p) that merge, all the records in the two clusters are also connected by transitivity. Since the clusters containing r and s merged at some point, r and s are thus connected under D, T, and R_p. Conversely, suppose that r and s are connected as the sequence $[r_1(=r), \ldots, r_n(=s)]$ under D, T, and R_p. If r and s are in the same cluster in R_p, they are already clustered together. Otherwise, all the clusters that contain $r_1, \ldots, r_n$ eventually merge together according to the HC_S algorithm, clustering r and s together. □

Lemma 2.3.6 directly implies that HC_S returns a unique solution. Suppose that there are two possible solutions for $E(R_p)$: F_1 and F_2. Without loss of generality, suppose that the records r and s are in the same cluster in F_1, but not so in F_2. Then r and s are connected under D, T, and R_p according to F_1 and Lemma 2.3.6, but not connected according to F_2, which is a contradiction.

We now prove that the HC_S algorithm is general incremental. In Definition 2.3.2.1, suppose that the three conditions hold, i.e., $P_1 \subseteq P_2$, $F_1 \in \bar{E}(P_1)$, and $F_2 \in \bar{E}(F_1 \cup (P_2 - P_1))$. Since HC_S returns a unique solution regardless of the order of records resolved, the ER results $E(P_2)$ and $E(F_1 \cup (P_2 - P_1))$ are both unique. □

2.4 Ordered List of Records

We now propose an ordered list of records as a format for hints. In comparison to a partition hierarchy, a list of records tries to maximize the number of matching records identified when the list is resolved sequentially. Two significant advantages are that the ER algorithm

itself does not have to change in order to exploit the information in a record list and that there is no required storage space for the hint. On the downside, finding the right ordering of records in order to guide the ER algorithm to find matching records as much as possible is a non-trivial task where the best solution depends on the ER algorithm itself. We propose general techniques for constructing record lists either from a partition hierarchy or from sampling. We then discuss how a record list can be used by ER algorithms.

2.4.1 Generation

We propose methods for efficiently constructing a list of records. The following section uses a partition hierarchy for generation. We also discuss how record lists can be generated using distance estimation functions.

Using Partition Hierarchies

We propose a technique for generating record lists based on a partition hierarchy. Assuming that an ER algorithm resolves records in the input list from left to right, a desirable feature of a record list is to order the records such that the ER algorithm can minimize the number of fully identified entities at any point of time. A fully identified entity is one where the ER algorithm has found all the matching records for that entity. For example, given a record list $[r_1, r_2, r_3]$ where r_1 refers to the same entity as r_2, an ER algorithm fully identifies the entity for $\{r_1, r_2\}$ after resolving the first two records and fully identifies the entity for $\{r_3\}$ after resolving the last record. Another input list could be $[r_3, r_1, r_2]$ where one entity (i.e., $\{r_3\}$) is already identified after resolving the first record in the list. The first list is better as a record list in a sense that the only record match between r_1 and r_2 was found early on. The second list is worse because $\{r_3\}$ was fully identified early on, and the comparison between r_1 and r_3 was unnecessary and could have been done after matching r_1 and r_2. That is, if we are only able to do one record comparison, then we will find the correct answer when using the record list $[r_1, r_2, r_3]$ and not when using the list $[r_3, r_1, r_2]$.

In general, we want to minimize the entities that are fully identified because they generate unnecessary comparisons with newer records resolved. We will later capture this idea by minimizing the expected number of fully-identified entities when the record list is

resolved sequentially from left to right. While we can use other orderings for generating a record list hint, our generation focuses on ER algorithms that follow the guideline in Section 2.4.2 where records in the front of the list are compared first.

Given a partition hierarchy H with L levels, we assume each of the partitions $P_1, \ldots, P_L$ are equally likely to be the ER answer. That is, each partition has the same chance of being the correct ER result of R and is thus a possible world of the records resolved. Suppose that we resolve a subset S of R. For each partition P_j, we estimate the number of clusters that are fully identified by resolving S as $nEntities_j(S) = \Sigma_{c \in P_j} \frac{|c \cap S|}{|c|}$. Since each partition is equally likely to be the answer, we define the overall estimate of the number of entities fully identified by resolving S as $\Sigma_{j=1 \ldots L}(\frac{1}{L} \times nEntities_j(S))$.

For example, suppose that the partition hierarchy H_1 has 3 levels where $P_1 = \{\{r_1, r_2\}, \{r_3\}, \{r_4, r_5\}\}$, $P_2 = \{\{r_1, r_2, r_3\}, \{r_4, r_5\}\}$, and $P_3 = \{\{r_1, r_2, r_3, r_4, r_5\}\}$. Each partition is equally likely to be the ER answer. Suppose that we resolve the set of records $S = \{r_1, r_2, r_4, r_5\}$, which is a subset of R. Then according to our definition of $nEntities$, the estimated number of entities identified in P_1 is $\frac{2}{2} + \frac{2}{2} = 2$ because all records in $\{r_1, r_2\}$ and $\{r_4, r_5\}$ have been resolved. For P_2, the estimation is $\frac{2}{3} + \frac{2}{2} = \frac{5}{3}$ because 2 out of 3 records in $\{r_1, r_2, r_3\}$ and all records in $\{r_4, r_5\}$ have been resolved. For P_3, the estimation is $\frac{4}{5}$. Our overall estimate for the actual number of entities identified $nEntities(S)$ is thus $\frac{1}{3} \times (2 + \frac{5}{3} + \frac{4}{5}) = \frac{67}{45}$, i.e., about 1.5 entities identified.

One could extend our model by allowing each partition to have its own probability of being the ER answer. That is, for each possible world P_i, we add a probability w_i indicating the confidence we have on that possible world. Given that the sum of the weights is 1, the estimated number of fully identified entities for the set S resolved would be $\Sigma_{j=1 \ldots L}(w_j \times nEntities_j(S))$. While we have considered the extension, we have chosen the current simple scheme for two reasons. First, setting the probabilities for each partition is difficult in practice. Second, the simple scheme usually performs as well as any other scheme using different probabilities (see Section 2.7.6).

We now define an optimal record list. Intuitively, we would like to minimize the number of entities fully identified at any point in time given that the ER algorithm resolves the records in the input list from left to right. We define a prefix set of a list to be the set of records from the beginning of the list. For example, the prefix sets of the list $[r_1, r_2]$ are $\{\}$,

$\{r_1\}$, and $\{r_1, r_2\}$.

Definition 2.4.0.1. *A record list H of R is* optimal *if any prefix set P of H has a minimum value of $\Sigma_{j=1...L}(\frac{1}{L}\times nEntities_j(P))$ among all subsets of R with size $|P|$.*

Interestingly, we can always derive a record list that is optimal according to Definition 2.4.0.1. A key observation is that $nEntities\ (S) = \Sigma_{r\in S}\ nEntities(\{r\})$, which says that the expected number of entities identified by a set S is the sum of the expected numbers of entities identified by the records in S. For example, according to H_1 (defined above), $nEntities(\{r_1, r_3\}) = \frac{1}{L} \times (nEntities_1(\{r_1, r_3\}) + nEntities_2(\{r_1, r_3\}) + nEntities_3(\{r_1, r_3\})) = \frac{1}{L} \times ((\frac{1}{2} + \frac{1}{1}) + \frac{2}{3} + \frac{2}{5}) = \frac{1}{3} \times (\frac{1}{2} + \frac{1}{3} + \frac{1}{5}) + \frac{1}{3} \times (\frac{1}{1} + \frac{1}{3} + \frac{1}{5}) = nEntities(\{r_1\}) + nEntities(\{r_3\})$. Hence, by simply sorting the records in R by their $nEntities$ values in increasing order (remember, we want to *minimize* the number of fully-resolved records), we can derive a record list where any prefix set has an optimal $nEntities$ value.

Algorithm 5 derives an optimal record list according to Definition 2.4.0.1. Using Steps 5–8 we compute the estimated number of entities for each record. According to H_1 above, record r_3 has a $nEntities$ value of $\frac{1}{3} \times (\frac{1}{1} + \frac{1}{3} + \frac{1}{5}) = \frac{46}{90}$. Similarly, r_1 and r_2 each have a value of $\frac{1}{3} \times (\frac{1}{2} + \frac{1}{3} + \frac{1}{5}) = \frac{31}{90}$. Finally, records r_4 and r_5 each have a value of $\frac{1}{3} \times (\frac{1}{2} + \frac{1}{2} + \frac{1}{5})$ = $\frac{36}{90}$ (the fractions were not reduced for easy comparison). We now sort the records by their $nEntities$ values (Step 9) in increasing order. In our example, we can produce the record list $H' = [r_1, r_2, r_4, r_5, r_3]$. (Records with the same $nEntities$ value can swap positions within the list.) As a simple verification, no prefix set of size 2 has a $nEntities$ value smaller than that of $\{r_1, r_2\}$, which is $\frac{31}{90} + \frac{31}{90} = \frac{62}{90}$.

We now show that Algorithm 5 returns an optimal record list.

Proposition 2.4.1. *Algorithm 5 returns an optimal record list.*

Proof. We first prove that $nEntities(S) = \Sigma_{r\in S} nEntities(\{r\})$. The reason is that $nEntities(S)$ $= \Sigma_{j=1...L}(\frac{1}{L} \times \Sigma_{c\in P_j} \frac{|S\cap c|}{|c|}) = \Sigma_{r\in S}\Sigma_{j=1...L}(\frac{1}{L} \times \Sigma_{c\in P_j} \frac{|\{r\}\cap c|}{|c|}) = \Sigma_{r\in S} nEntities(\{r\})$. Now given a prefix S of R sorted by $nEntities$ values, we show that $nEntities(S) \leq nEntities(S')$ for any $S' \subseteq R$. Suppose that there exists an $S' \subseteq R$ such that $nEntities(S) > nEntities(S')$ while $|S| = |S'|$. Then there exists an $r' \notin S$ where $nEntities(\{r'\})$ is smaller than the

ALGORITHM 5: Generating an optimal record list from a partition hierarchy

1: **Input:** the set of records R, a hint $H = \{P_1, \ldots, P_L\}$
2: **Output:** an optimal record list H'
3: **for** $r \in R$ **do**
4: $nEntities(\{r\}) = 0$
5: **for** $i = 1 \ldots L$ **do**
6: **for** $c \in P_i$ **do**
7: **for** $r \in c$ **do**
8: $nEntities(\{r\}) \leftarrow nEntities(\{r\}) + \frac{1}{L} \times \frac{1}{|c|}$
9: $H' \leftarrow$ Sorted records in R by their $nEntities$ values in increasing order
10: **return** H'

$nEntities$ value of the $|S|$th record in the record list. However, we have just contradicted the fact that the record list is sorted by the $nEntities$ values. □

The complexity of Algorithm 5 is $O(|R| \times (L + log(|R|)))$ because of the loop in Steps 5–8 and the sorting of records in Step 9. In special cases, however, the sorting can be done in linear time. For example, if there is only one level P in the hierarchy, then for a record r, $nEntities(\{r\}) = \Sigma_{c \in P} \frac{|\{r\} \cap c|}{|c|} = \frac{1}{|c_r|}$ where c_r is the cluster in P containing r. Hence, we can sort the records in R by the sizes of clusters that contain them in decreasing order. If we are given a maximum cluster size constant $MaxSize$ of P, we can create $MaxSize$ buckets and assign each record r to the bucket with the index $|c_r|$. We can then generate an optimal record list by iterating through all the buckets. The sorting can thus be done in $O(|R|)$ time. Using our example above, suppose that $P = \{\{r_1, r_2\}, \{r_3\}, \{r_4, r_5\}\}$. Then one optimal record list is $[r_1, r_2, r_4, r_5, r_3]$ because the cluster in P containing r_3 has a size of 1 while the cluster sizes for the other four records are all 2. Here, we can sort the five records using two buckets (i.e., $MaxSize = 2$).

Obviously, there are other ways to generate record lists using a partition hierarchy. For example, one could simply return the records in one of the partitions of the partition hierarchy. Our approach is a general way to produce record lists and has theoretical guarantees of minimizing the expected number of entities identified at any point in time.

ALGORITHM 6: Generating a record list based on a list of pairs

1: **Input:** a list of pairs with their estimated distances $L = [\langle r_1, s_1\rangle, \langle r_2, s_2\rangle, \ldots]$
2: **Output:** a record list H
3: $H \leftarrow []$
4: Sort L by estimated distances in increasing order
5: **for** $\langle r, s\rangle \in L$ **do**
6:     **if** $r \notin H$ **then**
7:         $H \leftarrow H + r$
8:     **if** $s \notin H$ **then**
9:         $H \leftarrow H + s$
10: **return** H

Using Distance Estimation Functions

We can also use an inexpensive distance estimator function $e(r, s)$ (application specific or sampling based) to generate a record list. Algorithm 6 shows how we can generate a list that resembles the given pair list. For example, given the pair list $[\langle r_1, r_2\rangle, \langle r_1, r_3\rangle, \langle r_2, r_3\rangle]$, we first read the pair $\langle r_1, r_2\rangle$ and append the records r_1 and r_2 to H. For the next pair $\langle r_1, r_3\rangle$, we only need to append r_3 to H because r_1 already exists in H. Hence, we generate the record list $H = [r_1, r_2, r_3]$. Of course, the record list H does not necessarily preserve all the information in the pair list. While some information cannot avoid being lost, we make the best effort to place the most likely matching records up front in the record list.

Unlike Algorithm 2 where a partition hierarchy hint can be constructed without sorting the list of pairs, a record list requires the list of pairs to be sorted by the estimated distances of the pairs. Hence, the complexity of Algorithm 6 is $O(|R|^2 \times log(|R|))$ where R is the set of input records. Moreover, a space complexity of $O(|R|^2)$ is required to store the estimated distances of all record pairs. In the case where there is limited time or space, we can approximate the result of Algorithm 6 by only retaining the top-k closest pairs where k is a parameter reflecting the limited time or space. A heap structure can be used to store the top-k pairs while estimating the pairwise distances in the sampling scheme. We then consider all the other pairs to be infinitely distanced. The time complexity of Algorithm 6 is then $O(k \times log(k))$ while the space complexity $O(k)$. In Section 2.7.3, we experimentally show that limiting k can significantly improve the time and space requirements with almost no decrease in quality.

2.4.2 Use

A record list can be applied to any ER algorithm that accepts as input a record list. A key advantage of using record lists is that the ER algorithm itself does not have to change. The following principle can be employed to benefit from a record-list hint:

- If there is flexibility in the order of which records are resolved, resolve the records in the front of the list first.

Again, our goal is to help the ER algorithm with hints to efficiently return an answer F' that has high precision and recall relative to the unmodified answer F.

As an example, we consider hierarchical clustering based on a Boolean match function [10] (called HC_B), which can benefit from record lists. The HC_B algorithm combines matching pairs of clusters in any order until no clusters match with each other. The comparison of two clusters can be done using an arbitrary function that receives two clusters and returns true or false, using the Boolean comparison function B to compare pairs of records. For example, suppose we have $R = \{r_1, r_2, r_3\}$ (which can also be viewed as a list of three singleton clusters) and the comparison function B where $B(r_1, r_2)$ = true, $B(r_2, r_3)$ = true, but $B(r_1, r_3)$ = false. Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs (e.g., a record r_2 has an ID of 2) from each cluster using B. For instance, when comparing $\{r_1, r_2\}$ with $\{r_3\}$, we return the result of $B(r_1, r_3)$. Depending on the order of clusters compared, the HC_B algorithm can merge $\{r_1\}$ and $\{r_2\}$ first, or $\{r_2\}$ and $\{r_3\}$ first. In the first case, the final ER result is $\{\{r_1, r_2\}, \{r_3\}\}$ (because the clusters $\{r_1\}$ and $\{r_2\}$ match, but $\{r_1, r_2\}$ and $\{r_3\}$ do not match) while in the second case, the ER result is $\{\{r_1, r_2, r_3\}\}$ (the clusters $\{r_2\}$ and $\{r_3\}$ match, and then $\{r_1\}$ and $\{r_2, r_3\}$ match). Now given a record list $[r_1, r_2, r_3]$ (the ordering is arbitrary and is set to illustrate the behavior of HC_B), the HC_B algorithm first compares r_1 and r_2. If we set the work limit W to one record comparison, then HC_B will terminate returning $\{\{r_1, r_2\}, \{r_3\}\}$.

2.5 Using Multiple Hints

Until now, we have assumed that one hint is generated per block. However, depending on the type of hint and the number of attributes used to generate the hint, we may have to generate multiple hints in order to accurately capture the order information of record pairs. For example, suppose that we are resolving people records and there are two ways to order the pairs of records: by their name or address similarities. If we are generating a list of pairs hint, then we could choose one of two attributes – name or address – and use it to sort the pairs into one hint. Another option is to combine the name and address similarity into one similarity (e.g., by taking a weighted sum of the values) and generate one hint. Finally, we can generate two separate hints for the two attributes. In this section, we assume that multiple hints of the same type are generated (corresponding to the last case in the above example) and discuss three options on how to exploit them while resolving records.

The first straightforward option is to repeatedly resolve the block of records for each hint and combine the results. For example, we could resolve the people with the closest names first and then resolve those with the closest addresses first and union the matching records. While this method is easy to apply to any type of hint, the overall runtime of resolving records may slow down due to redundant record comparisons for different hints.

Another option is to merge the multiple orderings into one ordering and then resolve the records once using this new combined hint. For example, given two sorted lists of records, we can merge the lists by sorting the records according to their sum of ranks in the two sorted lists. As another example, if we are combining two partition hierarchies, we could combine each level by performing a meet operation on the corresponding partitions. While combining hints may result in a loss of information of the ordering of pairs, the main advantage is that there is only one hint to use when resolving records and thus no redundant record comparisons.

The final option is to exploit the multiple hints simultaneously without combining them into one hint. For example, if there are two sorted lists of record pairs, then we can start reading the first pairs of records from both hints. If any record pair from one hint has already been read from the other hint, then we can read the next pair of records. While this option has the potential to fully exploit the ordering information of all the hints, deciding

how exactly we can exploit the multiple hints is not obvious.

2.6 Determining Which Hint to Use

As mentioned in Section 2.1.2, an ER algorithm may only be compatible with some types of hints (or with none at all), depending on the data structures and processing used. In this section we provide some hint selection guidelines and then illustrate how the guidelines apply to the ER algorithms we have already introduced.

If the ER algorithm compares pairs of records, and there is an estimator function e that is cheaper than the distance function d, a pair-list hint may be useful. If there is no estimator function e, then sampling techniques can be used to estimate the other distances. Next, if the ER algorithm clusters records based on their relative distances, then a hierarchy hint could be useful for focusing on the relatively closer records first. Finally, if the ER algorithm performs a sequential scan of records when resolving them, a record list hint may help compare the records that are more likely to match first.

Figure 2.4 summarizes our three hint types and the techniques used to generated them (see Section 2.7.1 for details). The figure also shows the ER algorithms we used in Sections 2.2 through 2.4 to illustrate each hint type. Although we could use a hierarchy hint or a record-list hint for the SN algorithm, the pair-list hint can be used most naturally because SN basically compares pairs of records that are likely to match in a given order. We use a partition hierarchy hint for the HC_S algorithm because HC_S can naturally resolve subsets of R with the guidance of the partition hint. While HC_S can also use a record list as a hint, the record list is designed to work better for ER algorithms that resolve records sequentially. For the HC_B algorithm we use a record lists hint because HC_B sequentially resolves its records. The HC_B algorithm could also use a partition hierarchy as its hint. However, we would have to modify HC_B and thus change its efficient algorithm for comparing records.

2.7 Experimental Results

In this section, we evaluate pay-as-you-go ER on real data sets and show how creating and using hints can improve the ER quality given a limit on how much work can be done. We

Hint	Generated from	ER algorithm
Pair list (PL)	Cheap dist. functions Sampling	SN
Hierarchy (H)	Sorted records Hash functions Sampling	HC_S
Record list (RL)	Partition hierarchy Complete sampling Partial sampling	HC_B

Figure 2.4: Hints to generate and ER algorithms to run

assume that blocking [77] is used (see Section 2.1), as it is in most ER applications with massive data. With blocking, the input records are divided into separate blocks using one or more key fields. For instance, if we are resolving products, we can partition them by category (books, movies, electronics, etc.). Then the records within one block are resolved independently from the other blocks. This approach lowers accuracy because records in separate blocks are not compared, but makes resolution feasible. (See [71, 109] for more sophisticated approaches.) From our point of view, the use of blocking means that we can read a full block (which can still span many disk blocks) into memory, perform resolution using hints, and then move on to the next block. In our experiments we thus evaluate the cost of resolving a single block, except for Section 2.7.8 where we perform scalability experiments by resolving multiple blocks. Keep in mind that these costs should be multiplied by the number of blocks. Finally in our experiments, we generate *one hint* for each block. Our approach can easily be extended to multiple hints using the techniques described in Section 2.5.

We start by describing our experimental setting in Section 2.7.1. In Section 2.7.2, we show how using hints can improve the ER quality with limited amounts of work. In Section 2.7.3, we investigate the CPU time and space overhead for creating hints and discuss the tradeoffs between the overhead and benefit of using hints. In Section 2.7.4 we investigate the right number of levels in a partition hierarchy hint. In Section 2.7.5, we explore the impact of the sample size on the accuracy of hints using sampling techniques. In Section 2.7.6, we experiment on record lists generated from partition hierarchies using an extended model where partitions in the hierarchy now have different confidence values.

In Section 2.7.7, we discuss how partition hierarchy hints can still enhance ER algorithms that are not incremental. In Section 2.7.8, we show how hints can be used to enhance ER in practical scenarios where the datasets can be very large. Finally, in Section 2.7.9, we show how our hint generation algorithms scale.

2.7.1 Experimental setting

In this section, we describe the settings used for our experiments. Our algorithms were implemented in Java, and our experiments were run on a 2.4GHz Intel(R) Core 2 processor with 4 GB of RAM.

Quality Metric

Since ER results may now be incomplete, it is important to measure the quality of an intermediate ER result. We compare an intermediate result with a "Gold Standard," which is the result of running ER on the same dataset to the end. Notice that we are *not* measuring the correctness of the ER algorithm itself, but instead determining how "close" the intermediate results are to the exhaustive result. Since ER results are partitions of the input set of records, we consider all the input records in the same output cluster to be identical. For instance, if records r and s are clustered into $\{r, s\}$ and then clustered with t, all three records r, s, t are considered to be the same (i.e., to match).

Suppose that the Gold Standard G contains the record pairs that match for the exhaustive solution while set S contains the matching pairs for the intermediate result. Then the precision Pr is $\frac{|G \cap S|}{|S|}$ while the recall Re is $\frac{|G \cap S|}{|G|}$. If the precision Pr is always 1 (i.e., the incremental algorithm always reports true matches), we use Re, the fraction of matching pairs found, as our quality metric. Otherwise, we can use the F_1 metric, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, as the quality metric (for more general ER metrics, see [73]).

We will use recall as our metric for all the experiments sections (except for Section 2.7.7) because the precision is always 1 for any intermediate ER result.

Real Data

The comparison shopping dataset we use was provided by Yahoo! Shopping and contains millions of records that arrive on a regular basis from different online stores and must be resolved before they are used to answer customer queries. Each record contains attributes including the title, price, and category of an item. We experimented on a random subset of 3,000 shopping records that had the string “iPod” in their titles and 2 million shopping records. When scaling ER on 2 million shopping records (see Section 2.7.8), the average block size was 124 records while the maximum block size was 6,082 records. Hence, the random subset of 3,000 shopping records can be considered as one (relatively large) block. We also experimented on a hotel dataset provided by Yahoo! Travel where tens of thousands of records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users. We experimented on a random subset of 3,000 hotel records located in the United States. Each hotel record contains attributes including the name, address, city, state, zip code, latitude, longitude, and phone number of a hotel. Again, the 3,000 hotel records can be considered as one block. While the 3K shopping and hotel datasets fit in memory, the 2 million shopping dataset did not fit in memory and had to be stored on disk.

Hints and ER Algorithms

For our experiments we use the three ER algorithms used to illustrate our hints (and summarized earlier in Figure 2.4). In this sub-section we provide some implementation details for the ER algorithms used.

The SN algorithm uses a Boolean match function for comparing two records. When comparing shopping records, the Boolean match function B compares the titles, prices, and categories. When comparing hotel records, B compares the states, cities, zip codes, and the names of the two hotels. We generate a pair list using cheap distance functions or from sampling. When generating record lists using cheap distance functions, we used the estimate function $e(r, s) = |Rank(r) - Rank(s)|$ using the title (name) attributes of shopping (hotel) records as the sort key. When generating pair lists using sampling, we only computed and stored the top-($(w-1) \times |R| - \frac{w \times (w-1)}{2}$) closest pairs (i.e., the number

of record pairs that would be compared by SN given the window size w) to limit the time and space overhead. We also used a random sample of 10 records.

The HC_S algorithm uses a distance function for comparing two records. When the comparing shopping records, the distance function D measures the Jaro distance [111] between the titles of two records. For the hotel records, D measures the Jaro distance of the names of two records. We generate partition hierarchies in three ways: using sorted records, hash functions, and sampling. By default, we set the number of levels of a partition hierarchy to 5. While increasing the number of levels helps us find more matching records early on, the benefits diminish from a certain point (see Section 2.7.4). The partition hierarchies based on sorted lists were balanced binary trees with the highest level containing a single cluster with all input records. The partition hierarchies based on hash functions used the prefixes of titles (names) as the hash values of shopping (hotel) records. When generating partition hierarchies using sampling, we clustered records with similar titles (for the shopping dataset) or names (for the hotel dataset) using several string comparison thresholds. We randomly selected 10 records for our samples. (In Section 2.7.5, we show that small sample sizes are sufficient for reasonable results.) A partition hierarchy is suitable for the HC_S algorithm because the hint suggests sets of records to resolve first, and the HC_S algorithm can easily resolve subsets of records at a time.

The HC_B algorithm uses a Boolean match function for comparing two records. When comparing shopping records, the Boolean match function B compares the titles, prices, and categories. When comparing hotel records, B compares the states, cities, zip codes, and the names of the two hotels. We generate a record list from a partition hierarchy (generated with hash functions) and from sampling. When generating a partition hierarchy used for constructing a record list, we used minhash signatures [57] generated from titles (names) as the hash values of shopping (hotel) records. When generating record lists using sampling, we tested two schemes. For the complete sampling scheme, we computed and stored all the estimate distances of pairs (i.e., $\frac{|R|\times(|R|-1)}{2}$ pairs) and generated a record list. For the partial sampling scheme, we only computed and stored the top-(5 $\times|R|$) closest pairs to limit the time and space overhead. In both schemes, we used a random sample of 10 records.

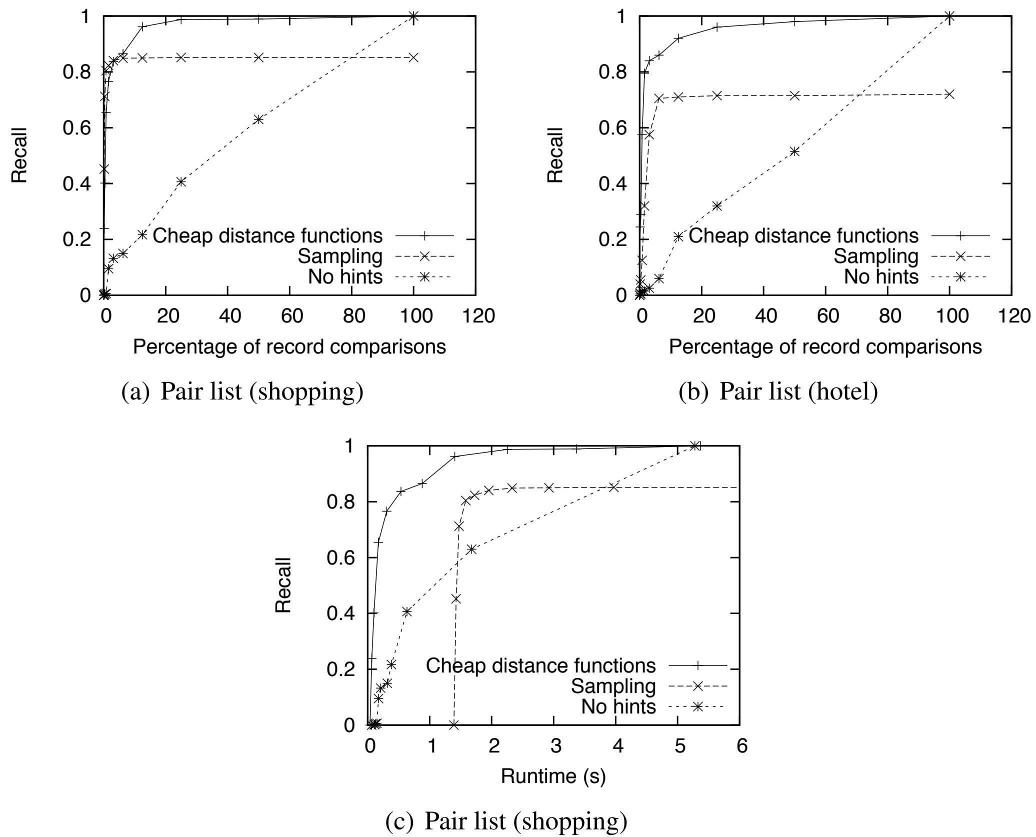
In all our algorithms, we avoid expensive comparisons when possible by comparing in phases. For example, when comparing two shopping records, we compare the category,

price, and title attributes, in that order. If the categories do not match, we avoid comparing the prices and titles. If the categories match, but not the prices, we avoid comparing the titles. This way, we can avoid many expensive title string comparisons. When experimenting on large datasets, we use various blocking techniques (see Section 2.7.8) to further scale ER. While more optimizations can be used on the base ER algorithms, our focus is to show the *relative* benefits of using hints compared to when they are not used.

2.7.2 Hint Benefit

In this section, we explore the benefits of using hints by measuring the recall values for various ER algorithms using different hints. Figure 2.5(a) shows how a pair list can help the SN algorithm compare the most likely matching record pairs for 3,000 shopping records. We experimented on the SN algorithm using two types of hints. Recall that the SN algorithm first sorts the records by a certain key. In our implementation, we sorted the records by their titles and then slid a window of size 100, comparing only the record pairs within the same window. The first hint we used was to order the pairs of records according to their difference in rank according to the sorted list. That is, the difference in rank was considered the distance between two records. The second hint we used estimated the pairwise distance between the records using the sampling technique (see Section 2.2.1) and compared the records with the closest estimated distance first. In our experiments, we set the sample size to 10 records. (In Section 2.7.5, we show that even a sample this small produces reasonable results.) Notice that when using the sampling technique, the SN algorithm does not use a sliding window on a sorted list of the records, but simply compares the pairs of records as dictated by the pair list.

As more records are compared using the match function B, the quality of SN using hints rapidly increases. For example, the quality of SN using a pair list generated from cheap distance functions achieves 0.96 recall with only 12.5$\%$ of the record comparisons required when running SN without hints. The quality of SN using the sampling technique achieves 0.8 recall with 0.78$\%$ of the entire work. While the sampling techniques gives a high recall early on, it does not give 1.0 recall even after performing as many comparisons as the SN algorithm without hints. The reason is that there are still matching record pairs



(a) Pair list (shopping)

(b) Pair list (hotel)

(c) Pair list (shopping)

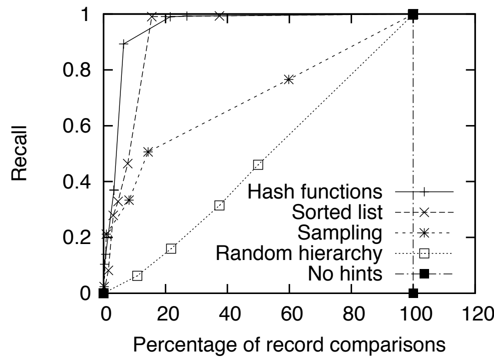
that would have been found by SN without hints, but are further down the pair list and will eventually be compared if more pairs are compared (recall that the SN algorithm only compares a small fraction of the total record pairs using a sliding window). In Section 2.7.5, however, we show that the sampling technique is actually very good at finding all matching pairs that are not necessarily within the same window. Finally, the recall of SN without hints increases linearly with more record comparisons.

Figure 2.5(d) shows how a partition hierarchy can help the HC_S algorithm to quickly identify matching records for 3,000 shopping records. The bottom-right plot (in Figure 2.5(d)) shows the progress of the original HC_S algorithm where records are clustered only after all pairs of base records are compared. Notice that the clustering of records does not involve record comparisons, which is why the original HC_S algorithm has a jump in recall from 0 to 1 when 100% of the record comparisons are done. The actual runtime for the second clustering step is very small (0.004s). The random hierarchy plot shows how

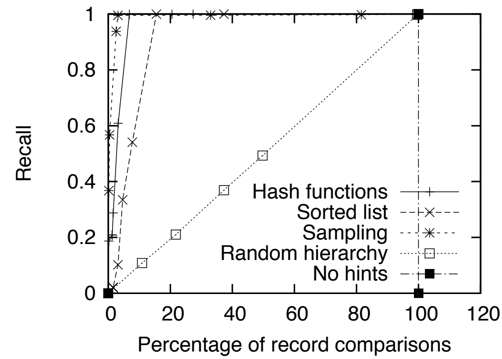
a randomized partition hierarchy helps the ER quality. Here, the records are clustered in a random fashion without any similarity comparisons. As a result, the plot shows a linear increase of recall as the number of record comparisons increases. The other three plots use partition hierarchies generated from a sorted list, hash functions, and sampling. Among them, a partition hierarchy based on sampling gives the slowest increase in recall where we get 0.51 recall with 14% of the comparisons HC_S uses without hints. The main reason for the relatively low recall is that the partitions in the hierarchy were highly skewed where some clusters in a partition were very large. As a result, the partitions in the hierarchy were not "pinpointing" the likely matching records. Moreover, setting the thresholds for creating the partitions was not a trivial task, making this approach relatively difficult to use. When using a partition hierarchy hint generated from a sorted list, we achieve 0.99 recall with 16% of the total comparisons of HC_S without hints. Finally, when using a hint generated using hash functions, we achieve a similar result of 0.89 recall using 6.5% of the total comparisons.

Figure 2.5(g) shows how record lists can help the HC_B algorithm to identify matching records early without modifying the ER algorithm itself. Again, we experimented on 3,000 shopping records. When using a record list generated from a partition hierarchy, we obtain 0.61 recall with 50% of the comparisons used by HC_B without hints. Record lists generated from complete or partial sampling give similar results where we obtain 0.67 recall with 50% of the total comparisons. In contrast, the HC_B algorithm without hints obtains 0.47 recall for 50% of its comparisons. While the complete and partial sampling schemes produce near-identical recall results against the number of record comparisons done, we will see in Section 2.7.3 that the partial sampling scheme outperforms the complete sampling scheme in recall against the actual ER runtime. Although the record list does not generally improve HC_B as much as partition hierarchies improve HC_S, the main advantage is that all these benefits were achieved without modifying the HC_B algorithm itself.

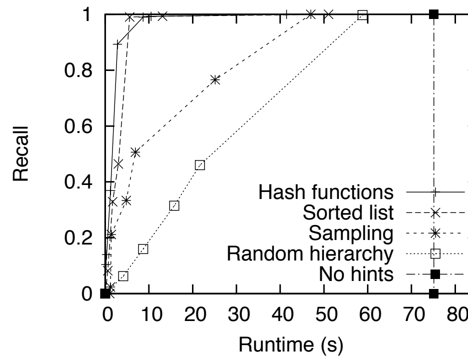
Figures 2.5(b), 2.5(e), and 2.5(h) show the hint results when resolving 3,000 hotel records. Unlike the shopping dataset where multiple records can match, the records in the hotel datasets mostly come from two data sources that do not have duplicates within themselves, so relatively few clusters have a size larger than 2. The hotel results show that a partition hierarchy based on sampling or any record list performs better on hotel data



(d) Partition hierarchy (shopping)



(e) Partition hierarchy (hotel)



(f) Partition hierarchy (shopping)

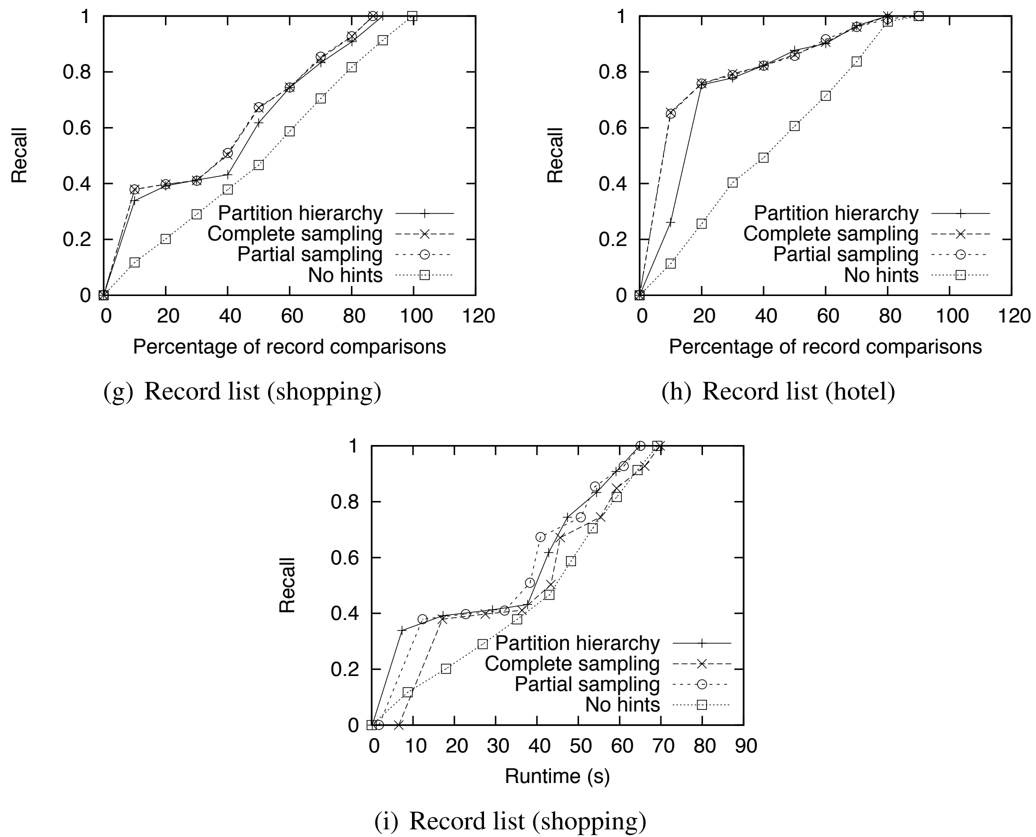
than when they are used on shopping data. Figures 2.5(c), 2.5(f), and 2.5(i) show the recall values of ER algorithms against runtime and will be explained in Section 2.7.3.

2.7.3 Hint Overhead

In this section we explore the CPU and memory space overhead of using hints. We first explore the time and space overhead of constructing and using hints. We then show the tradeoffs between the overhead and benefit of using hints from various perspectives.

Time and Space Overhead

The time overhead of a hint consists of the time to construct the hint and the time to use the hint. While we will measure the construction time for hints, the time overhead of using the hints themselves is not significant. The usage time overhead for accessing a pair list is



(g) Record list (shopping)

(h) Record list (hotel)

(i) Record list (shopping)

Figure 2.5: Recall of ER algorithms using hints against work or runtime, 3K shopping/hotel records

a simple iteration of the pairs in the list. The usage time overhead for accessing a partition hierarchy is an iteration of the clusters from the bottom partition to top. There is no time overhead for using a record list because we simply reorder the input list of records.

The "Time Overhead" column in Figure 2.6 shows the construction time overhead for each type of hint in Figure 2.4 (we explain the space overhead later). The sub-column head Sho3K means 3,000 shopping records while the sub-column head Ho3K means 3,000 hotel records. Each construction time overhead was produced by dividing the construction time of a hint by the CPU time for running the ER algorithm without using any hints. For example, the construction time for a partition hierarchy based on hash functions using 3,000 shopping records is 0.0001x the time for running the HC_S algorithm without hints.

Hint	**Generation**	**Time Overhead**		**Space Overhead (Const/Use)**	
		Sho3K	**Ho3K**	**Sho3K**	**Ho3K**
Pairs List	Cheap dist. functions	0.005	0.19	22 / 22	7.8 / 7.8
	Sampling	0.16	3.56	22 / 22	7.8 / 7.8
Hierarchy	Sorted records	4E-4	2E-4	0.07 / 0.07	0.02 / 0.02
	Hash functions	1E-4	1E-4	0.08 / 0.08	0.03 / 0.03
	Sampling	0.02	0.01	0.08 / 0.08	0.03 / 0.03
Record List	Partition hierarchy	7E-4	0.01	0.08 / 0	0.03 / 0
	Complete sampling	0.09	1.07	349 / 0	119 / 0
	Partial sampling	0.02	0.31	1.15 / 0	0.4 / 0

Figure 2.6: Time and space hint construction overhead depending on the type of hint, 3K shopping/hotel records

The overhead for constructing pair lists based on cheap functions depends on the number of pairs compared (which depends on the window size w). The larger the window size, the larger the construction time overhead. The overhead for constructing pair lists based on sampling is more expensive because all record pairs are compared before taking the top matching pairs. The time overhead for resolving 3,000 hotel records is 3.56x, which means that the time to construct the hint takes longer than running the ER algorithm itself. In this case, it is better to simply run the ER algorithm. The overhead for constructing partition hierarchies based on sorting or hashing is very small compared to running the HC_S algorithm because the record comparisons in HC_S are relatively expensive. Even if sampling is used (which requires a runtime quadratic in the number of input records), the construction time overhead is 0.02x for shopping records because the cost for estimating distances is much cheaper than computing the real distances. The overhead for constructing a record list from a partition hierarchy is relatively small compared to running the HC_B algorithm because, again, the record comparisons in HC_B are relatively expensive. However, when constructing a record list with complete sampling, the time overhead for HC_S resolving 3,000 hotel records is 1.07x. The partial sampling scheme significantly improves the complete sampling scheme where the time overhead for the same hint and data is 0.31x. Note that this improvement comes with almost no penalty in recall (see Figure 2.5(h)).

The "Space Overhead (Const/Use)" column in Figure 2.6 shows the space overhead for each type of hint. The space overhead of a hint consists of the memory space needed

for constructing the hint and the memory space needed to use the hint while running ER. Both of these costs can be significant and will be explored. The words "Const" and "Use" indicate the construction space overhead and usage-space overhead, respectively. The construction space overhead of a hint was computed by dividing the memory space needed for creating the hint by the memory space needed to store the input record list. The usage-space overhead of a hint was computed by dividing the memory space needed for storing the constructed hint by the memory space of the input record list. For example, the construction space overhead of a record list based on a partition hierarchy is 0.08x the space needed to store 3,000 shopping records while the space needed to store and use that hint (i.e., the usage overhead) is 0. Note that the space overhead is dependent on the size of the input records (i.e., if the records are larger, then the space overhead will decrease).

The space overhead for pair lists is proportional to the number of record pairs stored (which depends on the window size w). While the current space overhead for shopping records is 22x, one could reduce the window size to reduce the overhead if necessary. (Of course, reducing the number of pairs stored comes at a price of reducing the recall of SN.) The space overhead is same regardless of the how the list was made because the sampling technique store exactly the same number of record pairs as when using cheap functions. The space overhead for partition hierarchies based on sorted records and hash functions is reasonably small (0.07–0.08x for shopping records) because the hierarchy size is linear in the number of records. A partition hierarchy based on sampling has a reasonable construction space overhead (0.08x for shopping records) because we do not actually store the pairwise distance estimates computed by the sampling technique. The record-list hint based on a hierarchy hint has a construction space overhead of 0.08x because the partition hierarchy hint was based on hash functions. The record-list hint based on complete sampling has a large construction space overhead (349x for shopping records) because of the quadratic space required. This result is the largest space overhead a sampling scheme can have where all distance estimates between records are sorted and stored. The partial sampling scheme, however, shows a much lower and reasonable space overhead (1.15x for shopping records). We achieve this significant improvement with near-identical recall results (see Figures 2.5(g) and 2.5(h)). Finally, both record-list hints do not have usage-space overhead.

Tradeoff between Time Overhead and Benefit

We now observe how the construction time overhead of a hint actually affects the overall runtime of ER. We experiment on 3,000 shopping records. Figures 2.5(c), 2.5(f), and 2.5(i) show the recall values of ER results as a function of the ER runtime. The plots do not differ significantly from Figures 2.5(a), 2.5(d), and 2.5(g), respectively. While the construction time overhead are reflected in the plots, only the plots for using pair lists based on sampling and record lists based on complete and partial sampling show visible construction time overhead. When using pair lists based on sampling, it takes 1.45 seconds for SN to perform better than SN without hints. We also observe that the runtime needed to cluster records by HC_S after the pairwise distances is negligible (0.004s) compared to the total ER runtime. The results show that hints can benefit ER in runtime even with the construction time overhead.

We demonstrate how hints are helpful in finding "most" of the matching record pairs efficiently. Figure 2.7 shows how efficient hints are when obtaining 0.8 recall using 3,000 shopping records. For each hint type, we measure its construction time overhead (x-value). We then measure the time for the ER algorithm using the hint to achieve 0.8 recall divided by the ER runtime without hints (y-value). For example, a partition hierarchy generated from sampling for SN takes 0.01x the time to run HC_S without hints (x-value). Also, the time for HC_S with this hint to get 0.8 recall takes 0.37x the time to run HC_S without hints (y-value). Hence, the total runtime of SN using the hierarchy is 0.38x the runtime for SN without hints. Notice that in the case where the sum of the x and y values of a point is 1 (i.e., if the point is on the diagonal line $X + Y = 1$), then running ER with hints to obtain 0.8 recall takes the same time as running ER fully without hints. Hence, a hint is useful when its point is below the diagonal line. Our results show that all hints have points below the diagonal line, which means that our hints can efficiently identify 80% of the matching record pairs.

Figure 2.8 shows how the construction time of a hint can affect the point when using a hint starts to help. At one extreme, if there is no construction time, then hints can improve ER progress within a short time. On the other hand, if the construction time is very large, it may take many record comparisons until the overhead starts to pay off. For each hint type,

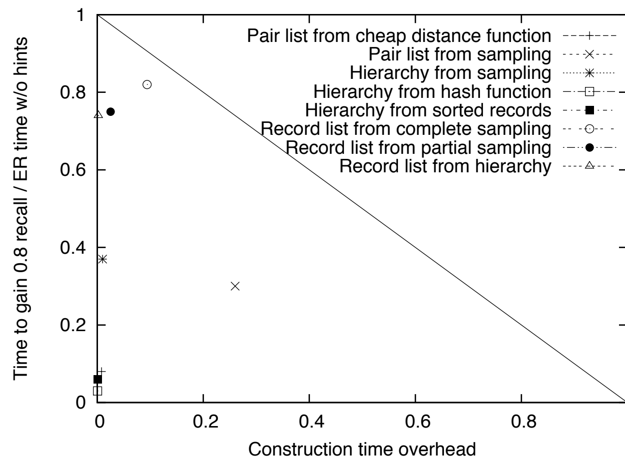


Figure 2.7: Construction time versus time to obtain 0.8 recall, 3K shopping records

we vary the construction time and convert it into number of record comparisons performed. For example, suppose HC_S does Z record comparisons without using hints. Then we can set the construction time of a partition hierarchy based on sampling to be equivalent to, say, 35% of Z. For each construction time, we also derive the number of record comparisons when ER using hints starts to achieve higher recall than ER without hints. When using a partition hierarchy based on sampling it takes about 46% of Z comparisons for the overhead of constructing the hint to pay off (see the right-most black-circle in Figure 2.8). The "ideal" plot would be exactly the $Y = X$ plot where no matter the overhead of constructing the hint, we immediately start benefitting by using the hints. For each hint, there is a point where the hint can no longer benefit ER with larger construction times. For example, if the construction overhead for a partition hierarchy based on sampling exceeds 35% of Z (i.e., if we go beyond the right-most black-circle), then the HC_B algorithm using the hint can never perform better than HC_B without hints.

2.7.4 Choosing the Number of Levels

Figure 2.9 shows the impact of the number of levels in the recall achieved by a given number of record comparisons. We resolved 3,000 shopping records using the HC_S algorithm using a hierarchy hint generated from the records sorted by their titles. Each hierarchy has $\{R\}$ as its highest-level partition and was a binary tree where each cluster $\{r_1, \ldots, r_n\}$ had

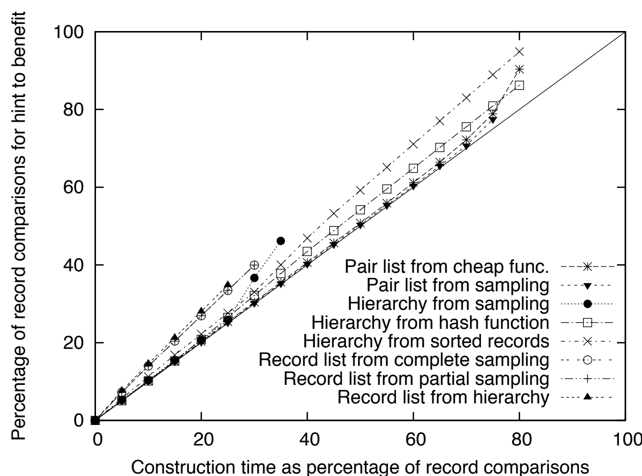


Figure 2.8: Construction time impact on hint payoff point, 3K shopping records

exactly two children $\{r_1, \ldots, r_{\frac{n}{2}}\}$ and $\{r_{\frac{n}{2}+1}, \ldots, r_n\}$. As a result, the more levels there are in the hint (increasing from 3 to 9), the steeper the recall curve becomes. For example, while using a hint with 3 levels gives a 65% recall for 25% of the total comparisons done by HC_S without hints, using a hint with 7 levels gives a 98% recall with only 15% of the total comparisons. Starting from 7 levels, however, the recall improvement becomes negligible. When the number of levels increase from 3 to 9, the hint construction time overhead ranges from 1.4E-4x to 5.3E-4x and the space overhead 0.07x to 0.08x. Hence, the time and construction space overhead do not significantly change with varying numbers of levels.

2.7.5 Sampling Performance

Figure 2.10 shows how the sample size affects the sampling scheme. We resolved 3,000 shopping records using a pair list as a hint where we simply compared pairs of records using a Boolean match function B following the order in the hint and performed a transitive closure at the end. The sample sizes ranged from 2 to 1000 and were chosen randomly from the input set of records. We also ran the naïve method where records were compared in a random order. The ER result was compared with the entire result of comparing all pairs of records using B and performing a transitive closure at the end. As a result, even a sample size of 2 produced a result significantly better than the random comparisons result and close

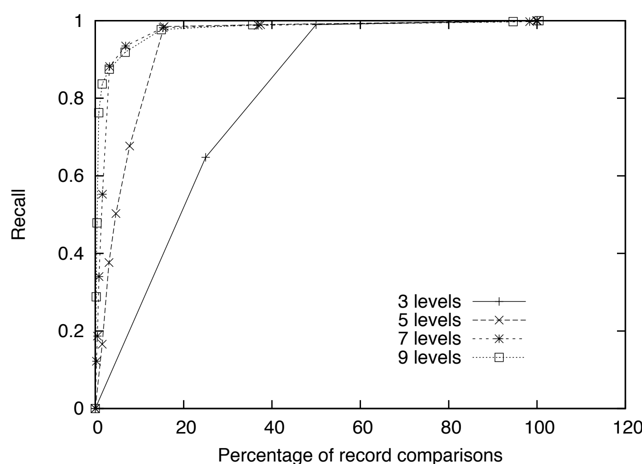


Figure 2.9: Number of levels impact on recall, 3K shopping records

to the result using a sample size of 1,000. Notice that the sample size = 2 plot sometimes performs better than the sample size = 10 plot. This result implies that simply having a larger sample size does not guarantee strictly better ER results. In summary, our sampling results show that small sample sizes suffice for near-optimal results.

2.7.6 Using Weights on Partition Hierarchy Levels

We consider the scenario where the HC_B algorithm uses a record list hint generated from a partition hierarchy. In Section 2.4.1, we discussed an extension of the partition hierarchy model where each partition can have a weight (or confidence value) associated with it. In this section, we vary the weights when generating record lists and see the impact the weights have on the quality of HC_B. We can use Algorithm 5 to generate an optimal record list by replacing Step 8 with the line "$nEntities(\{r\}) \leftarrow nEntities(\{r\}) + w_i \times \frac{1}{|c|}$."

Figure 2.11 shows for each combination of weights, the final recall of running HC_B on 3,000 shopping records where the work limit is set as half the number of comparisons HC_B would have performed without hints. For example, if w_1 = 1 and all the other weights are 0, we only use the bottom-most partition of the hierarchy for generating the record list. In Figure 2.11, the recall values range from 0.431 to 0.634. Not surprisingly, the minimum recall occurs when w_5 = 1, which means that only the highest-level partition was used to generate the record list. Any other weight assignment gives better results than setting w_5

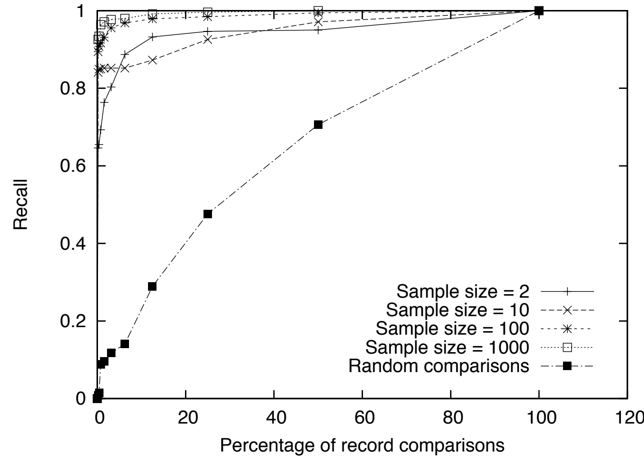


Figure 2.10: List of pairs using sampling, 3K shopping records

= 1. The recall when all weights have equal values (i.e., each weight is $\frac{1}{5}$) is 0.618, which is not significantly lower than the highest recall possible. We conclude that using equal weights is a reasonable strategy with the benefit that one does not have to fine-tune the weights of the hierarchy.

2.7.7 Non-incremental ER algorithms

We now experiment with ER algorithms that do not satisfy the general incremental property (see Definition 2.3.2.1). A non-incremental ER algorithm is not guaranteed to return a correct ER result when using Algorithm 4 for resolving clusters (see below). In this section, we experiment on the complete-link hierarchical clustering algorithm (called HC_C), which is identical to the HC_S algorithm (see Section 2.3.2) except that the distance between two clusters is defined by the maximum pairwise distance between their records. To see how using a partition hint can incorrectly alter the ER result of HC_C, suppose that we have R = $\{r_1, r_2, r_3\}$ where the pairwise distances are $D(r_1, r_2)$ = 2, $D(r_2, r_3)$ = 2, and $D(r_1, r_3)$ = 5 with a given threshold T = 2. The ER result would be $\{\{r_1, r_2\}$, $\{r_3\}\}$ because $\{r_1\}$ and $\{r_2\}$ match while $\{r_1, r_2\}$ and $\{r_3\}$ do not (having a distance of 5). However, if the partition hierarchy hint contains one partition $\{\{r_1\}$, $\{r_2, r_3\}\}$, then HC_C will resolve $\{r_2, r_3\}$ first and will merge r_2 and r_3. Since $\{r_1\}$ and $\{r_2, r_3\}$ do not match, the ER result is $\{\{r_1\}$, $\{r_2, r_3\}\}$. However, this result can never occur when running $E(R)$ without

Weights					**Recall**	**Weights**					**Recall**
w_1	w_2	w_3	w_4	w_5		w_1	w_2	w_3	w_4	w_5	
1	0	0	0	0	0.623	0	0	0	0	1	0.431
0	1	0	0	0	0.598	1/2	0	0	0	1/2	0.623
1/2	1/2	0	0	0	0.620	0	1/2	0	0	1/2	0.598
0	0	1	0	0	0.535	1/3	1/3	0	0	1/3	0.620
1/2	0	1/2	0	0	0.634	0	0	1/2	0	1/2	0.535
0	1/2	1/2	0	0	0.591	1/3	0	1/3	0	1/3	0.634
1/3	1/3	1/3	0	0	0.621	0	1/3	1/3	0	1/3	0.591
0	0	0	1	0	0.491	1/4	1/4	1/4	0	1/4	0.621
1/2	0	0	1/2	0	0.620	0	0	0	1/2	1/2	0.491
0	1/2	0	1/2	0	0.591	1/3	0	0	1/3	1/3	0.620
1/3	1/3	0	1/3	0	0.620	0	1/3	0	1/3	1/3	0.591
0	0	1/2	1/2	0	0.529	1/4	1/4	0	1/4	1/4	0.620
1/3	0	1/3	1/3	0	0.620	0	0	1/3	1/3	1/3	0.529
0	1/3	1/3	1/3	0	0.585	1/4	0	1/4	1/4	1/4	0.620
1/4	1/4	1/4	1/4	0	0.618	1/5	1/5	1/5	1/5	1/5	0.618

Figure 2.11: Weights impact on accuracy, 3K shopping records

hints. We measure the accuracy of an intermediate ER result using the F_1 measure defined in Section 1.2. Our experiments were done using 3,000 shopping records.

Figure 2.12 shows the accuracy results of running HC_C against the number of record comparisons performed. Among all schemes, using a partition hierarchy hint generated from a hash function produces an ER result with 0.98 accuracy using only 6% of the record comparisons required for a naïve approach without hints. The experiments show that partition hierarchy hints can produce highly-accurate ER results with few record comparisons even if the ER algorithms are not general incremental.

2.7.8 Early Termination on Large Datasets

We now scale our techniques on 0.5–2 million shopping records. Since the records do not fit in memory, we used blocking techniques as described in the beginning of Section 2.7. We used minhash signatures [57] for distributing the records into blocks. For the shopping dataset, we extracted 3-grams from the titles of records. We then generated a minhash signature for each records, which is an array of integers where each integer is generated by

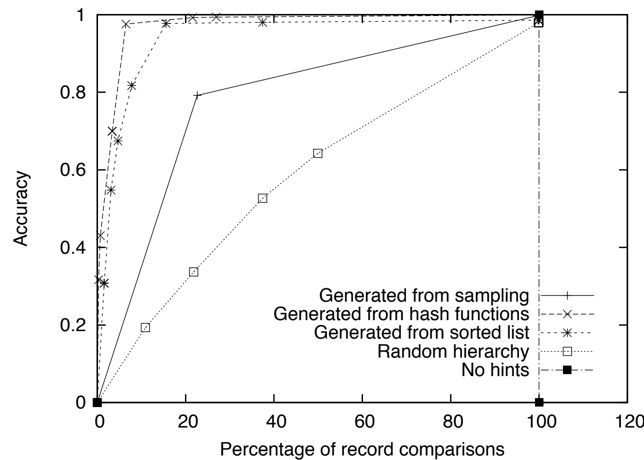


Figure 2.12: Non-incremental algorithm accuracy, 3K shopping records

applying a random hash function to the 3-gram set of the record.

While hints can help maximize the ER quality, it is not obvious exactly when to stop ER without knowledge on how many more matching records need to be identified. We compare three possible schemes on when to terminate ER:

- *No Limit*: We simply run ER without hints to the end.
- *Popcorn Scheme (Limit Rate)*: We stop when the rate of newly found matching pairs drops below a threshold. The analogy is making popcorn where we stop cooking when the frequency of pops drops below a certain level.
- *TV Dinner Scheme (Limit Computation)*: We limit the number of record comparisons based on the number of records to be resolved. The analogy is heating a TV Dinner in a microwave oven for a fixed amount of time as specified by the cooking instructions.

We used the HC_S algorithm and partition hierarchy hints generated from sorted lists. The first Popcorn scheme is useful when we want to maximize recall and yet minimize the runtime as much as possible. In our implementation, we terminate ER when the rate of finding new matching pairs among all record pairs compared drops below 1%. For example, for the next 200 record pairs compared, if fewer than 2 pairs matched, then we terminated HC_S. The rate was checked after each level iteration in the hierarchy. The second TV

Scheme	Runtime(hrs)			Recall		
	0.5M	**1M**	**2M**	**0.5M**	**1M**	**2M**
No Limit	0.4	1.8	15	1.0	1.0	1.0
Popcorn	0.12	0.37	2.5	0.98	0.98	0.99
TV Dinner	0.08	0.26	1.3	0.6	0.6	0.6

Figure 2.13: Runtime and recall for different schemes, 2M shopping records

Dinner scheme is useful when there is only a given amount of time for the application to run. In our experiments, we set the computation limit to be 10% of the total number of record pairs in the current set of records to be resolved. For example, when resolving a cluster of size 20, we ran at most about $\frac{1}{10} \times \frac{20 \times 19}{2} = 19$ record comparisons.

Figure 2.13 shows how the two schemes perform compared to when ER runs without hints. We measured the entire ER runtimes including the hint construction times and the IO costs for reading and writing blocks on disk. However, the bottleneck for the entire ER process was the CPU time to resolve the blocks in memory. While the Popcorn scheme tends to give better recall, it does not guarantee termination within a given amount of time. On the other hand, while the TV Dinner scheme has the advantage of having a predictable runtime, it may not always give the best recall results. The runtime improvements (at most 11.5x) are not as high as what we observed in the 3,000 shopping dataset results. (According to Figure 2.5(f), we can obtain 0.99 recall about 18x faster than running ER without hints using partition hierarchies generated from sorted lists on 3,000 shopping records.) The reason is that in our scenario many blocks were not large enough for hints to help as much (i.e., the overhead of constructing hints did not pay off as much), so the average benefit of using hints was relatively low. Nevertheless, using hints can still significantly improve the runtime of ER on large datasets (by 3.3–11.5x) while still obtaining high recall.

2.7.9 Scalability of Generating Hints

Table 2.14 shows the scalability results for generating hints. The construction times for hints scale well with the exception of generating a record list using complete sampling (for 2M records, the memory overflowed). However, by using partial sampling instead, we can obtain scalability with minimal loss in quality (see Section 2.7.2).

Hint	Generation	0.5M	1M	2M
PL	Cheap dist. fns	4	9	20
	Sampling	47	120	309
H	Sorted records	3	6	12
	Hash functions	8	11	16
	Sampling	40	130	379
RL	Par. hierarchy	11	19	31
	Com. sampling	291	1256	OOM
	Par. sampling	53	158	597

Figure 2.14: Hint generation time (secs), 2M shopping records

2.8 Related Work

Most of the ER work in the literature has focused on optimizing the overall runtime. In contrast, our approach takes a pay-as-you-go approach that optimizes the *intermediate results* of ER. Our approach is useful when either the data set is too large to resolve within a reasonable amount of time, or when there is not even enough time to resolve a small data set.

Blocking techniques [71, 8, 54] focus on improving the overall runtime of ER where the records are divided into possibly overlapping blocks, and the blocks are resolved one at a time. Locality sensitive hashing [43] is a method for performing probabilistic dimension reduction of high-dimensional data and can also be used as a blocking technique. A number of works [6, 25] propose efficient similarity joins. Our pay-as-you-go techniques improve blocking by also exploiting the ordering of record pairs according to their likelihood of matching to produce the best intermediate ER results.

A line of ER work [31, 101] implicitly uses hints by comparing record pairs in the order of their similarity. More recently, a framework for clustering records based on similarity join results [53, 52] has been proposed. Here, the duplication detection framework consists of two stages: an efficient similarity join, which returns similarities between likely matching records, and a clustering stage where records are clustered based on the given similarities. While these systems may already use hints, we believe our work is the first to explicitly identify and study a wide range of hints that yield results early.

Another line of work proposes similarity search [116, 20, 115] techniques where indices

or blocking criteria are used for quickly finding the records that are likely to match with a single record. In contrast, our work focuses on resolving all the records (instead of just one) by using hints, which provide information on the best record pairs that are more likely to match.

There has been a recent surge of work on pay-as-you-go information integration [69, 61] on large scale data. Our work is in the same spirit of these works where we incrementally resolve records given the limited amount of time and resources we have. Our work focuses on the ER domain and improves existing ER algorithms to produce results in a pay-as-you-go fashion using hints.

2.9 Conclusion

We have proposed a pay-as-you-go approach for Entity Resolution (ER) where given a limit in resources (e.g., work, runtime) we attempt to make the maximum progress possible. We introduce the concept of hints, which can guide an ER algorithm to focus on resolving the more likely matching records first. Our techniques are effective when there are either too many records to resolve within a reasonable amount of time or when there is a time limit (e.g., real-time systems). We proposed three types of hints that are compatible with different ER algorithms: a sorted list of record pairs, a hierarchy of record partitions, and an ordered list of records. We have also proposed various methods for ER algorithms to use these hints. Our experimental results evaluated the overhead of constructing hints as well as the runtime benefits for using hints. We considered a variety of ER algorithms and two real-world data sets. The results suggest that the benefits of using hints can be well worth the overhead required for constructing and using hints. Many interesting problems remain to be solved, including a more formal analysis of different types of hints and a general guidance for constructing and updating the "best" hint for any given ER algorithm.

Chapter 3

Evolving Rules

We explore the problem of incrementally updating ER results. In some cases ER result may not be produced just once, but constantly improved based on better understanding of the data, the schema, and the logic that examines and compares records. In particular, here we focus on changes to the logic that compares two records. We call this logic the *match rule*, and it can be a Boolean function that determines if two records represent the same entity, or a distance function that quantifies how different (or similar) the records are. Initially we start with a set of records S, then produce a first ER result E_1 based on S and a rule B_1. Some time later rule B_1 is improved yielding rule B_2, so we need to compute a new ER result E_2 based on S and B_2. The process continues with new rules B_3, B_4 and so on.

A naïve approach would compute each new ER result from scratch, starting from S, a potentially very expensive proposition. Instead, in this chapter we explore an incremental approach, where we compute E_2 based on E_1. Of course for this approach to work, we need to understand how the new rule B_2 relates to the old one B_1, so we can understand what changes incrementally in E_1 to obtain E_2. As we will see, our incremental approach may yield large savings over the naïve approach, but not in all cases.

To motivate and explain our approach, consider the following example. Our initial set of people records S is shown in Figure 3.1. The first rule B_1 (see Figure 3.2) says that two records match if predicate p_{name} evaluates to true. Predicates can in general be quite complex, but for this example assume that predicates simply perform an equality check. The ER algorithm calls on B_1 to compare records and groups together records with name

Record	**Name**	**Zip**	**Phone**
r_1	$John$	54321	123-4567
r_2	$John$	54321	987-6543
r_3	$John$	11111	987-6543
r_4	Bob	$null$	121-1212

Figure 3.1: Records to resolve

Match Rule	**Definition**
B_1	p_{name}
B_2	$p_{name} \wedge p_{zip}$
B_3	$p_{name} \wedge p_{phone}$

Figure 3.2: Evolving from rule B_1 to rule B_3

"John", producing the result $\{\{r_1, r_2, r_3\}, \{r_4\}\}$. (As we will see, there are different types of ER algorithms, but in this simple case most would return this same result.)

Next, say users are not satisfied with this result, so a data administrator decides to refine B_1 by adding a predicate that checks zip codes. Thus, the new rule is B_2 shown in Figure 3.2. The naïve option is to run the same ER algorithm with rule B_2 on set S to obtain the partition $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. (Only records r_1 and r_2 have the same name and same zip code.) This process repeats much unnecessary work: For instance, we would need to compare r_1 with r_4 to see if they match on name and zip code, but we already know from the first run that they do not match on name (B_1), so they cannot match under B_2.

Because the new rule B_2 is stricter than B_1 (we define this term precisely later on), we can actually start the second ER from the first result $\{\{r_1, r_2, r_3\}, \{r_4\}\}$. That is, we only need to check each cluster separately and see if it needs to split. In our example, we find that r_3 does not match the other records in its cluster, so we arrive at $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. This approach only works if the ER algorithm satisfies certain properties and B_2 is stricter than B_1. If B_2 is not stricter and the ER algorithm satisfies different properties, there are other incremental techniques we can apply. Our goal in this chapter is to explore these options: Under what conditions and for what ER algorithms are incremental approaches *feasible*? And in what scenarios are the savings over the naïve approach significant?

In addition, we study a complementary technique: *materialize* auxiliary results during one ER run, in order to improve the performance of future ER runs. To illustrate, say that

when we process $B_2 = p_{name} \wedge p_{zip}$, we concurrently produce the results for each predicate individually. That is, we compute three separate partitions, one for the full B_2, one for rule p_{name} and one for rule p_{zip}. The result for p_{name} is the same $\{\{r_1, r_2, r_3\}, \{r_4\}\}$ seen earlier. For p_{zip} it is $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$. As we will see later, the cost of computing the two extra materializations can be significantly lower than running the ER algorithm three times, as a lot of the work can be shared among the runs.

The materializations pay off when rule B_2 evolves into a related rule that is not quite stricter. For example, say that B_2 evolves into $B_3 = p_{name} \wedge p_{phone}$, where p_{phone} checks for matching phone numbers. In this case, B_3 is not stricter than B_2 so we cannot start from the B_2 result. However, we can start from the p_{name} result, since B_3 is stricter than p_{name}. Thus, we independently examine each cluster in $\{\{r_1, r_2, r_3\}, \{r_4\}\}$, splitting the first cluster because r_2 has a different phone number. The final result is $\{\{r_1, r_3\}, \{r_2\}, \{r_4\}\}$. Clearly, materialization of partial results may or may not pay off, just like materialized views and indexes may or may not help. Our objective here is, again, to study when is materialization *feasible* and to illustrate scenarios where it can pay off.

In summary, our contributions in this chapter are as follows:

- We formalize rule evolution for two general types of record match rules: Boolean match functions and distance-based functions. We identify two desirable properties of ER algorithms (*rule monotonic* and *context free*) that enable efficient rule evolution. We also contrast these properties to two properties mentioned in the literature (*order independent* and *incremental*). We categorize a number of existing ER algorithms based on the properties they satisfy. We then propose efficient rule evolution techniques that use one or more of the four properties (Sections 3.1 and 3.3). We believe that our results can be a useful guide for ER algorithm designers: if they need to handle evolving rules efficiently, they may want to build algorithms that have at least some of the properties we present.
- We experimentally evaluate (Section 3.4) the rule evolution algorithms for various ER algorithms using actual comparison shopping data from Yahoo! Shopping and hotel information from Yahoo! Travel. Our results show scenarios where rule evolution can be faster than the naïve approach by up to several orders of magnitude. We also illustrate the time and space cost of materializing partial results, and argue that these

costs can be amortized with a small number of future evolutions. Finally, we also experiment with ER algorithms that do not satisfy our properties, and show that if one is willing to sacrifice accuracy, one can still use our rule evolution techniques.

3.1 Match-based Evolution

We consider rule evolution for ER algorithms that cluster records based on Boolean match rules. (We consider ER algorithms based on distance functions in Section 3.3.) We first formalize an ER model that is based on clustering. We then discuss two important properties for ER algorithms that can significantly enhance the runtime of rule evolution. We also compare the two properties with existing properties for ER algorithms in the literature. Finally, we present efficient rule evolution algorithms that use one or more of the four properties.

3.1.1 Match-based Clustering Model

We define a Boolean match rule B as a function that takes two records and returns `true` or `false`. We assume that B is commutative, i.e., $\forall r_i, r_j, B(r_i, r_j) = B(r_j, r_i)$. A Boolean match rule is identical to a match function, and the two terms are used interchangeably.

Suppose we are given a set of records $S = \{r_1, \ldots, r_n\}$. An ER algorithm receives as inputs a partition P_i of S and a Boolean comparison rule B, and returns another partition P_o of S. A partition of S is defined as a set of clusters $P = \{c_1, \ldots, c_m\}$ such that $c_1 \cup \ldots \cup c_m = S$ and $\forall c_i, c_j \in P$ where $i \neq j$, $c_i \cap c_j = \emptyset$.

We require the input to be a partition of S so that we may also run ER on the output of a previous ER result. In our motivating example in the beginning of this chapter, the input was a set of records $S = \{r_1, r_2, r_3, r_4\}$, which can be viewed as a partition of singletons $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}\}$, and the output using the comparison rule $B_2 = p_{name} \wedge p_{zip}$ was the partition $P_o = \{\{r_1, r_2\},\{r_3\},\{r_4\}\}$. If we run ER a second time on the ER output $\{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$, we may obtain the new output partition $P_o = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ where the cluster $\{r_1, r_2\}$ accumulated enough information to match with the cluster $\{r_3\}$.

How exactly the ER algorithm uses B to derive the output partition P_o depends on the

specific ER algorithm. The records are clustered based on the results of B when comparing records. In our motivating example (beginning of this chapter), all pairs of records that matched according to $B_2 = p_{name} \wedge p_{zip}$ were clustered together. Note that, in general, an ER algorithm may not cluster two records simply because they match according to B. For example, two records r and s may be in the same cluster $c \in P_o$ even if $B(r, s)$ = `false`. Or the two records could also be in two different clusters $c_i, c_j \in P_o$ $(i \neq j)$ even if $B(r, s)$ = `true`.

We also allow input clusters to be un-merged as long as the final ER result is still a partition of the records in S. For example, given an input partition $\{\{r_1, r_2, r_3\}$,$\{r_4\}\}$, an output of an ER algorithm could be $\{\{r_1, r_2\}$,$\{r_3, r_4\}\}$ and not necessarily $\{\{r_1, r_2, r_3\}$,$\{r_4\}\}$ or $\{\{r_1, r_2, r_3, r_4\}\}$. Un-merging could occur when an ER algorithm decides that some records were incorrectly clustered [102].

Finally, we assume the ER algorithm to be *non-deterministic* in a sense that different partitions of S may be produced depending on the order of records processed or by some random factor (e.g., the ER algorithm could be a randomized algorithm). For example, a hierarchical clustering algorithm based on Boolean rules (see Section 3.1.2) may produce different partitions depending on which records are compared first. While the ER algorithm is non-deterministic, we assume the match rule itself to be deterministic, i.e., it always returns the same matching result for a given pair of records.

We now formally define a valid ER algorithm.

Definition 3.1.1. *Given any input partition P_i of a set of records S and any Boolean match rule B, a* valid *ER algorithm E non-deterministically returns an ER result $E(P_i$,$B)$ that is also a partition P_o of S.*

We denote all the possible partitions that can be produced by the ER algorithm E as $\bar{E}(P_i$, $B)$, which is a set of partitions of S. Hence, $E(P_i$,$B)$ is always one of the partitions in $\bar{E}(P_i$,$B)$. For example, given P_i = $\{\{r_1\}$, $\{r_2\}$, $\{r_3\}\}$, $\bar{E}(P_i$,$B)$ could be $\{\{\{r_1, r_2\}$, $\{r_3\}\}$, $\{\{r_1\}$, $\{r_2, r_3\}\}\}$ while $E(P_i$, $B)$ = $\{\{r_1, r_2\}$, $\{r_3\}\}$.

A rule evolution occurs when a Boolean comparison rule B_1 is replaced by a new Boolean comparison rule B_2. An important concept used throughout the paper is the relative strictness between comparison rules:

Definition 3.1.2. *A Boolean comparison rule* B_1 *is* stricter than *another rule* B_2 *(denoted as* $B_1 \leq B_2$*) if* $\forall r_i, r_j$, $B_1(r_i, r_j)$ = `true` *implies* $B_2(r_i, r_j)$ = `true`.

For example, a comparison rule B_1 that compares the string distance of two names and returns `true` when the distance is lower than 5 is stricter than a comparison rule B_2 that uses a higher threshold of, say, 10. As another example, a comparison rule B_1 that checks whether the names and addresses are same is stricter than another rule B_2 that only checks whether the names are same.

3.1.2 Properties

We introduce two important properties for ER algorithms – *rule monotonic* and *context free* – that enable efficient rule evolution for match-based clustering.

Rule Monotonic

Before defining the rule monotonic property, we first define the notion of refinement between partitions.

Definition 3.1.3. *A partition* P_1 *of a set* S refines *another partition* P_2 *of* S *(denoted as* $P_1 \leq P_2$*) if* $\forall c_1 \in P_1, \exists c_2 \in P_2$ *s.t.* $c_1 \subseteq c_2$.

For example, given the partitions $P_1 = \{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$ and $P_2 = \{\{r_1, r_2, r_3\}, \{r_4\}\}$, $P_1 \leq P_2$ because $\{r_1, r_2\}$ and $\{r_3\}$ are subsets of $\{r_1, r_2, r_3\}$ while $\{r_4\}$ is a subset of $\{r_4\}$.

We now define the rule monotonic property, which guarantees that the stricter the match rule, the more refined the ER result.

Definition 3.1.4. *An ER algorithm is* rule monotonic *($\mathcal{RM}$) if, for any three partitions* P, P_o^1, P_o^2 *and two match rules* B_1 *and* B_2 *such that*

- $B_1 \leq B_2$ *and*
- $P_o^1 \in \bar{E}(P, B_1)$ *and*
- $P_o^2 \in \bar{E}(P, B_2)$

then $P_o^1 \leq P_o^2$.

An ER algorithm satisfying $\mathcal{RM}$ guarantees that, if the match rule B_1 is stricter than B_2, the ER result produced with B_1 refines the ER result produced with B_2. For example, suppose that $P = \{\{r_1\},\{r_2\},\{r_3\},\{r_4\}\}$, $B_1 \leq B_2$, and $E(P_i, B_1) = \{\{r_1, r_2, r_3\}, \{r_4\}\}$. If the ER algorithm is $\mathcal{RM}$, $E(P_i, B_2)$ can only return $\{\{r_1, r_2, r_3\}, \{r_4\}\}$ or $\{\{r_1, r_2, r_3, r_4\}\}$.

Context Free

The second property, context free, tells us when a subset of P_i can be processed "in isolation" from the rest of the clusters.

Definition 3.1.5. *An ER algorithm is* context free *($\mathcal{CF}$) if for any four partitions P, P_i, P_o^1, P_o^2 and a match rule B such that*

- $P \subseteq P_i$ *and*
- $\forall P_o \in \bar{E}(P_i, B)$, $P_o \leq \{\bigcup_{c \in P} c \,, \bigcup_{c \in P_i - P} c\}$ *and*
- $P_o^1 \in \bar{E}(P, B)$ *and*
- $P_o^2 \in \bar{E}(P_i - P, B)$

then $P_o^1 \cup P_o^2 \in \bar{E}(P_i, B)$.

Suppose that we are resolving $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}\}$ with the knowledge that no clusters in $P = \{\{r_1\}, \{r_2\}\}$ will merge with any of the clusters in $P_i - P = \{\{r_3\}, \{r_4\}\}$. Then for any $P_o \in \bar{E}(P_i, B)$, $P_o \leq \{\{r_1, r_2\}, \{r_3, r_4\}\}$. In this case, an ER algorithm that is $\mathcal{CF}$ can resolve $\{\{r_1\}, \{r_2\}\}$ independently from $\{\{r_3\}, \{r_4\}\}$, and there exists an ER result of P_i that is the same as the union of the ER results of $\{\{r_1\}, \{r_2\}\}$ and $\{\{r_3\}, \{r_4\}\}$.

Existing ER Properties

To get a better understanding of $\mathcal{RM}$ and $\mathcal{CF}$, we compare them to two existing properties in the literature: incremental and order independent.

An ER algorithm is incremental [59, 58] if it can resolve one record at a time. We define a more generalized version of the incremental property for our ER model where any subsets of clusters in P_i can be resolved at a time.

Definition 3.1.6. *An ER algorithm is* general incremental *($\mathcal{GI}$) if for any four partitions P, P_i, P_o^1, P_o^2, and a match rule B such that*

- *$P \subseteq P_i$ and*
- *$P_o^1 \in \bar{E}(P, B)$ and*
- *$P_o^2 \in \bar{E}(P_o^1 \cup (P_i - P), B)$*

then $P_o^2 \in \bar{E}(P_i, B)$.

For example, suppose we have $P = \{\{r_1\}, \{r_2\}\}$, $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, and $P_o^1 = \{\{r_1, r_2\}\}$. That is, we have already resolved P into the result P_o^1. We can then add to P_o^1 the remaining cluster $\{r_3\}$, and resolve all the clusters together. The result is as if we had resolved everything from scratch (i.e., from P_i). Presumably, the former way (incremental) will be more efficient than the latter.

The $\mathcal{GI}$ property is similar to the $\mathcal{CF}$ property, but also different in a number of ways. First $\mathcal{GI}$ and $\mathcal{CF}$ are similar in a sense that they use two subsets of P_i: P and $P_i - P$. However, under $\mathcal{GI}$, $P_i - P$ is not resolved until P has been resolved. Also, $\mathcal{GI}$ does not assume P and $P_i - P$ to be independent (i.e., a cluster in P may merge with a cluster in $P_i - P$).

We now explore the second property in the literature. An ER algorithm is order independent ($\mathcal{OI}$) [59] if the ER result is same regardless of the order of the records processed. That is, for any input partition P_i and match rule B, $\bar{E}(P_i, B)$ is a singleton (i.e., $\bar{E}(P_i, B)$ contains exactly one partition of S).

ER Algorithm Categorization

To see how the four properties $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$ hold in practice, we consider several ER algorithms in the literature: SN, HC_B, HC_{BR}, and ME. While the original definitions of all four ER algorithms assume a set of records S as an input, we provide simple extensions for the algorithms to accept a set of clusters P_i as in Definition 3.1.1.

SN The sorted neighborhood (SN) algorithm [54] was defined in Section 2.2.2.

Proposition 3.1.7. *The SN algorithm is $\mathcal{RM}$, but not $\mathcal{CF}$.*

Proof. We prove that the SN algorithm is $\mathcal{RM}$. Given any partition P and two match rules B_1 and B_2 such that $B_1 \leq B_2$, the set of pairs of matching records found by B_1 is clearly a subset of that found by B_2, during the first phase of the SN algorithm. As a result, the transitive closure of the matching pairs by B_1 refines the transitive closure result of B_2 (i.e., $P_o^1 \leq P_o^2$).

We prove that the SN algorithm is not $\mathcal{CF}$ using a counter example. Suppose that the input partition is P_i = $\{\{r_1\}, \{r_2\}, \{r_3\}\}$, and we sort the records in P_i by their record IDs (e.g., the record r_2 has the ID of 2) into the sorted list [r_1, r_2, r_3]. Given the match rule B, suppose that only the pair r_1 and r_3 match with each other. Using a window size of 2, we do not identify any matching pairs of records because r_1 and r_3 are never in the same window according to the sorted list. Hence, $E(P_i, B) = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. To apply the $\mathcal{CF}$ property, we set $P = \{\{r_1\}, \{r_3\}\}$ and $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. The first two conditions in Definition 3.1.5 are satisfied because $P \subseteq P_i$ and $\forall P_o \in \bar{E}(P_i, B) = \{\{\{r_1\}, \{r_2\}, \{r_3\}\}\}$, $P_o \leq \{\bigcup_{c \in P}, \bigcup_{c \in P_i}\} = \{\{r_1, r_3\}, \{r_2\}\}$. Also, $E(P, B)$ is always $\{\{r_1, r_3\}\}$ (because r_1 and r_3 surely match being in the same window) and $E(P_i - P, B)$ is always $\{\{r_2\}\}$. However, $E(P, B) \cup E(P_i - P, B) = \{\{r_1, r_3\}, \{r_2\}\} \not\subseteq \bar{E}(P_i, B) = \{\{\{r_1\}, \{r_2\}, \{r_3\}\}\}$, violating the $\mathcal{CF}$ property. □

HC_B Hierarchical clustering based on a Boolean match rule [10] (which we call HC_B) was defined in Section 2.4.2.

Proposition 3.1.8. *The HC_B algorithm is $\mathcal{CF}$, but not $\mathcal{RM}$.*

Proof. We prove that the HC_B algorithm is $\mathcal{CF}$. Given the four partitions P, P_i, P_o^1, P_o^2 of Definition 3.1.5, suppose that $P_o^1 \cup P_o^2 \notin \bar{E}(P_i, B)$. We then prove that the four conditions of Definition 3.1.5 cannot all be satisfied at the same time. We first assume that the first, third, and fourth conditions are satisfied. That is, $P \subseteq P_i$, $P_o^1 \in \bar{E}(P, B)$, and $P_o^2 \in \bar{E}(P_i - P, B)$. Now suppose when deriving $E(P_i, B)$ that we "replay" all the merges among the clusters of P that were used to derive P_o^1 and then "replay" all the merges among the clusters of $P_i - P$ that were used to derive P_o^2. We thus arrive at the state $P_o^1 \cup P_o^2$, and can further merge any matching clusters until no clusters match to produce a possible ER result in $\bar{E}(P_i, B)$. Since $P_o^1 \cup P_o^2 \notin \bar{E}(P_i, B)$, there must have been new merges among clusters

in P_o^1 and P_o^2. Since none of the clusters within P_o^1 or clusters within P_o^2 match with each other, we know that there must have been at least one more merge between a cluster in P_o^1 and a cluster in P_o^2 when deriving $E(P_i, B)$. As a result, the second condition cannot hold because there exists an ER result $P_o \in \bar{E}(P_i, B)$ such that $P_o \not\leq \{\bigcup_{c \in P} c, \bigcup_{c \in P_i - P} c\}$ because there exists a cluster in P_o that contains records in P as well as $P_i - P$. Hence we have proved that all four conditions can never be satisfied if $P_o^1 \cup P_o^2 \notin \bar{E}(P_i, B)$.

We prove that the HC_B algorithm is not $\mathcal{RM}$ using a counter example. Consider the example we used for illustrating the HC_B algorithm where $\bar{E}(P_i, B)$ was $\{\{\{r_1, r_2\}, \{r_3\}\}, \{\{r_1\}, \{r_2, r_3\}\}\}$. Now without having to define a new Boolean match rule, by setting $B_1 = B$, $B_2 = B$, $P_o^1 = \{\{r_1, r_2\}, \{r_3\}\}$, and $P_o^2 = \{\{r_1\}, \{r_2, r_3\}\}$, we see that $P_o^1 \not\leq P_o^2$ (although $B_1 \leq B_2$), contradicting the $\mathcal{RM}$ property. This result suggests that any ER algorithm that can produce more than one possible partition given any P_i and B is not $\mathcal{RM}$. We show in Proposition 3.1.13 the equivalent statement that any ER algorithm that is $\mathcal{RM}$ always returns a unique solution. □

$\mathbf{HC_{BR}}$ We define the HC_{BR} algorithm as a hierarchical clustering algorithm based on a Boolean match rule (just like HC_B). In addition, we require the match rule to match two clusters whenever at least one of the records from the two clusters match according to B. (This property is equivalent to the representativity property in reference [10].) For example, a cluster comparison function that compares all the records between two clusters using B for an existential match is representative. That is, given two clusters $\{r_1, r_2\}$ and $\{r_3, r_4\}$, the cluster comparison function returns `true` if at least one of $B(r_1, r_3)$, $B(r_1, r_4)$, $B(r_2, r_3)$, or $B(r_2, r_4)$ returns `true`.

We can prove that the HC_{BR} is both $\mathcal{RM}$ and $\mathcal{CF}$ (see Proposition 3.1.11). We first define the notation of connectedness. Two records r and s are connected under B and P_i if there exists a sequence of records $[r_1\ (= r), \ldots, r_n\ (= s)]$ where for each pair (r_i, r_{i+1}) in the path, either $B(r_i, r_{i+1})$ = `true` or $\exists c \in P_i$ s.t. $r_i \in c, r_{i+1} \in c$. Notice that connectedness is "transitive," i.e., if r and s are connected and s and t are connected, then r and t are also connected.

Lemma 3.1.9. *Two records r and s are connected under B and P_i if and only if r and s are in the same cluster in $P_o \in \bar{E}(P_i, B)$ using the HC_{BR} algorithm.*

Proof. Suppose that r and s are in the same cluster in P_o. If r and s are in the same cluster in P_i, then r and s are trivially connected under B and P_i. Otherwise, there exists a sequence of merges of the clusters in P_i that grouped r and s together. If two clusters c_a and c_b in P_i merge where $r \in c_a$ and $s \in c_b$, then r and s are connected because there is at least one pair of records $r' \in c_a$ and $s' \in c_b$ such that r' and s' match (i.e., $B(r', s')$ = `true`), and r is connected to r' while s is connected to s'. Furthermore, we can prove that any record in $c_a \cup c_b$ is connected with any record in a cluster c_c that merges with $c_a \cup c_b$ using a similar argument: we know there exists a pair of records $r' \in c_a \cup c_b$ and $s' \in c_c$ that match with each other, and r is connected to r' while s is connected to s', which implies that r and s are connected under B and P_i. By repeatedly applying the same argument, we can prove that any r and s are connected if they end up in the same cluster in P_o.

Conversely, suppose that r and s are connected as the sequence $[r_1(= r), \ldots, r_n(= s)]$ under B and P_i. If r and s are in the same cluster in P_i, they are already clustered together in P_o. Otherwise, all the clusters that contain $r_1, \ldots, r_n$ eventually merge together according to the HC_{BR} algorithm, clustering r and s together in P_o. □

We next prove that HC_{BR} returns a unique solution.

Proposition 3.1.10. *The HC_{BR} algorithm always returns a unique solution.*

Proof. Suppose that HC_{BR} produces two different output partitions for a given partition P_i and match rule B, i.e., $\bar{E}(P_i, B) = \{P_o^1, P_o^2, \ldots\}$. Then there must exist two records r and s that have merged into the same cluster according to one ER result, but not in the same cluster for the other ER result. Suppose that r and s are in the same cluster in P_o^1, but in separate clusters in P_o^2. Since r and s are clustered together in P_o^1, they are connected by Lemma 3.1.9. Hence, r and s must also be clustered in P_o^2 again by Lemma 3.1.9, contradicting our hypothesis that they are in different clusters in P_o^2. Hence, HC_{BR} always returns a unique partition. □

Finally, we prove that HC_{BR} is both $\mathcal{RM}$ and $\mathcal{CF}$.

Proposition 3.1.11. *The HC_{BR} algorithm is both $\mathcal{RM}$ and $\mathcal{CF}$.*

Proof. The HC_{BR} algorithm is $\mathcal{CF}$ because the HC_B algorithm already is $\mathcal{CF}$. To show that the HC_{BR} algorithm also is $\mathcal{RM}$, suppose that $B_1 \leq B_2$. Then all the clusters that

match according to B_1 also match according to B_2. Hence for any $P_o^1 \in \bar{E}(P_i, B_1)$, we can always construct an ER result $P_o^2 \in \bar{E}(P_i, B_2)$ (which is unique by Proposition 3.1.10) where $P_o^1 \leq P_o^2$ by performing the exact same merges done for P_o^1 and then continuing to merge clusters that still match according to B_2 until no clusters match according to B_2. □

ME The Monge Elkan (ME) clustering algorithm (we define a variant of the algorithm in [74] for simplicity) first sorts the records in P_i (i.e., we extract all the records from the clusters in P_i) by some key and then starts to scan each record. For example, suppose that we are given the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, and we sort the records in P_i by their names (which are not visible in this example) in alphabetical order into the sorted list of records $[r_1, r_2, r_3]$. Suppose we are also given the Boolean match rule B where $B(r_1, r_2)$ = `true`, but $B(r_1, r_3)$ = `false` and $B(r_2, r_3)$ = `false`. Each scanned record is then compared with clusters in a fixed-length queue. A record r matches with a cluster c if $B(r, s)$ = `true` for any $s \in c$. If the new record matches one of the clusters, the record and cluster merge, and the new cluster is promoted to the head of the queue. Otherwise, the new record forms a new singleton cluster and is pushed into the head of the queue. If the queue is full, the last cluster in the queue is dropped. In our example, if the queue size is 1, then we first add r_1 into the head of the queue, and then compare r_2 with $\{r_1\}$. Since r_2 matches with $\{r_1\}$, we merge r_2 into $\{r_1\}$. We now compare r_3 with the cluster $\{r_1, r_2\}$ in the queue. Since r_3 does not match with $\{r_1, r_2\}$, then we insert $\{r_3\}$ into the head of the queue and thus remove $\{r_1, r_2\}$. Hence, the only possible ER result is $\{\{r_1, r_2\}, \{r_3\}\}$ and thus $\bar{E}(P_i, B) = \{\{\{r_1, r_2\}, \{r_3\}\}\}$. In general, ME always returns a unique partition.

Proposition 3.1.12. *The ME algorithm does not satisfy $\mathcal{RM}$ or $\mathcal{CF}$.*

Proof. We prove that the ME algorithm is not $\mathcal{RM}$ using a counter example. Suppose that the input partition is $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, and we sort the records by their record IDs (e.g., the record r_2 has the ID of 2) into the sorted list of records $[r_1, r_2, r_3]$. Suppose that $B_1(r_1, r_3)$ = `true`, but $B_1(r_1, r_2)$ = `false` and $B_1(r_2, r_3)$ = `false`. Compared to B_1, the only difference of B_2 is that $B_2(r_2, r_3)$ = `true`. Clearly, $B_1 \leq B_2$. Using a queue size of 2, $E(P_i, B_1)$ returns $\{\{r_1, r_3\}, \{r_2\}\}$ only because r_1 and r_2 are inserted into the queue separately, and r_3 then merges with $\{r_1\}$. On the other hand, $E(P_i, B_2)$ returns $\{\{r_1\},$

$\{r_2, r_3\}\}$ because r_1 and r_2 are inserted into the queue separately, and r_3 matches with $\{r_2\}$ first. Since $E(P_i, B_1) = \{\{r_1, r_3\}, \{r_2\}\} \not\leq \{\{r_1\}, \{r_2, r_3\}\} = E(P_i, B_2)$, the $\mathcal{RM}$ property does not hold.

We prove that the ME algorithm is not $\mathcal{CF}$ using a counter example. Suppose that the input partition is $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$, and we sort the records by their IDs into the sorted list of records $[r_1, r_2, r_3]$. Suppose that $B(r_1, r_3) = \texttt{true}$, but $B(r_1, r_2) = \texttt{false}$ and $B(r_2, r_3) = \texttt{false}$. Using a queue size of 1, we do not identify any matching pairs because r_1 is never compared with r_3 because once r_2 enters the queue, $\{r_1\}$ is pushed out of the queue. Hence, $E(P_i, B)$ is always $\{\{r_1\}, \{r_2\}, \{r_3\}\}$. Using Definition 3.1.6, suppose we set $P = \{\{r_1\}, \{r_3\}\}$ and $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. The first condition is satisfied because $P \subseteq P_i$. The second condition is satisfied because $\forall P_o \in \bar{E}(P_i, B)$, $P_o = \{\{r_1\}, \{r_2\}, \{r_3\}\} \leq \{\bigcup_{c \in P} c, \bigcup_{c \in P_i - P} c\} = \{\{r_1, r_3\}, \{r_2\}\}$. Also, P_o^1 is always $\{\{r_1, r_3\}\}$ because r_3 matches with $\{r_1\}$ while $\{r_1\}$ is in the queue, and P_o^2 is always $\{\{r_2\}\}$. As a result, $P_o^1 \cup P_o^2 = \{\{r_1, r_3\}, \{r_2\}\} \notin \bar{E}(P_i, B) = \{\{\{r_1\}, \{r_2\}, \{r_3\}\}\}$, contradicting the $\mathcal{CF}$ property. □

The venn diagram in Figure 3.3 shows which ER algorithms satisfy which of the four properties. The SN^2 and HC_B^2 algorithms are variants of the SN and HC_B algorithms, respectively, and are used to prove Propositions 3.1.17 and 3.1.18, respectively. For now, ignore the HC_{DS} and HC_{DC} algorithms, which are distance-based clustering algorithms covered in Section 3.3.2.

We first add the $\mathcal{OI}$ property into the venn diagram in Figure 3.3. Proposition 3.1.13 shows that the $\mathcal{OI}$ property includes the $\mathcal{RM}$ property. Proposition 3.1.14 shows that the ME algorithm is $\mathcal{OI}$, but not $\mathcal{RM}$. Also, $\mathcal{OI}$ partially overlaps with $\mathcal{CF}$, but does not contain it. According to Proposition 3.1.15, the HC_{BR} algorithm is both $\mathcal{OI}$ and $\mathcal{CF}$. According to Proposition 3.1.16, the HC_B algorithm is $\mathcal{CF}$, but not $\mathcal{OI}$. Finally, Proposition 3.1.14 shows that the ME algorithm is $\mathcal{OI}$, but not $\mathcal{CF}$.

Proposition 3.1.13. *Any ER algorithm that is $\mathcal{RM}$ is also $\mathcal{OI}$.*

Proof. Suppose that we are given an $\mathcal{RM}$ ER algorithm E, and the $\mathcal{OI}$ property does not hold. Then there exists a partition P_i and match rule B such that $\bar{E}(P_i, B)$ contains at least two different partitions P_x and P_y. Without loss of generality, we assume that $P_x \not\leq P_y$. However, we violate Definition 3.1.4 by setting $B_1 = B$, $B_2 = B$, $P_o^1 = P_x$, and $P_o^2 = P_y$

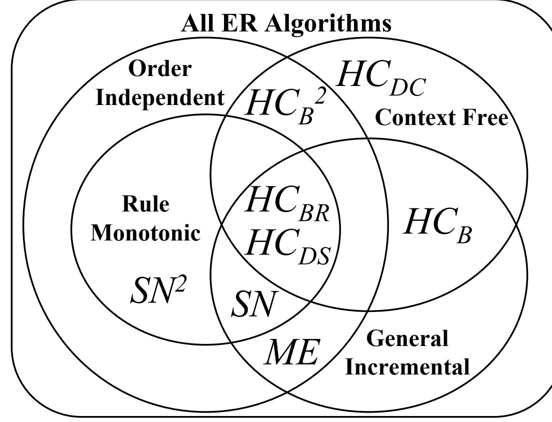


Figure 3.3: ER Algorithms satisfying properties

because $B_1 \leq B_2$, but $P_o^1 = P_x \not\leq P_y = P_o^2$. Hence, E cannot satisfy the $\mathcal{RM}$ property, a contradiction. □

Proposition 3.1.14. *The ME algorithm is $\mathcal{OI}$ and $\mathcal{GI}$, but not $\mathcal{RM}$ or $\mathcal{CF}$.*

Proof. Proposition 3.1.12 shows that ME does not satisfy $\mathcal{RM}$ or $\mathcal{CF}$. The ME algorithm is $\mathcal{OI}$ because it first sorts the records in P_i before resolving them with a sliding window and thus produces a unique solution. The ME algorithm is $\mathcal{GI}$ because $E(P_o^1 \cup (P_i - P)$, $B)$ always returns the same result as $E(P_i, B)$. That is, ME extracts all the records from its input partition before sorting and resolving them, and $P_o^1 \cup (P_i - P)$ contains the exact same records as those in P_i (i.e., $\bigcup_{c \in P_o^1 \cup (P_i - P)} c = \bigcup_{c \in P_i} c$). □

Proposition 3.1.15. *The HC_{BR} algorithm is $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$.*

Proof. Proposition 3.1.11 shows that HC_{BR} is $\mathcal{RM}$ and $\mathcal{CF}$. The HC_{BR} algorithm is also $\mathcal{OI}$ by Proposition 3.1.10. To show that HC_{BR} is $\mathcal{GI}$, consider the four partitions P, P_i, P_o^1, P_o^2 of Definition 3.1.6, and suppose that the three conditions $P \subseteq P_i$, $P_o^1 \in \bar{E}(P$, $B)$, and $P_o^2 \in \bar{E}(P_o^1 \cup (P_i - P), B)$ hold. We show that $P_o^2 \in \bar{E}(P_i, B)$. Starting from P_i, P_o^2 is the result of "replaying" the merges used to produce P_o^1, and then "replaying" the merges used to produce a possible result of $E(P_o^1 \cup (P_i - P), B)$. Since no clusters in P_o^2 match with each other, P_o^2 is also a possible result for $E(P_i, B)$. Hence, $P_o^2 \in \bar{E}(P_i$, $B)$. □

Proposition 3.1.16. *The HC_B algorithm is $\mathcal{CF}$ and $\mathcal{GI}$, but not $\mathcal{OI}$ (and consequently not $\mathcal{RM}$).*

Proof. Proposition 3.1.8 shows that HC_B is $\mathcal{CF}$. The proof that HC_B is $\mathcal{GI}$ is identical to the proof for HC_{BR} and is omitted.

The HC_B algorithm does not satisfy $\mathcal{OI}$ because, depending on the order of records compared, there could be several possible ER results. For example, given the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and the match rule B, suppose that $B(r_1, r_2)$ = `true` and $B(r_2, r_3)$ = `true`, but $B(r_1, r_3)$ = `false`. Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs from each cluster using B. For instance, when comparing $\{r_1, r_2\}$ with $\{r_3\}$, we return the result of $B(r_1, r_3)$. Then depending on the order of clusters in P_i compared, the HC_B algorithm either produces $\{\{r_1, r_2\}, \{r_3\}\}$ or $\{\{r_1, r_2, r_3\}\}$. □

We now add the $\mathcal{GI}$ property into the venn diagram in Figure 3.3. The $\mathcal{GI}$ property partially intersects with the other three properties $\mathcal{RM}$, $\mathcal{CF}$, and $\mathcal{OI}$, and does not include any of them. Proposition 3.1.15 shows that HC_{BR} is $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$. Hence, $\mathcal{GI}$ intersects with the other three properties. Proposition 3.1.16 shows that the HC_B algorithm is $\mathcal{GI}$, but not $\mathcal{OI}$ (and consequently not $\mathcal{RM}$). Proposition 3.1.14 shows that the ME algorithm is $\mathcal{GI}$, but not $\mathcal{CF}$. Hence, none of the other three properties include $\mathcal{GI}$. Proposition 3.1.17 shows the existence of an ER algorithm that is $\mathcal{RM}$ (the same algorithm is also $\mathcal{OI}$), but not $\mathcal{GI}$. Proposition 3.1.18 shows the existence of an ER algorithm that is $\mathcal{CF}$, but not $\mathcal{GI}$. Hence, $\mathcal{GI}$ does not include any of the other three properties.

Proposition 3.1.17. *There exists an ER algorithm that is $\mathcal{RM}$ (and consequently $\mathcal{OI}$ as well), but not $\mathcal{GI}$ or $\mathcal{CF}$.*

Proof. We define a variant of the SN algorithm called SN^2. Recall that the SN algorithm involves identifying the matching pairs of records by first sorting the records from P_i and then comparing records within the same sliding window. At the end, we produce a transitive closure of all the matching records. During the transitive closure, however, we define SN^2 to also consider all the records within the same cluster in the input partition P_i as matching. For example, suppose we have $P_i = \{\{r_1, r_2\}, \{r_3\}\}$ and a match rule B such that $B(r_1, r_3)$

= `true`, but $B(r_1, r_2)$ = `false` and $B(r_2, r_3)$ = `false`. Given a window size of 2, suppose that we sort the records in P_i by record ID into the list [r_1, r_2, r_3]. Then the original SN algorithm sorts will return $\{\{r_1\}, \{r_2\}, \{r_3\}\}$ because r_1 and r_3 are never compared in the same window. However, the new SN^2 algorithm will consider r_1 and r_2 as matching because they were in the same cluster in the input partition P_i. Hence, the result of SN^2 is $\{\{r_1, r_2\}, \{r_3\}\}$.

We prove that the SN^2 algorithm is $\mathcal{RM}$. Given any partition P and two match rules B_1 and B_2 such that $B_1 \leq B_2$, the set of pairs of matching records found by B_1 is clearly a subset of that found by B_2, during the first phase of the SN algorithm where we find all the matching pairs of records within the same sliding window at any point. In addition, the set of matching pairs of records identified from P_i is the same for both B_1 and B_2. As a result, the transitive closure of the matching pairs by B_1 refines the transitive closure result of B_2 (i.e., for any $P_o^1 \in \bar{E}(P_i, B_1)$ and $P_o^2 \in \bar{E}(P_i, B_2)$, $P_o^1 \leq P_o^2$).

We prove that the SN^2 algorithm is not $\mathcal{GI}$ using a counter example. Suppose that we have $P_i = \{\{r_1, r_2\}, \{r_3\}\}$ and a match rule B such that $B(r_1, r_3)$ = `true`, but $B(r_1, r_2)$ = `false` and $B(r_2, r_3)$ = `false`. Given a window size of 2, suppose that we sort the records in P_i by record ID into the list [r_1, r_2, r_3]. As a result, $\bar{E}(P_i, B) = \{\{\{r_1\}, \{r_2\}, \{r_3\}\}\}$ because r_1 and r_3 are never compared within the same window. Using Definition 3.1.6, suppose that we set $P = \{\{r_1\}, \{r_3\}\}$ and $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. As a result, $P_o^1 = E(P, B) = \{\{r_1, r_3\}\}$ because r_1 and r_3 surely match being in the same window. Also, $P_o^2 = E(P_o^1 \cup (P_i - P), B) = E(\{\{r_1, r_3\}, \{r_2\}\}, B) = \{\{r_1, r_3\}, \{r_2\}\}$ because r_1 and r_3 are in the same cluster of the input partition $P_o^1 \cup (P_i - P)$. Since $E(P_i, B)$ is always $\{\{r_1\}, \{r_2\}, \{r_3\}\}$, $P_o^2 \notin \bar{E}(P_i, B) = \{\{\{r_1\}, \{r_2\}, \{r_3\}\}\}$. Hence, the $\mathcal{GI}$ property is not satisfied. (On the other hand, the original SN algorithm is $\mathcal{GI}$; see Proposition 3.1.19.)

To prove that SN^2 is not $\mathcal{CF}$ (in order to show that Figure 3.3 correctly categorizes SN^2), we can directly use the proof of Proposition 3.1.7 that was used to show SN is not $\mathcal{CF}$. □

Proposition 3.1.18. *There exists an ER algorithm that is $\mathcal{CF}$ and $\mathcal{OI}$, but not $\mathcal{GI}$ or $\mathcal{RM}$.*

Proof. We define a variant of the HC_B algorithm (called HC_B^2) where all matching clusters are merged, but only in a certain order. For example, we can define a total ordering between

pairs of clusters where the clusters with the "smallest record IDs" are merged first. We first define the following notations: $MinID_1 = min\{min_{r\in c_1}$ ID(r), $min_{r\in c_2}$ ID$(r)\}$, $MaxID_1 = max\{min_{r\in c_1}$ ID(r), $min_{r\in c_2}$ ID$(r)\}$, $MinID_2 = min\{min_{r\in c_3}$ ID(r), $min_{r\in c_4}$ ID$(r)\}$, and $MaxID_2 = max\{min_{r\in c_3}$ ID(r), $min_{r\in c_4}$ ID$(r)\}$ where the ID(r) function returns the ID of r. We merge the pair of clusters (c_1, c_2) before the pair of matching clusters (c_3, c_4) if either $MinID_1 < MinID_2$ or both $MinID_1 = MinID_2$ and $MaxID_1 < MaxID_2$. For example, the matching clusters $\{r_3, r_9\}$ and $\{r_6, r_7\}$ merge before the matching clusters $\{r_4, r_8\}$ and $\{r_6, r_7\}$ because $MinID_1 = 3$, $MaxID_1 = 6$, $MinID_2 = 4$, $MaxID_2 = 6$ and thus $MinID_1 < MinID_2$. In general, we can use any total ordering of pairs of clusters. As a result of using the total ordering, the HC_B^2 algorithm always produces a unique ER result. For example, suppose that we have $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and the boolean match rule B where $B(r_1, r_2)$ = `true` and $B(r_2, r_3)$ = `true`, but $B(r_1, r_3)$ = `false`. Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs from each cluster using B. For instance, when comparing $\{r_1, r_2\}$ with $\{r_3\}$, we return the result of $B(r_1, r_3)$. Then the ER result $E(P_i, B) = \{\{r_1, r_2\}, \{r_3\}\}$. The clusters $\{r_1\}$ and $\{r_2\}$ merge before $\{r_2\}$ and $\{r_3\}$ because $MinID_1 = 1 < 2 = MinID_2$. Once $\{r_1\}$ and $\{r_2\}$ merge, $\{r_1, r_2\}$ does not match with $\{r_3\}$ because $B(r_1, r_3)$ = `false`.

We show that the HC_B^2 algorithm is $\mathcal{CF}$. Given the four partitions P, P_i, P_o^1, P_o^2, suppose that the four conditions in Definition 3.1.5 are satisfied. That is, $P \subseteq P_i$, $\forall P_o \in \bar{E}(P_i, B)$, $P_o \leq \{\bigcup_{c\in P} c\ , \bigcup_{c\in P_i - P} c\}$, $P_o^1 \in \bar{E}(P, B)$, and $P_o^2 \in \bar{E}(P_i - P, B)$. Since HC_B^2 returns a unique solution, there is exactly one $P_o \in \bar{E}(P_i, B)$. Suppose that P_o was generated via a sequence of cluster merges $M_1, M_2, \ldots$ where each M involves a merge of two clusters. Since $P_o \leq \{\bigcup_{c\in P} c\ , \bigcup_{c\in P_i - P} c\}$, we can split the sequence into merges M_1^a, $M_2^a, \ldots$ that only involve clusters in P and merges $M_1^b, M_2^b, \ldots$ that only involve clusters in $P_i - P$. We can run the first batch of merges $M_1^a, M_2^a, \ldots$ to produce a possible result of $E(P, B)$ and run the second batch of merges $M_1^b, M_2^b, \ldots$ to produce a possible result of $E(P_i - P, B)$. Since HC_B^2 returns a unique solution, both ER results $E(P, B)$ and $E(P_i - P, B)$ are in fact unique and thus are equal to P_o^1 and P_o^2, respectively. The union of P_o^1 and P_o^2 is equivalent to the result of running the merges $M_1, M_2, \ldots$, i.e., $E(P_i, B)$. Hence, $P_o^1 \cup P_o^2 \in \bar{E}(P_i, B)$.

The HC_B^2 algorithm is $\mathcal{OI}$ because the merges are done in a fixed order.

We show that the HC_B^2 algorithm does not satisfy $\mathcal{GI}$ using a counter example. Suppose that we have $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and the Boolean match rule B where $B(r_1, r_2) =$ `true` and $B(r_2, r_3) =$ `true`, but $B(r_1, r_3) =$ `false`. Also assume that, whenever we compare two clusters of records, we simply compare the records with the smallest IDs from each cluster using B. For instance, when comparing $\{r_1, r_2\}$ with $\{r_3\}$, we return the result of $B(r_1, r_3)$. Then the ER result $E(P_i, B) = \{\{r_1, r_2\}, \{r_3\}\}$ because $\{r_1\}$ and $\{r_2\}$ merge first (before $\{r_2\}$ and $\{r_3\}$) and then $\{r_1, r_2\}$ does not match with $\{r_3\}$ because $B(r_1, r_3) =$ `false`. However, in Definition 3.1.6 suppose that we set $P = \{\{r_2\}, \{r_3\}\}$ and $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. Then $P_o^1 = E(P, B) = \{\{r_2, r_3\}\}$ because $\{r_2\}$ matches with $\{r_3\}$. Also $P_o^2 = E(P_o^1 \cup (P_i - P), B) = E(\{\{r_2, r_3\}, \{r_1\}\}, B) = \{\{r_1, r_2, r_3\}\}$ because $\{r_2, r_3\}$ and $\{r_1\}$ match. Since $E(P_i, B)$ is always $\{\{r_1, r_2\}, \{r_3\}\}$, $P_o^2 \notin \bar{E}(P_i, B) = \{\{\{r_1, r_2\}, \{r_3\}\}\}$. Hence, the $\mathcal{GI}$ property is not satisfied.

We show that the HC_B^2 algorithm is not $\mathcal{RM}$ using a counter example. We use the same example in the previous paragraph where $E(P_i, B) = \{\{r_1, r_2\}, \{r_3\}\}$. Now suppose that we are given a stricter match rule B_1 where $B_1(r_2, r_3) =$ `true`, but $B_1(r_1, r_2) =$ `false` and $B_1(r_1, r_3) =$ `false`. Then $E(P_i, B_1) = \{\{r_1\}, \{r_2, r_3\}\}$ because only $\{r_2\}$ and $\{r_3\}$ match. Hence, although $B_1 \leq B$, $E(P_i, B) = \{\{r_1, r_2\}, \{r_3\}\} \not\leq E(P_i, B_1) = \{\{r_1\}, \{r_2, r_3\}\}$, violating $\mathcal{RM}$. □

Finally, Proposition 3.1.19 shows that the SN algorithm is $\mathcal{RM}$, $\mathcal{OI}$, and $\mathcal{GI}$, but not $\mathcal{CF}$.

Proposition 3.1.19. *The SN algorithm is $\mathcal{RM}$ (and consequently $\mathcal{OI}$) and $\mathcal{GI}$, but not $\mathcal{CF}$.*

Proof. Proposition 3.1.7 shows that SN is $\mathcal{RM}$ (and consequently $\mathcal{OI}$), but not $\mathcal{CF}$. We prove that the SN algorithm is $\mathcal{GI}$. Recall that SN extracts all records in the input partition before sorting and resolving them. Hence, the ER result is the same for different input partitions as long as they contain the same records. For example, $E(\{\{r_1, r_2\}, \{r_3\}\}, B) = E(\{\{r_1\}, \{r_2, r_3\}\}, B)$ because the two input partitions contain the same records r_1, r_2, and r_3. In Definition 3.1.6, we know that $E(P_o^1 \cup (P_i - P), B)$ always returns the same result as $E(P_i, B)$ because $P_o^1 \cup (P_i - P)$ contains the same records as those in P_i. Thus, for any $P_2 \in \bar{E}(P_o^1 \cup (P_i - P), B)$, $P_2 \in \bar{E}(P_b, B)$. □

3.1.3 Materialization

To improve our chances that we can efficiently compute a new ER result with rule B_2, when we compute earlier results we can materialize results that involve predicates likely to be in B_2. When we compute an earlier result $E(P_i, B_1)$ where say $B_1 = p_1 \wedge p_2 \wedge p_3$, we can also materialize results such as $E(P_i, p_1)$, $E(P_i, p_2)$, $E(P_i, p_1 \wedge p_2)$, and so on. The most useful materializations will be those that can help us later with $E(P_i, B_2)$. (See Section 3.2.) For concreteness, here we will assume that we materialize *all* conjuncts of B_1 (in our example, $E(P_i, p_1)$, $E(P_i, p_2)$, and $E(P_i, p_3)$).

Instead of serially materializing each conjunct, however, we can amortize the common costs by materializing different conjuncts in a concurrent fashion. For example, parsing and initializing the records can be done once during the entire materialization. More operations can be amortized depending on the given ER algorithm. For example, when materializing conjuncts using an ER algorithm that always sorts its records before resolving them, the records only need to be sorted once for all materializations. In Section 3.4.5, we show that amortizing common operations can significantly reduce the time overhead of materializing conjuncts. A partition of the records in S can be stored compactly in various ways. One approach is to store sets of records IDs in a set where each inner set represents a cluster of records. A possibly more space-efficient technique is to maintain an array A of records (where the ID is used as the index) where each cell contains the cluster ID. For example, if r_5 is in the second cluster, then $A[5] = 2$. If there are only a few clusters, we only need a small number of bits for saving each cluster ID. For example, if there are only 8 clusters, then each entry in A only takes 3 bits of space.

3.1.4 Rule Evolution

We provide efficient rule evolution techniques for ER algorithms using the properties. Our first algorithm supports ER algorithms that are $\mathcal{RM}$ and $\mathcal{CF}$. As we will see, rule evolution can still be efficient for ER algorithms that are only $\mathcal{RM}$. Our second algorithm supports ER algorithms that are $\mathcal{GI}$. Before running the rule evolution algorithms, we materialize ER results for conjuncts of the old match rule B_1 by storing a partition of the input records S (i.e., the ER result) for each conjunct in B_1 (see Section 3.2 for possible optimizations).

To explain our rule evolution algorithms, we review a basic operation on partitions. The *meet* of two partitions P_1 and P_2 (denoted as $P_1 \wedge P_2$) returns a new partition of S whose members are the non-empty intersections of the clusters of P_1 with those of P_2. For example, given the partitions $P_1 = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ and $P_2 = \{\{r_1\}, \{r_2, r_3, r_4\}\}$, the meet of P_1 and P_2 becomes $\{\{r_1\}, \{r_2, r_3\}, \{r_4\}\}$ since r_2 and r_3 are clustered in both partitions. We also show the following lemma regarding the meet operation.

Lemma 3.1.20. *If* $\forall i, P \leq P_i$ *then* $P \leq \bigwedge P_i$

Proof. We prove by induction on the number k of partitions that are combined with the meet operation.

Base case: $k = 1$. Obviously, $P \leq P_1$.

Induction: Suppose that $P \leq \bigwedge_{i \in \{1,\ldots,k\}} P_i$. We then show that the same equation holds when k increases by 1. Any pair of records r, s that are clustered in P are also clustered in $\bigwedge_{i \in \{1,\ldots,k\}} P_i$ and P_{k+1} by the induction hypothesis. Since r and s are clustered in both $\bigwedge_{i \in \{1,\ldots,k\}} P_i$ and P_{k+1}, r and s are also clustered in the meet of the two partitions, i.e., $\bigwedge_{i \in \{1,\ldots,k+1\}} P_i$. Hence any pair of records that are clustered in P are also clustered in $\bigwedge_{i \in \{1,\ldots,k+1\}} P_i$. □

Algorithm 7 performs rule evolution for ER algorithms that are both $\mathcal{RM}$ and $\mathcal{CF}$. The input requires the input partition P_i, the old and new match rules (B_1 and B_2, respectively), and a hash table H that contains the materialized ER results for the conjuncts of B_1. The conjuncts of a match rule B is denoted as $Conj(B)$. For simplicity, we assume that B_1 and B_2 share at least one conjunct. Step 3 exploits the $\mathcal{RM}$ property and meets the partitions of the common conjuncts between B_1 and B_2. For example, suppose that we have $B_1 = p_1 \wedge p_2 \wedge p_3$ and $B_2 = p_1 \wedge p_2 \wedge p_4$. Given $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}\}$, say we also have the materialized ER results $E(P_i, p_1) = \{\{r_1, r_2, r_3\}, \{r_4\}\}$ and $E(P_i, p_2) = E(P_i, p_3) = \{\{r_1\}, \{r_2, r_3, r_4\}\}$. Since the common conjuncts of B_1 and B_2 are p_1 and p_2, we generate the meet of $E(P_i, p_1)$ and $E(P_i, p_2)$ as $M = \{\{r_1\}, \{r_2, r_3\}, \{r_4\}\}$. By $\mathcal{RM}$, we know that $E(P_i, B_2)$ refines M because B_2 is stricter than both p_1 and p_2. That is, each cluster in the new ER result is contained in exactly one cluster in the meet M. Step 4 then exploits the $\mathcal{CF}$ property to resolve for each cluster c of M, the clusters in P_i that are subsets of c (i.e., $\{c' \in P_i | c' \subseteq c\}$). Since the clusters in different $\{c' \in P_i | c' \subseteq c\}$'s do not merge with

ALGORITHM 7: Rule evolution given $\mathcal{RM}$ and $\mathcal{CF}$

1: **input:** The input partition P_i, the match rules B_1, B_2, the ER result for each conjunct of B_1, the hash table H containing materializations of conjuncts in B_1
2: **output:** The output partition $P_o \in \bar{E}(P_i, B_2)$
3: Partition $M \leftarrow \bigwedge_{conj \in Conj(B_1) \cap Conj(B_2)} H(conj)$
4: **return** $\bigcup_{c \in M} E(\{c' \in P_i | c' \subseteq c\}, B_2)$

each other, each $\{c' \in P_i | c' \subseteq c\}$ can be resolved independently. As a result, we can return $\{\{r_1\}\} \cup E(\{\{r_2\}, \{r_3\}\}, B_2) \cup \{\{r_4\}\}$ as the new ER result of B_2.

In order to prove that Algorithm 7 is correct, we first prove three lemmas below. For simplicity, we denote $\{c' \in P_i | c' \subseteq c\}$ as IN(P_i, c). For a set of clusters Q, we define IN(P_i, Q) = $\bigcup_{c \in Q}$IN(P_i, c).

Lemma 3.1.21. *For an $\mathcal{RM}$ algorithm, $\forall P_o \in \bar{E}(P_i, B_2)$, $P_o \leq M$.*

Proof. We first use the $\mathcal{RM}$ property to prove that $\forall P_o \in \bar{E}(P_i, B_2)$, $P_o \leq M$. For each $conj \in Conj(B_1) \cap Conj(B_2)$, $B_2 \leq conj$. Hence, by $\mathcal{RM}$, $\forall P_o^1 \in \bar{E}(P_i, B_2)$ and $\forall P_o^2 \in \bar{E}(P_i, conj)$, $P_o^1 \leq P_o^2$. Since $M = \bigwedge_{conj \in Conj(B_1) \cap Conj(B_2)} E(P_i, conj)$, we conclude that $\forall P_o \in \bar{E}(P_i, B_2)$, $P_o \leq M$ using Lemma 3.1.20. (Notice that M is unique because each $E(P_i, conj)$ is also unique by $\mathcal{RM}$ and $\mathcal{CF}$.) □

Lemma 3.1.22. *Say we have an algorithm that is $\mathcal{RM}$ and $\mathcal{CF}$, an initial partition P_i, and two rules B_1, B_2 with conjuncts $Conj(B_1)$, $Conj(B_2)$. Let $M = \bigwedge_{conj \in Conj(B_1) \cap Conj(B_2)} E(P_i, conj)$. For any $W \subseteq M$, E(IN(P_i, W), B_2) $\leq W$.*

Proof. We use $\mathcal{CF}$ to prove that for any $W \subseteq M$, E(IN(P_i, W), B_2) $\leq W$. To avoid confusion with the initial set of clusters P_i, we use the symbol P_i' instead of P_i when using Definition 3.1.5. We satisfy the first two conditions in Definition 3.1.5 by setting P = IN(P_i, W) and $P_i' = P_i$. The first condition, $P \subseteq P_i'$, is satisfied because IN(P_i, W) is a subset of P_i by definition. The second condition, $\forall P_o \in \bar{E}(P_i', B_2)$, $P_o \leq \{\bigcup_{c \in P} c, \bigcup_{c \in P_i' - P} c\}$, is satisfied because we know by Lemma 3.1.21 that $\forall P_o \in \bar{E}(P_i, B_2)$, $P_o \leq M$. Also, we know that $W \subseteq M$. Thus, for $P_o^1 = E(P, B_2)$ and $P_o^2 = E(P_i' - P, B_2)$, $P_o^1 \cup P_o^2 = E(P_i', B_2)$. (Notice that E returns a unique solution being $\mathcal{RM}$ and $\mathcal{CF}$.) Since $E(P_i', B_2) \leq M$, $P_o^1 = E$(IN(P_i, W), B_2) $\leq W$, which also implies that E(IN(P_i, W), B_2) $\leq M$. □

Lemma 3.1.23. *Given the same setup as in Lemma 3.1.22, let $Y \subseteq M$ and $Z \subseteq M$ such that $Y \cap Z = \emptyset$ and $Y \cup Z = W$ (note: $W \subseteq M$). Let $Q = E(IN(P_i, W), B_2)$ (there is only one solution). Then $Q \leq \{\bigcup_{c \in Y} c\ , \bigcup_{c \in Z} c\}$.*

Proof. Suppose that $Q \nleq \{\bigcup_{c \in Y} c\ , \bigcup_{c \in Z} c\}$. Then a cluster in Q must have one cluster from Y and one from Z. Since $Q \leq M$ (by Lemma 3.1.22), however, we arrive at a contradiction. □

Proposition 3.1.24. *Algorithm 7 correctly returns a partition $P_o \in \bar{E}(P_i, B_2)$.*

Proof. We use $\mathcal{CF}$ to prove that $P_o = \bigcup_{c \in M} E(\{c' \in P_i | c' \subseteq c\}, B_2) \in \bar{E}(P_i, B_2)$. Suppose that $M = \{c_1, \ldots, c_{|M|}\}$. We omit the B_2 from any expression $E(P, B_2)$ for brevity (B_1 is not used in this proof). To avoid confusion with the initial set of clusters P_i, we use the symbol P'_i instead of P_i when using Definition 3.1.5 later in this proof. We define the following notation: $\alpha(k) = \bigcup_{c \in \{c_1,\ldots,c_k\}} E(\text{IN}(P_i, c))$ and $\beta(k) = \bigcup_{c \in M - \{c_1,\ldots,c_k\}} \text{IN}(P_i, c)$. Our goal $\bigcup_{c \in M} E(\{c' \in P_i | c' \subseteq c\}, B_2) \in \bar{E}(P_i, B_2)$ can thus be written as $\alpha(|M|) \in \bar{E}(P_i)$. To prove that $\alpha(|M|) \in \bar{E}(P_i)$, we first prove a more general statement: any $\alpha(k) \cup E(\beta(k)) \in \bar{E}(P_i)$ for $k \in \{0, \ldots, |M|\}$. Clearly, if our general statement holds, we can show that $\alpha(|M|) \in \bar{E}(P_i)$. We use induction on the number of partitions k processed in isolation.

Base case: $k = 0$. Then any $\alpha(0) \cup E(\beta(0)) = E(P_i) \in \bar{E}(P_i)$.

Induction: Suppose the equation above holds for $k = n$ (i.e., any $\alpha(n) \cup E(\beta(n)) \in \bar{E}(P_i)$). We want to show that the equation also holds when $k = n + 1$ where $n + 1 \leq |M|$.

We first show that $E(\text{IN}(P_i, c_{n+1})) \cup E(\beta(n+1)) = E(\beta(n))$ using $\mathcal{CF}$. We satisfy the first two conditions of Definition 3.1.5 by setting $P = \text{IN}(P_i, c_{n+1})$ and $P'_i = \beta(n)$. The first condition, $P \subseteq P'$, is satisfied because $\text{IN}(P_i, c_{n+1})$ is a subset of $\beta(n)$ by definition. The second condition, $\forall P_o \in \bar{E}(P'_i)$, $P_o \leq \{\bigcup_{c \in P} c\ , \bigcup_{c \in P'_i - P} c\}$, is satisfied by Lemma 3.1.23 by setting $Y = \{c_{n+1}\}$ and $Z = \{c_{n+2}, \ldots, c_{|M|}\}$. Hence, for $P_o^1 = E(P)$ and $P_o^2 = E(P'_i - P)$, $P_o^1 \cup P_o^2 = E(\text{IN}(P_i, c_{n+1})) \cup E(\beta(n+1)) = E(\beta(n))$.

Now $\alpha(n+1) \cup E(\beta(n+1)) = \alpha(n) \cup E(\text{IN}(P_i, c_{n+1})) \cup E(\beta(n+1))$, which is equal to some result $\alpha(n) \cup E(\beta(n))$. Using the induction hypothesis, we know that any $\alpha(n) \cup E(\beta(n)) \in \bar{E}(P_i)$, so $\alpha(n+1) \cup E(\beta(n+1)) \in \bar{E}(P_i)$ as well, which proves our induction step. □

Proposition 3.1.25. *The complexity of Algorithm 7 is* $O(c \times |S| + \frac{|S|^c}{z^c} \times g(\frac{|P_i| \times z^c}{|S|^c}, \frac{|S|}{|P_i|}))$ *where* S *is the set of records in the input partition of records* P_i*,* c *is the number of common conjuncts between* B_1 *and* B_2*,* z *is the average cluster size for any partition produced by a conjunct, and* $g(N, A)$ *is the complexity of the ER algorithm* E *for an input partition containing* N *clusters with an average size of* A *records.*

Proof. The complexity of Algorithm 7 can be computed by adding the cost for meeting partitions of the common conjuncts (Step 3) and the cost for running ER on the clusters in M (Step 4). In Step 3, we perform $c - 1$ meets, which takes about $O(c \times |S|)$ time where the meet operation can be run in $O(|S|)$ time [73]. Given a record r, the probability of some other record s clustering with r is $\frac{z-1}{|S|-1}$ because each cluster has an average size of z. The probability for s to be in the same cluster with r for all the c meeting partitions is thus $(\frac{z-1}{|S|-1})^c$, assuming that all conjuncts cluster records independently. As a result, the expected number of records to be clustered with r in M is $(|S| - 1) \times (\frac{z-1}{|S|-1})^c$. Hence, the average cluster size of M is $1 + (|S| - 1) \times (\frac{z-1}{|S|-1})^c \approx \frac{z^c}{|S|^{c-1}}$ records. The expected number of clusters in M is thus approximately $\frac{|S|}{\frac{z^c}{|S|^{c-1}}} = \frac{|S|^c}{z^c}$. Each $\{c' \in P_i | c' \subseteq c\}$ (where $c \in M$) has on average $\frac{|P_i|}{\frac{|S|^c}{z^c}} = \frac{|P_i| \times z^c}{|S|^c}$ clusters of P_i where the average size of each cluster in P_i is $\frac{|S|}{|P_i|}$. The complexity of Step 4 is thus $O(\frac{|S|^c}{z^c} \times g(\frac{|P_i| \times z^c}{|S|^c}, \frac{|S|}{|P_i|}))$. Hence, the total algorithm complexity is $O(c \times |S| + \frac{|S|^c}{z^c} \times g(\frac{|P_i| \times z^c}{|S|^c}, \frac{|S|}{|P_i|}))$. □

While Algorithm 7 does not improve the complexity of the given ER algorithm E running without rule evolution, its runtime can be much faster in practice because the overhead for meeting partitions is not high (Step 3), and there can be large savings by running ER on small subsets of P_i (i.e., the $\{c' \in P_i | c' \subseteq c\}$'s) (Step 4) rather than on the entire partition P_i.

The rule evolution algorithm for ER algorithms that are only $\mathcal{RM}$ is identical to Algorithm 7 except for Step 4, where we can no longer process subsets of P_i independently. However, we can still run Step 4 efficiently using global information. We revisit the sorted neighborhood ER algorithm (SN) in Section 3.1.2. Recall that the first step of SN is to move a sliding window on a sorted list of records, comparing records pairwise only within the same window of size W. (The second step is a transitive closure of all matching pairs.)

In Step 4, we are able to resolve each $\{c' \in P_i | c' \subseteq c\}$ ($c \in M$) using the same window size W as long as we also use the global sort information of the records to make sure only the records that would have been in the same window during the original run of SN should be compared with each other. Suppose that we have $B_1 = p_{name} \wedge p_{zip}$, $B_2 = p_{name} \wedge p_{phone}$, and the initial set $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}, \{r_5\}\}$. We set the sort key to be the record ID (e.g., r_4 has the ID 4). As a result, the records are sorted into the list $[r_1, r_2, r_3, r_4, r_5]$. Using a window size of W= 3, suppose we materialize $E(P_i, p_{name})$ = $\{\{r_1, r_3, r_5\}, \{r_2\}, \{r_4\}\}$ because r_1 and r_3 matched when the window covered $[r_1, r_2, r_3]$ and r_3 and r_5 matched when the window covered $[r_3, r_4, r_5]$. The records r_1 and r_5 only match during the transitive closure in the second step of SN. The meet M in Algorithm 7 is also $\{\{r_1, r_3, r_5\}, \{r_2\}, \{r_4\}\}$ because there is only one common conjunct p_{name} between B_1 and B_2. Thus, we only need to resolve the set $\{r_1, r_3, r_5\}$ using B_2. However, we must be careful and should not simply run $E(\{r_1, r_3, r_5\}, B_2)$ using a sliding window of size 3. Instead, we must take into account the global ordering information and never compare r_1 and r_5, which were never in the same window. Thus, if $B_2(r_1, r_3)$ = `false`, $B_2(r_3, r_5)$ = `false`, and $B_2(r_1, r_5)$ = `true`, the correct ER result is that none of r_1, r_3, r_5 are clustered. While we need to use the global sort information of records, our rule evolution is still more efficient than re-running SN on the entire input P_i (see Section 3.4).

Algorithm 3 performs rule evolution for ER algorithms that only satisfy the $\mathcal{GI}$ property. Algorithm 3 is identical to Algorithm 7 except that Step 4 is replaced with the code "**return** $E(\bigcup_{c \in M} E(\{c' \in P_i | c' \subseteq c\}, B_2), B_2)$". Since the $\mathcal{RM}$ property is not satisfied anymore, we can no longer assume that the meet M is refined by the ER result of B_2. Hence, after each $\{c' \in P_i | c' \subseteq c\}$ is resolved, we need to run ER on the union of the results (i.e., the outermost ER operation in Step 4) to make sure we found all the matching records. The $\mathcal{GI}$ property guarantees that the output P_o is equivalent to a result in $\bar{E}(P_i, B_2)$. Using the same example for Algorithm 7, we now return $E(\{\{r_1\}\} \cup E(\{r_2, r_3\}, B_2) \cup \{\{r_4\}\}, B_2)$.

There are two factors that make Algorithm 3 efficient for certain ER algorithms. First, each cluster in M is common to several ER results and thus contains records that are likely to be clustered. An ER algorithm may run faster by resolving clusters that are likely to match first. Second, there are fewer clusters for the outer E operation to resolve compared

to when E runs on the initial partition P_i. An ER algorithm may run faster when resolving fewer (but larger) clusters. While not all ER algorithms that are $\mathcal{GI}$ will speed up from these two factors, we will see in Section 3.4 that the HC_B algorithm indeed benefits from Algorithm 3.

The complexity of Algorithm 3 can be computed by adding the cost for meeting partitions and the cost for running ER on clusters. In comparison to Algorithm 7, the additional cost is the outermost ER operation in Step 4. In practice, Algorithm 3 is slower than Algorithm 7, but can still be faster than running the ER algorithm E without rule evolution.

Proposition 3.1.26. *Algorithm 3 correctly returns an ER result $P_o \in \bar{E}(P_i, B_2)$.*

Proof. Suppose $M = \{c_1, c_2, \ldots, c_{|M|}\}$. For this proof, we denote $\{c' \in P_i | c' \subseteq c\}$ as IN(P_i, c). We also omit B_2 from each $E(P, B_2)$ expression for brevity (B_1 is not used in this proof). We define the following notations: $\alpha(k) = \bigcup_{c \in M - \{c_1,\ldots,c_k\}} E(\text{IN}(P_i, c))$ and $\beta(k) = \bigcup_{c \in \{c_1,\ldots,c_k\}} \text{IN}(P_i, c)$. To avoid confusion with the initial set of clusters P_i, we use P_i' instead of P_i when using Definition 3.1.6 later in this proof. To prove that $P_o = E(\alpha(0)) \in \bar{E}(P_i) = \bar{E}(\beta(|M|))$, we prove the more general statement that $P_o \in \bar{E}(\alpha(k) \cup \beta(k)))$ for $k \in \{0, \ldots, |M|\}$. Clearly, if our general statement holds, we can show that $P_o \in \bar{E}(\beta(|M|)) = \bar{E}(P_i)$ by setting $k = |M|$.

Base case: We set $k = 0$. Then $P_o = E(\alpha(0)) \in \bar{E}(\alpha(0)) = \bar{E}(\alpha(0) \cup \beta(0))$.

Induction: Suppose that our statement holds for $k = n$, i.e., $P_o = E(\alpha(0)) \in \bar{E}(\alpha(n) \cup \beta(n))$. We want to show that the same expression holds for $k = n + 1$ where $n + 1 \leq |M|$. We use the $\mathcal{GI}$ property by setting P = IN(P_i, c_{n+1}) and $P_i' = \alpha(n+1) \cup \beta(n+1)$. The first condition $P \subseteq P_i'$ is satisfied because $\beta(n+1)$ contains P. We then set $P_o^1 = E(P) = E(\text{IN}(P_i, c_{n+1}))$ and $P_o^2 = E(P_o^1 \cup (P_i' - P)) = E(E(\text{IN}(P_i, c_{n+1})) \cup \alpha(n+1) \cup \beta(n)) = E(\alpha(n) \cup \beta(n))$. The $\mathcal{GI}$ property tells us that $P_o^2 \in \bar{E}(P_i') = \bar{E}(\alpha(n+1) \cup \beta(n+1))$. Thus, any $E(\alpha(n) \cup \beta(n)) \in \bar{E}(\alpha(n+1) \cup \beta(n+1))$. Using our induction hypothesis, we conclude that $P_o = E(\alpha(0)) \in \bar{E}(\alpha(n) \cup \beta(n)) \subseteq \bar{E}(\alpha(n+1) \cup \beta(n+1))$. □

3.2 Materialization Strategies

Until now, we have used a general strategy for rule materialization where we materialize on each conjunct. In this section, we list possible optimizations for materializations given more application-specific knowledge. Our list is by no means exhaustive, and the possible optimizations will depend on the ER algorithm and match rules.

A group of conjuncts is "stable" if they appear together in most match rules. As a result, the group can be materialized instead of all individual conjuncts. For example, if the conjuncts p_1, p_2, and p_3 are always compared as a conjunction in a person records match rule, then we can materialize on $p_1 \wedge p_2 \wedge p_3$ together rather than on the three conjuncts separately. Hence, the time and space overhead of materialization can be saved.

If we know the pattern of how the match rule will evolve, we can also avoid materializing on all conjuncts. In the ideal case where we know that the match rule can only get stricter, we do not have to save any additional materializations other than the ER result of the old match rule. Another scenario is when we are only changing the postfix of the old match rule, so we only need to materialize on all the prefixes of the old match rule. For example, if we have the match rule $p_1 \wedge p_2 \wedge p_3$, then we can materialize on p_1, $p_1 \wedge p_2$, and $p_1 \wedge p_2 \wedge p_3$. If the ER algorithm is both $\mathcal{RM}$ and $\mathcal{CF}$, then the ER result of $p_1 \wedge p_2$ can be computed efficiently from the ER result of p_1, and the ER result of $p_1 \wedge p_2 \wedge p_3$ from that of $p_1 \wedge p_2$.

3.3 Distance-based Evolution

We now consider rule evolution on distance-based clustering where records are clustered based on their relative distances instead of the Boolean match results used in the match-based clustering model. We first define our match rule as a distance function. We then define the notion of strictness between distance match rules and define properties analogous to those in Section 3.1.2. Finally, we provide a model on how the distance match rule can evolve and present our rule evolution techniques.

3.3.1 Distance-based Clustering Model

A distance match rule is defined as a commutative distance function D that returns a non-negative distance between two records instead of a Boolean function as in Section 3.1. For example, the distance between two person records may be the sum of the distances between their names, addresses, and phone numbers. Records are now clustered based on their relative distances with each other. The details on how exactly D is used for the clustering differs for each ER algorithm. In hierarchical clustering using distances [70], the closest pairs of records are merged first until a certain criterion is met. A more sophisticated approach [21] may cluster a set of records that are closer to each other compared to records outside, regardless of the absolute distance values. Other than using a distance match rule instead of a Boolean match rule, the definition of a valid ER algorithm remains the same as Definition 3.1.1. Compared to the general ER model in Chapter **??**, there is the notion of plugging in a distance function into an ER algorithm as opposed to using an ER algorithm with a fixed distance function.

In order to support rule evolution, we model D to return a *range* of possible non-negative distances instead of a single non-negative distance. For example, the distance $D(r_1, r_2)$ can be all possible distances within the range [13, 15]. We denote the minimum possible value of $D(r_1, r_2)$ as $D(r_1, r_2).min$ (in our example, 13) and the maximum value as $D(r_1, r_2).max$ (in our example, 15). As a result, an ER algorithm that only supports single-value distances must be extended to support ranges of values. The extension is specific to the given ER algorithm. However, in the case where the distance match rule only returns single value ranges, the extended algorithm must be identical to the original ER algorithm. Thus, the extension for general distances is only needed for rule evolution and does not change the behavior of the original ER algorithm.

A rule evolution occurs when a distance match rule D_1 is replaced by a new distance match rule D_2. We define the notion of relative strictness between distance match rules analogous to Definition **??**.

Definition 3.3.1. *A distance match rule D_1 is* stricter than *another rule D_2 (denoted as $D_1 \leq D_2$) if $\forall r, s$, $D_1(r, s).min \geq D_2(r, s).min$ and $D_1(r, s).max \leq D_2(r, s).max$.*

That is, D_1 is stricter than D_2 if its distance range is always within that of D_2 for any

record pair. For example, if $D_2(r, s)$ is defined as all the possible distance values within [$D_1(r, s).min{-}1$, $D_1(r, s).max{+}1$], then $D_1 \leq D_2$ (assuming that $D_1(r, s).min \geq 1$).

3.3.2 Properties

We use properties analogous to $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$ from Section 3.1.2 for the distance-based clustering model. The only differences are that we now use distance match rules instead of Boolean match rules (hence we must replace all B's with D's) and Definition 3.3.1 instead of Definition **??** for comparing the strictness between distance match rules. To show how the properties hold in practice, we consider two distance-based clustering algorithms: HC_{DS} and HC_{DC}.

HC_{DS} The Single-link Hierarchical Clustering algorithm [45, 70] (HC_{DS}) merges the closest pair of clusters (i.e., the two clusters that have the smallest distance) into a single cluster until the smallest distance among all pairs of clusters exceeds a certain threshold T. When measuring the distance between two clusters, the algorithm takes the smallest possible distance between records within the two clusters. Suppose we have the input partition P_i = {{r_1}, {r_2}, {r_3}} where $D(r_1, r_2)$ = 2, $D(r_2, r_3)$ = 4, and $D(r_1, r_3)$ = 5 (we later extend HC_{DS} to support ranges of distances) with T = 2. The HC_{DS} algorithm first merges r_1 and r_2, which are the closest records and have a distance smaller or equal to T, into $\{r_1, r_2\}$. The cluster distance between $\{r_1, r_2\}$ and $\{r_3\}$ is the minimum of $D(r_1, r_3)$ and $D(r_2, r_3)$, which is 4. Since the distance exceeds T, $\{r_1, r_2\}$ and $\{r_3\}$ do not merge, and the final ER result is {$\{r_1, r_2\}$, $\{r_3\}$}.

We extend the HC_{DS} algorithm by allowing ranges of distances to be returned by a distance match rule, but only comparing the minimum value of a range with either another range or the threshold T. That is, $D(r, s)$ is considered a smaller distance than $D(u, v)$ if $D(r, s).min \leq D(u, v).min$. Also, $D(r, s)$ is considered smaller than T if $D(r, s).min \leq T$. For example, [3, 5] $<$ [4, 4] because 3 is smaller than 4, and [3, 5] $>$ T = 2 because 3 is larger than 2. The extended HC_{DS} algorithm is trivially identical to the original HC_{DS} algorithm when D only returns a single value.

Proposition 3.3.2 shows that the HC_{DS} algorithm is $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$. As a

result, the HC_{DS} algorithm can use Algorithm 7 (with minor changes; see Section 3.3.3) for rule evolution.

Proposition 3.3.2. *The extended HC_{DS} algorithm is $\mathcal{RM}$, $\mathcal{CF}$, $\mathcal{GI}$, and $\mathcal{OI}$.*

Proof. We first define the notation of connectedness for HC_{DS}. Two records r and s are connected under D, T, and P_i if there exists a sequence of records [r_1 (= r), ..., r_n (= s)] where for each pair (r_i, r_{i+1}) in the path, either $D(r_i, r_{i+1}).min \leq T$ or $\exists c \in P_i$ s.t. $r_i \in c, r_{i+1} \in c$. Notice that connectedness is "transitive," i.e., if r and s are connected and s and t are connected, then r and t are also connected.

We now prove the following Lemma.

Lemma 3.3.3. *Two records r and s are connected under D, T, and P_i if and only if r and s are in the same cluster in E(P_i, D) using the HC_{DS} algorithm.*

Proof. The proof is identical to the proof in Lemma 3.1.9, except that we use D and T instead of B for comparing records. □

We prove that the extended HC_{DS} algorithm is $\mathcal{RM}$ (and thus $\mathcal{OI}$). Given the input partition P_i and two distance match rules D_1 and D_2 where $D_1 \leq D_2$, we want to show that for $P_o^1 \in \bar{E}(P_i, D_1)$ and $P_o^2 \in \bar{E}(P_i, D_2)$, $P_o^1 \leq P_o^2$. By Lemma 3.3.3, any pair of records r and s that get clustered together in P_o^1 are connected under D_1, T, and P_i. If $D_1 \leq D_2$, we know that $D_2(r_i, r_{i+1}).min \leq D_1(r_i, r_{i+1}).min \leq T$. Hence, r and s are also connected under D_2, T, and P_i. By Lemma 3.3.3, r and s are guaranteed to be clustered in P_o^2 as well. As a result, $P_o^1 \leq P_o^2$. Since HC_{DS} is $\mathcal{RM}$, it is also $\mathcal{OI}$.

We prove that the extended HC_{DS} algorithm is $\mathcal{GI}$. Using Definition 3.1.6, suppose that the three conditions hold, i.e., $P \subseteq P_i$, $P_o^1 \in \bar{E}(P, D)$, and $P_o^2 \in \bar{E}(P_o^1 \cup (P_i - P), D)$. Since HC_{DS} is $\mathcal{OI}$ (proved in previous paragraph), the ER results $E(P_i, D)$ and $E(P_o^1 \cup (P_i - P), D)$ are both unique.

We prove that $E(P_i, D)$ and $E(P_o^1 \cup (P_i - P), D)$ are in fact the same partition. By Lemma 3.3.3, any pair r and s in $E(P_i, D)$ are connected under D, T, and P_i. As a result, there exists a sequence of records [r_1 (= r), ..., r_n (= s)] where for each pair (r_i, r_{i+1}) in the path, either $D(r_i, r_{i+1}).min \leq T$ or $\exists c \in P_i$ s.t. $r_i \in c, r_{i+1} \in c$. The fact that $D(r_i, r_{i+1}).min \leq T$ does not change when evaluating $E(P_o^1 \cup (P_i - P), D)$. If $\exists c \in P_i$

s.t. $r_i \in c, r_{i+1} \in c$, then $\exists c \in P_o^1 \cup (P_i - P)$ s.t. $r_i \in c, r_{i+1} \in c$ because none of the clusters in P_i are un-merged in $P_o^1 \cup (P_i - P)$. Hence, r_i and r_{i+1} are connected in $E(P_o^1 \cup (P_i - P), D)$, which means that r and s are connected in $E(P_o^1 \cup (P_i - P), D)$. Thus, r and s are clustered together by Lemma 3.3.3.

Conversely, suppose that r and s are clustered together in $E(P_o^1 \cup (P_i - P), D)$. Then by Lemma 3.3.3, r and s are connected under D, T, and $P_o^1 \cup (P_i - P)$. As a result, there exists a sequence of records [r_1 (= r), ..., r_n (= s)] where for each pair (r_i, r_{i+1}) in the path, either $D(r_i, r_{i+1}).min \leq T$ or $\exists c \in P_o^1 \cup (P_i - P)$ s.t. $r_i \in c, r_{i+1} \in c$. The fact that $D(r_i, r_{i+1}).min \leq T$ does not change when evaluating $E(P_i, D)$. If $\exists c \in P_o^1 \cup (P_i - P)$ s.t. $r_i \in c, r_{i+1} \in c$, then $\exists c \in P_o^1 = E(P, D)$ s.t. $r_i \in c, r_{i+1} \in c$ or $\exists c \in P_i - P$ s.t. $r_i \in c, r_{i+1} \in c$. In the latter case, we know that $\exists c \in P_i$ s.t. $r_i \in c, r_{i+1} \in c$ is true. In the former case, we know by Lemma 3.3.3 that r_i and r_{i+1} are connected under D, T, and P. As a result, r_i and r_{i+1} are also connected under D, T, and P_i because $P \subseteq P_i$. Combining the former and latter cases, r_i and r_{i+1} are always connected under D, T, and P_i. Hence, r and s are also connected under D, T, and P_i. By Lemma 3.3.3, r and s must be clustered together in $E(P_i, D)$. Hence, we have proved that $E(P_i, D)$ and $E(P_o^1 \cup (P_i - P), D)$ are the same.

Using the third condition $P_o^2 \in \bar{E}(P_o^1 \cup (P_i - P), D)$, we know that $P_o^2 \in \bar{E}(P_i, D)$, so the HC_{DS} algorithm is $\mathcal{GI}$.

Finally, we prove that the extended HC_{DS} algorithm is $\mathcal{CF}$. Given the four partitions P, P_i, P_o^1, P_o^2, suppose that the four conditions in Definition 3.1.5 are satisfied. That is, $P \subseteq P_i, \forall P_o \in \bar{E}(P_i, D)$, $P_o \leq \{\bigcup_{c \in P} c\, , \bigcup_{c \in P_i - P} c\}$, $P_o^1 \in \bar{E}(P, D)$, and $P_o^2 \in \bar{E}(P_i - P, D)$. Since HC_{DS} returns a unique solution, there is exactly one $P \in \bar{E}(P_i, D)$. Suppose that $E(P_i, D)$ generates a sequence of cluster merges M_1, M_2, ... where each M involves a merge of two clusters. Since $P \leq \{\bigcup_{c \in P} c\, , \bigcup_{c \in P_i - P} c\}$, we can split the sequence into merges M_1^a, M_2^a, ... that only involve clusters in P and merges M_1^b, M_2^b, ... that only involve clusters in $P_i - P$. We can run the first batch of merges M_1^a, M_2^a, ... to produce a possible result of $E(P, D)$ and run the second batch of merges M_1^b, M_2^b, ... to produce a possible result of $E(P_i - P, D)$. Since HC_{DS} returns a unique solution, both ER results $E(P, D)$ and $E(P_i - P, D)$ are in fact unique and thus are equal to P_o^1 and P_o^2, respectively. The union of P_o^1 and P_o^2 is equivalent to the result of running the merges M_1, M_2, ..., i.e.,

$E(P_i, D)$. Hence, $P_o^1 \cup P_o^2 \in \bar{E}(P_i, D)$. □

HC_{DC} The Complete-link Hierarchical Clustering (HC_{DC}) algorithm [70] is identical to the HC_{DS} algorithm except in how it measures the distance between two clusters. While the HC_{DS} algorithm chooses the smallest possible distance between records within the two clusters, the HC_{DC} algorithm takes the largest possible distance instead. For example, the cluster distance between $\{r_1, r_2\}$ and $\{r_3\}$ is the maximum of $D(r_1, r_3)$ and $D(r_2, r_3)$. We use the same extension used in HC_{DS} to support ranges of values for distances where only the minimum values of each range are compared to other ranges or thresholds.

Proposition 3.3.4 shows that the extended HC_{DC} algorithm is $\mathcal{CF}$ and $\mathcal{OI}$, but not $\mathcal{RM}$ or $\mathcal{GI}$. As a result, the extended HC_{DC} algorithm cannot use Algorithms 7 or 3 for rule evolution.

Proposition 3.3.4. *The extended HC_{DC} algorithm is $\mathcal{CF}$, but not $\mathcal{RM}$, $\mathcal{OI}$, or $\mathcal{GI}$.*

Proof. We prove that the extended HC_{DC} algorithm is $\mathcal{CF}$. Given the four partitions P, P_i, P_o^1, P_o^2, suppose that the four conditions in Definition 3.1.5 are satisfied. That is, $P \subseteq P_i, \forall P_o \in \bar{E}(P_i, D), P_o \leq \{\bigcup_{c \in P} c\, , \bigcup_{c \in P_i - P} c\}$, $P_o^1 \in \bar{E}(P, D)$, and $P_o^2 \in \bar{E}(P_i - P, D)$. We define a cluster merge M as two clusters merging. We also denote the distance of the two clusters merging during M as $M.dist$. Suppose that the sequence of cluster merges M_1^a, M_2^a, ... were used to generate P_o^1 while a sequence of cluster merges M_1^b, M_2^b, ... were used to generate P_o^2. Notice that both sequences are sorted by their distances (i.e., by $M.dist$). We now construct the sequence of merges $M_1, M_2, \ldots$ by merging the two sequences M_1^a, M_2^a, ... and M_1^b, M_2^b, ... sorted by distance. Running the sequence $M_1, M_2, \ldots$ on P_i in fact generates a possible result of $E(P_i, B)$. We sketch a proof by contradiction. Suppose that one of the merges M_i is incorrect, i.e., the two clusters being merged in M_i do not have the minimum distance among all pairwise distances between all clusters. First, we know that none of the cluster pairs that contain records in P have distances less than $M_i.dist$ because all the merges $M_1^a, \ldots, M_j^a$ where $M_j^a.dist < M_i.dist$ and $M_{j+1}^a.dist \geq M_i.dist$ have been performed. For similar reasons, none of the cluster pairs that contain records in $P_i - P$ have distances less than $M_i.dist$ either. Hence, the cluster pair with the minimum distance should be a pair of one cluster that consists of records

in P and another cluster that consists of records in $P_i - P$. However, by merging these two clusters, we are contradicting the second condition of Definition 3.1.5 where none of the records in P are supposed to merge with any records in $P_i - P$. Hence, the merge sequence $M_1, M_2, \ldots$ indeed produces a possible result of $E(P_i, B)$. Since running the sequence $M_1, M_2, \ldots$ produces an equivalent result as running the sequence $M_1^a, M_2^a, \ldots$ and then running the sequence $M_1^b, M_2^b, \ldots$, $P_o^1 \cup P_o^2 \in \bar{E}(P_i, D)$.

We prove that the extended HC_{DC} algorithm does not satisfy $\mathcal{OI}$ and consequently not $\mathcal{RM}$ using a counter example. Suppose that we have the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and a distance match rule where $D(r_1, r_2) = [2]$, $D(r_2, r_3) = [2]$, and $D(r_1, r_3) = [4]$. We are also given the threshold value $T = 3$. If r_1 and r_2 merge first, then the algorithm terminates without more merges because the next minimum distance is 4 (i.e., the distance between $\{r_1, r_2\}$ and $\{r_3\}$), which exceeds T. Similarly, if r_2 and r_3 merge first, then the algorithm terminates without any more merges. Thus, HC_{DC} returns two possible ER results $\{\{r_1, r_2\}, \{r_3\}\}$ or $\{\{r_1\}, \{r_2, r_3\}\}$, violating the $\mathcal{OI}$ property and thus the $\mathcal{RM}$ property.

Finally, we prove that the extended HC_{DC} algorithm is not $\mathcal{GI}$ using a counter example. Suppose that we have the input partition $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and a distance match rule where $D(r_1, r_2) = [2]$, $D(r_2, r_3) = [3]$, and $D(r_1, r_3) = [4]$. We are also given the threshold value $T = 3$. Then the ER result $E(P_i, D)$ is always $\{\{r_1, r_2\}, \{r_3\}\}$ because $\{r_1\}$ and $\{r_2\}$ merge first (having the shortest distance 2), and then no clusters match anymore. Using Definition 3.1.6, suppose we set $P = \{\{r_2\}, \{r_3\}\}$ and $P_i = \{\{r_1\}, \{r_2\}, \{r_3\}\}$. Then $P_o^1 \in \bar{E}(P, D)$ is always $\{\{r_2, r_3\}\}$ because $\{r_2\}$ and $\{r_3\}$ merge together. As a result, $P_o^2 \in \bar{E}(P_o^1 \cup (P_i - P), D)$ is always $\{\{r_1\}, \{r_2, r_3\}\}$ because $\{r_1\}$ does not match with $\{r_2, r_3\}$ (having a distance of 4). Thus, $P_o^2 \notin \bar{E}(P_i, D) = \{\{\{r_1, r_2\}, \{r_3\}\}\}$, violating the $\mathcal{GI}$ property. □

We have now completed the venn diagram in Figure 3.3, which shows the results of Propositions 3.3.2 and 3.3.4.

3.3.3 Rule Evolution

While we used the CNF structures of match rules to perform rule evolution in Section 3.1.4, the distance match rules are not Boolean expressions. Instead, we define a model on how the distance match rule can evolve. We assume that each distance $D_1(r, s)$ changes by at most $f(D_1(r, s))$ where f is a positive function that can be provided by a domain expert who knows how much D_1 can change. Examples of f include a constant value (i.e., each distance can change by at most some constant c) or a certain ratio of the original distance (i.e., each distance can change by at most X percent). As a result, $D_1(r, s).max + f(D_1(r, s)) \geq D_2(r, s).max$ and $D_1(r, s).min - f(D_1(r, s)) \leq D_2(r, s).min$. As a practical example, suppose that D_1 returns the sum of the distances for the names, addresses, and zip codes, and D_2 returns the sum of the distances for the names, addresses, and phone numbers. If we restrict the zip code and phone number distances to be at most 10, then when D_1 evolves to D_2, we can set f = 10. Or if the zip code and phone number distances are always within 20% of the D_1 distance, then f = 0.2×D_1.

Given D_1 and D_2, we can now define a third distance match rule $D_3(r, s)$ = [max $\{D_1(r, s).min - f(D_1(r, s))$, 0}, $D_1(r, s).max + f(D_1(r, s))$], which satisfies $D_3 \geq D_1$ and $D_3 \geq D_2$. (Notice that our definition ensures all the possible distances of D_3 to be non-negative.) Compared to the Boolean clustering model, rule D_3 acts as the "common conjuncts" between D_1 and D_2. As a result, we now materialize the ER result of D_3, $E(P_i$, $D_3)$, instead of the ER results for all the conjuncts in the first match rule. We also update Algorithm 7 in Section 3.1.4 by replacing Step 3 with "Partition $M \leftarrow H(D_3)$" where H is a hash table that only contains the result $E(P_i$, $D_3)$ for the match rule D_3.

Example We illustrate rule evolution for the HC_{DS} algorithm using the updated Algorithm 7. Suppose we are given the input partition P_i = $\{\{r_1\}$, $\{r_2\}$, $\{r_3\}\}$ and the distance match rule D_1 where $D_1(r_1, r_2)$ = [2], $D_1(r_2, r_3)$ = [4], and $D_1(r_1, r_3)$ = [5]. We use the threshold T = 2 for termination. If we are given $f(d) = 0.1 \times d$, D_3 is defined as $D_3(r_1, r_2)$ = [1.8, 2.2], $D_3(r_2, r_3)$ = [3.6, 4.4], and $D_3(r_1, r_3)$ = [4.5, 5.5]. We then materialize the ER result M = $E(P_i$, $D_3)$. Among the records, only r_1 and r_2 match having $D_3(r_1, r_2).min$ = 1.8 $\leq T$ = 2. Once the clusters $\{r_1\}$ and $\{r_2\}$ merge, $\{r_1, r_2\}$ and $\{r_3\}$ do not match because $D_3(r_1$, $r_3).min$ = 4.5 and $D_3(r_2$, $r_3).min$ = 3.6, both exceeding T. Hence M = $\{\{r_1, r_2\}$,

$\{r_3\}\}$. Suppose we are then given D_2 such that $D_2(r_1, r_2)$ = [2.2], $D_2(r_2, r_3)$ = [3.9], and $D_2(r_1, r_3)$ = [4.9] (notice that indeed $D_2 \leq D_3$). We then return $\bigcup_{c \in M} E(\{c' \in P_i | c' \subseteq c\}$, $D_2)$ using the same threshold T = 2. For the first cluster in M, we run $E(\{\{r_1\}, \{r_2\}\}, D_2)$. Since $D_2(r_1, r_2).min$ = 2.2 $> T$, $\{r_1\}$ and $\{r_2\}$ do not merge. The next partition $\{\{r_3\}\}$ is a singleton, so our new ER result is $\{\{r_1\}, \{r_2\}, \{r_3\}\}$, which is identical to $E(P_i, D_2)$.

3.4 Experimental Evaluation

Evaluating rule evolution is challenging since the results depend on many factors including the ER algorithm, the match rules, the rate and type of evolution, and the materialization strategy. Obviously there are many cases where evolution and/or materialization are not effective, so our goal in this section is to show there are realistic cases where they can pay off, and that in some cases the savings over a naïve approach can be significant. The savings can be very important in scenarios where data sets are large and where it is important to obtain a new ER result as quickly as possible (think of national security applications where it is critical to respond to new threats as quickly as possible).

For our evaluation, we assume that blocking is used, as it is in most ER applications with massive data. From our point of view, the use of blocking means that we can read a full block (which can still span many disk blocks) into memory, perform resolution (naïve or evolutionary), and then move on to the next block. In our experiments we thus evaluate the cost of resolving a single block. Keep in mind that these costs should be multiplied by the number of blocks.

There are three metrics that we use to compare ER strategies: CPU, IO and storage costs. (Except for Section 3.4.7, we do not consider accuracy since our evolution techniques do not change the ER result, only the cost of obtaining it.) We discuss CPU and storage costs in the rest of this section, leaving a discussion of IO costs to Section 3.4.2. In general, CPU costs tend to be the most critical due to the quadratic nature of the ER problem, and because matching/distance rules tend to be expensive. In Section 3.4.2 we argue that IO costs do not vary significantly with or without evolution and/or materialization, further justifying our focus here on CPU costs.

We start by describing our experimental setting in Section 3.4.1. Then in Sections 3.4.3

and 3.4.4, we discuss the CPU costs of ER evolution compared to a naïve approach (ignoring materialization costs, if any). In Section 3.4.5 we consider the CPU and space overhead of materializing partitions. Note that we do not discuss the orthogonal problem of *when* to materialize (a problem analogous to selecting what views to materialize). In Section 3.4.6 we discuss total costs, including materialization and evolution.

3.4.1 Experimental Setting

We experiment on a comparison shopping dataset provided by Yahoo! Shopping and a hotel dataset provided by Yahoo! Travel. We evaluated the following ER algorithms: SN, HC_B, HC_{BR}, ME, HC_{DS}, and HC_{DC}. Our algorithms were implemented in Java, and our experiments were run on a 2.4GHz Intel(R) Core 2 processor with 4GB of RAM.

Real Data The comparison shopping dataset we use was provided by Yahoo! Shopping and contains millions of records that arrive on a regular basis from different online stores and must be resolved before they are used to answer customer queries. Each record contains various attributes including the title, price, and category of an item. We experimented on a random subset of 3,000 shopping records that had the string "iPod" in their titles and a random subset of 1 million shopping records. We also experimented on a hotel dataset provided by Yahoo! Travel where tens of thousands of records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users. We experimented on a random subset of 3,000 hotel records located in the United States. While the 3K shopping and hotel datasets fit in memory, the 1 million shopping dataset did not fit in memory and had to be stored on disk.

Match Rules Table 3.1 summarizes the match rules used in our experiments. The *Type* column indicates whether the match rules are Boolean match rules or distance match rules. The *Data* column indicates the data source: shopping or hotel data. The *match rules* column indicates the match rules used. The first two rows define the Boolean match rules used on the shopping and hotel datasets. For the shopping datasets, B_1^S compares the titles and categories of two shopping records while B_2^S compares the titles and prices of shopping records. For the hotel data, B_1^H compares the states, cities, zip codes, and names of two

Table 3.1: Match Rules

Type	Data	Match rules
Boolean	Shopping	$B_1^S : p_{ti} \wedge p_{ca}$ $B_2^S : p_{ti} \wedge p_{pr}$
Boolean	Hotel	$B_1^H : p_{st} \wedge p_{ci} \wedge p_{zi} \wedge p_{na}$ $B_2^H : p_{st} \wedge p_{ci} \wedge p_{zi} \wedge p_{sa}$
Distance	Shopping	$D_1^S : Jaro_{ti}$ $D_2^S : Jaro_{ti}$ changes randomly within 5%
Distance	Hotel	$D_1^H : Jaro_{na} + 0.05 \times Equals_{ci}$ $D_2^H : Jaro_{na} + 0.05 \times Equals_{zi}$

hotel records. The B_2^H rule compares the states, cities, zip codes, and street addresses of two hotel records. The last two rows define the distance match rules for the two datasets. For the shopping data, D_1^S measures the Jaro distance [111] between the titles of two shopping records while D_2^S randomly alters the distance of D_1^S by a maximum ratio of 5%. The Jaro distance returns a value within the range [0, 1], and gives higher values for closer records. For the hotel data, D_1^H sums the Jaro distance between the names of two records and the Equality distance between the cities of two records weighted by 0.05. We define the Equality distance to return 1 if two values are exactly the same and 0 if they are not the same. The D_2^H rule sums the Jaro distance between names with the Equality distance between the zip codes of two records weighted by 0.05. As a result, the D_1^H distance can alter by at most the constant 0.05.

ER and Rule Evolution Algorithms We experiment rule evolution on the following ER algorithms: SN, HC_B, HC_{BR}, ME, HC_{DS}, and with HC_{DC}. (We do *not* experiment with rule evolution using the join ER model, but focus on the clustering ER models.) Table 3.2 summarizes for each ER algorithm which section it was defined in and which rule evolution algorithm is used. The HC_{DS} and HC_{DC} distanced-based clustering algorithms terminate when the minimum distance between clusters is smaller than the threshold 0.95 (recall that *closer* records have *higher* Jaro + Equality distances). Although the ME and HC_{DC} algorithms do not satisfy the $\mathcal{RM}$ property, we can still use Algorithm 7 to efficiently produce new ER results with small loss in accuracy. Notice that, although ME is $\mathcal{GI}$, Algorithm 3 is not efficient because of the way ME extracts all records from the input

Table 3.2: ER and rule evolution algorithms tested

ER algorithm	Section	Rule evolution algorithm used
SN	3.1.4	Algorithm for SN in Section 3.1.4
HC_B	3.1.4	Algorithm 3
HC_{BR}	3.1.4	Algorithm 7
ME	3.1.4	Algorithm 7
HC_{DS}	3.3.2	Algorithm 7 (for distance-based clustering)
HC_{DC}	3.3.2	Algorithm 7 (for distance-based clustering)

partition P_i (without exploiting any of the clusters in P_i) and sorts them again. Both the HC_{DS} and HC_{DC} algorithms use Algorithm 7 adjusted for the distance-based clustering model (see Section 3.3.3).

3.4.2 Evaluating IO costs

We discuss the corresponding IO costs and argue that the materialization IO costs are less significant than the CPU costs. Using our blocking framework, we can analyze the overall runtime of an ER process. The basic operations of an ER process are described in Table 3.3. The operations are categorized depending on whether they are disk IO consuming operations or CPU time consuming operations.

To compare the overall performance of an ER process using rule evolution and a naïve ER process without rule evolution, we consider the scenario where we run ER once using an old match rule and then perform one rule evolution using a new match rule. A naïve ER process without rule evolution would roughly require initializing the records, creating the blocks, and reading and resolving the blocks twice. An ER process using rule evolution on the other hand would require the same process above plus the additional work of creating and using rule materializations minus running ER on all blocks during the rule evolution. The decompositions of the two approaches for our one rule evolution scenario are shown in Table 3.4. Notice that the listed operations are not necessarily run sequentially. For example, for the naïve approach, the R_B and E operations are actually interleaved because each block is read and then resolved before the next block is read.

The IO overhead of using rule evolution compared to the IO cost of the naïve approach

Table 3.3: Basic operations in blocking ER framework

Operation	Description
IO time consuming operations	
R_F	Read records from input file
R_B	Read all blocks to memory
W_B	Write out all blocks to disk
R_M	Read all materializations to memory
W_M	Write all materializations to disk
O	Write the output ER result to disk
CPU time consuming operations	
I	Initialize records (trim attributes not used in rules)
E	Run ER on all blocks (one block at a time)
M	Create materializations for all blocks (one at a time)
V	Run rule evolution (using materializations) on all blocks (one at a time)

Table 3.4: Decomposition of ER processes for one rule evolution

ER process	Decomposition
Naïve	R_F,I,W_B,R_B,E,O,R_B,E,O
Using rule evolution	R_F,I,W_B,R_B,E,O,M,W_M,R_B,R_M,V,O

can thus be written as $\frac{R_M+W_M}{R_F+W_B+2\times R_B+2\times O}$. Since the size of the materializations is usually much smaller than the size of the entire set of records (see Section 3.1.3), the additional IOs for rule evolution is also smaller than the IOs for reading and writing the blocks. Thus, the IO costs do not vary significantly with or without evolution and/or materialization.

3.4.3 Rule Evolution Efficiency

We first focus on the CPU time cost of rule evolution (exclusive of materialization costs, if any) using blocks of data that fit in memory. For each ER algorithm, we use the best evaluation scheme (see Section 3.4.1) given the properties of the ER algorithm. Table 3.5 shows the results. We run the ER algorithms SN, HC_B, and HC_{BR} using the Boolean match rules in Table 3.1 on the shopping and hotel datasets. When evaluating each match rule, the conjuncts involving string comparisons (i.e., p_{ti}, p_{na}, and p_{sa}) are evaluated last because they are more expensive than the rest of the conjuncts. We also run the HC_{DS}

Table 3.5: ER algorithm and rule evolution runtimes

ER algorithm	Sh1K	Sh2K	Sh3K	Ho1K	Ho2K	Ho3K
ER algorithm runtime (seconds)						
SN	0.094	0.152	0.249	0.012	0.027	0.042
HC_B	1.85	7.59	17.43	0.386	2.317	5.933
HC_{BR}	3.56	19.37	48.72	0.322	1.632	4.264
HC_{DS}	8.33	40.38	111	5.482	27.96	73.59
Ratio of ER algorithm runtime to rule evolution runtime						
SN	4.09	4.22	4.45	1.2	1.93	2
HC_B	1.5	1.84	2.07	1.27	1.3	1.27
HC_{BR}	162	807	1218	36	136	237
HC_{DS}	298	708	918	322	499	545

algorithm using the distance match rules in Table 3.1 on the two datasets. Each column head in Table 3.5 encodes the dataset used and the number of records resolved in the block. For example, Sh1K means 1,000 shopping records while Ho3K means 3,000 hotel records. The top five rows of data show the runtime results of the naïve approach while the bottom five rows show the runtime improvements of rule evolution compared to the naïve approach. Each runtime improvement is computed by dividing the naïve approach runtime by the rule evolution runtime. For example, the HC_{BR} algorithm takes 3.56 seconds to run on 1K shopping records and rule evolution is 162 times faster (i.e., having a runtime of $\frac{3.56}{162}$ = 0.022 seconds).

As one can see in Table 3.5, the improvements vary widely but in many cases can be very significant. For the shopping dataset, the HC_{BR}, and HC_{DS} algorithms show up to orders of magnitude of runtime improvements. The SN algorithm has a smaller speedup because SN itself runs efficiently. The HC_B algorithm has the least speedup (although still a speedup). While the rule evolution algorithms for SN, HC_{BR}, and HC_{DS} only need to resolve few clusters at a time (i.e., each $\{c' \in P_i | c' \subseteq c\}$ in Algorithm 7), Algorithm 3 for the HC_B algorithm also needs to run an outermost ER operation (Step 4) to resolve the clusters produced by the inner ER operations. The hotel data results show worse runtime improvements overall because the ER algorithms without rule evolution ran efficiently.

3.4.4 Common Rule Strictness

The key factor of the runtime savings in Section 3.4.3 is the strictness of the "common match rule" between the old and new match rules. For match-based clustering, the common match rule between B_1 and B_2 comprises the common conjuncts $Conj(B_1) \cap Conj(B_2)$. For distance-based clustering, the common match rule between D_1 and D_2 is D_3, as defined in Section 3.3.3. A stricter rule is more selective (fewer records match or fewer records are within the threshold), and leads to smaller clusters in a resolved result. If the common match rule yields smaller clusters, then in many cases the resolution that starts from there will have less work to do.

By changing the thresholds used by the various predicates, we can experiment with different common rule strictness, and Figure 3.4 summarizes some of our findings. The horizontal axis shows the strictness of the common rule: it gives the ratio of record pairs placed by the common rule within in a cluster to the total number of record pairs. For example, if an ER algorithm uses p_{ti} to produce 10 clusters of size 10, then the strictness is $\frac{10\times\binom{10}{2}}{\binom{100}{2}}$ = 0.09. The lower the ratio is, the stricter the common rule, and presumably, fewer records need to be resolved using the new match rule.

The vertical axis in Figure 3.4 shows the runtime improvement (vs. naïve), for four algorithms using our shopping data match rules in Table 3.1. The runtime improvement is computed as the runtime of the naïve approach computing the new ER result divided by the runtime of rule evolution. As expected, Algorithms SN, HC_{BR}, and HC_{DS} achieve significantly higher runtime improvements as the common comparison rule becomes stricter. However, the HC_B algorithm shows a counterintuitive trend (performance decreases as strictness increases). In this case there are two competing factors. On one hand, having a stricter common match rule improves runtime for rule evolution because the computation of each $E(\{c' \in P_i | c' \subseteq c\}, B_2)$ in Step 4 becomes more efficient. On the other hand, a common comparison rule that is too strict produces many clusters to resolve for the outermost ER operation in Step 4, increasing the overall runtime. Hence, although not shown in the plot, the increasing line will eventually start decreasing as strictness decreases.

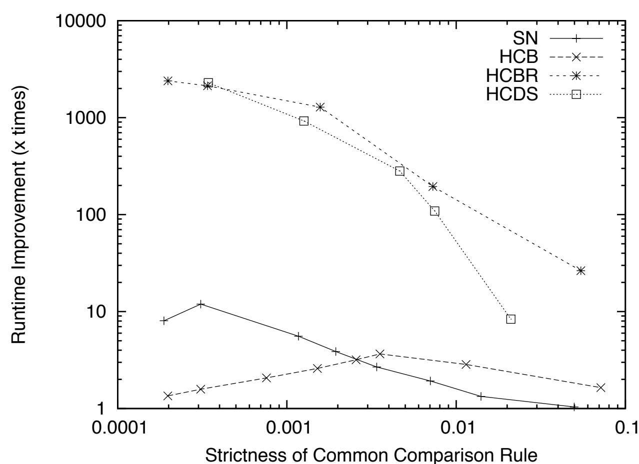


Figure 3.4: Degree of change impact on runtime, 3K shopping records

3.4.5 Materialization Overhead

In this section we examine the CPU and space overhead of materializations, independent of the question of what conjuncts should be materialized. Recall that materializations are done as we perform the initial resolution on records S. Thus the materialization can piggyback on the ER work that needs to be done anyway. For example, the parsing and initialization of records can be done once for the entire process of creating all materializations and running ER for the old match rule. In addition, there are other ways to amortize work, as the resolution is concurrently done for the old rule and the conjuncts we want to materialize. For example, when materializing for the SN and ME algorithms, the sorting of records is only done once. For the HC_B and HC_{BR} algorithms, we cache the merge information of records. For the HC_{DS} and HC_{DC} algorithms, the pairwise distances between records are only computed once. We can also compress the storage space needed by materializations by storing partitions of record IDs.

Table 3.6 shows the time and space overhead of materialization in several representative scenarios. In particular, we use Algorithms SN, HC_B, HC_{BR}, and HC_{DS} on 3K shopping and hotel records, and assume *all* conjuncts in the old rule are materialized.

The *Time O/H* columns show the time overhead where each number is produced by dividing the materialization CPU time by the CPU runtime for producing the old ER result. For example, materialization time for the SN algorithm on 3K shopping records is 0.52x

Table 3.6: Time overhead (ratio to old ER algorithm runtime) and space overhead (ratio to old ER result) of rule materialization, 3K records

ER algorithm	Sho3K		Ho3K	
	Time O/H	Space O/H	Time O/H	Space O/H
SN	0.52 (0.02)	0.28	1.14 (0.27)	0.14
HC_B	0.87 (0.04)	0.14	3.18 (0.71)	0.1
HC_{BR}	11 (3E-6)	0.14	13.28 (1.06)	0.1
HC_{DS}	0.44	0.07	0.61	0.02

the time for running $E(P_i,B_1^S)$ using SN. Hence, the total time to compute $E(P_i, B_1^S)$ and materialize all the conjuncts of B_1^S is 1+0.52 = 1.52 times the runtime for $E(P_i, B_1^S)$ only. The numbers in parentheses show the time overhead when we do *not* materialize the most expensive conjunct. That is, for SN, HC_B, and HC_{BR} in the shopping column we only materialize p_{ca}; in the hotel column, we only materialize p_{st}, p_{ci}, and p_{zi} (without p_{na}).

For the shopping dataset, the SN and HC_B algorithms have time overheads less than 2 (i.e., the number of conjuncts in B_1^S) due to amortization. For the same reason, HC_{DS} has a time overhead below 1. The HC_{BR} algorithm has a large overhead of 11x because each common conjunct tends to produce larger clusters compared to $E(P_i, B_1^H)$, and HC_{BR} ran slowly when larger clusters were compared using the expensive p_{ti} conjunct.

The hotel dataset shows similar time overhead results, except that the time overheads usually do not exceed 4 (i.e., the number of conjuncts in B_1^H) for the match-based clustering algorithms.

The *Space O/H* columns show the space overhead of materialization where each number was produced by dividing the memory space needed for storing the materialization by the memory space needed for storing the old ER result. For example, the materialization space for the SN algorithm on 3K shopping records is 0.28x the memory space taken by $E(P_i, B_1^S)$ using SN. The total required space is thus 1+0.28 = 1.28 times the memory space needed for $E(P_i, B_1^S)$. The space overhead of materialization is small in general because we only store records by their IDs.

3.4.6 Total Runtime

The speedups achievable at evolution time must be balanced against the cost of materializations during earlier resolutions. The materialization cost of course depends on what is materialized: If we do not materialize any conjuncts, as in our initial example in Section 1, then clearly there is no overhead. At the other extreme, if the initial rule B_1 has many conjuncts and we materialize all of them, the materialization cost will be higher. If we have application knowledge and know what conjuncts are "stable" and likely to be used in future rules, then we can only materialize those. Then there is also the amortization factor: if a materialization can be used many times (e.g., if we want to explore many new rules that share the materialized conjunct), then the materialization cost, even if high, can be amortized over all the future resolutions.

We study the total run time (CPU and IO time for original resolution plus materializations plus evolution) for several scenarios. We experiment on 0.25 to 1 million shopping records (multiple blocks are processed). Our results illustrate scenarios where materialization does pay off. That is, materialization and evolution lowers the total time, as compared to the naïve approach that runs ER from scratch each time. Of course, one can also construct scenarios where materialization does not pay off.

We measure the total runtimes of ER processes as defined in Section 3.4.2 where we run ER once using an old match rule and then perform one rule evolution using a new match rule. We experimented on 0.25 to 1 million random shopping records and used the following Boolean match rules for the SN, HC_B, and HC_{BR} algorithms: $B_1 = p_{ca} \wedge p_{ti}$ (same as B_1^S in Table 3.1) and $B_2 = p_{ca} \wedge p_{pr}$. In addition, we only materialized on the conjunct p_{ca} instead of on both conjuncts in B_1. The time overheads for materializing p_{ca} were shown in parentheses in Figure 3.6. For the HC_{DS} algorithm, we used D_1^S and D_2^S in Table 3.1. We used minhash signatures [57] for distributing the records into blocks. For the shopping dataset, we extracted 3-grams from the titles of records. We then generated a minhash signature for each records, which is an array of integers where each integer is generated by applying a random hash function to the 3-gram set of the record.

Figure 3.5 shows our total time results where we measured the total runtimes of running

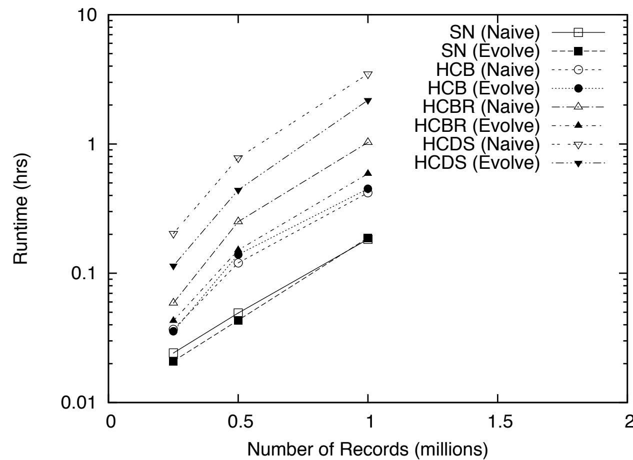


Figure 3.5: Scalability, 1M shopping records

ER on B_1 and then evolving once to B_2. Each rule evolution technique and its corresponding naïve approach use the same shape for points in their plots. For example, the rule evolution runtime plot for the SN algorithm uses *white square* points while the naïve SN approach uses *black square* points. In addition, all the naïve approach plots use white shapes while the rule evolution plots use black shapes. Our results show that the total runtimes for the SN and HC_B algorithms do not change much because the runtime benefits of using rule evolution more or less cancels out the runtime overheads of using rule evolution. For the HC_{BR} and HC_{DS} algorithms, however, the runtime benefits of rule evolution clearly exceed the overheads. While we have shown the worst case scenario results where only one evolution occurs, the improvements will most likely increase for multiple rule evolutions using the same materializations.

3.4.7 Without the Properties

So far, we have only studied scenarios where one or more of the properties needed for our rule evolution techniques held. We now consider a scenario where the necessary properties do not hold. In this case, we need to use the naïve approach to get a correct answer. From our previous results, however, we know that the naïve approach can be very expensive compared to rule evolution. The alternatives are to fix the ER algorithm to satisfy one of the properties or to run one of our rule evolution algorithms even though we will not get

correct answers. We investigate the latter case and see if we can still return ER results with minimum loss in accuracy.

We experiment on two ER algorithms that do not satisfy the $\mathcal{RM}$ property and thus cannot use Algorithm 7: the ME and HC_{DC} algorithms. While the ME algorithm is still $\mathcal{GI}$ and can thus use Algorithm 3, there is no runtime benefit because in ME all the records in P_i are extracted and sorted again regardless of the clusters in P_i.

To measure accuracy, we compare a rule evolution algorithm result with the corresponding result of the naïve approach. We consider all the records that merged into an output cluster to be identical to each other. For instance, if the clusters $\{r\}$ and $\{s\}$ merged into $\{r, s\}$ and then merged with $\{t\}$ into $\{r, s, t\}$, all three records r, s, t are considered to be the same. Suppose that the correct answer A contains the set of record pairs that match for the naïve solution while set B contains the matching pairs for the rule evolution algorithm. Then the precision Pr is $\frac{|A \cap B|}{|B|}$ while the recall Re is $\frac{|A \cap B|}{|A|}$. Using Pr and Re, we compute the F_1-measure, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, and use it as our accuracy metric.

Table 3.7 shows the runtime and accuracy results of running Algorithm 7 as the rule evolution algorithm on datasets that fit in memory. The columns show the dataset used and the number of records resolved. The top two rows of data show the runtimes for the naïve approach. The middle two rows of data show the runtime improvements of rule evolution compared to the naïve approaches. Each runtime improvement is computed by dividing the naïve approach runtime by the rule evolution runtime (not including the materialization costs). Overall, the runtime of ME improves by 1.67x to 5.53x while the runtime of HC_{DC} improves by 501x to 2386x. The bottom two rows of data show the accuracy values of each ER result compared to the correct result produced by the naïve approach. The accuracy results are near-perfect for the ME algorithm while being at least 0.85 for HC_{DC}. The experiments show that rule evolution may produce highly-accurate ER results even if the ER algorithms do not satisfy any property while still significantly enhancing the runtime performance of rule evolution.

Table 3.7: Runtime and accuracy results for ER algorithms without the properties

ER algorithm	Sh1K	Sh2K	Sh3K	Ho1K	Ho2K	Ho3K
ER algorithm runtime (seconds)						
ME	0.094	0.162	0.25	0.015	0.033	0.051
HC_{DC}	8.08	39.2	105	5.51	28.1	73.57
Ratio of ER algorithm runtime to rule evolution time						
ME	5.53	5.23	5.43	1.67	2.06	2.04
HC_{DC}	674	1509	2386	501	879	1115
F_1 accuracy of rule evolution						
ME	0.94	0.95	0.97	1.0	1.0	0.997
HC_{DC}	0.93	0.86	0.85	1.0	0.999	0.999

3.5 Related Work

Entity resolution involves comparing records and determining if they refer to the same entity or not. Most of the work falls into one of the ER models we consider: match-based clustering [54, 10] and distance-based clustering [12, 70]. While the ER literature focuses on improving the accuracy or runtime performance of ER, they usually assume a fixed logic for resolving records. To the best of our knowledge, our work is the first to consider the ER result update problem when the logic for resolution itself changes.

One of the recent challenges in information integration research is called Holistic Information Integration [49] where both schema and data issues are addressed within a single integration framework. For example, schema mapping can help with understanding the data and thus with ER while ER could also provide valuable information for schema mapping. Hence, schema mapping and ER can mutually benefit each other in an iterative fashion. While our work does not address the schema mapping problem, we provide a framework for iteratively updating ER results when the comparison logic (related to the schema) changes.

Another related problem is updating clustering results when the records (data) change. Some work explores the problem of clustering data streams. Charikar et al. [19] propose incremental clustering algorithms that minimize the maximum cluster diameter given a stream of records. Aggarwal et al. [1] propose the CluStream algorithm, which views a

stream as a changing process over time and provides clustering over different time horizons in an evolving environment. An interesting avenue of further research is to combine clustering techniques for both evolving data and rules. Since our rule evolution techniques are based on materializing ER results, we suspect that the same techniques for evolving data can be applied on the materialized ER results.

Materializing ER results is related to the topic of query optimization using materialized views, which has been studied extensively in the database literature [22, 44]. The focus of materialized views, however, is on optimizing the execution of SQL queries. In comparison, our work solves a similar problem for match rules that are Boolean or distance functions. Our work is also related to constructing data cubes [51] in data warehouses where each cell of a data cube is a view consisting of an aggregation (e.g., sum, average, count) of interests like total sales. In comparison, rule evolution stores the ER results of match rules. Nonetheless, we believe our rule evolution techniques can improve by using techniques from the literature above. For example, deciding which combinations of conjuncts to materialize is related to the problem of deciding which views to materialize.

3.6 Conclusion

In most ER scenarios, the logic for matching records evolves over time, as the application itself evolves and as the expertise for comparing records improves. In this chapter we have explored a fundamental question: when and how can we base a resolution on a previous result as opposed to starting from scratch? We have answered this question in two commonly-used contexts, record comparisons based on Boolean predicates and record comparisons based on distance (or similarity) functions. We identified two properties of ER algorithms, rule monotonic and context free (in addition to order independent and general incremental), that can significantly reduce runtime at evolution time. We also categorized several popular ER algorithms according to the four properties.

In some cases, computing an ER result with a new rule can be much faster if certain partial results are materialized when the original ER result (with the old rule) is computed. We studied how to take advantage of such materializations, and how they could be computed efficiently by piggybacking the work on the original ER computation.

Our experimental results evaluated the cost of both materializations and the evolution itself (computing the new ER result), as compared to a naïve approach that computed the new result from scratch. We considered a variety of popular ER algorithms (each having different properties), two data sets, and different predicate strictness. The results illustrate realistic cases where materialization costs are relatively low, and evolution can be done extremely quickly.

Overall, we believe our analysis and experiments provides guidance for the ER algorithm designer. The experimental results show the potential gains, and if these gains are attractive in an application scenario, our properties help us design algorithms that can achieve such gains.

Chapter 4

Joint Entity Resolution

In this chapter, we focus on the problem of joint ER where multiple datasets of different entity types are resolved together. Addressing a setting in which relationships between the datasets exist, the result of resolving one dataset may benefit the resolution of another dataset. For example, the fact that two (apparently different) authors a_1 and a_2 have published exactly the same papers could be strong evidence that the two authors are in fact the same. Also, by identifying the two authors to be the same, we can further deduce that two papers p_1 and p_2 written by a_1 and a_2, respectively, are more likely to be the same paper as well. This reasoning can easily be extended to more than two datasets. For example, resolving p_1 and p_2 can now help us resolve the two corresponding venues v_1 and v_2. Compared to resolving each dataset separately, joint ER can achieve better accuracy by exploiting the relationships between datasets.

Among the existing papers on joint ER [31, 15, 27, 81, 90, 7], few have focused on scalability, which is crucial in resolving large data (e.g., hundreds of millions of people records crawled from the Web). The solutions that do address scalability propose custom joint ER algorithms for efficiently resolving records. However, given that there exist ER algorithms that are optimized for specific types of records (e.g., there could be an ER algorithm that specializes in resolving authors only and another ER algorithm that is good at resolving venues), replacing all the ER algorithms with a single joint ER algorithm that customizes to all types of records may be challenging for the application developer. Instead, we propose a flexible framework for joint ER where one can simply "plug in" existing ER

algorithms, and our framework can then schedule and resolve the datasets individually using the given ER algorithms.

While many previous joint ER approaches assume that all the datasets are resolved at the same time in memory using one processor, our framework allows efficient resource management by resolving a few datasets at a time in memory using multiple processors. Our framework extends blocking techniques. In addition, our framework resolves multiples types of data and divides the resolution based on the data type. Our approach may especially be useful when there are many large datasets that cannot be resolved altogether in memory. Thus one of the key challenges is determining a good sequence for resolution. For instance, should we resolve all venues first, and then all papers and then all authors? Or should we consider a different order? Or should we resolve some venues first, then some related papers and authors, and then return to resolve more venues? And we may also have to resolve a type of record multiple times, since subsequent resolutions may impact the work we did initially.

As a motivating example, consider two datasets P and V (see Table 4.1) that contain paper records and venue records, respectively. (Note that Table 4.1 is a simple example used for illustration. In practice, the datasets can be much larger and more complex.) The P dataset has the attributes Title and Venue while V has the attributes Name and Papers. For example, P contains record p_1 that has the title "The Theory of Joins in Relational Databases" presented at the venue v_1. The Papers field of a V record contains a set of paper records because one venue typically has more than one paper presented.

Suppose that two papers are considered the same and are clustered if their titles and venues are similar while two venues are clustered if their names and papers are similar. Say that we resolve the paper records first. Since p_1 and p_3 have the exact same title, p_1 and p_3 are considered the same paper. We then resolve the venue records. Since the names of v_1 and v_2 are significantly different, they cannot match based on name similarity alone. Luckily, using the information that p_1 and p_3 are the same paper, we can infer that v_1 and v_2 are most likely the same venue. (In fact "ACM TODS" and "ACM Trans. Database Syst." stand for the same journal.) We can then re-resolve the papers in case there are newly matching records. This time, however, none of the papers match because of their different titles. Hence, we have arrived at a joint ER result where P was resolved into the partition

Paper	Title	Venue
p_1	The Theory of Joins in Relational Databases	v_1
p_2	Efficient Optimization of a Class of Relational ...	v_1
p_3	The Theory of Joins in Relational Databases	v_2
p_4	Optimizing Joins in a Map-Reduce Environment	v_3

Venue	Name	Papers
v_1	ACM TODS	$\{p_1, p_2\}$
v_2	ACM Trans. Database Syst.	$\{p_3\}$
v_3	EDBT	$\{p_4\}$

Table 4.1: Papers and Venues

$\{\{p_1, p_3\}, \{p_2\}, \{p_4\}\}$ while V was resolved into $\{\{v_1, v_2\}, \{v_3\}\}$. Notice that we have followed the sequence of resolving papers, venues, then papers again.

Given enough resources, we can improve the runtime performance by exploiting parallelism and minimizing unnecessary record comparisons. For example, if we have two processors, then we can resolve the papers and venues concurrently. As a result, p_1 and p_2 match with each other. After the papers and venues are resolved, we resolve the venues again, but only perform the incremental work. In our example, since p_1 and p_2 matched in the previous step, and p_1 was published in the venue v_1 while p_2 was published in the venue v_2, we only need to check if v_1 and v_2 are the same venue and can skip any comparison involving v_3. Notice that the papers do not have to be resolved at the same time because none of the venues merged in the previous step. However, after v_1 and v_2 are identified as the same venue, we resolve the papers once more. Again, we only perform the incremental work necessary where we resolve the three records p_1, p_2, and p_3 (because v_1 and v_2 matched in the previous step), but not p_4. In total, we have concurrently resolved the papers and venues, then incrementally resolved the venues, and then incrementally resolved the papers. If the incremental work is much smaller than resolving a dataset from the beginning, the total runtime may improve.

In the case where the same dataset is resolved multiple times, an interesting question to ask is whether we should use the exact same ER algorithm for resolving that dataset again. For example, the paper dataset above was resolved twice: once before resolving the venues and once after. Given that the venues are resolved, we may want to "re-train" the

ER algorithm for that context using machine learning techniques. For instance, we may want to weight venue similarity more (relative to paper title similarity) now that venues have been resolved. Thus, in this chapter we propose a *state-based* training method that trains an ER algorithm based on the current state of the other datasets being resolved.

In summary, we make the following contributions:

- We present a modular joint ER framework, where existing ER algorithms, tuned to a particular type of records, can be effectively used. We define the *physical executions* of multiple ER algorithms that produce joint ER results (Section 4.1).
- We introduce the concept of a scheduler, whose output (a *logical execution plan*) specifies the order for datasets to be resolved in order to produce "correct" joint ER results (Section 4.2).
- We show how the joint ER processor uses an execution plan to physically execute the ER algorithms and return a joint ER result while satisfying resource constraints (Section 4.3).
- We propose a state-based training method that uses the current state of other datasets for fine tuning joint ER algorithms (Section 4.4).
- We experiment on both synthetic and real data to demonstrate the behavior and scalability of joint ER. We then use real data to explore the accuracy of state-based training (Section 4.5).

4.1 Framework

In this section, we define the framework for joint entity resolution. Figure 4.1 shows the overall architecture of our system. Given an influence graph (defined in Section 4.2.1), a scheduler constructs a logical "execution plan," which specifies a high-level order for resolving datasets using the ER algorithms. Next, given the physical resource constraints (e.g., memory size and number of CPU cores), the joint ER processor uses the execution plan to "physically execute" the ER algorithms on the given datasets using the available resources to produce a joint ER result.

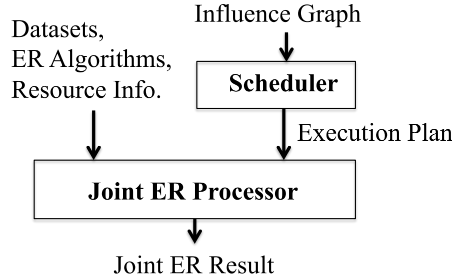


Figure 4.1: System Architecture

In the following sections, we define our ER model for resolving multiple types of data. We then formalize joint ER using the ER algorithms. We then formalize physical executions of ER algorithms that can return joint ER results.

4.1.1 ER Model

Each *record* r is of one entity type. For example, a record can represent a paper or a venue. A *dataset* R contains records of the same type. Thus we may have a dataset for papers and a dataset for venues. There may also be more than one dataset containing records of the same entity type. For example, we could have two datasets containing paper records. As a result, we also allow one dataset to be split into multiple datasets. For instance, a common technique to resolve a large set of records R is to split it into smaller sets or blocks. In this case, each block is viewed as one dataset. The entire collection of datasets is denoted as the set $\mathcal{D}$. We also assume that a record of one entity type may *refer to* another record of a different type. For example, if a paper p was published in a venue v, then p may refer to v by containing a pointer to v. (Further details on references can be found in Section 4.3.4.) We define a cluster of records c to be a set of records. A partition of the dataset R is defined as a set of clusters $P = \{c_1, \ldots, c_m\}$ where the following properties hold: $c_1 \cup \ldots \cup c_m = R$ and $\forall c_i, c_j \in P$ where $i \neq j$, $c_i \cap c_j = \emptyset$.

An ER algorithm E_R resolves a single dataset R. Given a dataset $R = \{r_1, \ldots, r_n\}$, E_R receives as input a partition P_R of R and returns another partition P'_R of R. A partition is a general way to represent the output of an ER algorithm because the clusters provide the lineage information on which records end up representing the same entity. In addition, one

could optionally merge the records within the same cluster to produce composite records. We also require the input of ER to be a partition of R so that we may also run ER on the output of a previous ER result. In our motivating example in the beginning of this chapter, the input for papers was a set of records $R = \{p_1, p_2, p_3\}$, which can be viewed as a partition of singletons $P_R = \{\{p_1\}, \{p_2\}, \{p_3\}\}$, and the output was the partition $P'_R = \{\{p_1, p_3\},\{p_2\}\}$. We say that an ER algorithm *resolves* R if we run ER on a partition of R to produce another partition of R.

During the resolution of R, E_R may use the information of records in other datasets that the records in R refer to. For example, if paper records refer to author records, we may want to use the information on which authors are the same when resolving the papers. To make the potential dependence on other data sets explicit, we write the invocation of E_R on P_R as $E_R(P_R, \mathcal{P}_{-R})$ where $\mathcal{P}_{-R} = \{P_Y | Y \in \mathcal{D} - \{R\}\}$ represents the partitions of the remaining datasets. For example, if $\mathcal{D} = \{R, S\}$, and the partitions of R and S are P_R and P_S, respectively, then $\mathcal{P}_{-R} = \{P_S\}$ and running E_R on P_R returns the result of $E_R(P_R, \mathcal{P}_{-R})$. Notice that the records in R may not refer to records in all the datasets in $\mathcal{P}_{-R}$.

We now define a valid ER algorithm that resolves a dataset.

Definition 4.1.1. *Given any input partition P_R of a set of records R, a* valid *ER algorithm E_R returns an ER result $P'_R = E_R(P_R, \mathcal{P}_{-R})$ that satisfies the two conditions:*

1. *P'_R is a partition of R*
2. *$E_R(P'_R, \mathcal{P}_{-R}) = P'_R$*

The first condition says that E_R returns a partition P'_R of R. The second condition requires that the output is a fixed point result where applying E_R on P'_R will not change the result further unless the partitions in $\mathcal{P}_{-R}$ change. For example, say that there are two datasets $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2\}$ where $P_R = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and $P_S = \{\{s_1\}, \{s_2\}\}$. Say that E_R is a valid algorithm where $E_R(P_R, \{P_S\}) = \{\{r_1, r_2\}, \{r_3\}\} = P'_R$. Then we know by the second condition of Definition 4.1.1 that $E_R(P'_R, \{P_S\})$ returns P'_R as well. Notice that a valid ER algorithm is not required to be deterministic, and the ER result does not have to be unique.

4.1.2 Joint ER Model

Using valid ER algorithms, we can define a joint ER result on all the datasets $\mathcal{D}$ as follows.

Definition 4.1.2. *A* joint ER *result on $\mathcal{D}$ is the set of partitions $\{P_R | R \in \mathcal{D}$, P_R is a partition of R, and $E_R(P_R, \mathcal{P}_{-R}) = P_R\}$.*

A joint ER result is thus a fixed point: running any additional ER on any dataset does not change the results. Continuing our example from Definition 4.1.1, say we have the partitions $P'_R = \{\{r_1, r_2\}, \{r_3\}\}$ and $P'_S = \{\{s_1, s_2\}\}$. In addition, say that $E_R(P'_R, \{P'_S\})$ = P'_R, and $E_R(P'_S, \{P'_R\}) = P'_S$. Then according to Definition 4.1.2, $\{P'_R, P'_S\}$ is a joint ER result of R and S. Notice that a joint ER result is not necessarily unique because even a single dataset may have multiple ER results satisfying Definition 4.1.1.

4.1.3 Physical Execution

In our framework, we assume the datasets are resolved in parallel in synchronous steps. At each step, some datasets are resolved using valid ER algorithms while other datasets are left unchanged. As we will see in the example below, a resolution in a given step refers to the previous state of the datasets.

For example, Figure 4.2 pictorially shows a physical execution of three datasets R, S, and T where we first resolve R and S concurrently then S and T sequentially. (For now ignore the (i, j) annotations.) At Step 0, no resolution occurs, but each dataset $X \in \mathcal{D}$ is initialized as the partition $P^0_X = \{\{r\} | r \in X\}$. In our example, we initialize P^0_R, P^0_S, and P^0_T. After n steps of synchronous transitions, a partition of dataset X is denoted as P^n_X. We denote the entire set of partitions after Step n except for P^n_R as $\mathcal{P}^n_{-R} = \{P^n_Y | Y \in \mathcal{D} - \{R\}\}$ (e.g., $\mathcal{P}^1_{-R}$ in Figure 4.2 is $\{P^1_S, P^1_T\}$). There are two options for advancing each partition P^n_X into its next step partition P^{n+1}_X. First, we can simply set P^{n+1}_X to P^n_X without change, which we pictorially express as a double line from P^n_X to P^{n+1}_X. For example, we do not run ER on P^0_T during Step 1 and thus draw a double line from P^0_T to P^1_T below. Second, we can run ER on P^n_X using the information in the other partitions $\mathcal{P}^n_{-X}$. We pictorially express the flow of information as arrows from P^n_X and the partitions in $\mathcal{P}^n_{-X}$ to P^{n+1}_X. For instance, producing P^2_S using ER may require the information of all the previous-step partitions, so

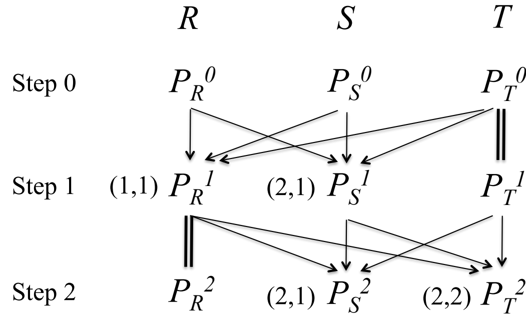


Figure 4.2: The Physical Execution (((R), (S)),((),(S, T)))

we draw three arrows from P_R^1, P_S^1, and P_T^1 to P_S^2. Notice that we do not allow ER to use the information of partitions in more than 1 step behind because in general it is more helpful to use the most recent partition information possible.

The datasets resolved in the same step can be resolved in parallel by several machines. After each step, a synchronization occurs where datasets are re-distributed to different machines for the next step of processing. (In Section 4.3, we elaborate on how datasets are distributed to machines.) We call this step-wise sequence of resolutions a *physical execution*. For each dataset R being resolved, we also specify the machine number m resolving R and the execution order o of R by m during the current step as (m, o). For example, in Step 2, the dataset S is the 1st dataset to be resolved by machine 2.

The physical execution information can be compactly expressed as a nested list of three levels where the outer-most level specifies the steps, the middle level the machine order, and the inner-most level the order of datasets resolved within a single machine. Our physical execution can thus be represented as $(((R),(S)),((),(S,T)))$. Given a set of initial partitions $\{P_X^0|X \in \mathcal{D}\}$, a physical execution $\mathcal{T}$ produces the partitions $\{P_X^{|\mathcal{T}|}|X \in \mathcal{D}\}$ where $|\mathcal{T}|$ is the number of synchronous steps within $\mathcal{T}$. Throughout the chapter, we will omit the step numbers of partitions if the context is clear. Since a physical execution is a sequence of datasets to resolve, we can concatenate two physical executions $\mathcal{T}_1$ and $\mathcal{T}_2$ into one execution $\mathcal{T}_1 + \mathcal{T}_2$. For example, concatenating two physical executions $(((R)),((S),(T)))$ and $(((U)))$ becomes $(((R)),((S),(T)),((U)))$. We can access each synchronous execution within a physical execution using a list index notation. For instance, the first synchronous execution of $\mathcal{T}$ = $(((R)),((S),(T)))$ is $\mathcal{T}[1]$ = $((R))$.

Validity A correct physical execution should produce a joint ER result satisfying Definition 4.1.2. We capture this desirable property in the following definition.

Definition 4.1.3. *A* valid *physical execution $\mathcal{T}$ of the initial partitions of $\mathcal{D}$ returns a set of partitions $\{P_R|R \in \mathcal{D}\}$ that is the same as the partitions produced by running the physical execution $\mathcal{T}$ + $(((S)))$ for any $S \in \mathcal{D}$.*

Intuitively, resolving datasets after running a valid physical execution plan does not change the final result. In our motivating example in the beginning of this chapter, the physical execution $(((P)), ((V)), ((P)))$ is valid because running ER on the final partitions of P or V no longer generates new clusterings of records, so running the physical execution $(((P)), ((V)), ((P)))$ and then either $(((P)))$ or $(((V)))$ produces the same partitions as well. However, the physical execution $(((V)), ((P)))$ is not valid because we fail to cluster the venues v_1 and v_2 together, and resolving V again after the physical execution results in v_1 and v_2 clustering (i.e., executing $(((V)), ((P)))$ + $(((V)))$ = $(((V)), ((P)), ((V)))$ will produce a different result than executing $(((V)), ((P)))$).

We now prove that running valid ER algorithms using a valid physical execution returns a correct joint ER result.

Proposition 4.1.4. *Running valid ER algorithms on a valid physical execution of $\mathcal{D}$ returns a joint ER result of $\mathcal{D}$ satisfying Definition 4.1.2.*

Proof. We show that the result of running a valid physical execution $\mathcal{T}$ results in a set of partitions $\{P_R|R \in \mathcal{D}, P_R$ is a partition of $R, E_R(P_R, \mathcal{P}_{-R}) = P_R\}$. Since we run $\mathcal{T}$ on the initial partitions of the datasets in $\mathcal{D}$ using valid ER algorithms, we return a set of partitions of $\mathcal{D}$. We also know that running E_R on each final partition P_R of dataset R results in P_R because we know that running $\mathcal{T}$ produces the same result as running $\mathcal{T}$ + $(((R)))$ by the condition in Definition 4.1.3. □

Notice that while all valid physical executions return correct joint ER results, the converse is not true. That is, not all joint ER results that satisfy Definition 4.1.2 can be produced by valid physical executions. The reason is that Definition 4.1.2 does not require a step-wise execution of ER algorithms on the datasets for producing the joint ER result. For example, suppose that we only have one dataset $R = \{r, r'\}$ and an ER algorithm E_R where

$E_R(\{\{r\},\{r'\}\}) = \{\{r\},\{r'\}\}$ and $E_R(\{\{r,r'\}\}) = \{\{r,r'\}\}$. Then a physical execution can only produce the joint ER result $\{\{\{r\},\{r'\}\}\}$ by running ER on the initial partition of R. However, there is another correct joint ER result $\{\{\{r,r'\}\}\}$, which cannot be produced by running E_R. Since our joint ER results are based on running ER algorithms on datasets, however, our desirable outcome is a valid physical execution that leads to a joint ER result satisfying Definition 4.1.2.

Feasibility There may be a limit to the available resources for joint ER. For example, there may be a fixed amount of memory we can use at any point. Or there may be a fixed number of CPU cores we can use at the same time. In this chapter, we only restrict the number of processors and define a physical plan to be feasible if it can be executed using the given number of processors.

Definition 4.1.5. *A* feasible *physical execution $\mathcal{T}$ satisfies the following condition.*

- $\forall i \in \{1,\ldots,|\mathcal{T}|\}, |\mathcal{T}[i]| \leq$ *number of processors*

We denote the estimated running time of E_R on R as $t(E_R,R)$. For example, if E_R has a runtime quadratic to the size of its input and $|R|$ = 1000, then we can estimate the runtime $t(E_R,R)$ as 10^6. The total runtime of a physical execution $\mathcal{T}$ is then $\Sigma_{i=1\ldots|\mathcal{T}|}$ max$\{\Sigma_{R\in C} t(E_R,R) | C \in \mathcal{T}[i]\}$. For example, suppose we have the physical execution $\mathcal{T}$ = $(((R,S),(T)),((U)))$ and running E_R on R and E_S on S both take 3 hours, running E_T on T takes 5 hours, and running E_U on U takes 1 hour. Then the estimated total runtime is max$\{3+3,5\}$+max$\{1\}$ = 7 hours.

Our goal is to produce a valid and feasible physical execution that minimizes the total runtime. This problem can be proved to be NP-hard [40], so evaluating every possible physical execution may not be acceptable for resolving a large number of datasets. Hence in the following sections, we provide a step-wise approach where we first produce an “execution plan” that represents a class of valid physical executions (Section 4.2) and then produce feasible and efficient physical executions based on the execution plan (Section 4.3).

4.2 Scheduler

The scheduler receives an "influence graph" that captures the relationships among datasets and produces a logical execution plan using the influence graph. We identify which logical execution plans are correct in the sense that they can be used to produce valid physical executions. We propose an algorithm that generates efficient execution plans that satisfy desirable properties and are thus likely to result in fast physical executions. We also discuss how to construct influence graphs when blocking techniques are used.

4.2.1 Influence Graph

An influence graph G of the datasets $\mathcal{D}$ is generated by the application developer and captures the semantic relationships between the datasets. The vertices in G are exactly the datasets in $\mathcal{D}$, and there is an edge from dataset R to S if R "influences" S. We define the influence relationship between two datasets as follows.

Definition 4.2.1. *A dataset R* influences *another dataset S (denoted as $R \rightarrow S$) if there exist partitions P_R of R and P_S of S such that the physical execution $(((R)), ((S)))$ applied to P_R and P_S may give a different result than when $(((S)))$ is applied.*

In our motivating example in the beginning of this chapter, the dataset P of papers influences the dataset V of venues because of the following observations. First, clustering the two papers $\{p_1\}$ and $\{p_3\}$ in P_P^1 resulted in the two venues $\{v_1\}$ and $\{v_2\}$ in P_V^2 clustering as well. Thus the entire physical execution $(((P)), ((V)))$ on the initial partitions P_P^0 and P_V^0 produces the partitions $P_P^2 = \{\{p_1, p_3\}, \{p_2\}\}$ and $P_V^2 = \{\{v_1, v_2\}\}$. On the other hand, if $\{p_1\}$ and $\{p_3\}$ had not been clustered, then $\{v_1\}$ and $\{v_2\}$ would not have clustered because the two venues have a low string similarity for names. So applying the physical execution $(((V)))$ on P_P^0 and P_V^0 produces the partitions $P_P^1 = \{\{p_1, p_3\}, \{p_2\}\}$ and $P_V^1 = \{\{v_1\}, \{v_2\}\}$. Hence by Definition 4.2.1, P influences V. An influence graph could possibly be generated automatically based on the ER algorithms. For example, the scheduler may consider R to influence S if the code in E_R for comparing two records r and r' in R also compares the S records that r and r' refer to.

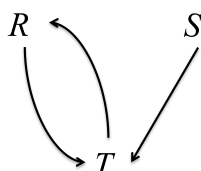


Figure 4.3: An Influence Graph

The influence relationships among multiple datasets can be expressed as a graph. For example, suppose that there are three datasets R, S, and T where R influences T, T influences R, and S influences T. In this case, we can create an influence graph G (shown in Figure 4.3) that contains three vertices R, S, and T and three directed edges: $R \rightarrow T$, $T \rightarrow R$, and $S \rightarrow T$.

The influence graph provides guidance on the order for resolving datasets. According to the influence graph in Figure 4.3, it seems clear that we should resolve S before T. However, it is unclear how to order R and T. One possible physical execution we could use is $(((S)),((R)),((T)))$. However, since T influences R, we may want to resolve R one more time after resolving T just in case there are newly merging (splitting) clusters in the partition P_R of R. As a result, we might end up running the physical execution $(((S)),((R)),((T)),((R)))$ instead. Furthermore, after resolving the last R, we may want to resolve T again just in case the partition P_T of T may change and so on. In general, we can only figure out the correct physical execution by actually running the ER algorithms until the ER results converge according to Definition 4.1.2. In the next section, we define the execution plan as a way to capture multiple possible physical executions into a more compact logical expression.

There are several ways to construct an influence graph. An automatic method is to view the ER algorithm where we draw an edge from R to S if the ER algorithm for S uses any information in R. Another method is to draw the edges based on the known semantics of the datasets. For example, while resolving papers may influence the resolution of venues, it is unlikely that the resolved papers would influence say phone numbers. When constructing an influence graph, It is desirable to avoid unnecessary edges in the influence graph. Intuitively, the fewer the edges, the scheduler can exploit the graph to generate "efficient" plans for resolving the datasets as we shall see in Section 4.2.2. On the other hand, if

most datasets influence each other, then there is not much optimization the scheduler can perform.

In the case where blocking techniques are used on a dataset R, we can construct an influence graph with fewer edges. Recall that blocking divides R into possibly overlapping smaller datasets $R_1, \ldots, R_k$ where each R_i fits in memory. In general, there can be edges between any R_i and R_j. If we assume that the resolutions of one block do not affect the resolutions in another block, however, we may remove the edges between the R_i's.

Furthermore, if blocking is used on multiple datasets, there are several possible ways for reducing the edges of the influence graph depending on the application. We illustrate the choices by considering a scenario of two datasets where R is a set of people and S is a set of organizations. Suppose that R influences S because some people may be involved in organizations. We assume that both R and S are too large to fit in memory and are thus blocked into smaller datasets. Hence, we would like to know which individual blocks of R influence which blocks in S. In the most general case, a person can be involved in any organization. Thus, each block in R influences all the blocks in S. However, suppose we know that the blocking for R and S is done on the country of residence and that the organizations in S are only domestic. By exploiting these application semantics, we can reduce the number of edges by only drawing an edge from each block of R to its corresponding block in S in the same country. In Section 4.2.3, we show how reducing the edges in the influence graph can improve the efficiency of joint ER.

4.2.2 Execution Plan

While an influence graph can be used to directly generate a physical execution, it is more convenient to add a level of abstraction and generate an intermediate execution plan first. The analogy is a DBMS producing a logical query plan for a given query before generating the actual physical query plan. As a result, the joint ER processor can be flexible in generating the actual physical execution based on the high-level execution plan. In this section, we show how the scheduler generates an execution plan based on an influence graph.

Name	Syntax	Description
Concurrent Set	$\{\dots\}$	Resolve datasets concurrently once
Fixed-point Set	$\{\dots\}+$	Resolve datasets until convergence

Table 4.2: Execution Plan Syntax

Syntax and Properties

An execution plan L is a sequence of concurrent and fixed-point sets (see Table 4.2). A concurrent set contains one or more datasets that are resolved concurrently once. A fixed-point set contains one or more datasets to resolve together until "convergence." That is, the datasets in a fixed-point set F are resolved possibly more than once in a sequence of concurrent sets to return the set of partitions defined below. (Note that $\mathcal{P}_{-R}$ below is still defined as $\{P_X|X \in \mathcal{D} - \{R\}\}$ and not as $\{P_X|X \in F - \{R\}\}$. That is, when resolving $R \in F$, the ER algorithm E_R may still use the information in a dataset outside F.)

Definition 4.2.2. *A* converged result *of a fixed-point set* F *is the set of partitions* $\{P_R|R \in F, P_R$ *is a partition of* R*, and* $E_R(P_R, \mathcal{P}_{-R}) = P_R\}$.

For example, datasets with the influence graph of Figure 4.3 can be resolved by the execution plan L = ($\{S\}, \{R, T\}+$) (see Section 4.2.2 for details on the execution plan generation) where S is resolved once, and R and T are repeatedly resolved until convergence. Hence, one possible physical execution for L is ($((S), ()), ((R), (T)), ((R), (T)), \dots$) where the number of $((R), (T))$'s depends on the contents of the records. (In Section 4.3.2, we show other techniques for generating physical executions of fixed-point sets that can improve the physical execution runtime.)

We can access a concurrent or fixed-point set of an execution plan using a list index notation. For instance, for the execution plan L above, $L[1]$ = $((S), ())$ and $L[2]$ = $((R), (T))$. Since an execution plan is a sequence of fixed-point and concurrent sets, one can concatenate two execution plans into a longer sequence.

We would like to define "good" execution plans that can lead to valid physical executions satisfying Definition 4.1.3. For example, suppose that the influence graph G contains two datasets R and S, and that R influences S. Then the execution plans ($\{R, S\}+$) and ($\{R\}$, $\{S\}$) seem to be good because for each R resolved, either S is resolved with R in

the same fixed-point set (in the first plan) or after R (in the second plan). However, the execution plan ($\{S\}$,$\{R\}$) may not lead to a correct joint ER result because resolving S again after the resolution of R may generate a different joint ER result.

We capture the desirable notion above into the property of conformance below.

Definition 4.2.3. *An execution plan L* conforms to *an influence graph G of $\mathcal{D}$ if*

1. $\forall R \in \mathcal{D}$, $\exists i$ *s.t.* $R \in L[i]$ *and*
2. $\forall R, S \in \mathcal{D}$ *s.t.* R *has an edge to* S *in* G, *for each* $L[i]$ *that contains* R, *either*
   - $L[i]$ *is a fixed-point set and* $S \in L[i]$ *or*
   - $\exists j$ *where* $j > i$ *and* $S \in L[j]$.

For example, suppose there are two datasets R and S where R influence S according to the influence graph G. Then the execution plans ($\{R, S\}$+) and ($\{R\}$, $\{S\}$) conform to G because in both cases, R and S are resolved at least once and S is either resolved with R in the same fixed-point set or resolved after R. However, the execution plan ($\{S\}$) does not conform to G because R is not resolved at least once (violating the first condition of Definition 4.2.3). Also, the execution plan ($\{R, S\}$) does not conform to G because S is neither resolved with R in the same fixed-point set nor resolved after R (violating the second condition). In Proposition 4.3.2 (see Section 4.3), we show that an execution plan that conforms to an influence graph G results in a valid physical execution given that the physical execution terminates.

Construction

A naïve execution plan that conforms to an influence graph G of the datasets $\mathcal{D}$ is ($\{\mathcal{D}\}$+), which contains a single fixed-point set containing all the datasets in $\mathcal{D}$. The following Lemma shows that ($\{\mathcal{D}\}$+) conforms to G.

Lemma 4.2.4. *Given an influence graph G of the datasets $\mathcal{D}$, the execution plan ($\{\mathcal{D}\}$+) conforms to G.*

Proof. The first condition of Definition 4.2.3 is satisfied because all the datasets in $\mathcal{D}$ are contained in the one fixed-point set of ($\{\mathcal{D}\}$+). The second condition holds as well because for any datasets R and S where R influences S, S is in the same fixed-point set as R. □

However, ($\{\mathcal{D}\}$+) is not an "efficient" execution plan in terms of the runtime of the physical execution since we would need to repeatedly resolve all the datasets until convergence. We thus explore various optimizations for improving an execution plan in general. We assume the scheduler only has access to the influence graph and not the runtime and memory physical statistics of the datasets. Hence, we use heuristics that are likely to improve the execution plan. The analogy is logical query optimization in database systems where pushing down selects in a query plan most likely (but not necessarily) improves the query execution time. An important requirement for the optimization techniques is that the final joint ER result must still satisfy Definition 4.1.2 (although the result does not have to be unique). We present three optimization techniques for execution plans and then present an algorithm that can produce efficient execution plans according to the optimization criteria.

Remove Redundant Datasets We can improve the efficiency of an execution plan by removing datasets that are redundant. For example, in the execution plan $(\{R, S\}+, \{R, S\})$, running ER on the concurrent set $\{R, S\}$ is unnecessary because the fixed-point set $\{R, S\}+$ already returns a converged result of R and S. We say that an execution plan L is *minimal* if for each dataset $X \in \mathcal{D}$, X is inside at most one concurrent or fixed-point set in L. For example, $(\{R, S\}+, \{R, S\})$ is not a minimal execution plan because R and S occur in two sets while $(\{R, S\}+)$ is a minimal plan.

Avoid Large Fixed-Point Sets A minimal execution plan is not necessarily the most efficient one. Consider the naïve execution plan ($\{\mathcal{D}\}$+), which contains a single fixed-point set containing all the datasets. While ($\{\mathcal{D}\}$+) is minimal, it may not be the most efficient execution plan if the fixed-point set can be divided into smaller concurrent and fixed-point sets. For example, suppose that $\mathcal{D}$ = $\{R, S, T\}$, and the execution plan L = $(\{S\}, \{R, T\}+)$, which conforms to the influence graph of Figure 4.3. When resolving the datasets in the order of L, we only need to resolve S once and do not have to worry

about the ER result of S after resolving R and T together. However, if we were to run the execution plan ($\{\mathcal{D}\}$+), we might have to resolve S multiples times because we do not have any information of which datasets can safely be resolved exactly once.

Hence, our second optimization for execution plans is to avoid large fixed-point sets. As a result, we can save the time needed to check for convergence as we illustrated above. Given an execution plan L that conforms to the influence graph G, we say that L is *compact* if no fixed-point set in L can be divided into smaller concurrent or fixed-point sets while still guaranteeing that L conforms to G. For example, suppose we have the two datasets R and S where R influences S. Then the execution plan $L = (\{R\}, \{S\})$ is compact because L conforms to G and does not contain any fixed-point sets. However, the plan $(\{R, S\})$ is not compact because it does not conform to G. Also, the plan $(\{R, S\}+)$ is not compact as well because the fixed-point set $\{R, S\}+$ can be divided into the sequence of two concurrent sets $\{R\}$ and $\{S\}$.

Notice that a minimal execution plan is not necessarily compact and vice versa. For instance, if there are two datasets R and S that do not influence each other, the plan L_1 = $(\{R, S\}+)$ is minimal while L_2 = $(\{R\}, \{S\}, \{R\}, \{S\})$ is not minimal. However, L_2 is compact while L_1 is not.

Maximize Parallelism Assuming we have enough cores and memory, we can improve an execution plan by placing more datasets in the same concurrent set to resolve concurrently. For example, if R and S do not influence each other, and the execution plan is $(\{R\}, \{S\}, \{T, U\}+)$, then given that we have two cores and enough memory, we could use the better plan $(\{R, S\}, \{T, U\}+)$ where we resolve R and S concurrently. In general, we say an execution plan is *maximally parallel* if for each concurrent or fixed-point set S resolved, we resolve as many datasets that are not influenced by other unresolved datasets as possible. For example, suppose we have the influence graph $R \rightarrow S$, $T \rightarrow U$. Then an execution plan $(\{R\}, \{S, T\}, \{U\})$ is not maximally parallel because T could have been resolved with R during the first concurrent set. On the other hand, the execution plan $(\{R, T\}, \{S, U\})$ is maximally parallel because while the first concurrent set $\{R, T\}$ is being resolved, neither S nor U can be resolved as well because R influences S and T influences U.

ALGORITHM 8: Constructing an execution plan

```
   input  : An influence graph G
   output: An execution plan L
 1 Group each strongly connected component of G into a node in the graph G';
 2 For each pair of nodes n, n' ∈ G', draw an edge from n to n' if ∃R ∈ n and ∃S ∈ n' where R
   has an edge to S in G;
 3 repeat
 4 |   Z ← nodes in G' that are not pointed from other nodes;
 5 |   if a node n ∈ Z has a size |n| > 1 then
 6 |   |   L ← L + (n+);
 7 |   |   remove n from G';
 8 |   else
 9 |   |   L ← L + (⋃_{n∈Z} n);
10 |   |   remove nodes in Z from G';
11 until G' is empty;
12 return L;
```

Construction Algorithm Algorithm 8 uses the three heuristics above to produce execution plans that conform to the given influence graph G. To illustrate Algorithm 8, suppose that the influence graph G contains four datasets R, S, T, and U and has three edges $R \rightarrow T, T \rightarrow R$, and $S \rightarrow T$. In Step 1, we identify the strongly connected components of G, which are $\{R, T\}$, $\{S\}$, and $\{U\}$. In Step 2, we create the graph G' where each strongly connected component in G forms a single node in G'. A node in G' is thus a set of datasets that are strongly connected in G. A node n in G' points to another node n' if a dataset in n points to any dataset in n' according to G. Hence, G' in our example has three nodes $n_1 = \{R, T\}$, $n_2 = \{S\}$, and $n_3 = \{U\}$ where n_2 points to n_1 (since S influences T). This construction guarantees that G' is always a directed acyclic graph. Starting from Step 3, we identify the nodes in G' that are not influenced by any other node. In our example, we identify the set $Z = \{\{S\}, \{U\}\}$. Since both nodes in Z have a size of 1, we add to L the combined set $\{S, U\}$ (Step 9). Next, we update Z to $\{\{R, T\}\}$. Since the node $\{R, T\}$ has two datasets, it is added to L as a fixed-point set $\{R, T\}+$ (Step 6). As a result, our final execution plan is $L = (\{S, U\}, \{R, T\}+)$. Finally we return L as the output in Step 12.

The following proposition shows the correctness of Algorithm 8.

Proposition 4.2.5. *Given an influence graph G of the datasets $\mathcal{D}$, Algorithm 8 returns an*

execution plan that conforms to G.

Proof. The first condition of Definition 4.2.3 is satisfied because every dataset in $\mathcal{D}$ is allocated to exactly one fixed-point or concurrent set in L by construction. We now prove the second condition of Definition 4.2.3. Suppose that R influences S in G. First, suppose there is a directed path from S to R in G. Then R and S form a strongly connected component and are thus grouped into the same node n in Step 1. In Step 6, n is added to L as a fixed-point set because n contains at least two datasets. Thus the first OR-condition of the second condition is satisfied. Second, if there is no directed path from S to R in G, then by Steps 4 and 8 of the algorithm, the node n containing S is added to L after the node n' containing R, satisfying the second OR-condition of the second condition. As a result, the second condition of Definition 4.2.3 is always satisfied. □

The following proposition shows the efficiency of the execution plans produced by Algorithm 8.

Proposition 4.2.6. *An execution plan produced by Algorithm 8 is minimal, compact, and maximally parallel.*

Proof. An execution plan L generated by Algorithm 8 is minimal because any dataset $X \in \mathcal{D}$ is not resolved in more than one concurrent or fixed-point set.

Next, we prove that L is also compact where none of the fixed-point sets in L can be split into smaller fixed-point sets while still guaranteeing that L conforms to G. Suppose that Algorithm 8 returns an execution plan L, and we split a fixed-point set $\mathcal{S}$ in L into two sets A and B. Since the datasets in $\mathcal{S}$ form a strongly connected component C in G, the sets A and B form two subgraphs C_1 and C_2 of C that are mutually connected in G. As a result, there exists a pair of datasets $R \in C_1$ and $S \in C_2$ where R influences S and a pair of datasets $R' \in C_1$ and $S' \in C_2$ where S' influences R'. Since L conforms to G and R influences S, the fixed-point set of C_2 must follow the fixed-point set of C_1. However, since S' influences R', the fixed-point set of C_1 must follow the fixed-point set of C_2, which is a contradiction. Thus, C_1 and C_2 must be combined into a single fixed-point set for L to conform to G.

Finally, the execution plan L is also maximally parallel because we add as many datasets that are not influenced by other datasets as possible in Step 8. □

We now show that Algorithm 8 runs in linear time.

Proposition 4.2.7. *Given an influence graph G with V vertices and E edges, Algorithm 8 runs in $O(|V| + |E|)$ time.*

Proof. Identifying the strongly connected components in G (Step 1) can be done in $O(|V| + |E|)$ time using existing method such as Tarjan's algorithm [96]. Next, generating the execution plan in Steps 3–9 can be done in $O(|V| + |E|)$ time by keeping track of nodes without incoming edges and removing edges from the nodes added to L. Hence, the total complexity of Algorithm 8 is $O(|V| + |E|)$. □

Again, the optimizations used in the scheduler are heuristics that are likely (but not guaranteed) to improve the actual runtime of the physical execution. For example, maximizing parallelism is only effective when we have enough cores and memory to resolve datasets concurrently. For now, we consider the execution plans produced by Algorithm 8 to be reasonably efficient for our experiments. In Section 4.3, we discuss how to produce efficient physical executions based on the optimized execution plans generated from Algorithm 8.

4.2.3 Exploiting the Influence Graph

The more we know about what data sets do *not* influence others, the better the scheduler can exploit the given influence graph and improve the efficiency of the execution plan. We illustrate this point by considering a blocking scenario where two datasets R and S are blocked into R_1, R_2 and S_1, S_2, respectively. We also assume that the blocks of R may influence the blocks of S and vice versa, but the blocks within the same dataset do not influence each other.

In the case where each block in R influences all the blocks in S and vice versa, we need to draw edges from R_i to S_j and S_j to R_i for $i, j \in \{1, 2\}$, resulting in a total of 8 edges. However, the more strongly connected the influence graph, the more difficult it becomes for the scheduler to generate an efficient execution plan. In our example, the scheduler generates the execution plan L_1 = ($\{R_1, R_2, S_1, S_2\}$+) using Algorithm 8.

Now suppose we know through application semantics that each block in R only influences its corresponding block in S with the same index and vice versa. As a result, we only need to draw four influence edges: R_1 to S_1, S_1 to R_1, R_2 to S_2, and S_2 to R_2. Hence, the scheduler generates the execution plan L_2 = ($\{R_1, S_1\}$+, $\{R_2, S_2\}$+) using Algorithm 8. The plan L_2 is more efficient than L_1 in a sense that we can exploit the information that R_1 and S_1 can be resolved separately from R_2 and S_2.

In an extreme case, we might know that R_2 does not influence any block in S. That is, only R_1 and S_1 influence each other. As a result, the execution plan is now L_3 = ($\{R_1, S_1\}$+, $\{R_2, S_2\}$) where we exploit that fact that R_2 and S_2 do not influence each other. Again, the plan L_3 is more efficient than L_2 where we know R_2 and S_2 only need to be resolved once.

4.3 Joint ER Processor

In this section, we discuss how the joint ER processor uses an execution plan to generate a valid physical execution. We first discuss how a concurrent set can be resolved within a constant factor of the optimal schedule. Next, we discuss how to resolve fixed-point sets. We then prove that sequentially resolving the concurrent and fixed-point sets according to the execution plan produces a valid physical execution as long as the execution terminates. Finally, we introduce expander functions, which can be used to significantly enhance the efficiency of joint ER.

4.3.1 Concurrent Set

When resolving a concurrent set, we are given a set of datasets to resolve once and a number of processors that can run in parallel. The overall runtime is thus the maximum runtime among all processors. We would like to assign the datasets to the processors such that the overall runtime is minimized.

We use the List scheduling algorithm [46] for scheduling datasets to processors. Whenever there is an available processor, we assign a dataset to that processor that start running ER. This algorithm is guaranteed to have a runtime within $(2 - \frac{1}{p})$ times the optimal schedule time where p is the number of processors. A key advantage of the List scheduling

algorithm is that it is an "on-line" algorithm, i.e., executed while ER is running. Predicting the runtime of ER is challenging because it may depend on how other datasets have been resolved. While there are other scheduling work that improve on the constant bound, they do not make significant improvements and are off-line, i.e., executed in advanced of any actual invocations of ER algorithms.

4.3.2 Fixed-Point Set

We now resolve a fixed-point set, where we are given a number of datasets that must be resolved at least once until convergence on parallel processors. The goals is to again minimize the overall runtime of the processors by scheduling the resolutions of datasets to the processors.

We propose an on-line iterative algorithm for resolving a fixed-point set. Again, the advantage of an on-line algorithm is that we do not have to worry about estimating the runtime of ER. The idea is to repeatedly resolve the fixed-point set, but only the datasets that need to be resolved according to the influence graph. Initially, all datasets need to be resolved. However, after the first step, we only resolve the datasets that are influenced by datasets that have changed in the previous step. For example, suppose we are resolving the fixed-point set $\{R, S, T\}+$ where R and T influence each other while S and T influence each other. We first resolve all three datasets using the greedy algorithm in Section 4.3.1. Say that only R and S had new partitions. Then we only need to resolve T in the next step. If T no longer has new partitions after its resolution, then we terminate.

Finding the optimal schedule for fixed-point sets is even more challenging than that of resolving a concurrent set. In our example above, since T is resolved twice, we might not have needed to resolve T during the first step. However, performing this optimization requires a deterministic way of figuring out if T will be resolved in future steps and whether it is indeed sufficient to resolve T just once, which are both hard to predict. We plan to study such optimizations in future work.

In general, resolving a fixed-point set is not guaranteed to converge. In order to see when we are guaranteed termination, we first categorize influence as either positive or negative. A positive influence from R to S occurs when for some P_R and P_S there exists

two records $s, s' \in S$ that are not clustered when we run the physical execution $(\{S\})$, but are clustered when we run the physical execution $(\{R\}, \{S\})$. Conversely, a negative influence occurs when s and s' are clustered when we run $(\{S\})$, but not clustered when we run $(\{R\}, \{S\})$. Note that an influence can be both positive and negative.

We now show that, if all influences are positive, then the iterative algorithm for fixed-point sets always returns a converged set of partitions for a fixed-point set.

Proposition 4.3.1. *If all the influences among the datasets in a fixed-point set F are not negative, then the iterative algorithm for fixed-point sets terminates and produces a set of partitions satisfying Definition 4.2.2.*

Proof. Suppose we have the fixed-point set F and repeatedly resolve the datasets that can potentially change. If all the influences are non-negative, clusters can only merge and never split. Since there is only a finite number of possible merges, the partitions of the datasets eventually converge into a set of partitions $\{P_R | R \in F\}$. In addition, since no clusters merge anymore, $E_R(P_R, \mathcal{P}_{-R}) = P_R$ for each dataset $R \in F$. Hence, there exists a converged result satisfying Definition 4.2.2 for the datasets in F. □

In the case where both negative and positive influences occur, the fixed-point set may not have a joint ER result. For example, say that two authors a_1 and a_2 merging influences a cluster containing two papers p_1 and p_2 to split, but then if p_1 and p_2 split, a_1 and a_2 split as well. Also, if a_1 and a_2 split, say that p_1 and p_2 merge, and p_1 and p_2 merging causes a_1 and a_2 to merge as well. As a result, the splits and merges continue indefinitely, and there is no fixed-point result that satisfies Definition 4.1.2.

If negative influences prevent joint ER from terminating, we can restrict the number of ER invocations in an execution plan L. Even if we do not have a fixed-point result that satisfies Definition 4.1.2, we may still improve accuracy compared to the simple case where the ER algorithms are run in isolation. For instance, if we are resolving the fixed-point set $\{R, S\}+$ where R and S influence each other, we can limit the repetition to, say, 2. Although the physical execution $(\{R, S\}, \{R, S\})$ is not guaranteed to return a joint ER result satisfying Definition 4.1.2, it may be close enough with a reasonable amount of runtime spent. In Section 4.5.3, we show that in practice, the ER results converge quickly (within 3 resolutions per dataset) while producing the correct joint ER result.

4.3.3 Joint ER Algorithm

We discuss how the joint ER processor uses an execution plan to generate a physical execution. Our joint ER algorithm first initializes each dataset by creating a set of singleton clusters of the records. The algorithm then sequentially resolves each concurrent or fixed-point set using the algorithms in Sections 4.3.1 and 4.3.2, respectively.

We show that the resulting physical execution is valid as long as it terminates.

Proposition 4.3.2. *Given an execution plan L that conforms to an influence graph G of the datasets $\mathcal{D}$, if the joint ER algorithm using L terminates, then the resulting physical execution is valid.*

Proof. Suppose that we have the datasets $\mathcal{D}$ and an influence graph G. Given an execution plan L that conforms to G, suppose that the joint ER algorithm terminates, producing a physical execution $\mathcal{T}$. First, the result of the basic algorithm is a set of partitions for all the datasets in $\mathcal{D}$ because we start from the initial partitions of $\mathcal{D}$ and only use valid ER algorithms on the partitions. Second, we show that running $\mathcal{T}$ produces the same partitions as running $\mathcal{T}$+ [$\{R\}$] for any $R \in \mathcal{D}$. For each dataset R, we show that, after running $\mathcal{T}$, its partition P_R cannot change further by running [$\{R\}$]. Suppose P_R has changed while running [$\{R\}$]. Then there must be some dataset X that was resolved after the last resolution of R in $\mathcal{T}$ and influenced the resolution of R during the execution of [$\{R\}$]. If X was in the same concurrent set as R, we have a contradiction because X should have been in the same fixed-point set as R by Definition 4.2.3. If X was in the same fixed-point set as R, then since the joint ER algorithm terminates, we know that the resolved fixed-point set satisfies Definition 4.2.2. Hence, resolving R again should not change P_R further, contradicting our assumption. Finally, X could be in a concurrent or fixed-point set other than R's set. Since L conforms to G, however, there must have been another resolution of R after the resolution of X in $\mathcal{T}$, which is a contradiction. Hence, $\mathcal{T}$ is a valid physical execution. □

4.3.4 Expander Function

We optimize a physical execution where we focus on minimizing redundant computation as much as possible when a dataset is resolved multiple times. A key property we satisfy is

that the resulting joint ER result is the same regardless of the optimization.

A record $r \in R$ *refers to* a record $s \in S$ if r contains a pointer to s. For example, if the paper record r contains an attribute with a label "venue" and value "ACM TODS," and s represents the venue ACM TODS, then r refers to s. Or the venue attribute could simply contain s's ID. In general, r could refer to more than one record in S. Hence, we denote the set of S records r refers to as $\mathcal{R}_S(r)$ where $\mathcal{R}_S(r) \subseteq S$. In our motivating example in the beginning of this chapter, $\mathcal{R}_P(v_1) = \{p_1, p_2\}$ while $\mathcal{R}_V(p_1) = \{v_1\}$. Conversely, the set of records in S that refer to r is denoted as $\mathcal{R}_S^{-1}(r)$. In our motivating example, $\mathcal{R}_P^{-1}(v_1)$ is again $\{p_1, p_2\}$. However, if p_1 did not refer to v_1, then $\mathcal{R}_P^{-1}(v_1)$ would be $\{p_2\}$ while $\mathcal{R}_P(v_1)$ is still $\{p_1, p_2\}$.

Compared to the case where all datasets are resolved in isolation, joint ER has the overhead of possibly resolving a dataset multiple times. To reduce this overhead, we would like to narrow down the set of records influenced by a previous resolution and only run ER on those records. For example, suppose that we have the execution plan ($\{R\}$, $\{S\}$, $\{R\}$) and have already resolved the first R and S. When resolving the second R, we would like to avoid running ER on the entire P_R. Instead, the idea is to run ER on only a small subset of clusters $O \subseteq P_R$ and produce the same result as resolving the entire P_R again. Determining O can be done by exploiting the given ER algorithm as we describe below. We note that the idea of avoiding resolution from scratch does not always work for all ER algorithms. That is, certain ER algorithms may perform global operations and thus require that all the records are resolved from the beginning. An in-depth study of the properties that enable incremental ER can be found in reference [103].

To construct the candidate records to resolve, we first construct the set of records M that contains R records that are referenced by records in other datasets that have changed since the last time R was resolved. That is, if the records in $c \subseteq S$ are newly clustered, and the records in c refer to the records in $c' \subseteq R$, then we add the records in c' to M. We repeat this operation for all datasets that influence R. However, resolving just R' may not produce a correct result. For example, suppose we want to compare the records r and r' when resolving R the second time. However, in order to see if r and r' are indeed the same entity, we might have to resolve the two records along with another record r'' because r can only match with r' if r' matches and clusters with r''. Hence, we must run ER on a

sufficiently large superset of R' that would guarantee a correct ER result.

An *expander function* X_R for dataset R produces this superset by receiving M and returning a set of clusters $O \subseteq P_R$ such that running ER on O produces a result just as if we have run ER on the entire partition P_R. More formally, we define a valid expander function as follows.

Definition 4.3.3. *Given an input partition P_R of R for ER and a set of records M to resolve, a* valid *expander function X_R for the ER algorithm E_R satisfies the following condition:*

- $E_R(P_R) = E_R(X_R(M)) \cup (P_R - X_R(M))$

If $|X_R(M)|$ is much smaller than $|P_R|$, then running ER on $X_R(M)$ can be much faster than running ER on the entire P_R. We note that not all ER algorithms have a valid expander function that returns a set $X_R(M)$ smaller than P_R.

To illustrate an expander function, we use the sorted neighborhood technique [54] (SN) as our ER algorithm. The SN algorithm first sorts the records in P_R (i.e., we extract all the records from the clusters in P_R) using a certain key assuming that closer records in the sorted list are more likely to match. For example, suppose that we have the input partition P_R = $\{\{r_1\}$, $\{r_2\}$, $\{r_3\}\}$ and sort the clusters by their names (which are not visible in this example) in alphabetical order to obtain the list Z = (r_1, r_2, r_3). The SN algorithm then slides a fixed-sized window on the sorted list of records and compares all the pairs of records that are inside the same window at any point. More formally, we only compare the records r and r' where $|Rank(r, Z) - Rank(r', Z)| < W$ where $Rank(r)$ denotes the index of r within Z while W denotes the window size. If the window size is 2 in our example, then we compare r_1 with r_2 and then r_2 with r_3, but not r_1 with r_3 because they are never in the same window. We thus produce pairs of records that match with each other. After collecting all the pairs of records that match, we perform a transitive closure on all the matching pairs of records to produce a partition P'_R of records. For example, if r_1 matches with r_2 and r_2 matches with r_3, then we merge r_1, r_2, r_3 together into the output P'_R = $\{\{r_1, r_2, r_3\}\}$.

Given a previous SN result P_R and a set of records M to re-resolve, we define the expander function X_R to return the set of records O = $\bigcup_{c \in Y} c$ where $Y = \{c \mid c \in P_R \wedge \exists r \in c, r' \in M$ s.t. $|Rank(r, Z) - Rank(r', Z)| < M\}$. (We later prove that O correctly

contains all the records that may potentially match with records in M.) For example, if we have the sorted list Z = (r_1, r_2, r_3, r_4) and M = $\{r_1, r_2\}$, then given the partition P_R = $\{\{r_1, r_2\}$, $\{r_3\}$, $\{r_4\}\}$, we need to resolve M's records with r_3 because the cluster $\{r_3\}$ overlaps with one of the windows (r_2, r_3) of r_2. However, we do not have to resolve the records of M with r_4 because no window of r_1 or r_2 overlaps with the cluster $\{r_4\}$.

Proposition 4.3.4. *The function X_R is a valid expander function for the SN algorithm satisfying Definition 4.3.3.*

Proof. Suppose that we are given a previous SN result P_R and a set of records M to re-resolve. Then for each $r \in M$, the SN algorithm must resolve r with any record r' that is within a cluster in P_R that overlaps with any window of any record in M. We revisit our example above where we have the sorted list Z = (r_1, r_2, r_3, r_4) and M = $\{r_1, r_2\}$. Then given the partition P_R = $\{\{r_1, r_2\}$, $\{r_3\}$, $\{r_4\}\}$, we need to resolve the records of M with r_3 because r_2 and r_3 are in the same window and might match because $r_2 \in M$. In general, any record satisfying the condition above must be in the output of X_R because there is a chance of a transitive match of records $r, s_1, \ldots, s_k, r'$ where r matches with s_1, s_2 matches with s_3, ..., and s_k matches with r'.

We now show that if r' does not satisfy the above condition, then r and r' are guaranteed not to match when SN is run on the entire P_R. Since r and r' are not within the same window, then r and r' are never directly compared. Also, since the cluster of r' does not overlap with any window of any record in M, there is no chance of a transitive match of records $r, s_1, s_2, \ldots, s_k, r'$ either. In our example above, we do not have to match the records of M with r_4 because we know that r_3 and r_4 did not previously match with each other according to P_R and will not match now either because both r_3 and r_4 are not in M.

Our definition $X_R(M) = \bigcup_{c \in Y} c$ (where $Y = \{c \mid c \in P_R \wedge \exists r \in c, r' \in M$ s.t. $|Rank(r, Z) - Rank(r', Z)| < M\}$) exactly captures the condition above. Hence, the result of running E_R only on $X_R(M)$, i.e., $E_R(X_R(M)) \cup (P_R - X_R(M))$, is equivalent to the result of running E_R on P_R. □

We can now improve the resolution of a fixed-point set by using expander functions. Instead of running $E_R(P_R, \mathcal{P}_{-R})$ for each dataset R, we run $E_R(X_R(Cand(P_R)), \mathcal{P}_{-R})$

where $Cand(P_R)$ returns the records in R referenced by newly-clustered records in other datasets.

4.4 ER Algorithm Training

In a joint ER setting, training an ER algorithm may be challenging. For example, say that we are physically executing $(((R)), ((S)), ((R)))$. Say that we first train the ER algorithm for R and resolve R. After resolving S, however, we may want to "re-train" the ER algorithm to resolve R for the second time in order to reflect the ER result of S. In this section, we propose a state-based training technique that can train an ER algorithm based on the state of other datasets being resolved.

4.4.1 Match Rule

We focus on a specific type of ER algorithm that uses a *match rule* to decide if records represent the same real-world entity. A match rule can be a Boolean match rule that determines if two records represent the same entity, or a distance match rule that quantifies how different (or similar) the records are. A Boolean match rule B is defined as a function that takes two records and returns true or false. We assume that B is commutative, i.e., $\forall r_i, r_j, B(r_i, r_j) = B(r_j, r_i)$. For example, two person records may match according to B if their names and addresses are similar. A distance match rule D is defined as a commutative distance function that returns a non-negative distance between two records instead of a Boolean value. Alternatively, a distance can be viewed as a similarity between 0 (completely different) and 1 (identical). For example, the distance between two person records may be the sum of the distances between their names, addresses, and phone numbers. Of course, not all ER algorithms use match rules to resolve records, and not all ER algorithms have parameters that can be trained. Nevertheless, there are many ER algorithms that do use match rules whose parameters are learned.

How exactly the ER algorithm uses B (or D) to derive the output partition P'_R depends on the specific ER algorithm. For example, say that records r and s match, s and t match, but r and t do not match according to a Boolean match rule B. If the ER algorithm performs

a connected component operation on the matching records, then the output is $\{\{r, s, t\}\}$. On the other hand, if the ER algorithm only clusters the records that all match with each other, then the answer may now be $\{\{r, s\}, \{t\}\}$ or $\{\{r\}, \{s, t\}\}$. However, we assume that, if two records match according to a match rule, then they are very likely to end up in the same cluster in the ER result.

We assume that each record $r \in R$ consists of a set of attributes $r.\mathcal{A}$. For illustration purposes, we can think of each attribute $a \in r.\mathcal{A}$ as a label-value pair, although this view is not essential for our work. As an example, the following record may represent the person Bob:

$$r = [name : \{\text{Bob}\}, address : \{\text{123Main}\}, age : \{30\})$$

The value of the attribute a of r is denoted as $r.a$. For instance, $r.name$ = {Bob}.

We then assume that the Boolean match rule can be expressed in the following form: $B(r, s) = w_0 + \Sigma_{i=1,\ldots,k} w_i \times Sim_i(r, s) \geq 0$. Each similarity function Sim_i computes the similarity of r and s using the attributes in r and s.

4.4.2 Training

We now train the weights $w_0, \ldots, w_k$ of a Boolean match rule B. (Training a distance match rule can be done in a similar fashion.) For each record pair (r, s), the attribute similarities form a "feature vector" $f(r, s) = (Sim_1(r, s), \ldots, Sim_k(r, s))$ that represents (r, s). We also define the vector of weights $w = (w_1, \ldots, w_k)$. Using f and w, the match rule $B(r, s)$ can be rewritten as the following equation.

$$B(r, s) = w \times [1, f(r, s)]^T \geq 0.$$

We can thus formalize the optimization problem for training w as follows:

$$\begin{array}{ll} \min_{w_0, w} & \frac{1}{2}||w||^2 + C \sum_{p \in R \times R} \xi_p \\ \text{s.t.} & y(p)(w \times f(p)^T + w_0) \geq 1 - \xi_p, p \in R \times R \\ & \xi_p \geq 0, p \in R \times R \end{array}$$

The parameter C is the penalty parameter of the error term and $y(r, r')$ returns 1 if r and r' are clustered in the final ER result and -1 otherwise. The weights $w_0, \ldots, w_k$ can be trained using a standard SVM method using a linear kernel function [67] where the optimization problem below is equivalent to finding the optimal margin classifier for the matching and non-matching pairs of records according to the gold standard.

We can train on a small sample (say 10%) of the datasets in $\mathcal{D}$ where we know the correct joint ER results. These samples will be our gold standard, which "represent" $\mathcal{D}$ in a sense that the match rules that work well for the samples will also work well for resolving the entire datasets in $\mathcal{D}$ for the entire joint ER process. Notice that for each resolution of the dataset R, we may train a separate set of weights using our sample datasets.

The training above assumes that the given ER algorithm will most likely cluster the records that match according to the match rule. If the given ER algorithm is more complex and cannot be properly trained by the optimization problem above, then one could either use a more complex optimization process to train the weights $w_0, \ldots, w_k$ or use a custom training (provided by the application developer) that adjusts the parameters of the ER algorithm. Notice that in the latter case, the match rule does not have to be a linear combination of similarity values anymore and can be any black-box function.

4.4.3 State-based Training

If all the datasets are resolved in isolation, we would only need to train the ER algorithm for each dataset once. Since we are in a joint ER environment, however, training based on which other datasets have been resolved can be useful. For example, suppose that we are resolving papers mostly based on their title comparison results. However, if an author dataset that influences the paper dataset is resolved, then it might help to put more emphasis on the author comparison result as well. Moreover, if the papers have previously been resolved once already, then we would actually like to give less emphasis on the title comparison results.

We define the *state* of the current resolution of a dataset X to be the result of the physical execution before resolving X. We would like to train a set of weights W used by the ER algorithm E_X for each possible state. For instance, if the entire physical execution

is $(((R)),((S)),((R)))$, then we train R based on the states $()$ and $(((R)),((S)))$.

In general, it is difficult to exactly predict the entire physical execution unless the joint ER processor follows a pre-defined pattern of resolving datasets. Simply training on all possible states is impractical if there is an infinite number of states. A reasonable assumption is that the resolution of the sampled datasets used for training will generate the same physical execution as that of the full data. In Section 4.5.3, we demonstrate that state-based training can significantly improve the accuracy of training without states.

4.5 Experimental Results

We first evaluate our joint ER techniques on synthetic datasets and show the runtime behavior of our techniques. We then evaluate the scalability and behavior of joint ER on a large real dataset (called the Spock dataset [92]). Finally we evaluate our training techniques using another real dataset (called the Cora dataset) to demonstrate state-based training. Our algorithms were implemented in Java, and our experiments were run on a 2.4GHz Intel(R) Core 2 processor with 4 GB of RAM.

4.5.1 Synthetic Data Experiments

We evaluate the runtime behavior of joint ER using synthetic data. The main advantage of synthetic data is that they are much easier to generate for different scenarios and provide more insights into the operation of our joint ER algorithms.

Table 4.3 shows the parameters used for generating the synthetic data and the default values for the parameters. We first create d datasets where each dataset contains records that represent a total of s entities. For each entity, there are u records that represent that entity. As a result, each dataset contains $s \times u$ records. While each dataset thus has $200 \times 5 = 1,000$ records as a default, one could easily scale this data to much larger sizes. Each record r in a dataset contains one integer value $r.v$. While a record may contain many attributes in practice, we simplify our model and assume that $r.v$ represents all the attributes of r that are not references to records in other datasets. In addition, we later use the values to "emulate" the match rule, i.e., if two records have values that are close, then they will more likely be

Param.	Description	Val.
Data Generation		
d	Number of datasets	15
s	Number of entities per dataset	200
u	Number of duplicate records per entity	5
i	Value difference between consecutive entities	10
v	Maximum deviation of value per entity	5
Match Rule		
a	Value similarity weight	0.5
t	Record comparison threshold	0.5
Resource		
p	Number of processors	2

Table 4.3: Parameters for generating synthetic data

considered the same entity by the match rule. We assume that the entities of a dataset have the values 0, i, $2 \times i$, ..., $(s-1) \times i$. A record that represents an entity with a value of e contains a value randomly selected from $[e, e+v]$. If a dataset R influences S, we create an attribute in each R record that refers to an existing record in S. When assigning references, we require that for any two records of the same entity, they can only refer to two records of the same entity of another dataset. The set of datasets influencing the dataset S is denoted as $\mathcal{I}(S)$. As defined in Section 4.3.4, the set of records in S referring to the record r is denoted as $\mathcal{R}_S^{-1}(r)$. When running joint ER, the parameter p indicates the number of CPU processors that can resolve datasets concurrently.

The match rule B compares two records and returns true if they are similar and false otherwise. When comparing the records r and r', B considers two similarities: the value similarity, which compares the values $r.v$ and $r'.v$, and the reference similarity, which compares the references of r and r' to records in other datasets. We use the parameter a to balance the value and reference similarities as follows:

$$B(r, r') = a \times \frac{1}{|r.v - r'.v| + 1} + \\ (1-a) \times \frac{\sum_{X \in \mathcal{I}(R)} I(r, r', X)}{\sum_{X \in \mathcal{I}(R)} I(r, r', X) + N(R)} \geq t$$

where $I(r, r', X) = |\mathcal{R}_X^{-1}(r) \cap \mathcal{R}_X^{-1}(r')|$ and $N(r, r') = |\{Y | Y \in \mathcal{I}(R) \wedge I(r, r', Y) = 0\}|$. That is, $I(r, r', X)$ is the number of records in X that refer to both r and r', and $N(R)$ is the number of datasets that influence R and that do not have any records that refer to both r and r'. The first term weighted by a is the normalized value similarity, which ranges from $\frac{1}{d+1}$ to 1, and increases as the values of r and r' are more similar. The second term weighted by $(1 - a)$ is the normalized reference similarity, which ranges from 0 to 1 and increases as more references in r and r' overlap. B returns true if the sum of these normalized values is larger or equal to the comparison threshold t.

For our ER algorithm we use the R-Swoosh algorithm [10], which uses a Boolean pairwise match rule to compare records and a pairwise merge function to combine two records that match into a composite record. When two records r and r' are merged, the composite record r'' contains either $r.v$ or $r'.v$ as its value.

We measure the runtime in terms of the "critical" number of record comparisons. That is, for each synchronous step of joint ER, we add the maximum number of record comparisons among all parallel running processors. For example, if processor 1 ran 10 comparisons in the first and second iterations while processor 2 ran 9 and 11 comparisons, respectively, then the critical number of record comparisons is $max\{9, 10\} + max\{10, 11\}$ = 10 + 11 = 21 comparisons. Although the record comparison time itself may change, we believe that counting the comparisons is a reasonable representation of the amount of work done.

Influence Graph Pattern

In this section, we study how our joint ER algorithm improves over a naïve implementation of joint ER that does not exploit the influence graph. In the naïve solution, all the datasets are repeatedly resolved until all of them converge. Notice that *all* the datasets need to be resolved for each step because, even if a dataset does not change in the current step, we do not know if a change in some other dataset might influence this dataset later on. Given a set $\mathcal{D}$ of datasets, the naïve solution is thus the most efficient way to resolve the execution plan ($\{\mathcal{D}\}$+) without exploiting the influence graph. We use the default settings in Table 4.3 to construct our datasets and do not use expander functions.

When constructing the influence graph used by our joint ER algorithm, we consider

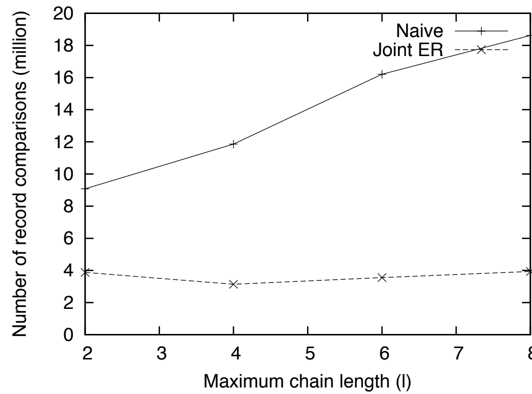


Figure 4.4: Linear structure results

two patterns: linear and random. In the linear model, the influence graph is a collection of "chains" where each chain contains datasets that form a linked list of influence in one direction. Given a maximum chain length of l, we use the first 10 datasets to create $n = \lfloor \frac{10}{l} \rfloor$ chains of length l and one more chain of length $10 - n \times l$ if $10 - n \times l > 0$. The remaining 5 datasets neither influence nor are influenced by other datasets. In the random model, we use a given probability c for creating a more connected structure. For any pair R and S in the first 10 datasets, R influences S with a probability of c. The remaining 5 datasets do not influence or are influenced by other datasets.

Figure 4.4 shows how adjusting l in the linear model influences the critical number of record comparisons. Since at least five datasets are redundantly resolved for each step, the total work of the naïve solution increases linearly for larger l values. In comparison, the joint ER algorithm does an almost constant amount of work for any l by only resolving the necessary datasets at each step. As a result, the joint ER algorithm outperforms the naïve solution by 2.3–4.7x.

Figure 4.5 shows how adjusting c in a random model influences the critical number of record comparisons. If c is small, only a few datasets influence each other, so our joint ER algorithm can avoid redundantly resolving datasets by exploiting the influence graph. If c is larger, then the first 10 datasets are more likely to be connected with each other, and our joint ER algorithm performs similarly to the naïve solution for those 10 datasets. Notice that, since the remaining five datasets do not influence and are not influenced by other

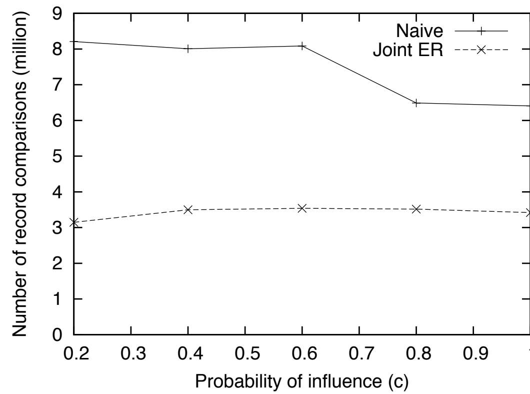


Figure 4.5: Random structure results

datasets, the joint ER algorithm outperforms the naïve solution by a certain degree even if $c = 1$. As a result, the joint ER algorithm outperforms the naïve solution by 1.8–2.6x.

In summary, the joint ER algorithm outperforms the naïve solution by exploiting the influence graph. If the influence graph has a linear structure, then the joint ER algorithm performs better for longer chains. If the influence graph has a random structure, then the joint ER algorithm performs better for sparser graphs.

Number of Iterations

We study how certain parameters influence the number of iterations of the joint ER process. In order to be able to interpret our results more clearly, we focus on a single cycle with two datasets (called R and S) that influence each other. We then run ER alternatively on the two datasets until both of them converge.

Figure 4.6 shows how the value similarity weight a influences the number of iterations. We count each resolution of R or S as one iteration. For example, the resolution of R, S, and R is viewed as three iterations. Each group of bars show the results for one a value. The nth bar in a group shows the number of record comparisons for the nth dataset resolved. That is, the first bar represents the result of resolving R, the second bar the result of S, the third bar the result of R, and so on. Hence, the number of bars per group is the number of iterations it took for resolving R and S together. For example, if $a = 0.1$, there are eight iterations. As a increases, the value similarity of B becomes the dominant factor

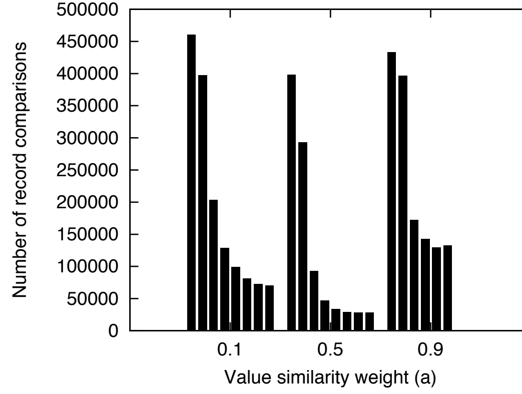


Figure 4.6: Value similarity weights versus iterations

when comparing records. As a result, there are fewer iterations (if $a = 0.9$, there are only six iterations) because the overlap in references have less impact on the comparison of records.

Expander Function

We now study the performance of expander functions in various scenarios. We use an expander function E that receives a record $r \in R$ and returns all the records in R that have a value within the range of $[\lfloor \frac{r.v}{i} \rfloor \times i, \lfloor \frac{r.v}{i} \rfloor \times i + v]$, which covers all the records that represent the same entity as r by our construction. We prove that E is valid as long as $i > v$ and $t \geq a \times \frac{1}{i-v+1}$. Suppose that $i > v$. Then E returns exactly the records that refer to the same entity as r. Also, for any two records r and r' in R that do not represent the same entity, $|r.v - r'.v| \geq i - v$. By construction of references, r and r' have no overlapping references to any other dataset, so the reference similarity is always 0. Hence, the weighted similarity of $B(r, r')$ is $a \times \frac{1}{|r.v - r.v'|+1} + (1-a) \times 0 \leq a \times \frac{1}{i-v+1}$. Hence, as long as $t > a \times \frac{1}{i-v+1}$, $B(r, r')$ is always false, and E is valid. Since E only returns the records that represent the same entity as r, E should be viewed as an optimal expander function that returns the best result possible. We compare the total number of critical comparisons among three methods: the naïve solution, the joint ER algorithm that does not use E, and the joint ER algorithm that uses E. We use a random model with an influence probability of $c = 0.3$ for generating the influence graph.

In Figure 4.7, we evaluate the performance of E by varying the comparison threshold

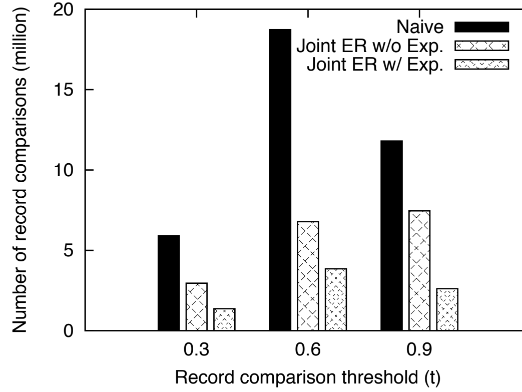


Figure 4.7: Threshold versus expander function performance

t. For E to be valid in our default setting, we need t to be larger than $a \times \frac{1}{i-v+1} = \frac{1}{60}$. If t = 0.3, most records are likely to match with each other, so there are fewer record comparisons performed because of the frequent merging of records. As t increases to 0.6, fewer records start to match, so more record comparisons are performed with fewer merges. In addition, more iterations are needed to completely resolve all the datasets, which further increases the number of record comparisons. If t increases to 0.9, then very few records match, so there are more record comparisons. However, there are fewer iterations as well, so the number of record comparisons actually decreases for the naïve solution and the joint ER solution with expander functions. Throughout the three configurations of t, the joint ER algorithm with E outperforms the naïve solution and the joint ER algorithm without E by 4.3–4.9x and 1.8–2.9x, respectively. Since we are experimenting on a best-case expander function, the performance of using any other expander function should be somewhere between our results of running joint ER with and without E.

Next in Figure 4.8, we vary the number of duplicates per entity u from 3 to 9. As u increases, the number of record comparisons increases because there are more records to compare. However, the number of iterations does not change for different u values. In the three configurations of u, the joint ER algorithm with E outperforms the naïve solution and the joint ER algorithm without E by 3.8–4.4x and 1.7–1.8x, respectively.

In summary, the joint ER algorithm using expander functions can outperform the other two techniques for various joint ER scenarios.

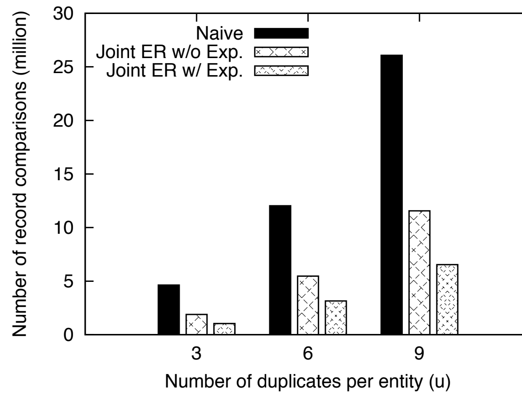


Figure 4.8: Number of duplicates versus expander function performance

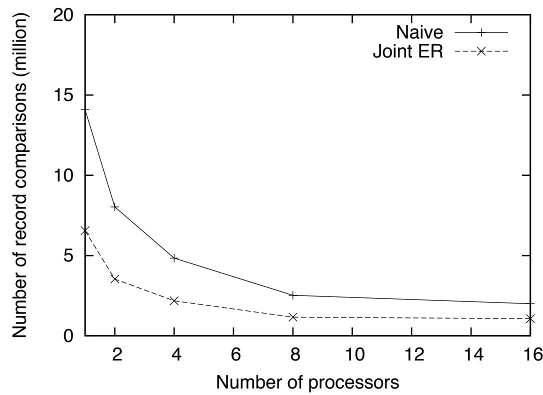


Figure 4.9: Number of processors versus record comparisons

Number of Processors

In Figure 4.9, we compare our joint ER algorithm with the naïve solution varying the number of processors p. Both plots are proportional to the plot $y = \frac{1}{p}$. If p = 1, joint ER outperforms the naïve solution by 2.1x in record comparisons. However, if p = 16, the naïve solution starts to perform relatively better (only 1.8x worse in record comparisons) because all datasets are being resolved at the same time, making convergence faster and thus reducing the number of iterations. Hence, our joint ER algorithms are more effective when there are fewer processors, i.e., when the resources are more scarce.

Exploiting Semantic Knowledge

We now resolve datasets using blocking techniques and show how reducing the edges in the influence graph by exploiting application semantics can improve the runtime performance. We experiment on 4–16 datasets that form a random influence graph where for each pair of datasets R and S, R influences S with probability 0.5. For half of the datasets, we increase each size from 1,000 to 10,000 records. We assume that only 1,000 records can be resolved in memory, so we divided each large dataset into 10 blocks of size 1,000. We use three scenarios for influence relations among blocks. In the first scenario, we assume a "quadratic" connection scenario where for every R that influences S, all the blocks of R influence S as well. In the case where S itself is divided into blocks, all the blocks of R influence each of the blocks of S. In the second scenario, we consider a "linear" connection scenario where each block in R only influences its corresponding block in S with the same index. In our third scenario, we consider an extreme "single" connection scenario where only one random block of R influences the corresponding block in S with the same index. Notice that the single connection scenario produces a best-possible influence graph that can be generated when R influences S. For the other parameters, we use the default settings in Table 4.3 to construct our datasets and do not use expander functions.

Figure 4.10 shows that, the single-connection scenario outperforms the quadratic-connection scenario by 1.6–3.9x and the linear-connection scenario by 1.6–3.4x depending on the number of datasets resolved. The improvement is due to the better exploitation of the influence graph by the scheduler.

4.5.2 Real Data Experiments

We now test the scalability of our joint ER algorithm on a real dataset provided by a commercial people search engine called Spock [92], which collects hundreds of millions of person information records from various websites such as Facebook, MySpace, and Wikipedia. The records are then resolved to generate one profile per person. The Spock schema contains various datasets that contain personal information, education, employment, addresses, and tags of people. We obtained a subset of the entire Spock dataset that contained information of about 70 million people whose names started with one of the characters 'c', 's',

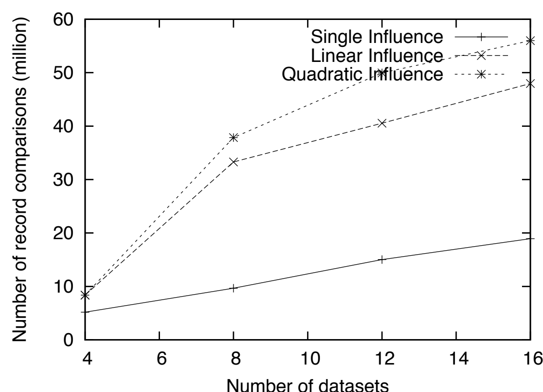


Figure 4.10: Blocking scenario versus record comparisons

or 'k'. We chose this particular subset because relatively many people have names that start with the three characters, increasing the chance for finding duplicates [1].

In our experiments, we used a simplified version of the Spock schema and resolve the following types of records: persons, addresses, schools, and jobs. For our scalability experiments, we generated subsets of the data of different sizes, as follows. First we select a random subset (called P) of the desired size from the 70 million person records. Then we select the addresses (called A), schools (called S), and jobs (called J) that refer to at least one person among the randomly selected person records. The largest dataset we generated in this fashion contains 1M people records, 0.8M addresses, 3.5K schools, and 6.6K jobs. We also generated datasets with 0.25M, 0.5M and 0.75M people records.

Our datasets may not be representative of the Spock data in terms of the number of duplicates per entity because we take a random sample of the entire 70 million person records. For example, if there are 100 duplicates per entity, and we take a 10% sample of the entire data, then there would be on average only 10 duplicates per entity in the sample set. An alternative sampling method that preserves the number of duplicates per entity is to take a random subset of the actual entities and collect all the records that refer to those entities. Since we did not have a gold standard for the Spock data, however, determining the actual entities was challenging.

Each record contains either values or references to other types of records. All the

[1] Spock was unable to give us all the data for legal reasons.

letters in the records were converted to lowercase. An address record in A contains a street address, city, and state, but no attributes that refer to other types of records. A person record in P contains a first name, last name, gender, age, city, and state, and also contains attributes that refer to records in A, S, and J. A school record in S record contains a name and an attribute that refers to records in P. A job record in J also contains a name and an attribute that refers to records in P. As a result, we used the influence graph where A influences P, P and S influence each other, and P and J influence each other. Hence, the execution plan generated by Algorithm 8 was ($\{A\}$ $\{P, S, J\}$+).

Since both A and P were too large to fit in memory, we used blocking techniques to divide the two datasets. The blocking on A was based on the first character of the city appended with the state for each address record. For example, if an address record has the city "stanford" and state "ca", then the blocking key is "sca". For the records that did not contain a state, there were only a few thousand of them so they were all put in a single block containing records without states. As a result, we may miss matches between records that have a state and those that do not have a state. For example, an address in "atlanta, ga" will never be compared with a record in "atlanta" without a state although the two addresses may be the same. On the other hand, the blocking prevents unnecessary matches that may occur due to the sparse information. That is, if the second "atlanta" was in the state "tx", then the two addresses should not be compared even if they look similar. The blocking on P was based on the first two characters of the last name appended with the state for each person record. For example, if a person record has the last name "smith" and the state "tx", then the blocking key is "smtx". (We used two characters of the last name instead of one because all last names started with a 'c', 's', or 'k'.) The records without a state were distributed by the first two characters of their last names into blocks containing records without states. The S and J datasets were small enough to fit in one block each. While the blocks containing the records were stored on disk, the influence graph and the information of which records were clustered together was kept in memory.

We use fixed-sized extents called segments to store the blocks on disk [109]. A segment acts as a unit of transfer for reading blocks into memory and thus cannot exceed the memory size. We allocate a fixed number of segments consecutively on disk and then randomly assign the blocks to the segments. The advantage of using segments is that the different

block sizes are evened out when they are randomly assigned to the segments. As a result, we can approximately do the same amount of work for each segment processed.

We use two ER algorithms to demonstrate that our framework can use different ER algorithms for each type of data.

- The Sorted Neighbor (SN) algorithm (see Section 4.3.4) was used for the A and P datasets. When resolving A, we sorted the records by their cities and then used a sliding window of size 100 for comparing the records. When comparing two address records, we performed a string similarity comparison of the street address using the Jaro distance function [111]. We considered the records to match if either the street addresses were near identical or if the street addresses were similar and the states were the same. When resolving P, we sorted the records by their last names and then used a sliding window of size 100 for comparing the records. When comparing two person records, we compared the appended first and last names using the Jaro distance function. If the names were similar, we also checked if the two records had the same age and gender or the same city and state or the same school or the same job to determine a match.

- The R-Swoosh algorithm (see Section 4.5.1) was used for resolving the S and J datasets. For both S and J, two records were considered to match if they either had near-identical names or had similar names and referred to at least one common P record.

Our setting thus illustrates the flexibility of our framework where one can plug in any ER algorithm for each dataset resolved.

Figure 4.11 shows the runtime of joint ER as the number of person records increases. The sizes of the other types of data are not shown in the x-axis, but increase in proportion to the number of person records. The different plots show the results for using 1–4 concurrent threads. For any number of threads, the joint ER runtime increases linearly to the number of records resolved, mainly because the SN algorithm has linear scalability. As the number of threads increases, the runtime improves in a sub-linear fashion. For example, the runtime for resolving the Spock data with 1M person records improves by 1.4x when increasing the

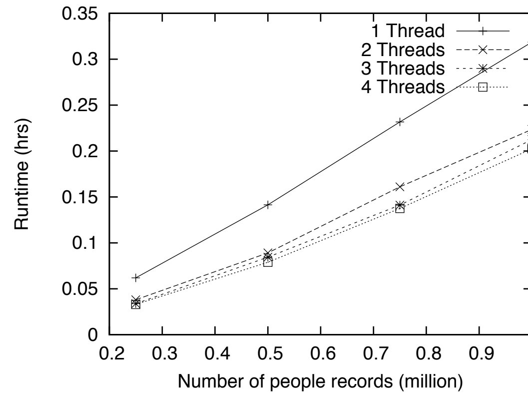


Figure 4.11: Scalability results on the Spock dataset

number of threads from 1 to 2, but only improves by 1.1x when increasing the number of threads from 2 to 4. The main reason is that the block sizes were not evenly distributed, so the workload of record comparisons was not evenly distributed to the threads as well. If all the blocks had the same size, then we would see runtime improvements more proportional to the number of threads. In addition, there is a fixed cost of initially distributing the records to the blocks on disk, which further reduces the benefit of concurrent processing. Nevertheless, the results show that our joint ER algorithm can scale to millions of records.

Figure 4.12 shows the scalability results as above, except that the largest dataset P is now resolved with the R-Swoosh algorithm instead of the SN algorithm. Since R-Swoosh has a quadratic complexity, the joint ER runtime increases quadratically as the number of person records increases. Compared to Figure 4.11, the joint ER runtimes are about an order of magnitude larger as well. The results show that the joint ER scalability heavily depends on performances of the specific ER algorithms plugged into the framework.

We now study the physical execution of running joint ER on the Spock data. Instead of showing all the details of the physical execution, we summarize by showing the number of segments that were read into memory for each dataset resolved. (Recall that a segment is a unit of transfer for reading blocks into memory.) For example, the summarized physical execution ((R:1), (S:2, T:3)) says that one segment containing the records of R was read into memory for resolution, and then two segments of S and three segments of T were read into memory for resolution. The first row of Table 4.4 shows the physical execution

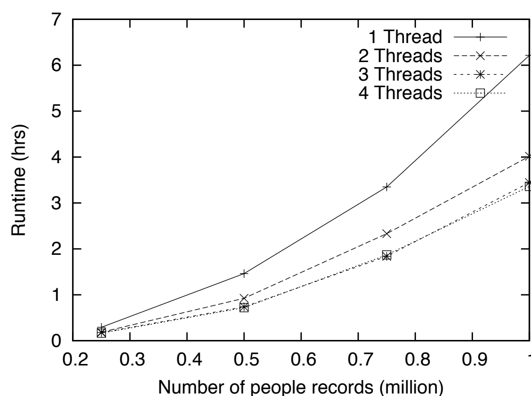


Figure 4.12: Scalability results when running R-Swoosh on P

Setting	**Physical Execution Summary**
Figure 4.11	((A:40), (P:40,S:1,J:1), (P:26,S:1,J:1), (P:1))
Figure 4.12	((A:40), (P:40,S:1,J:1), (P:26,S:1,J:1))

Table 4.4: Physical Execution Summary on Spock Data

using the setting of Figure 4.11 for resolving the Spock data with 1M people records using one thread where the SN algorithm was used to resolve P. The second row shows the execution when we use the setting of Figure 4.12 for resolving the same data using one thread where the R-Swoosh algorithm was used to resolve P. Compared to the physical execution in the first row, one fewer segment of P was read into memory for the second row because R-Swoosh is more "optimistic" in matching records compared to the SN algorithm and thus finds all the matching records early on. Both executions do not have many iterations because there were relatively few matching records. On average, 5% of the records in A clustered with other records for each block resolved (including repetitions where some blocks were resolved multiple times). For the other datasets, the percentages were 0.5–1.1%, 0.2–0.3%, and 0.4% for P, S, and J, respectively.

4.5.3 Training Accuracy

We now address the question on whether state-based training can improve the ER accuracy. If training the match rules once already produces perfect ER results, then there is no need to train the match rule multiple times. However, if producing correct ER results depends

Rule	Definition
B_P	$w_0 + w_1 \times S_T + w_2 \times S_A + w_3 \times O_V \geq 0$
B_V	$w_0 + w_1 \times S_N + w_2 \times O_P \geq 0$

Table 4.5: Cora data match rules

on the resolution of other datasets, then our techniques may help.

We evaluate state-based training using real data. We experiment on the Cora dataset, which is a publicly available list of 1,879 papers. From the paper records, we extracted 1,879 venues into a separate list. The gold standard for resolving the papers is given within the Cora dataset. For the venues, we manually created a gold standard by grouping the same venues together.

Table 4.5 shows the match rules for the papers and venues. We denote the paper dataset as P and the venue dataset as V. The paper match rule B_P compares the similarity of titles (S_T), the similarity of author lists (S_A), and the venue overlap (O_V) between papers. The venue match rule B_V compares the similarity of names (S_N) and the paper overlap (O_P) between venues. When computing the similarity for titles and names, we removed special characters and used the Jaro measure [111] to compute a string similarity ranging from 0 to 1. When comparing the author lists, we extracted the last names of the authors, sorted and concatenated them in alphabetical order, and used the Jaro distance to compute the string similarity. We emphasize that our match rules are not designed to be the most accurate rules. There are other attributes (date, publisher, volume, page numbers, etc.) that we could have exploited and better models [82] for constructing the match rules. Hence, we use our simple rules to clearly demonstrate how the joint training and resolution of datasets can significantly enhance accuracy compared to resolving each dataset individually.

To measure accuracy, we compare our ER results with the gold standard. We consider all the input records that clustered together to be identical to each other. For instance, if records r and s clustered into $\{r, s\}$ and then clustered with t, all three records r, s, t are considered to be the same. Suppose that the set G contains the set of record pairs that cluster in the gold standard while set S contains the matching pairs for our algorithm. Then the precision Pr is $\frac{|G \cap S|}{|S|}$ while the recall Re is $\frac{|G \cap S|}{|G|}$. Using Pr and Re, we compute the F_1-measure, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, and use it as our accuracy metric.

Iter.	Rule	State	w_0	w_1	w_2	w_3
Paper weights trained first						
1	B_P	()	-7.89	8.02	0.92	0
2	B_V	(P)	-8.99	9.99	2.22	n/a
3	B_P	(P, V)	-6.71	6.39	0.79	0.76
4	B_V	(P, V, P)	-8.99	9.99	2.22	n/a
Venue weights trained first						
1	B_V	()	-8.42	9.42	0	n/a
2	B_P	(V)	-8.1	7.73	1.36	0.83
3	B_V	(V, P)	-8.99	9.99	2.22	n/a
4	B_P	(V, P, V)	-6.39	5.25	1.44	0.89
5	B_V	(V, P, V, P)	-8.99	9.99	2.22	n/a

Table 4.6: Trained weights

The weight training was done on a random 10% subset of the papers and venues using the LIBSVM package [67]. We train the papers and venues in an alternating fashion. That is, we train the paper weights, and then the venue weights, then the paper weights again, and so on until all the weights converge. The top part of Table 4.6 shows the trained weights of B_P and B_V where the paper weights were trained first. For each state, we have listed the datasets resolved. Notice that after the first iteration, w_3 of B_P is trained to 0 because none of the venues have been merged yet, so comparing the venues of papers does not improve accuracy. The bottom part of Table 4.6 shows the weights when the venues are trained first. This time, in the first iteration, the weight w_2 of B_V is trained to 0 because none of the papers have been resolved and merged yet. Notice that within three resolutions of the papers and venues, we have produced a fixed point joint ER result, which suggests that the number of resolutions per dataset is small in practice.

Table 4.7 shows the trained weights without considering the states of resolved datasets. When training the paper weights, since no venues have merged, the training results are identical to the paper weights after the first iteration in the top part of Table 4.6. Similarly, for the venues, the training results are the same as the venue weights after the first iteration in the bottom part of Table 4.6.

Using the trained weights, we now run the Swoosh algorithm on each dataset following the same sequence as that of the training. Hence if B_P was trained first, we run ER on

Rule	w_0	w_1	w_2	w_3
B_P	-7.89	8.02	0.92	0
B_V	-8.42	9.42	0	n/a

Table 4.7: Trained weights without states

Iteration	**Type**	**Pr**	**Re**	**F1**
Papers resolved first				
1	P	0.83	0.92	0.87
2	V	0.69	0.88	0.78
3	P	0.83	0.96	0.89
4	V	0.69	0.88	0.78
5	P	0.83	0.96	0.89
Venues resolved first				
1	V	0.94	0.54	0.69
2	P	0.83	0.96	0.89
3	V	0.84	0.89	0.86
4	P	0.83	0.96	0.89
5	V	0.84	0.89	0.86

Table 4.8: Accuracy results of training with states

the papers first, then run ER on the venues, and then on the papers, and so on until the ER results converge. If B_V was trained first, we run ER on the venues first. The top part of Table 4.8 shows the progression of accuracy as each dataset is resolved when the papers are resolved first. The bottom part of Table 4.8 shows the corresponding results when the venues are resolved first. As a result, the accuracy of the resolved papers is 0.89 regardless which dataset was resolved first while the accuracy of the resolved venues is 0.78 if the papers were resolved first and 0.86 if the venues were resolved first.

We now compare our state-based training results with the training results without states. Table 4.9 shows the accuracy results when the states are not considered. Notice that the paper and venue results are identical to the first iteration results of the top and bottom parts of Table 4.8, respectively. As a result, using the state information has improved the accuracy of papers by 2% and that of the venues by 9–17%. The larger improvement for venues matches our intuition because many identical venues that have significantly different names (e.g., the names "NIPS" and "Advances in Neural Information Processing

Type	Pr	Re	F1
P	0.83	0.92	0.87
V	0.94	0.54	0.69

Table 4.9: Accuracy results of training without states

Systems" represent the same conference) started to match once they shared many papers.

In summary, we have shown that state-based training can improve the accuracy of ER compared to training without states. In general, state-based training does not always outperform training without states. For example, if training without states already produces perfect match rules, then there is no benefit in retraining the rules. Hence, state-based training is mainly useful when references play a significant role in matching records as in our demonstration.

4.6 Related Work

Most of the ER work [65] has focused on resolving one dataset. In contrast, our approach attempts to resolve multiple datasets of records at the same time, which can significantly improve the accuracy of ER. Although several recent works [109, 83] have proposed general scalable ER algorithms for a single entity type of data, they do not discuss how to coordinate the resolution of multiple types of records that refer to each other.

Many works have considered joint ER focusing on accuracy only. Dong et al. [31] presents an ER method where record pairs of different datasets are resolved simultaneously. Bhattacharya et al. [15] proposes joint ER techniques for a specific domain (citations) using hard-coded ER algorithms. Several probabilistic models for joint ER has been proposed as well. Culotta et al. [27, 28] uses conditional random fields to improve the propagation of information when resolving records of multiple datasets. Domingos et al. [81] supports a restricted version of joint ER where comparison results of basic attributes are shared when resolving records. Markov Logic Networks [90] can also be used to specify a rich set of constraints for joint ER. While the above approaches significantly enhance the accuracy of joint ER, they do not scale to large datasets and do not provide a general framework for custom ER algorithms either.

Recently, Arasu et al. [7] have proposed joint ER techniques based on a declarative language for constraints. Unlike previous approaches, their work focuses on scalability as well as accuracy of joint ER. A declarative set of constraints is reflected in a graph of records with different types of edges that indicate the likelihood of two records matching. Then a correlation clustering algorithm is used to connect records while minimizing the number of violated constraints in the graph. In comparison, our work provides generality where one can simply plug in her application-specific ER algorithm and get joint ER results. In addition, our joint ER algorithm can resolve few datasets at a time based on an execution plan, providing flexibility in resource (e.g., CPU, memory) management.

Job scheduling [18] and software pipelining [3] are related topics to our problem of assigning datasets to processors. The goal is to assign fixed-length jobs (in software pipelining, an instruction is a job) that may depend on each other to processors in order to minimize the parallel runtime. While joint ER can also be viewed as a job scheduling problem, a major distinction is that it is very difficult to predict the runtime of ER on each dataset unlike job scheduling and software pipelining where each job has a fixed runtime. In addition, even if a dataset R influences S, we are not necessarily restricted to resolving S after R unlike in job scheduling and software pipelining where jobs must be processed by strictly following a dependence relation.

Several works have proposed training techniques for ER algorithms. Bilenko et al. [16] provides training for the specific domain of string similarity measures. Sarawagi et al. [87] proposes interactive learning techniques that minimize the work needed to achieve high accuracy of ER. Bhattacharya et al. [14] proposes unsupervised learning techniques for ER. While most of these work perform training once, we propose a state-based training technique where an ER algorithm of a dataset may be trained multiple times based on different resolution states of other datasets.

4.7 Conclusion

When performing entity resolution on multiple types of datasets, resolving records of one type can impact the resolution of other types of records. In this chapter, we have explored the following two questions: Given a limited amount of resources, what is the most

efficient schedule for resolving the datasets? How do we train the ER algorithms? We have answered the first question by proposing a flexible and modular resolution framework where existing ER algorithms that are developed for given record types can be plugged in and used with other ER algorithms. In our experiments, we have demonstrated that our joint ER method can resolve large datasets efficiently by scheduling and coordinating the individual ER algorithms. We have answered the second question by proposing a state-based training technique where ER algorithms can be trained multiple times based on the state of other datasets. We have demonstrated that our state-based training techniques can indeed improve the accuracy compared to training the ER algorithms only once.

Chapter 5

Entity Resolution with Negative Rules

Until now, we have assumed that ER is a "perfect" process in the sense that it cannot make mistakes and always correctly resolves records. In practice, however, the process for matching and merging records in ER may be application-specific, complex, and error-prone. The input records may contain ambiguous and not-fully specified data, and it may be impossible to capture all the application nuances and subtleties in whatever logic is used to decide when records match and how they should be merged. Thus, the set of resolved records (after ER) may contain "errors" that would be apparent to a domain specialist. For example, we may have a customer record with an address in a country we do not do business with. Or two different company records where the expert happens to know that one company recently acquired the other, so they are now the same entity.

An effective way to reduce ER errors is to define *integrity constraints* that should be satisfied by the data [50, 35]. The constraints may be written by people different from the application writers, to avoid making the same mistake twice. After (or while) the application runs, the constraints are independently checked, and inconsistencies flagged. Of course, in an ideal world, the application writers would enforce all integrity constraints perfectly, and integrity checking would be unnecessary. However, we do not live in an ideal world and integrity checking represents a useful "sanity check."

Integrity constraints tell us what data states are invalid but do not tell us how to arrive at a valid state. In this chapter we study how to modify the ER process, in light of some integrity constraints that we call *negative rules*, so that we arrive at a set of resolved records

	Name	SSN	Gender
r_1	Pat	999-04-1234	
r_2	Patricia		F
r_3	Pat	999-04-1234	M

Figure 5.1: A list of people

that satisfy the constraints. Furthermore, since in general there can be more than one valid resolved set, we also discuss how a domain expert can guide the ER process to arrive at a desirable and valid set of records using various methods for resolving records. We also explore properties of the negative rules that make this directed ER process more efficient.

Motivating Example Consider the three people records shown in Figure 5.1, that are to be resolved. We would like to merge records that actually refer to the same person. Suppose the match function compares r_1 and r_2 first and returns a match because they have similar names. Records r_1 and r_2 are thus merged into a new record r_{12}:

r_{12}	Pat, Patricia	999-04-1234	F

Now suppose that r_{12} matches with r_3 since they have similar names and an identical social security number. The result is a new record r_{123}:

r_{123}	Pat, Patricia	999-04-1234	M, F

In this case, r_{123} is the answer of the ER process.

However, it is easy to see there are problems with this solution. These problems can be identified by "negative rules," i.e., constraints that define inconsistent states. In this example, say we have a rule that states that one person cannot have two genders, and hence record r_{123} violates the constraint. The reader may of course wonder why this constraint was not enforced by the merge function that combined r_{12} with r_3. There are two reasons. One reason is that the person writing the merge function may be unaware of this gender constraint or enforced it incorrectly. Keep in mind that the constraints in practice will be much more complex than what our simple example shows. For instance, the merge function (or the negative rule) may be a complex computer program that considers many

factors in making a decision. It may have numerous "patches" added over time by different people. Furthermore, the match and negative rules may be written by different people (as mentioned earlier), so it is not surprising that the rules can reach conflicting decisions.

A second reason why the gender constraint was not enforced by the merge function may be that the constraint is "fixable". In this application it may be acceptable to have a record with two genders *during* the resolution (as opposed to in the final answer), because future merges may resolve the gender. For example, say r_{123} were to merge with another record that indicated that Pat was Male. Then the merge function may eliminate the Female gender because there is now more evidence that Pat is male. In this scenario it is okay to temporarily generate r_{123} since it is useful in constructing a valid final record. However, it is not okay to leave r_{123} in the final answer.

To resolve the gender inconsistency, say we unmerge r_{123} back into $\{r_{12}, r_3\}$. In our example, the set $\{r_{12}, r_3\}$ may still not be a valid ER answer: We may have a negative rules stating that no two final records should have the same social security number. In our case, the problem occurred because r_1 was initially merged with r_2 instead of r_3.

The reader may wonder why the ER process did not first merge r_1 and r_3 since they are clearly a better match than r_1, r_2. First, our example is simple, and in practice there may be no obvious ordering to the merges. Furthermore, the person coding the match function may not be aware of the SSN check that will be performed by the negative rule. Second, an inherent feature (some would say weakness) of pairwise matching is that merge decisions are done without global analysis, a pair of records at a time. This feature is what makes the approach simple and appealing to some applications, but is also the feature that can introduce problems like the one illustrated by our example. Our approach here will be to fix these problems via the definition of negative rules.

In our simple example, we can arrive at two possible solutions that satisfy the negative rules presented above. One solution occurs when we unmerge r_{12} and re-merge r_1 and r_3, resulting in $\{r_{13}, r_2\}$. The other is when we simply discard r_3, resulting in $\{r_{12}\}$. Note that $\{r_1, r_2\}$ is not a good solution because it is not "maximal," i.e., r_1 and r_2 could have been merged without problems. The precise definition of a valid solution will be given in the next section.

Interestingly, many inconsistencies in real-world data can be captured with negative

rules that examine one or two records at a time. For example, we can easily apply our rules to hotel data saying that no hotel can have two different street numbers on the same street and that no two hotels with different names can have the same street name, street number, and phone number.

In this chapter we address precisely the identification and handling of inconsistent ER answers.

- We start by summarizing the ER model of this chapter (Section 5.1.1).
- We then define the concept of negative rules (Section 5.1.2), both *unary negative rules* that detect internal inconsistencies within one record, and *binary negative rules* that detect problems involving a pair of records (as in our example).
- We formally define what is the correct ER answer in the presence of such negative rules (Section 5.1.3).
- We define simple properties of the match, merge, and negative rules that make it easier to find the correct solutions (Section 5.3), and we present algorithms that find a solution based on guidance from a domain expert (Sections 5.2, 5.4).
- We experimentally evaluate our algorithms using actual comparison shopping data from Yahoo! Shopping and hotel information data from Yahoo! Travel (Sections 5.5, 5.6).
- We discuss related work in Section 5.7 and conclude in Section 5.8.

5.1 ER-N Model

In this section, we formalize negative rules and define correct ER results based on the negative rules. We then introduce two approaches for avoiding inconsistencies.

5.1.1 ER

We start with an instance $I = \{r_1, \ldots, r_n\}$, which is a set of records.

Match and Merge Functions A *match* function M determines if two records r_1 and r_2 refer to the same real-world entity. If the records match, $M(r_1, r_2)$ = `true`. We denote this as $r_1 \approx r_2$. Otherwise, $M(r_1, r_2)$ = `false` ($r_1 \not\approx r_2$).

A *merge* function μ merges two records into one. The function is only defined for matching records. The result of $\mu(r_1, r_2)$ is denoted as $\langle r_1, r_2 \rangle$.

We assume two basic properties for M and μ – idempotence and commutativity. Idempotence says that any record matches itself, and merging a record with itself yields the same record. Commutativity says that, if r_1 matches r_2, then r_2 matches r_1. Additionally, the merged results of r_1 and r_2 should be identical regardless of the merge ordering.

- *Idempotence:* $\forall r,\ r \approx r$ and $\langle r, r \rangle = r$.
- *Commutativity:* $\forall r_1, r_2, r_1 \approx r_2$ iff $r_2 \approx r_1$, and if $r_1 \approx r_2$, then $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.

We believe that most match and merge functions will naturally satisfy these properties. Even if they do not, they can easily be modified to satisfy the properties. To illustrate the second point, suppose that idempotence does not hold because the records have very little information (e.g., a person named John isn't necessarily identical to another person named John). In that case, we can be more strict in determining if two records are the same by conducting a bitwise comparison between the records or comparing the sources from which the records originated.

Merge Closure A merge closure $\bar{R}$ contains all the possible records that can be generated from R using M and μ.

Definition 5.1.1. *The* merge closure $\bar{R}$ *of* R *satisfies the following conditions:*

1. $R \subseteq \bar{R}$
2. $\forall r_1, r_2 \in \bar{R}$ *s.t.* $r_1 \approx r_2$, $\langle r_1, r_2 \rangle \in \bar{R}$.
3. *No strict subset of* $\bar{R}$ *satisfies conditions 1,2.*

We present an algorithm for computing $\bar{R}$ in Algorithm 9. (It is shown in [10] that Algorithm 9 is optimal in the sense that no algorithm makes fewer record comparisons in the

ALGORITHM 9: Computing the merge closure ($\bar{R}$)

1: **input:** a set R of records
2: **output:** the merge closure of R, $\bar{R}$
3: $\bar{R} \leftarrow \emptyset$
4: **while** $R \neq \emptyset$ **do**
5: $\quad r \leftarrow$ a record from R
6: $\quad$ remove r from R
7: $\quad$ **for all** records r' in $\bar{R}$ **do**
8: $\quad\quad$ **if** $r \approx r'$ **then**
9: $\quad\quad\quad merged \leftarrow \langle r, r' \rangle$
10: $\quad\quad\quad$ **if** $merged \notin R \cup \bar{R} \cup \{r\}$ **then**
11: $\quad\quad\quad\quad R \leftarrow R \cup \{merged\}$
12: $\quad \bar{R} \leftarrow \bar{R} \cup \{r\}$
13: **return** $\bar{R}$

worst case.) Note that the merge closure can possibly be infinite if a chain of merges produces new records indefinitely. In Section 5.3, we will present some additional properties for M and μ that prevent this case.

Domination We now define domination between records. Record r_1 is dominated by r_2 if both records refer to the same entity, but r_2's information "includes" that of r_1. That is, r_1 is redundant information and should be subsumed by r_2. What records dominate others is application dependent. We can assume that for a given application there is some partial order relation (i.e., a reflexive, transitive, and anti-symmetric binary relation) that tells us when domination exists. The domination of r_1 by r_2 is denoted as $r_1 \preceq r_2$. For example, in some application where merges simply collect all information in records, we may have $r_1 \preceq r_2$ whenever $r_2 = \langle r_1, r' \rangle$ (for some r'). We will use this domination in our examples unless stated otherwise. In Section 5.3, we present a canonical domination order that holds when some additional properties for M and μ are satisfied.

Domination on records can be naturally extended to instances as follows:

Definition 5.1.2. *Given two instances R_1, R_2, we say that R_1 is dominated by R_2 (denoted as $R_1 \preceq R_2$) if $\forall r_1 \in R_1, \exists r_2 \in R_2$ s.t. $r_1 \preceq r_2$.*

5.1.2 Negative Rules

A negative rule is a predicate that takes an arbitrary number of records and returns either `consistent` or `inconsistent`. Negative rules can be categorized according to their numbers of arguments. In our work, we consider unary and binary negative rules.

A unary negative rule N_1 checks if a record r is valid by itself. If r is internally inconsistent, $N_1(r)$ = `inconsistent` (denoted as $r \nleftrightarrow r$). Otherwise, $N_1(r)$ = `consistent` (denoted as $r \leftrightarrow r$). An internally inconsistent record should not exist in an ER solution.

A binary negative rule N_2 checks if two different records r_1 and r_2 can coexist. We require r_1 and r_2 to be different in order to make a clean distinction between unary and binary negative rules. If r_1 and r_2 are inconsistent, $N_2(r_1, r_2)$ = `inconsistent` (denoted as $r_1 \nleftrightarrow r_2$). Otherwise, $N_2(r_1, r_2)$ = `consistent` (denoted as $r_1 \leftrightarrow r_2$). Two inconsistent records cannot coexist in an ER solution.

Neither type of negative rules can be incorporated into the match and merge functions. As we illustrated in the beginning of this chapter, a unary negative rule cannot be supported by simply disallowing two records to merge into an internally inconsistent record because inconsistencies could be fixed in the future. Binary negative rules also do not fit in the match and merge functions for the same reason. Moreover, a match function only has a local view of two records and cannot tell whether the merged record will generate any new binary inconsistencies with other records "outside." Thus, negative rules cannot be enforced by modifying the match and merge functions.

We say that a set of records is inconsistent if there exists a single record violating a unary negative rule or a pair of records violating a binary negative rule.

We assume the basic commutativity property for negative rules. That is, if r_1 is inconsistent with r_2, then r_2 is also inconsistent with r_1.

- *Commutativity (Negative Rule):*$\forall r_1, r_2$ s.t. $r_1 \nleftrightarrow r_2$, then $r_2 \nleftrightarrow r_1$.

Finally, the negative rules are black-box functions that can be implemented in any way as long as they satisfy commutativity.

5.1.3 ER-N

We now formally define entity resolution with negative rules.

Definition 5.1.3. *Given an instance R and the merge closure, $\bar{R}$, an ER-N of R is a consistent set of records J that satisfies the following conditions:*

1. $J \subseteq \bar{R}$,
2. $\forall r \in \bar{R} - J$, *either*
   - $\exists r' \in J$ *s.t.* $r \preceq r'$ *or*
   - $J \cup \{r\}$ *is inconsistent,*
3. *No strict subset of J satisfies conditions 1 and 2.*
4. *No other instances satisfying conditions 1,2, and 3 dominate J.*

Intuitively, J is a maximal consistent subset of $\bar{R}$ (The first three conditions of Definition 5.1.3 imply that J is consistent; the proof can be done by contradiction). The second condition ensures the maximality by saying that any record from $\bar{R}$ that is not in J is either dominated by a record in J or introduces an inconsistency to J. The third condition ensures that J is consistent and has no dominated records. Lastly, the fourth condition filters out "undesirable" solutions that are dominated by other solutions. Returning to our example in Figure 5.1, suppose that every pair of records match and that the merge closure $\bar{R}$ is $\{r_1, r_2, r_3, r_{12}, r_{13}, r_{23}, r_{123}\}$. The instance $\{r_{13}, r_2\}$ is a valid ER-N solution because 1) $\{r_{13}, r_2\}$ is a subset of $\bar{R}$; 2) any other record from $\bar{R}$ (i.e., $r_1, r_3, r_{12}, r_{23}, r_{123}$) is either dominated by a record in $\{r_{13}, r_2\}$ ($r_1 \preceq r_{13}$, $r_3 \preceq r_{13}$) or introduces an inconsistency (unary: $r_{23} \nleftrightarrow r_{23}$, $r_{123} \nleftrightarrow r_{123}$; binary: $r_{12} \nleftrightarrow r_{13}$); 3) $\{r_{13}, r_2\}$ is consistent, so no records can be dropped; and 4) $\{r_{13}, r_2\}$ is not dominated by the only other solution, $\{r_{12}\}$. The instance $\{r_{12}\}$ is also a valid solution for the same reasoning. To clarify the role of the fourth condition (i.e., the first three conditions do not imply the fourth condition), notice that the instances $\{r_1, r_2\}$ and $\{r_2, r_3\}$ satisfy the first three conditions, but are dominated by the solution $\{r_{13}, r_2\}$. Hence, $\{r_1, r_2\}$ and $\{r_2, r_3\}$ are not valid solutions.

5.1.4 Resolving Inconsistencies

There are two general approaches for resolving records in the presence of negative rules:

- *Late Approach.* The merge and match functions are used to generate a set $ER(R)$, which is after-the-fact checked for inconsistencies. As inconsistencies are discovered, appropriate "fixes" (see below) are taken, with the guidance of a domain expert. We call this domain expert the *solver*, to differentiate this person from that ones writing match, merge and negative rules.

- *Early Approach.* With the help of a solver, we start identifying records that we want to be in the final answer J. Even before the final answer is known, we start "fixing" problems between the selected records in J and other records not yet selected.

In this chapter, we follow an early approach because the late approach involves backtracking (i.e., unmerging records), which can be very expensive. There are several ways inconsistencies can be "fixed" with the help of the solver:

- *Discard Data.* When an inconsistency is detected, the solver may decide to drop one of the records causing the problem. The dropped record will not be in the final answer.

- *Forced Merge.* The solver decides that two inconsistent records should have been merged and manually forces a merge. That is, it is deemed that the match function made a mistake. For example, if two hotels Comfort Inn and Comfort Inn Milton are the same hotels but mistakenly not matched by the match function, the negative rule could flag an inconsistency (because the names are suspiciously similar), and the solver could merge them.

- *Override Negative Rule.* The solver decides that the flagged record(s) are consistent after all, i.e., the negative rule was incorrect in flagging an error. For example, Comfort Inn and Comfort Inn Milton, which were flagged by the negative rule to be suspiciously similar, might be different hotels after all. The records(s) are then allowed in the final answer.

When we present our algorithms (Sections 5.2 and 5.4), we will use a Discard technique. However, after each algorithm, we summarize the changes that are necessary to

handle the other two approaches. In our experimental sections (Sections 5.5 and 5.6) we will address the accuracy and performance of the three approaches.

Note incidentally that with the Forced Merge and the Override NR approaches, we should also modify Definition 5.1.3 slightly, so that overridden negative rules do not count as inconsistencies, and so that forced merges are considered valid.

5.2 The GNR Algorithm

The GNR algorithm (General algorithm for Negative Rules) assumes the basic properties in Section 5.1 and that $\bar{R}$ is finite. We also assume that a solver makes decisions when there is a choice to be made. The solver looks at the records, and selects one that is "more desirable" to have in the final answer. If no solver is available, the algorithm could make the choice at random or based on heuristics (e.g., a record with more data fields is preferable to one with fewer). With human intervention, the algorithm will be guided to one of the possible solutions that is acceptable to the solver; without such guidance, the algorithm will still find a valid ER-N solution, but the solution may not be the "most desirable."

In our algorithm, the solver starts by choosing the non-dominated records from $\bar{R}$. The management of inconsistencies and domination are done by the algorithm. The algorithm is shown in Algorithm 10. The merge closure $\bar{R}$ is computed using Algorithm 9. Notice that we can automatically choose records that are non-dominated and consistent with every record in S because they will eventually be chosen by the solver.

To illustrate how the GNR algorithm works, we again refer to our motivating example in Figure 5.1. Again, assume that $\bar{R}$ (and thus S) is $\{r_1, r_2, r_3, r_{12}, r_{13}, r_{23}, r_{123}\}$. Since we assume that $r_i \preceq r_j$ whenever r_i was used to generate r_j, there is only one non-dominated record in $\bar{R}$, namely r_{123}. Thus, there is really no choice for the solver but to select r_{123} for the first iteration. However, r_{123} is internally inconsistent and is discarded (step 9). For the second iteration, the solver has a choice among $\{r_{12}, r_{13}, r_{23}\}$. Suppose the solver chooses r_{13}. At step 10, r_{13} is included in J. Then the records that are dominated by or inconsistent with r_{13} are removed from S, leaving $S = \{r_2, r_{23}\}$. Choosing r_2 and discarding r_{23} (since r_{23} is internally inconsistent) results in our final solution $\{r_{13}, r_2\}$. Notice that, if the solver had chosen r_{12} during the second iteration, the final solution would have been $\{r_{12}\}$.

ALGORITHM 10: The GNR Algorithm

1: **input:** a set R of records
2: **output:** J = ER-N(R)
3: $S \leftarrow \bar{R}$ /*Computed with Algorithm 9*/
4: $J \leftarrow \emptyset$
5: **while** $S \neq \emptyset$ **do**
6: $ndS \leftarrow$ the non-dominated records in S
7: $r \leftarrow$ a record from ndS chosen by the solver
8: $S \leftarrow S \diagdown \{r\}$
9: **if** $r \not\leftrightarrow r$ **then** continue (next iteration of loop)
10: $J \leftarrow J \cup \{r\}$
11: **for all** $r' \in S$ **do**
12: **if** $r' \not\leftrightarrow r$ **or** $r' \preceq r$ **then**
13: $S \leftarrow S \diagdown \{r'\}$
14: return J

Proposition 5.2.1. *The GNR algorithm returns a valid ER-N solution.*

Proof. The solution J should satisfy the four conditions in Definition 5.1.3. First, J is a subset of $\bar{R}$ because we are not creating any new records. Second, each record r in $\bar{R}$ that is not in J was discarded (step 9) due to an internal inconsistency or deleted from S (step 13) because r was either inconsistent with or dominated by a record being inserted into J. Third, no stricter subset of J satisfies the second condition because any r removed from J is not dominated by a record in $J \diagdown \{r\}$ and does not introduce an inconsistency to $J \diagdown \{r\}$. Finally, no other solution dominates J: Suppose there exists such a solution J' (i.e., $J \preceq J'$). Let [r_{s_1},r_{s_2},...,$r_{s_{|J|}}$] be the records of J ordered by when they were added to J by Algorithm 10. Looking at r_{s_1}, we know that r_{s_1} must also exist in J' because r_{s_1} is a non-dominated record in $\bar{R}$ (ignoring internally inconsistent records), and there exists a record in J' that dominates r_{s_1} (i.e., the record that dominates r_{s_1} can only be r_{s_1}). Next, define S_1 as the records in $\bar{R}$ that are neither dominated by nor inconsistent with r_{s_1}. (Note that S_1 is what remains of the original S, after the first iteration of Algorithm 10.) According to the second condition of ER-N, no record outside S_1 can be in $J \diagdown \{r_{s_1}\}$ or $J' \diagdown \{r_{s_1}\}$. Now looking at r_{s_2}, we can see that r_{s_2} must also exist in J' because r_{s_2} is a non-dominated record in S_1 (ignoring internally inconsistent records) and there exists a record in $J' \diagdown \{r_{s_1}\}$ that dominates r_{s_2}. After iterating through all the records of J in a similar fashion, we can

see that J is a subset of J'. Moreover, J' cannot have more records than J according to the third ER-N condition. Thus, we conclude that $J = J'$, which contradicts the assumption that the two instances are different. In conclusion, the GNR algorithm returns a valid ER-N solution. □

While the GNR algorithm discards records to resolve inconsistencies (see Section 5.1.4), it can also use alternative strategies for resolving records. First, the algorithm can be extended to support forced merges. Once the solver chooses a record (step 7), that record is compared with every record in the set S for new inconsistencies. The solver can then view all the inconsistent pairs detected in step 12 and manually merge the records that should have been merged. After the merges, we can re-run the merge closure to identify additional matches that occur. While this step guarantees accuracy, it can be very expensive. An alternative approach is to simply continue after the forced merge without re-running the merge closure.

Second, the GNR algorithm can also support overriding of negative rules using a similar process as for forced merges. Looking at the new inconsistencies in steps 9 and 12, the solver can manually override the inconsistencies that are considered incorrect. The decisions of the solver can be stored in a hash table along with the records involved. (Thus, two records are inconsistent only if the binary inconsistency rule says they are, and the pair of records is not in the override hash table.)

Finally, the solver can use a combination of all three strategies to resolve inconsistencies. For unary inconsistencies in step 9, the solver can either discard the record or override the negative rule. For binary inconsistencies in step 12, the solver can use one of the three strategies. If the binary rule is incorrect, the solver overrides the negative rule. If the binary rule is correct but merging the two records results in an inconsistent record, the solver discards a record. However, if the merging does not introduce an inconsistency then the solver uses the forced merge technique.

Human Effort An important metric for the GNR algorithm is the "human effort" made by the solver. Of course, human effort is very hard to model and is seldom quantified in our database community. Nevertheless, because the human solver plays a key role in entity

resolution with negative rules, we feel it is important to analyze human effort, even if our metric is far from perfect.

There are three ways the solver can be involved in the algorithm. First, the solver must choose records from the set ndS (step 7). Second, the solver must check whether a record is internally inconsistent during step 9 (but only if the unary negative rule returns `inconsistent`). Third, the solver must check a pair of records for inconsistencies during step 12 (only if the binary negative rule returns the result `inconsistent`). The effort made for each type of effort will vary depending on the strategy used by the solver.

Since it is difficult to predict the behavior of the solver, we use the following simple model as a surrogate of the human effort. For checking unary rules, we simply count the number of checked records. For binary rules, we count the number of pairs checked. For choosing records, the cost of selecting one record from a set of records ndS (step 7 in Algorithm 10) is $|ndS|$. The total human cost for choosing records is then the sum of costs for all such selections. For example, given a set of ten records with no inconsistencies or domination relationships, the total human effort for choosing all the ten records is $10 + 9 + ... + 2 = 54$. Notice that we do not count the effort for choosing the last record.

We caution that the human effort values we present, by themselves, are not very useful. The actual human effort will vary depending on the strategies we use for resolving inconsistencies. For example, the Discard Data and Forced Merged strategies can be run automatically and save most of the human effort while using a combination of strategies might require a significant amount of human effort (see Section 5.5.3). However, we believe the human effort values can be helpful in comparisons. For instance, if in Scenario A the cost is 10 times that in Scenario B, then we can infer that the solver will be significantly more loaded in Scenario A.

5.3 Properties for the Rules

Entity resolution is an inherently expensive operation (especially with negative rules) regardless of the solution used. In general, practitioners use two types of techniques to reduce the cost: blocking and exploiting properties.

If blocking techniques produce relatively small blocks, the GNR algorithm will be feasible. Also, note that the GNR algorithm becomes more attractive in scenarios where there are relatively few matches. (The more matches, the larger $\bar{R}$ becomes.)

A second general approach to reducing cost is to exploit properties of the match and merge functions to make it possible to find the correct solution with less effort. In this section we present such desirable properties: two for match and merge functions, and one for negative rules. In Section 5.4 we then use these properties to make the ER process significantly more efficient.

Of course, note that the properties we propose will not hold in all applications. If the properties do hold, then one will be able to achieve the improved performance. If the properties do not naturally hold, the solver may want to modify the rule so that the properties hold (e.g., by keeping more information in a merged record, one may be able to achieve the representativity property defined below). Finally, if the properties below definitely do not hold in a given application, the solver may nevertheless still want to use the efficient algorithm of Section 5.4, in order to get an answer in a reasonable time. The answer will *not* be correct because of the "wrong" algorithm we used for this case, but the answer may be "relatively close" to the correct answer.

The bound of incorrectness depends on the portion of "problematic" records that do not make the rules satisfy the properties. For example, the initial set of records R could conceptually be divided into two sets $X = \{x_1, x_2, \ldots, x_n\}$ and $Y = \{y_1, y_2, \ldots, y_m\}$ where the properties are satisfied when resolving the records in X while not necessarily so when resolving the records in Y. That is, Y includes all the records that could possibly generate inconsistencies. We suspect that the "incorrectness" of the ER solution would then be bounded by the fraction $\frac{Y}{X+Y}$ of the total records in the ER solution. In practice, the problematic record set Y is only a small fraction of the entire set of records (see Section 5.6.3). Further research is required to refine this intuitive "incorrectness" bound.

5.3.1 Match and Merge Functions

Two desirable properties for M and μ are associativity and representativity. Associativity says that the merge order is irrelevant. Representativity says that a merged record represents

its base records and matches with all records that match with the base records.

- *Associativity:* $\forall r_1, r_2, r_3$ such that $\langle r_1, \langle r_2, r_3 \rangle \rangle$ and $\langle \langle r_1, r_2 \rangle, r_3 \rangle$ exist, $\langle r_1, \langle r_2, r_3 \rangle \rangle = \langle \langle r_1, r_2 \rangle, r_3 \rangle$.
- *Representativity:* If $r_3 = \langle r_1, r_2 \rangle$ then for any r_4 such that $r_1 \approx r_4$, we also have $r_3 \approx r_4$.

Associativity and representativity together are somewhat strict, but powerful properties. Combined with the two basic properties, idempotence and commutativity, they are called the ICAR properties. It is shown in [10] that, given the ICAR properties, the merge closure of R is always finite.

Union Class of Match and Merge Functions There is a broad class of match and merge functions that satisfy the ICAR properties because they are based on union of values. We call this class the *Union Class*. The key idea is that each record maintains all the values seen in its base records. For example, if a record with name {John Doe} is merged with a record with name {J. Doe}, the result would have the name {John Doe, J. Doe}. Unioning values is convenient since we record all the variants seen for a person's name, a hotel's name, a company's phone number, and so on. Keeping the "lineage" of our records is important in many applications, and furthermore ensures we do not miss future potential matches. Notice that the actual presentation of this merged record to the *user* does not have to be a set, but can be any string operation result on the possible values (e.g., {John Doe}). Such a strategy is perfectly fine as long as the records only use the "underlying" set values for matching and merging. Two records match if there exists a pair of values from the records that match. In our example, say the match function compares a third record with name {Johnny Doe} to the merged record obtained earlier. If the function compares names, then it would declare a match if Johnny Doe matches either one of the two names. The match and merge functions in this Union Class satisfy the ICAR properties as long as the match function is reflexive and commutative (two properties that most functions have)

Beyond the Union Class, there are other rules that while not strictly in this class, also record in some way all the values they have encountered. For example, a record may represent the range of prices that have been seen. If the record is merged with another

record with a price outside the range, the range is expanded to cover the new value. Thus, the range covers all previously encountered values. Instead of checking if the prices in the records match exactly, the match function checks if price ranges overlap. It can be shown that match and merge functions that keep all values explicitly or in ranges also satisfy the ICAR properties.

Merge Domination If the ICAR properties are satisfied, we can use a natural domination order called merge domination.

Definition 5.3.1. r_1 is merge dominated by r_2 *(denoted* $r_1 \leqslant r_2$*), if* $r_1 \approx r_2$ *and* $\langle r_1, r_2 \rangle = r_2$.

Reference [10] shows that merge domination is a partial order on records given the ICAR properties. Merge domination is a natural way of ordering records and will be our default domination order when the ICAR properties hold.

5.3.2 Negative Rules

One desirable property for negative rules is called persistence. In many applications, inconsistencies tend to hold regardless of future merges. Persistence is defined for both unary and binary negative rules.

Unary persistence is defined on unary negative rules. The property states that an internally inconsistent record r stays inconsistent regardless of its merging with other records.

Binary persistence is defined on binary negative rules. This time, two inconsistent records r_1 and r_2 stay inconsistent regardless of their merging with other records. The only exception is when r_1 and r_2 merge together, either directly or indirectly. In that case, the binary inconsistency is resolved because the two records no longer coexist ($\langle r_1, r_2 \rangle$ could be internally inconsistent).

- *Unary Persistence:* If $r_1 \nleftrightarrow r_1$ and $r_3 = \langle r_1, r_2 \rangle$, then $r_3 \nleftrightarrow r_3$.
- *Binary Persistence:* If $r_1 \nleftrightarrow r_2$ and $\langle r_1, r_3 \rangle \neq r_2$, then $\langle r_1, r_3 \rangle \nleftrightarrow r_2$.

ALGORITHM 11: The ENR Algorithm

1: **input:** a set R of records
2: **output:** J = ER-N(R)
3: $P \leftarrow$ ER(R) /*e.g., using R-Swoosh*/
4: $C \leftarrow$ the set of connected components of inconsistent packages in P
5: **for all** $c_i \in C$ **do**
6: $J_i \leftarrow$ GNR($\bigcup_{p \in c_i} b(p)$)
7: $J = J_1 \cup ... \cup J_{|C|}$
8: return J

We believe persistence holds in many applications. Unary persistence mostly holds if the merge function is in the Union Class. For example, a hotel having two addresses will still have at least two addresses after merging with other records. Binary persistence is also reasonable – two hotels having the same address will still have the same address regardless of their merging with other hotels.

5.4 The ENR Algorithm

The ENR algorithm (Enhanced algorithm for Negative Rules; shown in Algorithm 11) exploits the properties in Section 5.3 (i.e., the ICAR and Persistence properties) to make the GNR algorithm efficient. Rather than looking at the entire merge closure of R, we would like to partition R and look at the merge closure of each partition. Note that the partitions here are different from the components produced by blocking techniques (see Section 5.3). Specifically, we do not assume any semantic knowledge, as exploited by blocking techniques. The partitioning can be done in two steps. First, we partition R into "packages" (introduced in [72] in another context) where two records generated from different packages do not match. Next, we deal with inconsistencies by connecting "inconsistent packages" into connected components so that two records generated from different components are always consistent with each other.

Packages partition R such that no two records generated from different packages match. (The two generated records may be inconsistent.) The (base) records of package p are denoted as $b(p)$, and the entire set of generated records (i.e., the merge closure) of p is denoted as $c(p)$. All the records in p can merge into a single representing record, which we

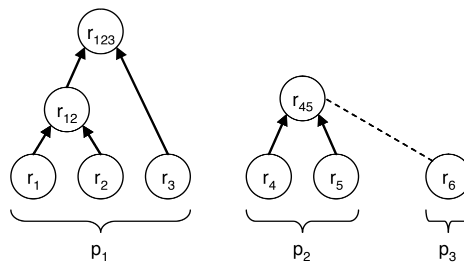


Figure 5.2: Package formation

denote as $r(p)$.

Packages are generated by running Algorithm 9, except that when we merge two records r and r' (step 9), we remove r and r' from further consideration. (Because of the ICAR properties any future record that would have matched r or r' will now match the merged record.) Furthermore, we do not explicitly remove dominated records at the end; the above optimization takes care of that. These two optimizations (plus a few other improvements) yield what is called the R-Swoosh Algorithm, which is studied in detail in [10]. From our point of view, the important point is that packages can be computed efficiently, given the ICAR properties and an algorithm like R-Swoosh.

Figure 5.2 illustrates the package formation step (ignore the dotted line for now). The bottom records are the input records, and the arrows show the merges that occur. In this example, three packages result. For instance, the leftmost package has record r_{123} as representative.

We next connect inconsistent packages together, forming connected components of inconsistent packages. We say two packages p and p' are inconsistent if their representing records, $r(p)$ and $r(p')$, are inconsistent. In our example in Figure 5.2, packages p_2 and p_3 are inconsistent because r_{45} and r_6 are inconsistent (dotted line). As a result, package p_1 forms one component by itself while packages p_2 and p_3 together form another component. To give an illustration why p_2 and p_3 should be connected although r_{45} and r_6 do not match, it could be the case that the name of the same hotel was written in different languages for r_{45} and r_6. While the match function might have considered the two records different because of the different names, the negative rule could help fix that error by connecting p_2

and p_3. Proposition 5.4.1 shows that no two records generated from two consistent packages are inconsistent. Thus, there are no inconsistencies between records generated from different components.

Proposition 5.4.1. *Consider two consistent packages p, p', i.e., $r(p) \leftrightarrow r(p')$. Then $\forall r_1 \in$ c(p), $r_2 \in$ c(p'), $r_1 \leftrightarrow r_2$.*

Proof. Suppose that $r_1 \not\leftrightarrow r_2$. By the definition of packages, r_1 and r_2 can each merge with other records into $r(p)$ and $r(p')$, respectively. Then according to binary persistence, $r(p)$ $\not\leftrightarrow r(p')$, which is a contradiction. □

Finally, we run the GNR algorithm on the records of each connected component of packages. Returning to our example in Figure 5.2, the first component contains the package p_1. Thus, we run the GNR algorithm on $b(p_1)$ = $\{r_1,r_2,r_3\}$. Notice that the solver only has to look at the merge closure of three records instead of the original six. Next, we run the GNR algorithm on the records of the second component containing package p_2 and p_3. In this case, we start with the set $b(p_2) \cup b(p_3)$ = $\{r_4,r_5,r_6\}$. Combining the results of running the GNR algorithm on the two components gives us the final ER-N solution.

Proposition 5.4.2. *The ENR algorithm returns a valid ER-N solution.*

Proof. It is sufficient to prove that running the ENR algorithm on R is equivalent to running the GNR algorithm on R. Adding all the merge closures of the partitions of R produced by the ENR algorithm results in $\bar{R}$ because records generated from different components are independent, i.e., they are consistent with each other and never match. Thus, the solver is looking at the same $\bar{R}$ for both algorithms. Rn the ENR algorithm, however, the solver is handling one subset of $\bar{R}$ at a time. □

While the ENR algorithm assumes the Discard approach, it can also support alternative strategies for resolving records. Since the ENR algorithm only plays a role in isolating inconsistencies, the actual algorithmic changes are all done on the GNR algorithm. Hence, the ENR algorithm does not change regardless of the strategy used.

5.5 Precision and Recall

To evaluate our GNR and ENR algorithms, there are two sets of issues to consider: accuracy and performance. In this section we consider accuracy, i.e., how and by how much can precision and recall of a solution be improved by using negative rules and our algorithms. In the following section we address the performance, i.e., the human effort and system runtime needed for resolving records with negative rules.

5.5.1 Experimental Setting

We ran our experiments on a hotel dataset provided by Yahoo! Travel. In this application, hundreds of thousands of records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users. Because of the volume of data, we used blocking techniques (see Section 5.3) to partition the data into independent blocks and then applied our algorithms on each block. In our experiments, we used a partition containing hotels in the United States; we will call these U.S. hotels from now on.

To evaluate accuracy, we used a "Gold Standard" G also provided by Yahoo. Gold standard G is a set of record pairs. If a pair (A, B) is in G, then input records A and B are considered by a domain expert to be the same hotel. If a pair A, B is not in G, then A and B represent different hotels. Set G turns out to be transitive, i.e., if (A, B) and (B, C) are in G, then (A, C) is also in G.

To evaluate an ER-N solution we proceed as follows. We consider all the input records that merged into an output record to be identical to each other. For instance, if hotels A and B merged into $\langle A, B\rangle$ and then merged with C, all three hotels are considered to be the same. Let S be the set of all pairs found to be equal. In our example, (A, B), (B, C) and (A, C) are all in S. Then the precision Pr is $\frac{|G \cap S|}{|S|}$ while the recall Re is $\frac{|G \cap S|}{|G|}$. In addition, we also used the F_1-measure, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, as a single metric for precision and recall.

The GNR and ENR algorithms were implemented in Java, and our experiments were run on a 2.0GHz Intel Xeon processor with 6GB of memory. Though our server had multiple processors, we did not exploit parallelism.

5.5.2 Rules

Since we did not have access to the proprietary code in Yahoo's match and merge functions, we developed our own rules, based on our understanding of how hotel records are handled. Our rules are union rules, as described in Section 5.3.1. That is, our merge function μ retains all the distinct values of the base records.

The match function M compares two hotel records using eight attributes: name, street address, city, state, zip, country, latitude, and longitude. When comparing two records, we do pairwise comparisons between all the possible attribute values from each record and look for a match. The names and street addresses are first compared using the Jaro-Winkler similarity measure [1] [60], to which a threshold T_M from 0 to 1 is applied to get a yes/no answer. We use the same threshold T_M for comparing names and addresses because they have similar string lengths. If the names and street addresses match, M returns `true` if at least one of the following holds:

- the cities, states, and countries are exactly the same.
- the zip codes and countries are exactly the same.
- the latitude and longitude values do not differ more than 0.1 degree (which corresponds to approximately 11.1km).

It is easy to show that the union operation for merging and existential comparison for matching guarantee the ICAR properties.

For our experiments we used two types of negative rules. Here we describe the first type, and the second type is discussed later on in this section. Our initial negative rules are based on the phone number attribute. This attribute is not as robust as say hotel name or city and zip code for determining matches, but is useful for detecting anomalies that should be checked by the solver.

In particular, our initial unary negative rule N_1 flags a hotel with different phone numbers. In order to precisely compare phone numbers, we first remove non-numeric characters (e.g., '(','), and '-'). We then compare each digit starting from the last position until we

[1] The Jaro-Winkler similarity measure returns a similarity score in the 0 to 1 range base on many factors, including the number of characters in common and the longest common substring.

have compared all the digits of either one of the phone numbers. We compare from the last digit because some phone numbers include area codes while others do not. For example, we consider "(650)123-4567" and "1234567" to be equal by trimming the first phone number into "6501234567" and then comparing the last seven digits. This strategy works very well in our dataset.

Our initial binary negative rule N_2 checks if two hotels have the same phone number. That is, N_2 does a pairwise phone number comparison between all the possible phones of the two records, looking for existing matches. N_2 uses the same phone number comparison function as N_1.

5.5.3 Strategies

We first resolved the records without using the negative rules, using only M and μ (i.e., just step 3 of the ENR Algorithm). We used as input 5,000 U.S. hotel records, and we used various thresholds for T_M. The solid line in Figure 5.3 shows the precision and recall curve for each threshold we used (ignore the other data points for now). Among them, the threshold that produces the highest F_1-measure is 0.74, and the point using that threshold is marked as the "Best Point." To give an idea on how many records actually merged together in the Best Point result, we show in Figure 5.4 the distribution of base records per output record. While most input records did not merge with any other record, a significant portion of the output records were formed by a merge of two input records.

Discard Strategy Next we ran the ENR Algorithm, using the negative rules and the threshold ($T_M = 0.74$) for the match function that yielded the Best Point. Recall that with the Discard strategy, the solver only has to select records for the final result. Any negative rule violations are simply corrected by removing records. When choosing records during step 7 in Algorithm 10, we emulated the solver's decisions by always selecting the record containing the largest number of base records from the set ndS.

The resulting precision and recall of the Discard strategy is shown in Figure 5.3. (Note that the dark triangle for the Discard strategy overlaps with the square for a scheme that is described below. Both schemes have approximately the same performance.) Compared to

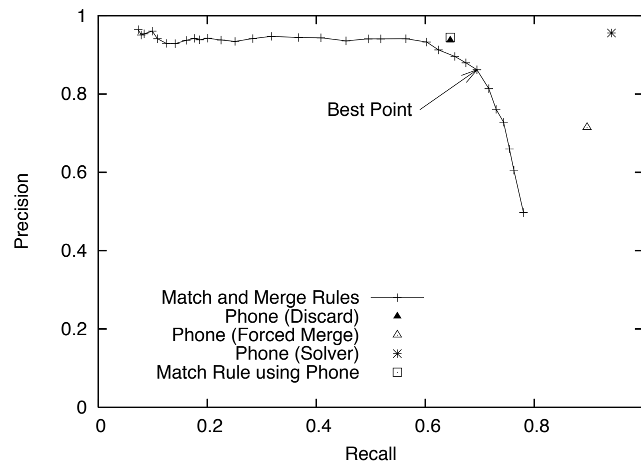


Figure 5.3: Precision and recall for different strategies

Record size	Number of records
1	3477
2	725
3	21
4	0
5	2

Figure 5.4: Distribution of base records per output record

the Best Point, the precision has increased while the recall has decreased. Intuitively, discarding records reduces incorrectly merged records (increasing the precision of merging), but may also mistakenly remove correct merges (decreasing the recall).

The advantage of the Discard strategy is that the human effort is relatively small compared to the Solver strategy (see below) because the human solver only needs to choose records and does not need to do any manual unary or binary checks. As a matter of fact, if records are selected based on size (as we did for our emulation), then the solver does no actual work.

Automatic Forced Merge Strategy An alternative to fixing inconsistencies by discarding records is to force the merge of records that violate the binary negative rule. When we run ENR in this fashion (everything else unchanged) we get the Forced Merge data point in Figure 5.3. We see that the Forced Merge strategy decreases the precision while increasing

the recall of the Best Point. Forcing inconsistent records to merge may create internally inconsistent records (decreasing the precision) and also find correct matches (increasing the recall).

The Forced Merge strategy is effective when there are many record matches that were not identified by the match and merge functions. Since we are merging inconsistent records *automatically*, without solver intervention, the solver cost is the same as for the Discard strategy.

Solver Strategy Finally, we tested a strategy where all negative rule violations are examined by the human solver, and he decides in each case whether it is best to force a merge, ignore the negative rule firing, or to discard a record. To emulate what a solver would do, we rely on the Gold Standard G. When a unary inconsistency is detected in record r (step 9 of GNR), we check if any pair of base records for r is *not* in G. If all pairs are in G, then we ignore the negative rule. When a binary rule violation is detected (step 12), we check if the records can be safely merged. If the merged record would only contain base record pairs in G, then we go ahead and force a merge. Using G to drive the algorithm is fair since we expect the human solver to make decisions that are consistent with those made by the domain expert who created the gold standard.

The accuracy of the solver strategy is shown in Figure 5.3. We can see that the Solver strategy significantly outperforms any strategy both in precision and recall. However, note that the solution is still not 100% correct. The reason is that the solver can only fix problems flagged by the negative rules. If an incorrect merge or a missing merge is not detected by the negative rules, then the problem is not brought to the solver's attention. Of course, the Solver strategy is more expensive for the solver, as he has to manually examine and resolve all records flagged by the negative rules. Thus, it is important to design negative rules that do not generate too many unnecessary checks.

At this point the reader may wonder, if checking phone numbers is so effective in detecting problems, why were phone numbers not checked by the match function? As we argued in the introduction, negative rules are integrity checks often developed after the match and merge functions are implemented. It is often safer not to embed integrity checks in the same code that is being checked. Furthermore, the match and merge functions may

be legacy code that is hard to modify, developed by programmers that did not have perfect knowledge.

It is also important to notice that adding phone number checks to our original match function *does not* give the same results as the Solver Strategy. For example, we could modify our match function so that hotels with different phone numbers do not match (effectively incorporating our unary negative rule into the match function). Figure 5.3 (point labeled "Match Function using Phone") shows the result of using the new match function with the Best Point threshold ($T_M = 0.74$). Compared to the Best Point, the precision increased to 0.944 while the recall dropped to 0.646. (Incidentally, the result is very similar to that of the Discard strategy.) Hence, accuracy is much better with the solver where hotels with questionable phone numbers are being examined by an expert, so opposed to simply not merged.

5.5.4 Other Negative Rules

To better understand how negative rules impact accuracy, we implemented a second type of rule. These rules, sometimes used in practice, flag "borderline cases" as suspicious so the solver checks them out. For instance, say two hotels r and s have very similar names and addresses, but not quite similar enough that the match function fires (or perhaps other attributes indicate a mismatch). Then we may want the solver to look at r and s to decide what to do.

In particular, our binary negative rule states two records are inconsistent if there exists a pair of names, one in each record, that have a string similarity over T_B, and a pair of street addresses also have a similarity over T_B. Our unary negative rule checks if the possible names and street addresses in a record are "too far apart" to be in the same record. Specifically, given a unary threshold T_U, a record is internally inconsistent if two possible names differ more than T_U and two possible street addresses differ more than T_U. We used the Jaro-Winkler similarity measure for all string comparisons. We call these rules the NameAddr negative rules.

Figure 5.5 shows the result of testing the NameAddr negative rules on several T_U and T_B thresholds using the Solver strategy. The nine points (black squares) are produced by

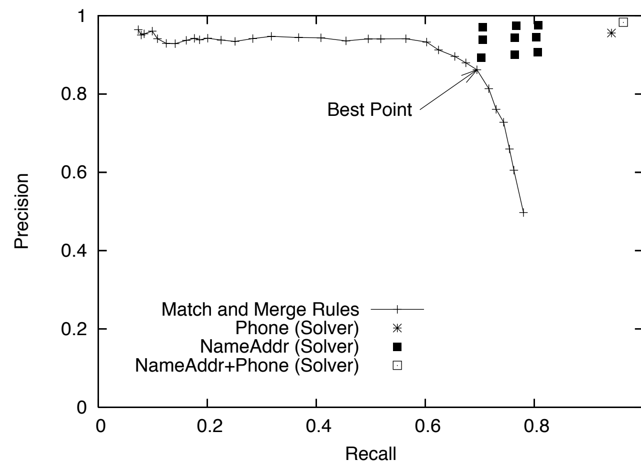


Figure 5.5: Precision and recall for various negative rules

assigning T_U the values 0.8, 0.9, and 0.99 while assigning T_B the values 0.75, 0.7, and 0.65. As T_U increases, precision increases, and as T_B decreases, recall increases. For example, the leftmost 3 points correspond to $T_B = 0.75$, while the bottom 3 points correspond to $T_U = 0.8$. The top, rightmost point is for T_U = 0.99 and T_B = 0.65. These trends are as one would expect: the unary rule detects more inconsistencies as T_U increases while the binary negative rule does so as T_B decreases. And the more inconsistencies that are flagged, the more opportunities the solver has to fix things (and of course, the more work for the solver).

Because our negative rules have parameters that let us vary how stringent they are, we can visualize the tradeoff between accuracy and solver cost. For example, Figure 5.6 shows how recall and cost relate. The horizontal axis is the recall achieved as we vary T_B (keeping T_U at its lowest value), and the vertical axis shows the solver cost. For example, the rightmost points are obtained with $T_B = 0.65$ and we get a recall of about 0.806. The top most curve shows our estimate for the selection cost; the middle curve shows the number of pairs of records manually checked by the solver for binary inconsistencies, and the bottom curve shows the number of records checked for unary inconsistencies. We can clearly see that achieving the higher recall comes at a price, as the solver needs to examine more records. Note that the record selection cost can be eliminated if we automate the record selection process, choosing the largest record as we did for our experiments here. The analogous precision-cost graph (not presented here) shows that, unlike recall, achieving a higher precision does not significantly increase the solver cost.

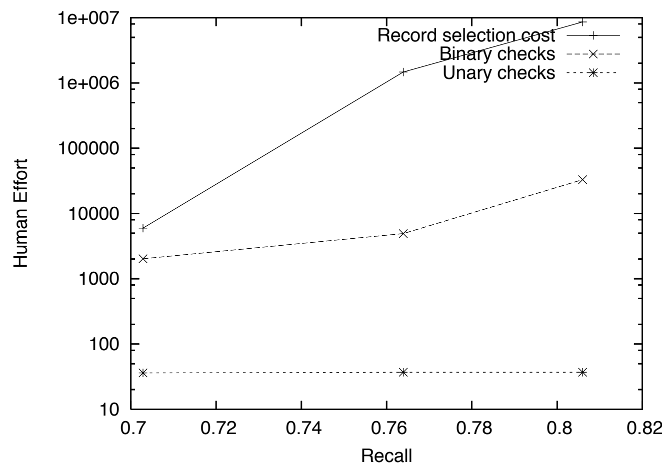


Figure 5.6: Human effort versus recall

We also combined the NameAddr negative rules with the phone number negative rules presented earlier. The combined unary (binary) negative rule returns inconsistent when either one of the two unary (binary) negative rules returns inconsistent. (In this case we set T_U and T_B to 0.99 and 0.65, respectively.) Figure 5.5 shows that the combined method gives the highest precision and recall. Intuitively, the combined rules identify the largest number of inconsistencies for the solver to check. However, as a result, the solver does the most work.

In summary, the precision and recall of an ER-N solution depends on the strategy used for resolving records as well as the negative rules. However, in general:

- The Discard strategy increases precision, but decreases recall.
- The Forced Merge strategy increases recall, but decreases precision.
- The Solver strategy lets a human decide how to fix inconsistencies on a case by case basis. The accuracy improvement depends on how effectively the negative rules find actual inconsistencies.

5.5.5 Choosing Negative Rules

In general, choosing the right number of negative rules that maximize the precision and recall with a reasonable solver cost requires application knowledge about "common errors"

Negative Rule(s)	Description
C	No hotel can have two different cities
L	No two hotels can have latitudes that differ less than 0.01 degree (i.e., 1.1km)
NA	The NameAddr negative rules defined in Section 5.5.4 where T_U=0.8 and T_B=0.75
P	The Phone negative rules defined in Section 5.5.2

Figure 5.7: List of negative rules

Combination	Precision/Recall	Human Effort
NA	0.7/0.89	5968
NA + C	0.7/0.91	6017
NA + L	0.87/0.91	4692245
NA + C + L	0.87/0.93	4722335
P	0.94/0.95	934

Figure 5.8: Results for various combinations of negative rules

of the match and merge functions. If the negative rules do not properly point out the errors, then the solver might end up checking unnecessary records without improving the precision or recall much.

Figure 5.7 shows several negative rules including the Phone and NameAddr negative rules defined in the previous sections. Figure 5.8 shows the precision, recall, and human effort results for various combinations of the negative rules. (We experimented on the 5,000 U.S. hotel records using the Solver strategy.) Adding the City negative rule to the NameAddress negative rules slightly increases the recall with a small additional human effort. The Latitude negative rule, on the other hand, significantly increases the precision, but also requires a much larger human effort because many different hotels can be within 1.1km (latitude) of each other. The Phone negative rule alone already gives a better precision and recall (while requiring a much smaller human effort) compared to previous combinations because it effectively pinpoints the errors of the match and merge functions.

The experiments show that finding the best negative rules requires a good understanding of the application and the match and merge functions. Carefully thought-out negative rules (like the Phone negative rules) will be able to find most of the real inconsistencies of the match and merge functions with little human effort. Other negative rules may either fail to

find many inconsistencies or end up increasing the human effort too much.

5.6 Performance

In this section, we address the performances of the GNR and ENR algorithms. First, we compare the human efforts of the two algorithms and show that the ENR algorithm performs significantly better than the GNR algorithm except for cases where binary inconsistencies occur frequently. Next, we compare the system runtimes of the algorithms by analyzing the major runtime factors and conducting scalability tests. We also ran our experiments on a comparison shopping dataset provided by Yahoo! Shopping, and the results are analogous to those of the hotel dataset.

5.6.1 Human Effort

Record Selection Cost and Rule Checks We measured the human efforts for the two algorithms on 1,000 to 5,000 U.S. hotel records using the phone number negative rules and the threshold $T_M = 0.74$. We used the Solver strategy from Section 5.5 for resolving records.

Figure 5.9 shows that the ENR algorithm requires much less solver effort than the GNR algorithm. The significantly larger selection cost for the GNR algorithm compared to the ENR algorithm is due to the highly redundant records views, which can be illustrated by the following example. Suppose that a set of initial records has a merge closure size of 100. Moreover, suppose that the initial records form ten connected components where each component has a merge closure size of 10. For simplicity, we ignore the inconsistency and domination relationships among records. For the GNR algorithm, the solver must view $\sum_{i=2}^{100} i = 5499$ records; for the ENR algorithm, the solver only needs to view $10 \times \sum_{i=2}^{10} i = 540$ records, which is about one-tenth the effort of the GNR algorithm effort. Although the merge closure is the same for both algorithms, the ENR algorithm saves a lot of redundant views because it partitions the merge closure into many smaller independent components.

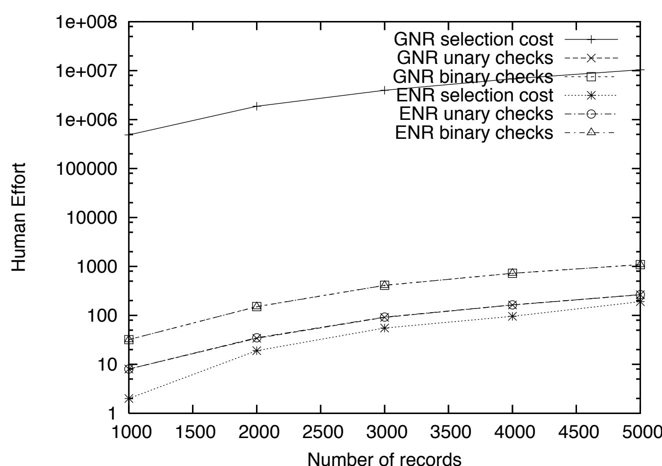


Figure 5.9: Human effort

Binary Density Impact However, the ENR algorithm does not always perform better than the GNR algorithm. In the case where many binary inconsistencies occur, the ENR algorithm loses the advantage of dividing the merge closure into many smaller components and handling one component at a time. To capture the degree of binary inconsistencies, we define the binary density measure as the ratio between the number of inconsistent record pairs in $\bar{R}$ and the number of all the possible record pairs in $\bar{R}$. For example, if five records are inconsistent with each other among ten records of $\bar{R}$, the binary density is $\binom{5}{2}/\binom{10}{2}$ = 10/45 $\approx$ 0.202. For our experiments, we used the NameAddr negative rules on 5,000 U.S. hotel records and varied the binary density by changing the binary threshold T_B. A lower T_B results in a higher binary density because a pair of records is more likely to be inconsistent.

Figure 5.10 shows how human effort and binary density relate. For small binary densities, the ENR algorithm has a much lower record selection cost than GNR because the component sizes are small, minimizing the time for running the GNR algorithm on each component. As the binary density increases, however, the components get larger, and running the GNR algorithm on them takes longer. For high binary densities, the benefits of the ENR algorithm disappear because the merge closure is no longer partitioned into smaller components, and the selection cost of ENR becomes close to that of the GNR algorithm.

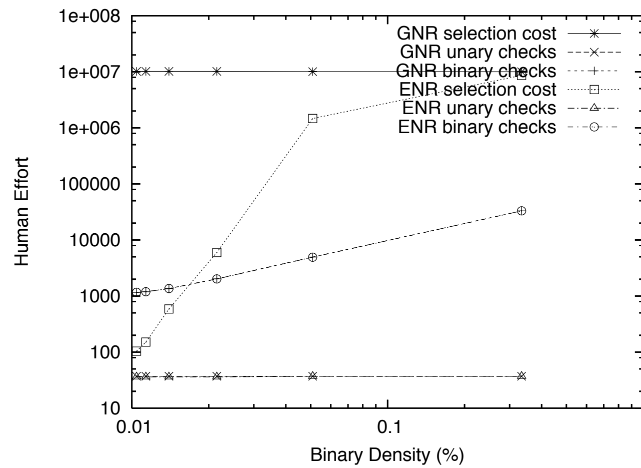


Figure 5.10: Binary density impact on human effort

5.6.2 System Runtime

Runtime decomposition Figure 5.11 shows the runtime decomposition of the GNR and ENR algorithms using the phone number negative rules on 5,000 U.S. hotel records. We show the total runtimes and major runtime factors for each algorithm. The majority of the GNR runtime is used for managing the domination relationships between records in the merge closure in order to find the non-dominated records in S (step 6 of GNR). The next longest task for GNR is invoking the binary negative rule. The major runtime factors for the ENR algorithm are running the R-Swoosh algorithm and connecting the inconsistent components. Comparing the total runtimes, the ENR algorithm is 2.2 to 2.5 times faster than GNR.

Scalability We conducted scalability tests for the GNR and ENR algorithms using the phone number negative rules on 1,000 to 27,000 records randomly selected (regardless of the country) from the entire hotel dataset provided by Yahoo!. (The entire dataset size was 27,049 records.) We used a slightly higher match threshold than usual ($T_M = 0.8$) in order to properly match non-U.S. hotels. For example, using the threshold $T_M = 0.74$ on the French hotels resulted in many different hotels incorrectly merging with each other. Figure 5.12 shows that the GNR algorithm cannot handle more than 17,000 records in a reasonable time while the ENR algorithm shows a quadratic growth in runtime. As the

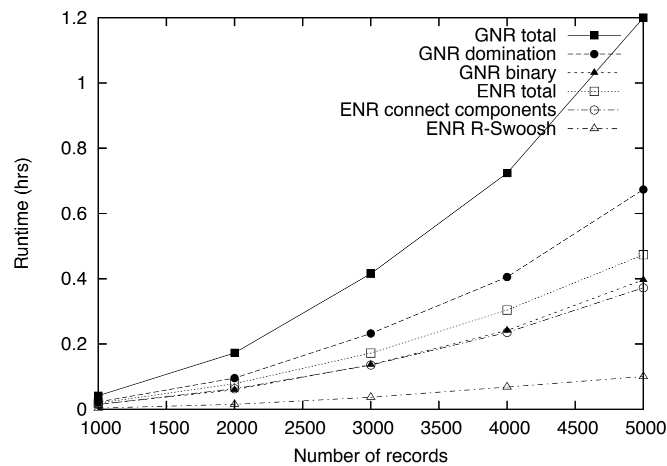


Figure 5.11: Runtime decomposition

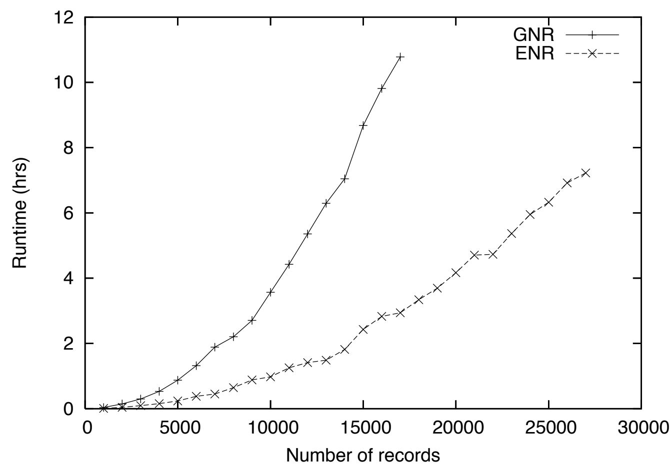


Figure 5.12: Scalability

dataset gets larger, ENR outperforms GNR by up to 3.89 times.

In summary, although the ENR algorithm outperforms the GNR algorithm both in human effort and scalability, ER-N is an inherently expensive process and thus only relatively small sets of records can be handled. Thus, large data sets need to be partitioned into smaller sets (e.g., using blocking techniques) that can be resolved in detail. How large a data set can be exhaustively resolved depends on the application. For example, in our recent work on scaling ER on 2 million Yahoo! Shopping records [109], the average block size was 124 records while the maximum block size was 6082 records. It is also possible to distribute the ER-N computations across multiple processors, in order to handle larger data

sets. We can use techniques similar to the ones in [9] to distribute the data and computation among processors. There are also applications that do not require an exhaustive comparison on the entire data set. For example, a technique called Data Dipping compares a given record only with a small subset of candidate records that are likely to match the record.

5.6.3 Without the Properties

So far, we have only studied scenarios where the properties for the negative rules (Section 5.3.2) hold. We now consider a scenario where the properties do not hold. (We still assume the ICAR properties hold for the match and merge functions. See reference [10] for an extensive study on using the R-Swoosh algorithm without the ICAR properties.) In this case, we need to run the GNR algorithm for a correct ER-N answer. From our previous results, however, GNR can be very expensive in runtime. The alternatives are to either modify the negative rules to satisfy the properties or run ENR even though we might not get the correct ER-N answer. In this section, we consider the second alternative and investigate how similar the ENR result is to the GNR result.

We use a modified version of the NameAddress negative rules (defined in Section 5.5.4) where we only compare the longest strings. Specifically, the binary negative rule now compares the longest names and addresses of the two records while the unary negative rule compares the two longest names and two longest addresses of a single record. Although the Commutativity property is satisfied, the Unary and Binary Persistence properties are not guaranteed because the longest name and address of a record could change after a record merge, possibly making a previously inconsistent record consistent.

Figure 5.13 shows a comparison of the GNR and ENR results when we use the modified NameAddress negative rules and vary the T_U and T_B thresholds. We experimented on the 5,000 U.S. hotel records using the Solver strategy. For each possible threshold pair, we show the total number of records for each result (columns 2 and 3) and the Jaccard similarity between the two results (column 4). Given a GNR solution G and ENR solution E, the Jaccard similarity between the two results is defined as $\frac{|G \cap E|}{|G \cup E|}$. The average Jaccard similarity is 99.41%, making the ENR result almost identical to the GNR result.

In summary, the ENR algorithm is a reasonable way to compute an ER-N result even

T_B/T_U	**GNR**	**ENR**	**Jaccard Similarity**
0.65/0.8	4445	4445	100.0
0.65/0.9	4445	4445	100.0
0.65/0.99	4445	4445	100.0
0.70/0.8	4478	4454	99.38
0.70/0.9	4478	4454	99.38
0.70/0.99	4478	4454	99.38
0.75/0.8	4525	4475	98.85
0.75/0.9	4525	4475	98.85
0.75/0.99	4525	4475	98.85

Figure 5.13: Result sizes and similarities

when the properties for the negative rules do not hold. The experimental results show that the incorrectness of the ENR result is very small in practice.

5.7 Related Work

Most of the ER work in the literature focuses on exploiting positive rules to improve the accuracy of record matching. In contrast, our ER-N model provides a general framework for both positive and negative rules where the match, merge, and negative rules are black-box functions.

A line of work [29, 30, 89, 13, 31] has addressed the use of negative rules. Doan et al. [29, 30] introduced constraints to perform sanity checks for object matching. Dong et al. [31] used dependency graphs while Bhattacharya and Getoor [13] used negative relational evidence to improve the accuracy of the constraints. Shen et al. [89] provided a probabilistic interpretation of constraints and categorized them according to their semantics. However, the constraints used in the work above are local in a sense that they only prevent two records from incorrectly matching. To the best of our knowledge, our work is the first to introduce binary negative rules, which require a global view of records for detecting inconsistencies.

A recent work [23] uses aggregate constraints to improve the accuracy of record clustering. Their goal is to partition the initial set of records such that the number of constraint violations is minimized. The textual similarity between tuples is used to restrict the search

space of partitions. Our work complements the above work in several ways. First, we guarantee a correct and maximal solution as opposed to using the constraints as a search heuristic. Second, we take a pairwise approach, which is an alternative to their clustering approach. Finally, we consider record merges, which a clustering approach does not directly support, and do integrity checks on the merged records. Record merges are important for our pairwise approach because they naturally provide the lineage for each record in the ER result, making it easy to view intermediate states of the ER process. Although negative rules could also be used in clustering approaches, we would need to define the notion of intermediate states for clusters.

A related topic to our work is maintaining integrity constraints in relational databases [50, 35]. Active database systems [110] use triggers and rules to provide mechanisms for integrity constraints. More recently, a line of research [24, 17] shows how to "repair" an inconsistent database into a consistent one while minimizing the difference. The possible repair actions are deleting, inserting, and modifying tuples. While the motivations of providing integrity constraints are similar to ours, the work above do not address the additional complexity of iteratively matching and merging records. Moreover, their focus is on specific constraints such as functional dependencies and inclusion dependencies.

Another related line of work is called statistical data editing [36, 112, 38] where missing or contradictory data is edited and imputed for intended analytic purposes. While statistical data editing is focused on fixing the data itself, our work complements this approach by improving the matching process of data (records) using negative rules.

An interesting analogy for negative rules can be found in a topic called non-monotonic reasoning [41, 79, 42]. Unlike conventional logic, a non-monotonic inference can later be retracted by contrary information. Although retracting inferences is similar to applying negative rules, the main focus of non-monotonic reasoning is to deduce whether a single statement is true or false. In contrast, we are trying to find all the records in the solution (in logic terminology, the theory) efficiently. Another difference is that, while non-monotonic reasoning does not alter its basic beliefs, an ER-N algorithm can discard base records that have incorrect data.

5.8 Conclusion

For the entity resolution process, unary and binary negative rules capture "sanity checks" written by domain specialists who are different from the ones writing the match and merge functions. As far as we know, our work is the first to formally define what *correct* and *maximal* entity resolution means in the context of negative rules (Section 5.1.3). Such a logical and formal foundation is critical for developing ER-N algorithms: it is easy to develop algorithms that apply rules in an ad-hoc fashion and give some sort of answer. However, here we have presented two algorithms that are proven to give the correct answer. Another aspect that is often overlooked is that entity resolution often requires human guidance to handle unexpected situations and erroneous real-world data. Our algorithms demonstrate how a human "solver" can guide the resolution process. One of our algorithms (GNR) represents a generic way to solve ER-N while the other (ENR) makes the GNR algorithm efficient by exploiting additional properties for the match, merge, and negative rules.

Neither of our solutions is perfect: the GNR algorithm can be expensive unless used for small data sets or when there are few matching records. The ENR algorithm is only guaranteed to return a correct solution if certain properties hold, and these properties may not hold in some applications. However, the algorithms can be used if one is willing to tolerate some loss in accuracy: As we discussed, one can partition the input data set and only run ER-N on a partition (bucket) at a time. Also, one can run the ENR algorithm when the properties do not hold, which introduces additional inaccuracies. Evaluating such inaccuracies is beyond the scope of this thesis.

There may of course be applications where negative rules simply introduce too much cost, even when shortcuts are taken. Entity resolution with negative rules is inherently expensive, but we believe it is important to understand the options and their costs, so that application developers can make informed decisions.

Chapter 6

A Model for Quantifying Information Leakage

In Chapters 2–5, we studied a set of scalable and general solutions for ER. The flip side of data integration, however, is that there is a danger of one's personal information being more exposed to the public. Nowadays Internet users are continually giving out sensitive information [76]: a user needs to give out her credit card data in order to purchase something; she may need to tell her drug store what drugs she needs; she needs to give her employer (and many others) her social security number; her airline needs her passport number, and so on. Each bit of information she releases represents a loss of privacy, and she never knows who may end up getting her information. For instance, her store may share her information with some advertiser; or her airline may give her passport number to some governments. The separate information losses can become much more serious if some adversary is able to gather and piece together the user's information using ER.

We study two aspects of the data privacy problem. The first problem, which is the focus of this chapter, is quantifying how leakage can increase (or decrease) as information is pieced together. We do not wish to view privacy as all-or-nothing; rather, we wish to view it as a continuous measure that can represent the severity of a user's information loss. And once we can quantify leakage, we can study strategies for reducing leakage (or increasing it if we want to take the point of view of a law-enforcement "adversary" trying to learn about possible criminals). The second problem is to go one step further and actually manage

information leakage. Chapter 7 focuses on reducing information leakage.

As a motivating example of quantifying information leakage, suppose that Alice has the following information: her name is Alice, her address is 123 Main, her phone number is 555, her credit card number is 999, her social security number is 000. We represent Alice's information as the record: p = {[N, Alice], [A, 123 Main], [P, 555], [C, 999], [S, 000]}. Suppose now that Alice buys something on the Web and gives the vendor a subset of her information, say r = {[N, Alice], [A, 123 Main], [C, 999]}. By doing so, Alice has already partially compromised her privacy.

We can quantify the information leakage by measuring how *correct* and *complete* the information in r is against p. In our example, 3 out of 3 of r's attributes were correct while 3 out of 5 of p's attributes were found in r. Uncertain and incorrect information are also important factors in measuring information leakage. If an adversary Eve is not *confident* about Alice's information, then although the information itself is correct, the leakage should be considered less than when Eve is absolutely confident about the data. Moreover, if Eve is absolutely sure about some incorrect information about Alice, then the information leakage should decrease in proportion to Eve's confidence. Returning to our example above, suppose that Alice also gives the same vendor the following information (through another purchase): {[N, Alice], [A, 777 Main], [C, 999], [X, 111]}. As a result, the vendor may only be half certain about Alice's address. In addition, if [X, 111] contains false information, the vendor now has an incorrect attribute X. Both errors should be factored in when computing the leakage.

The information leakage may also be affected by any *data analysis* performed by adversary Eve. For example, Eve may run an entity resolution (ER) operation to identify which pieces of information refer to the same person. To illustrate, Figure 6.1 shows the records of five people: r = {[N, Alice], [P, 123]}, s = {[N, Alice], [C, 999]}, t = {[N, Alice], [P, 987]}, u = {[N, Bob], [P, 333]}, and v = {[N, Carol], [S, 000]}. The record p above the line represents Alice's full information. Assuming that the name is a strong identifier for people, Eve may conclude that r, s, and t refer to the same person and merge their contents into $\langle r, s, t \rangle$ = {[N, Alice], [P, 123], [C, 999], [P, 987]} (denoted as the dotted lines connecting r, s, and t). As a result, Eve may obtain better information about Alice. However, the analysis itself may be costly if the data to resolve is very large and Eve does not have

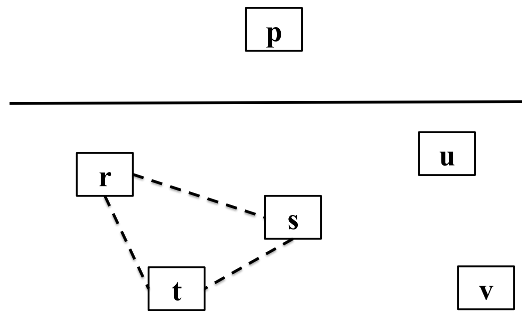


Figure 6.1: Information Leakage with Entity Resolution

sufficient resources to perform the analysis.

In summary, our contributions in this chapter are as follows:

- We formalize information leakage as a general measure of privacy. Our measure reflects the following factors: the correctness and completeness of the leaked data, the adversary's confidence on the data, and the adversary's analysis on the data (Section 6.1).
- We compare our information leakage model with two related privacy models in data publishing: k-anonymity and l-diversity (Section 6.2).
- We formulate various problems for managing information leakage that can be solved using our framework (Section 6.3).
- We propose efficient exact and approximate algorithms for computing information leakage (Section 6.4).
- We experimentally evaluate leakage in a synthetic environment, both to check that the model matches our intuition, as well as to test the scalability of the exact and approximate leakage algorithms. (Section 6.5).

6.1 Information Leakage Measure

We consider a scenario where the adversary Eve has one record r pertaining to Alice, which could be a piece of information collected from a social network profile, a homepage, or even a tweet. Record r contains a set of attributes, and each attribute consists of a label

and value. As an example, the following record may represent Alice:

$$r = \{[\text{N}, \text{Alice}], [\text{A}, 20], [\text{A}, 30], [\text{Z}, 94305]\}$$

Each attribute $a \in r$ is surrounded by square brackets and consists of one label $a.l$ and one value $a.v$. Notice that there are two ages for Alice. We consider [A, 20] and [A, 30] to be two separate pieces of information, even if they have the same label. Multiple label-value pairs with identical labels can occur when two records merge and the label-value pairs are simply collected. In our example, Alice may have reported her age to be 20 in one case, but 30 in another. (Equivalently, year of birth can be used instead of age.) Although we cannot express the fact that Alice has only one age (either 20 or 30), the confidences we introduce in Section 6.1.3 can be used to indicate the likelihood of each value.

In addition, each attribute label l has an weight w_l that reflects the relative importance of an attribute with label l. These weights will be used below to compute leakage, so that attributes with highly weighted labels will contribute more than those with lower weights. In our example, we may give the credit card label C a weight of w_C = 3 and the zip code label a weight of w_Z =1, to reflect that Alice considers her credit card number three times more important than her zip code. The absolute values of the weights are not important, only their relative sizes. Thus, if there are only three labels, giving them the weights 1, 2, and 3 is equivalent to giving them the weights 2, 4, and 6. We emphasize that the weights are assigned to labels and not on the individual attributes. We believe this simplification is useful, since giving weights to every possible attribute is not practical.

In our model we assume that different attributes are *not* correlated. However, in some cases the values for some attributes may depend on each other. For example, the birth date of a person depends on the age of a person because the birth date can be used to compute the age. In this case, if both the date of birth and the age are discovered by Eve, we do not want to account for the loss twice. We get around this problem by simply assuming that our model contains only one of the dependent attributes, e.g., either date of birth or age. In other cases, attributes may be correlated, but not equivalent. For example, phone number and address may be correlated: if we know the phone number we may be able to narrow down the possible addresses, and vice versa. We can model this situation

by assuming there are three attributes: J contains the "joint information," A contains the remaining address information, and P the remaining phone information. If Eve discovers Alice's phone number, she has values for J and P; if she discovers the address, she gets J and A, and if she has both address and phone, Eve has J, A and P. Now we can provide weights for the J, A and P labels, and not double count the correlated knowledge.

We also assume a reference record p that contains Alice's complete information. An interesting question is how much of Alice's information has been revealed by exposing the record r? One might say that even if one attribute is leaked, a privacy breach has occurred, so Alice has no privacy. On the other extreme, however, one may say that, since not all of Alice's information has been leaked, the privacy has not been breached. In order to capture the amount of information that has been leaked, we define the *record leakage* of the record r as its similarity against the reference record p (see Definition 6.1.1). Also given a database R (which is a set of records), we define the *information leakage* of R to be its similarity against p after the "data analysis" of the adversary (see Definition 6.1.2).

In the following sections, we discuss the main components of our measure and formally define record leakage and information leakage.

6.1.1 Correctness

The correctness measure of a record r reflects the portion of r's information that is correct according to p. We adapt the definition of precision from the information retrieval literature [70] to define correctness. The precision of the record r against the reference p is defined as follows:

$$Pr(r,p) = \frac{\Sigma_{a \in r \cap p} w_{a.l}}{\Sigma_{a \in r} w_{a.l}}$$

If $\Sigma_{a \in r} w_{a.l} = 0$, we define Pr to be 0. Suppose that p = {[N, Alice], [A, 20], [P, 123], [Z, 94305]} and r = {[N, Alice], [A, 20], [P, 111]}. Also say that w_N = 2 while the weights for all other labels are 1. Then the precision of r against p is $\frac{2+1}{2+1+1} = \frac{3}{4}$.

There are several ways to extend our definition of Pr. First, we can reflect the degree of error in computing information leakage where more correct information implies more leakage. For example, suppose that Alice is 30 years old. Then the information leakage when Eve guesses that Alice is 31 years old should be higher than the leakage when Eve suspects

Alice is 80 years old. Second, we can take into account the statistical background knowledge of the adversary when measuring the leakage. For instance, if knowing that someone has an average age may be less leakage than knowing that someone has an exceptional age.

6.1.2 Completeness

Even if the correctness of r is very high, r may not be significant if only a small fraction of p has been discovered. Hence, we also define the notion of how much of p is found by r, which we call the completeness of r. This time, we adapt the definition of recall from the information retrieval literature. In general, one could use any other measure to capture how much correct information was found by the record r.

We define the recall of r against p as follows.

$$Re(r,p) = \frac{\Sigma_{a \in r \cap p} w_{a.l}}{\Sigma_{a \in p} w_{a.l}}$$

If $\Sigma_{a \in p} w_{a.l} = 0$, we define Re to be 0. In our example, the recall of r against p is $\frac{2+1}{2+1+1+1}$ = $\frac{3}{5}$.

The information retrieval literature suggests the harmonic mean as one way of combining correctness and completeness. Given the correctness Pr and completeness Re, the weighted harmonic mean is defined as F = $\frac{1}{\alpha/Pr+(1-\alpha)/Re}$ = $\frac{(\beta^2+1)\times Pr\times Re}{\beta^2\times Pr+Re}$ where $\beta^2 = \frac{1-\alpha}{\alpha}$.

The F_1 measure sets β to 1 and thus gives equal weight to precision and recall. In information retrieval, the F_1 measure captures how relevant a search result is against a given query. In comparison, the information leakage measure quantifies the relevance of a record r against the correct answer p. We can combine the precision and recall to produce a single record leakage measure L^0 where the "0" superscript indicates the leakage computation without confidences.

$$L^0(r,p) = F_1(Pr(r,p), Re(r,p)) = \frac{2 \times Pr(r,p) \times Re(r,p)}{Pr(r,p) + Re(r,p)}$$

In our example, the F_1 value is $\frac{2\times 3/4\times 3/5}{3/4+3/5}$ = $\frac{2}{3}$.

6.1.3 Adversary Confidence

As mentioned earlier, the confidence that adversary Eve has on her data plays a role in computing leakage. For example, Eve may have gotten some information of r from an unreliable website. Or Eve may have heard rumors of the subject indirectly from someone else. If Eve is not so confident about r, then even if r has a high accuracy against p, the information leakage should be less than when Eve is more confident. Also, if Eve is highly confident about information that is not accurate, then the information leakage should be considered less than when Eve is not so confident about the inaccurate information.

To reflect the confidence of Eve, we extend our data model to have per-attribute confidence values. As a result, a record r consists of a set of attributes, and each attribute contains a label, a value, and a confidence (from 0 to 1) that captures the uncertainty of the attribute from Eve's point of view (the more Eve knows about Alice, the higher the confidence values). Any attribute that does not exist in r is assumed to have a confidence of 0. As an example, the following record may represent Alice:

$$r = \{[\mathrm{N}, \mathrm{Alice}, 1], [\mathrm{A}, 20, 0.5], [\mathrm{A}, 30, 0.4], [\mathrm{Z}, 94305, 0.3]\}$$

That is, Eve is certain about Alice's name, but is only 50% confident about Alice being 30 years old, 40% confident in Alice being 30 years old, and 30% confident about Alice's zip code 94305. For each attribute $a \in r$, we can access a's label $a.l$, a single value $a.v$, and confidence $a.c$. We assume that attributes in the reference p always have a confidence of 1 and omit the confidence values. No two attributes in the same record may have the same label and value pair.

The confidences within the same record are independent of each other and reflect "alternate worlds" for Eve's belief of the correct information of Alice. For example, if we have r = $\{$[name, Alice, 1], [age, 20, 0.4], [phone, 123, 0.5]$\}$, in Eve's view then there are four possible worlds: $\{$[name, Alice], [age, 20], [phone, 123]$\}$ with probability $0.4 \times 0.5 = 0.2$, $\{$[name, Alice], [age, 20]$\}$ with probability $0.4 \times (1 - 0.5) = 0.2$, $\{$[name, Alice], [phone, 123]$\}$ with probability $(1 - 0.4) \times 0.5 = 0.3$, and $\{$[name, Alice]$\}$ with probability $(1 - 0.4) \times (1 - 0.5) = 0.3$. We denote the possible worlds of a record r as the

set of records without confidences

$$W(r) = \{\{[a.l, a.v]|a \in r'\}|r' \in 2^r\}$$

where 2^r is the power set of r.

To help in our definition of leakage, we define the function $p(a, r)$ that simply gives the confidence of attribute a in record r:

$$p(a, r) = \begin{cases} b.c & \exists b \in r \text{ s.t. } a.l = b.l \wedge a.v = b.v \\ 0 & o.w. \end{cases}$$

We now extend our information leakage measure in the previous section to use confidences. Since, the confidence values of the attributes in r are independent, we can define the record leakage of r against the reference p as follows. (In Section 6.4, we show how to compute the record leakage efficiently.)

Definition 6.1.1. *Given confidence values, the record leakage of record r against the reference p is*

$$\begin{aligned} L(r, p) &= E[L^0(\bar{r}, p)] \\ &= \sum_{r' \in W(r)} (\prod_{a \in r'} p(a, r))(\prod_{a \notin r'} 1 - p(a, r)) L^0(r', p) \end{aligned}$$

where $\bar{r}$ is a random variable of r's possible worlds.

That is, we are computing the expected value of the F_1 value between a possible world and p. For example, suppose that p = {[N, Alice], [A, 20], [P, 123]} and r = {[N, Alice, 0.5], [A, 20, 1]}. Also say that w_N = 2 while all the other weights have a value of 1. There are two possible values for $\bar{r}$: r_1 = {[A, 20]} and r_2 = {[N, Alice], [A, 20]}. We then compute $L^0(r_1, p) = \frac{2\times1/1\times1/3}{1/1+1/3} = \frac{1}{2}$ and $L^0(r_2, p) = \frac{2\times2/2\times2/3}{2/2+2/3} = \frac{4}{5}$. Thus $L(r, p) = \frac{1}{2} \times L^0(r_1, p) + \frac{1}{2} \times L^0(r_2, p) = \frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times \frac{4}{5} = \frac{13}{20}$. Notice that L receives a record with confidences as its first input and a record without confidences as its second input. One can also define the precision and recall metrics using confidences by replacing $L^0(r', p)$ in Definition 6.1.1 with $Pr(r', p)$ and $Re(r', p)$, respectively.

$L(r, p)$ quantifies leakage when Eve has a single record r in her possession. What happens if Eve has a set of records R? There is no simple answer in this case, but we take a "conservative" approach and define leakage as the worst leakage that may occur when Eve looks at any one of her R records. That is, $L^0(R, p) = max_{r \in R} L(r, p)$. Note that we are overloading the symbol L: if the first parameter is a set, it refers to set leakage; if the first parameter is a single record, then it is record leakage. We use the "0" superscript to distinguish this basic set leakage from leakage after the adversary analyzes and possibly combines records (next subsection).

6.1.4 Adversary Effort on Data Analysis

In order to increase the information leakage, the adversary Eve may further improve the quality of the database by fixing errors, adding more information, or removing duplicates. We illustrate three possible data analysis operations below.

- *Error Correction*: The adversary Eve identifies and corrects erroneous data. For example, Eve may fix misspellings of words in the database.
- *Augment Information*: Eve fills in missing data either by inferring the data or copying the data from other sources. For example, if Eve knows the addresses of people, then she could fill in their zip codes automatically.
- *Entity Resolution* [105, 34]: Eve can identify the records that refer to the same real-world entity and merge them into composite records. For example, if Eve has the three records r, s, and t in the database and know that r and s both refer to the same person, then she can merge r and s by combining the information of the two records into a single record $\langle r, s \rangle$, resulting in a database with two records: $\langle r, s \rangle$ and t. An ER operation can also use background information when resolving the records.

Among the operations, we will focus on the entity resolution (ER) operation. To illustrate how a data analysis operation can improve information leakage, say that we are running the ER function E on the database R = $\{r$ = $\{$[N, Alice, 1], [P, 123, 1]$\}$, s = $\{$[N, Alice, 1], [C, 999, 1]$\}$, t = $\{$[N, Bob, 1], [P, 987, 1]$\}\}$. (Notice that we have set all the confidence values to 1 for simplicity.) Also suppose we have the reference record p = $\{$[N, Alice], [P, 123], [C, 999], [Z, 111]$\}$. Before running E, the information leakage is

$L^0(R,p) = max_{r \in R} L(r,p) = max\{\frac{2\times 2/2\times 2/4}{2/2+2/4}, \frac{2\times 2/2\times 2/4}{2/2+2/4}, 0\} = \frac{2}{3}$. While running E, suppose we conclude that r and s are the same person and produce the merged record $\langle r,s \rangle$ = $\{$[N, Alice, 1], [P, 123, 1], [C, 999, 1]$\}$. Then the new database $E(R)$ = $\{\langle r,s\rangle, t\}$ has the information leakage $L^0(E(R),p) = max_{r\in E(R)} L(r,p)$ = $max\{\frac{2\times 3/3\times 3/4}{3/3+3/4}, 0\}$ = $\frac{6}{7}$. Hence, by applying E, Eve has increased the information leakage of R from $\frac{2}{3}$ to $\frac{6}{7}$.

We can abstract any combination of data analysis operations as a function E, which receives the database R and returns another database $E(R)$ that may increase information leakage. For example, one can augment information to the database and then perform entity resolution.

While the above operations are powerful and may enhance information leakage, they require computation effort on Eve's side. For example, if a sophisticated ER algorithm takes quadratic time to run, then it may not be feasible to run the algorithm on all the hundreds of millions of people on the Web. If Eve uses a more relaxed and faster algorithm, then more records can be resolved.

To incorporate the adversary effort into our model, we define the cost function C that receives the adversary operation E and the database R, and returns the "cost" required to run E on R. The cost could be measured in computation steps, run time, or even in dollars. For instance, if has $O(n^2)$ complexity for resolving n records, then $C(E,R)$ could be $c \times |R|^2$ for some constant c.

Using the basic definition of information leakage and the data analysis operation E, we now define our information leakage measure as follows.

Definition 6.1.2. *Given an adversary operator E, the* information leakage *of R against p is $L(R,p,E)$ = $L^0(E(R),p)$ with the cost $C(E,R)$.*

For example, say that there are 1000 records in the database R, and $L^0(R,p)$ = 0.3. Also say that $C(E,R)$ = $c \times |R|^2$ where c = $\frac{1}{1000}$. Now suppose that the operator E improves the information leakage where $L^0(E(R),p) = 0.9$. Hence, the data analysis using E has revealed an additional information of 0.9−0.3 = 0.6 using a cost of $\frac{1}{1000} \times 1000^2 = 1000$. Notice that if E is an identity function where $E(R) = R$, then $L(R,p,E)$ reduces to the basic information leakage definition $L^0(R,p)$ = $max_{r\in R} L(r,p)$.

We can extend our model to a scenario where Eve not just has a database of records R, but also has a "query" of interest. For example, Eve may be focusing on a person with name Alice who is 30 years old. In this case, Eve's query is q = $\{$[name, Alice, 1], [age, 30, 1]$\}$. Eve can then look in R for records that are "related" to this Alice, thus expanding her information on this Alice. That is, starting with q, Eve may use ER to merge records in the database that refer to the same entity as q. Given an ER function E, we define the *dipping result* of q, $D(R, E, q)$, as the record $r \in E(R \cup \{q\})$ such that r is a merged result of q. For example, suppose we have the database R = $\{r$ = $\{$[N, Alice, 1], [P, 123, 1]$\}$, s = $\{$[N, Alice, 1], [C, 999, 1]$\}$, t = $\{$[N, Bob, 1], [P, 987, 1]$\}\}$. Also say that the ER function E merges all the records that have the same name. Given the query q = $\{$[N, Alice, 1]$\}$, we then obtain a dipping result $\langle r, s, q\rangle$ = $\{$[N, Alice, 1], [C, 999, 1], [P, 123, 1]$\}$ because both r and s have the same name as q. The information leakage of q can be defined as $L(D(R, E, q), p)$. (Reference [105] provides more details on ER and computing dipping results.)

6.2 Relationship to Other Measures

In this section, we compare our information leakage measure with two popular privacy models in data publishing. We first provide a detailed comparison of information leakage and the k-anonymity model [94]. Next, we briefly discuss how our measure relates to the l-diversity model [68]. Both models take an all-or-nothing approach where either all the records in a database are equally "safe" or none of the records are safe at all. In comparison, our information leakage model can be used to quantify various notions of privacy, e.g., the information leakage of an individual record within a database. Obviously, it is impossible to compare our model with every existing privacy model. For example, we do not directly compare our work with the t-closeness [66] or Differential Privacy [32] models. However, the same argument holds where information leakage can quantify various notions of privacy while the two existing models either deem the entire database safe or not safe.

Name	Zip	Age	Disease
Alice	111	30	Heart
Bob	112	31	Breast
Carol	115	33	Cancer
Dave	222	50	Hair
Pat	299	70	Flu
Zoe	241	60	Flu

Table 6.1: Database of Patients (R)

6.2.1 k-anonymity

In data publishing, the k-anonymity model [94] prevents the identity disclosure of individuals within a database. More formally, a database R satisfies k-anonymity if for every record $r \in R$, there exist $k-1$ other records in R that have the same "quasi-identifiers." The quasi-identifiers are attributes that can be linked with external data to uniquely identify individuals in the database. For example, suppose that all the records in R have three attributes: zip code, age, and disease. Given external information, if a person in R can be identified by looking at her zip code and age, then the quasi-identifiers are zip code and age. Also, a sensitive attribute is an attribute whose value for any particular individual must be kept secret from people who have no direct access to R. In our example, we consider the disease attribute as sensitive.

Table 6.1 illustrates a database R that contains the name, zip code, age, and disease information of patients. For example, patient 1 has the name Alice, zip code 111, an age of 30, and a heart disease. Table 6.2 shows the 3-anonymous version of R (called R_a) without the names. In order to anonymize a database, we "suppress" each quasi-identifier value by either replacing a number with a range of numbers or replacing one or more characters in a string with the same number of wild card characters. (A wild card character is denoted as '*' and represents any character.) For example, Dave's age 50 was suppressed to ≥ 50 while Alice's zip code 111 was suppressed to 1**. For any patient, there are two other patients that have the same zip code and age information. Each set of records that have the same quasi-identifiers form one equivalence class. We assume that the adversary Eve can view Table 6.2, but not Table 6.1.

The k-anonymity model takes an all-or-nothing approach where either all the records

Zip	**Age**	**Disease**
11*	3*	Heart
11*	3*	Breast
11*	3*	Cancer
2**	≥50	Hair
2**	≥50	Flu
2**	≥50	Flu

Table 6.2: 3-Anonymous Version of Table 6.1 (R_a)

satisfy k-anonymity or not. That is, if each record is indistinguishable from $k-1$ other records in terms of quasi-identifiers, the released database is considered "safe." Otherwise, the database is not safe. In comparison, the information leakage model is more general and can quantify a wider range of privacy settings.

First, we can study the information leakage for individuals. For example, suppose that we compare the information leakage of Alice and Zoe. (For simplicity, we assume all attribute weights have the same value 1.) Say that we first run an ER algorithm E that merges all the records in Table 6.2 that have the same zip code and age. As a result, there are two merged records: r_1: {[Zip, 11*, 1], [Age, 3*, 1], [Disease, Heart, 1], [Disease, Breast, 1], [Disease, Cancer, 1]} and r_2: {[Zip, 2**, 1], [Age, ≥50, 1], [Disease, Hair, 1], [Disease, Flu, 1]}. If the reference record of Alice is p_a = {[Name, Alice], [Zip, 111], [Age, 30], [Disease, Heart]}, the information leakage of Alice is $max\{L(r_1, p_a), L(r_2, p_a)\}$ = $max\{\frac{2\times 3/5\times 3/4}{3/5+3/4}, 0\}$ = $\frac{2}{3}$. Here, we have made the simplification that a suppressed value (e.g., 1**) is equal to its non-suppressed version (e.g., 111). In practice, one could view a suppressed value as the original value with a reduced confidence value. If the reference record of Zoe is p_b = {[Name, Zoe], [Zip, 241], [Age, 60], [Disease, Flu]}, then the information leakage of Zoe is $max\{L(r_1, p_b), L(r_2, p_b)\}$ = $max\{0, \frac{2\times 3/4\times 3/4}{3/4+3/4}\}$ = $\frac{3}{4}$. Again, we have simplified the comparison and considered a suppressed value (e.g, ≥50) to be equal to its unsuppressed version (e.g., 60). As a result, the information leakage of Zoe, $\frac{3}{4}$, is higher than that of Alice, $\frac{2}{3}$, although k-anonymity considers the records of both people to be equally safe.

Second, we can quantify the impact of background information on privacy. For example, say that the adversary knows additional information about Alice as shown in Table 6.2.

Name	Zip	Age
Alice	111	30

Table 6.3: Background Information (R_b)

The adversary can then combine the database R_b in Table 6.3 with Table 6.2 to measure Alice's information leakage. Using the same ER algorithm E as above, we now generate the two records r'_1: $\{$[Name, Alice, 1], [Zip, 11*, 1], [Age, 3*, 1], [Disease, Heart, 1], [Disease, Breast, 1], [Disease, Cancer, 1]$\}$ and r_2: $\{$[Zip, 2**, 1], [Age, $\geq$50, 1], [Disease, Hair, 1], [Disease, Flu, 1]$\}$. The information leakage of Alice is thus $max\{L(r'_1, p_a), L(r_2, p_a)\}$ = $max\{\frac{2\times4/6\times4/4}{4/6+4/4}, 0\}$ = $\frac{4}{5}$. Hence, in the presence of the background information R_b, Alice's information leakage has increased from $\frac{2}{3}$ to $\frac{4}{5}$.

6.2.2 l-diversity

The l-diversity model [68] enhances the k-anonymity model by ensuring that the sensitive attributes of each equivalence class have at least l "well-represented" values. For example, in Table 6.2, the first equivalence class contains 3 distinct diseases while the second equivalence class has 2 distinct diseases. If l = 3, then we would like to enforce each equivalence class to have at least 3 distinct diseases. Suppose that we change Zoe's disease in Table 6.2 from Flu to Influenza. Then the modified database R'_a satisfies 3-diversity because each equivalence class has at least 3 different diseases.

Although R'_a is considered safe by l-diversity, the fact that Influenza is semantically similar to the Flu may result in less privacy for Zoe. We now illustrate how the information leakage model can quantify this change in privacy. First suppose that E considers the diseases Flu and Influenza to be different. Then using the ER algorithm E defined above, we generate the two records r_1: $\{$[Zip, 11*, 1], [Age, 3*, 1], [Disease, Heart, 1], [Disease, Breast, 1], [Disease, Cancer, 1]$\}$ and r'_2: $\{$[Zip, 2**, 1], [Age, $\geq$50, 1], [Disease, Hair, 1], [Disease, Flu, 1], [Disease, Influenza, 1]$\}$. Thus the information leakage of Zoe is $max\{L(r_1, p_b), L(r'_2, p_b)\}$ = $max\{0, \frac{2\times3/5\times3/4}{3/5+3/4}\}$ = $\frac{2}{3}$. Now suppose that the operation E' is equivalent to E, but considers Influenza to be the same disease as the Flu and replaces all the occurrences of Influenza with Flu when merging records. In this case, the generated record r'_2 is now r''_2: $\{$[Zip, 2**, 1], [Age, $\geq$50, 1], [Disease, Hair, 1], [Disease,

Flu, 1]$\}$. As a result, the information leakage of Zoe becomes $max\{L(r_1, p_b), L(r''_2, p_b)\}$ = $max\{0, \frac{2\times3/4\times3/4}{3/4+3/4}\}$ = $\frac{3}{4}$. Hence, by exploiting the application semantics, the information leakage of Zoe has increased from $\frac{2}{3}$ to $\frac{3}{4}$. Notice that this measurement could not be done using the l-diversity model, which cannot capture the usage of application semantics.

6.3 Applications

Our information leakage framework can be used to answer a variety of questions as we show in the following sections. As we use our framework, it is important to keep in mind "who knows what". In particular, if Alice is studying leakage of her information, she needs to make assumptions as to what her adversary Eve knows (database R) and how she operates (the data analysis function E Eve uses). These types of assumptions are common in privacy work, where one must guess the sophistication and compute power of Eve. On the other hand, if Eve is studying leakage she will not have Alice's reference information p. However, she may use a "training data set" for known individuals in order to tune her data analysis operations, or say estimate how much she really knows about Alice. In the following sections, we formalize problems in information leakage both in Alice's point of view and in Eve's point of view.

6.3.1 Releasing Critical Information

In this section, we formalize problems for managing Alice's information leakage. Suppose that Alice tracks R, the information she has given out in the past. She now wants to release a new record r (e.g., her credit card information) which may fall in the hands of the adversary who might use the ER function E to resolve other records with r. Alice can compute the direct leakage involved in releasing the record r, i.e., $L(R \cup \{r\}, p, E)$. However, we may want to capture the information leaked by r only instead of computing the entire leakage of the database. We thus define the incremental leakage of r as follows.

$$I(R, p, E, r) = L(R \cup \{r\}, p, E) - L(R, p, E)$$

Since r may make it possible for Eve to piece together big chunks of information about Alice, the incremental leakage may be large, even if r contains relatively little data.

To illustrate incremental leakage for a critical piece of information, say that Alice wants to purchase a cellphone app from an online store and is wondering which credit card c_1 or c_2 she uses will lead to a smaller loss of her privacy. Each purchase requires Alice to submit her name, credit card number, and phone number. Due to Alice's previous purchases, the store already has some information about Alice.

In particular:

- Alice's reference information is $p = \{$[N, n_1], [C, c_1], [C, c_2], [P, p_1], [A, a_1]$\}$ where N stands for name, C for credit card number, P for phone, and A for address.
- The online store has two previous records $R = \{s = \{$[N, n_1, 1], [C, c_1, 1], [P, p_1, 1]$\}$, t = $\{$[N, n_1, 1], [C, c_2, 1]$\}\}$. (We omit the app information in any record for brevity.)
- The store accepts one of the two records $u = \{$[N, n_1, 1], [C, c_1, 1], [P, p_1, 1]$\}$ or $v = \{$[N, n_1, 1], [C, c_2, 1], [P, p_1, 1]$\}$ for the cellphone app purchase. Since Alice is purchasing an app, again no shipping address is required.

Suppose that two records refer to the same entity (or match) if their names and credit card numbers are the same or their names and phone numbers are the same, and that merging records simply performs a union of attributes. Also say that all weights w have the same value 1.

Then the information leakage of Alice before her purchase is $L(R,p) = max_{r \in E(R)} L(r,p)$ = $max_{r \in \{s,t\}} L(r,p)$ = $max\{\frac{2 \times 3/3 \times 3/5}{3/3+3/5}, \frac{2 \times 2/2 \times 2/5}{2/2+2/5}\}$ = $max\{\frac{3}{4}, \frac{4}{7}\}$ = $\frac{3}{4}$. If Alice uses c_1 and releases u to the store, then the information leakage is still $L(r,p) = \frac{2 \times 3/3 \times 3/5}{3/3+3/5} = \frac{3}{4}$ because u and s are identical and merge together, but not with t. If Alice uses c_2 and releases v instead, all three records merge together because v matches with both s and t. Hence the information leakage is $L(s+t+v,p) = \frac{2 \times 4/4 \times 4/5}{4/4+4/5} = \frac{8}{9}$. To compare Alice's two choices, we compute the incremental leakage values, i.e., the change in leakage values due to the app purchase. In our example, the incremental leakage of releasing u is $\frac{3}{4} - \frac{3}{4}$ = 0 while the incremental leakage of releasing v is $\frac{8}{9} - \frac{3}{4} = \frac{5}{36}$. Thus, in this case Alice should use the credit card c_1 to buy her app because it preserves more of her privacy.

6.3.2 Releasing Disinformation

Releasing disinformation can be an effective way to reduce information leakage. Given previously released information R, Alice may want to release either a single record or multiple records that can decrease the information leakage. We call records that are used to decrease the leakage *disinformation* records. Of course, Alice can reduce the information leakage by releasing arbitrarily large disinformation. However, disinformation itself has a cost. For instance, adding a new social network profile would require the cost for registering information. As another example, longer records could require more cost and effort to construct. We use $C(r)$ to denote the entire cost of creating r.

We define the problem of minimizing the information leakage using one or more disinformation records. Given one data analysis operator E, a set of disinformation records S and a maximum budget of C_{max}, the optimal disinformation problem can be stated as follows:

$$\begin{aligned} \text{minimize} \quad & L(R \cup S, p, E) \\ \text{subject to} \quad & \Sigma_{r \in S} C(r) \leq C_{max} \end{aligned}$$

The set of records S that minimizes the information leakage within our budget C_{max} is called an optimal disinformation.

We study the problem of releasing disinformation. Figure 6.2 shows a database R = $\{r, s, t, u, v\}$ where r and s refer to the entity p while t, u, and v refer to an entity other than p. The disinformation record can reduce the database leakage in two ways. First, a disinformation record can perform *self disinformation* by acting as disinformation itself and add its irrelevant information to a correct record. In our example, the disinformation record d_1 snaps with the correct record r and adds its own information to r. Second, a disinformation record can perform *linkage disinformation* by reducing the database leakage by linking irrelevant records in R to a correct record. For example, the disinformation record d_2 is linking the irrelevant record v to the correct record r and thus adding v's information to r. Using these two basic disinformation strategies, one can perform a combination of self and linkage disinformation as well.

When creating a record, we use a user-defined function called $Create(S, L)$ that creates a new minimal record that has a size less or equal to L and is guaranteed to match all the

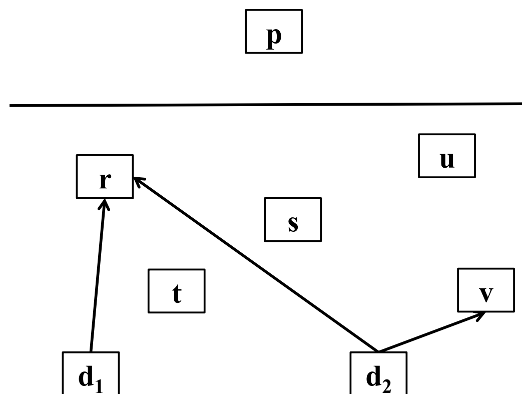


Figure 6.2: Self and Linkage Disinformation

records in the set S. If there is no record r such that $|r| \leq L$ and all records in S match with r, the $Create$ function returns the empty record $\{\}$. A reasonable assumption is that the size of the record produced by $Create$ is proportional to $|S|$ when $L > |S|$. We also assume a function called $Add(r)$ that appends a new attribute to r. The new attribute should be "incorrect but believable" (i.e., bogus) information. We assume that if two records r and s match, they will still match even if Add appends bogus attributes to either r or s. The $Create$ function is assumed to have a time complexity of $O(|S|)$ while the Add function $O(|r|)$. Reference [105] provides more detail how to use the $Create$ and Add functions to generate the optimal disinformation.

6.3.3 Enhancing a Composite Record

From the adversary Eve's point of view, there may also be interesting "optimization" questions to ask. Since Eve does not know Alice's full record p, the questions cannot be phrased in terms of p. Consider a composite record r_c that Eve has inferred from a set of facts in a set R. For whatever reason, Eve is very interested in r_c, but unfortunately there is some uncertainty in the attributes in r_c. We define r_p to be the same as r_c except that all confidences in r_c are set to 1 and omitted from the record. $L(r_c, r_p)$ is a measure of how certain r_c is: the closer $L(r_c, r_p)$ is to 1, the more certain Eve is of the information in r_c.

To improve $L(r_c, r_p)$ (i.e., make it closer to 1), Eve can try to increase her confidence in the attributes in R. For any given attribute $a = [l, v, c]$ in some $r_i \in R$, Eve can improve

the confidence of a by doing more research, bribing someone, issuing a subpoena, etc. The increase in the confidence of a will clearly have a cost associated with it. There are again many ways to model the cost, but for simplicity let us assume that the cost in changing the confidence from its current value of c to 1 is $C(a) = 1 - c$.

Now the question is, what is the most cost effective way to increase Eve's confidence in r_c. If Eve only wants to verify one attribute, then we want the one $a \in r_i$ that maximizes

$$\frac{L(r'_c, r_p) - L(r_c, r_p)}{C(a)}$$

where r'_c is the composite record Eve can infer when the confidence in a is increased to 1.

For example, suppose that we have the database R = $\{r_1$= $\{$[N, Alice, 1], [A, 20, 1]$\}$, r_2= $\{$[N, Alice, 0.9], [P, 123, 0.5], [C, 987, 1]$\}\}$ where N stands for name, A stands for age, P stands for phone, and C stands for credit card number. We assume that all weights w have the value 1. Suppose that r_1 and r_2 merges into r_c = $\{$[N, Alice, 1], [A, 20, 1], [P, 123, 0.5], [C, 987, 1]$\}$ where we take the maximum confidence value when merging two attributes with the same label and value pair. Then r_p = $\{$[N, Alice], [A, 20], [P, 123], [C, 987]$\}$. If we enhance the name in r_2 to have a confidence of 1, then r'_c = $\{$[N, Alice, 1], [A, 20, 1], [P, 123, 0.5], [C, 987, 1]$\}$ (which is identical to r_c), and $C([\text{N}, \text{Alice}, 0.9])$ = $1 - 0.9$ = 0.1. Then $\frac{L(r'_c,r_p)-L(r_c,r_p)}{C([\text{N,Alice,0.9}])}$ = $\frac{0}{0.1} = 0$. On the other hand, if we enhance the phone number of r_2, then r'_c = $\{$[N, Alice, 1], [A, 20, 1], [P, 123, 1], [C, 987, 1]$\}$ with the cost $C([\text{P}, 123, 0.5])$ = $1 - 0.5$ = 0.5. Then $\frac{L(r'_c,r_p)-L(r_c,r_p)}{C([\text{P,123,0.5}])}$ = $(\frac{2\times4/4\times4/4}{4/4+4/4} - (\frac{1}{2} \times \frac{2\times4/4\times4/4}{4/4+4/4} + \frac{1}{2} \times \frac{2\times4/4\times3/4}{4/4+3/4}))/0.5$ = $\frac{1-(1/2+3/7)}{0.5}$ = $\frac{1}{28}$. Hence, verifying the phone number in r_2 results in a better enhancement of r_c than verifying the name of r_2.

6.4 Computation

Computing information leakage efficiently is important because the amount of information (i.e., the number of attributes) within a record can be large in practice. Given a database R and a data analysis operation E, the information leakage (see Definition 6.1.2) can be computed by running $E(R)$, and then computing the maximum record leakage by iterating each record r in $E(R)$ and computing $L(r, p)$. (Since in general we do not have

any $E(R)$ prior knowledge, the only strategy is to iterate through the resulting $E(R)$ records.) Given that computing the record leakage $L(r,p)$ takes $f(|r|,|p|)$ time and running E on R takes $g(|R|)$ time, the total complexity of computing the information leakage is $O(g(|R|) + \sum_{r \in E(R)} f(|r|,|p|))$.

A naïve approach for computing the record leakage is to iterate through all possible worlds of r and add the record leakage values (without confidences) multiplied by their probabilities as shown in Definition 6.1.1. This solution has an exponential complexity of $O(2^{|r|} \times |r|)$. In the following sections, we propose efficient solutions for computing the record leakage. We first propose a $O(|p| \times |r|^2)$-time algorithm that computes the exact value of information leakage and assumes constant weights. We then propose a $O(|p| \times |r|)$-time approximate solution that works for arbitrary weights.

6.4.1 Exact Solution using Constant Weights

We now show how to compute record leakage in polynomial time given that all the weights have a constant value w. We first re-write $L(r,p)$ as follows. First, equations (1) and (2) result from the definitions of L^0, Pr, and Re.

$$L(r,p) = E\left[\frac{2 \times Pr(\bar{r},p) \times Re(\bar{r},p)}{Pr(\bar{r},p) + Re(\bar{r},p)}\right] \tag{6.1}$$

$$= E\left[\frac{2 \times \sum_{a \in \bar{r} \cap p} w_{a.l}}{\sum_{a \in \bar{r}} w_{a.l} + \sum_{a \in p} w_{a.l}}\right] \tag{6.2}$$

Equation (3) uses the linearity of expectation and sums the record leakage for each attribute in p. The notation $\bar{r}\backslash\{b\}$ indicates the possible world of r without the attribute b.

$$= 2 \times \sum_{b \in p} p(b,r) \times \tag{6.3}$$

$$E\left[\frac{w_{b.l}}{\sum_{a \in \bar{r}\backslash\{b\}} w_{a.l} + w_{b.l} + \sum_{a \in p} w_{a.l}}\right]$$

Equation (4) simplifies equation (3) using the assumption that all the weights have an equal value.

$$= \quad 2 \times \sum_{b \in p} p(b,r) \times E\left[\frac{1}{|\bar{r} \backslash \{b\}| + 1 + |p|}\right] \tag{6.4}$$

Equation (5) converts the expression $\frac{1}{X}$ to $\int_0^1 t^{X-1} dt$ and pushes the expectation operator within the integral.

$$= \quad 2 \times \sum_{b \in p} p(b,r) \times \int_0^1 E\left[t^{|\bar{r} \backslash \{b\}| + |p|}\right] dt \tag{6.5}$$

Equation (6) uses the fact that all the attributes in r are independent and converts $E\left[t^{|\bar{r} \backslash \{b\}|}\right]$ into $\prod_{a \in z} E\left[t^{X_a}\right]$ where X_a is a random variable that is 1 if a appears in $\bar{r}$ and 0 otherwise. We also remove any attribute in r that has the same label and value as b by iterating the attributes in $z = \{[c.l, c.v] | c \in r \wedge (c.l \neq b.l \vee c.v \neq b.v)\}$.

$$= \quad 2 \times \sum_{b \in p} p(b,r) \times \int_0^1 t^{|p|} \prod_{a \in z} E\left[t^{X_a}\right] dt \tag{6.6}$$

Finally, equation (7) computes the expected value of t^{X_a} as $p(a,r) \times t + (1 - p(a,r))$.

$$\begin{aligned} = \quad & 2 \times \sum_{b \in p} p(b,r) \times \\ & \int_0^1 t^{|p|} \prod_{a \in z} (p(a,r) \times t + (1 - p(a,r))) dt \end{aligned} \tag{6.7}$$

Algorithm 12 numerically evaluates equation (7). Steps 3–12 convert the innermost product into an expression of the form $Y_0 \times t^n + Y_1 \times t^{n-1} + ... Y_n$, where n is the number of attributes in z. The coefficients $Y_0, ... Y_n$ are stored in list Y in the algorithm. (List Z is an auxiliary list used to compute the Y values.) Steps 13 and 14 evaluate equation (7). The integral is applied to each $p(b,r) \times t^{|p|} \times Y_x \times t^{n-x}$ term in turn, yielding $p(b,r) \times \frac{Y_x}{|p|+n-x+1}$. (Note that $|Y| = n + 1$.)

As an illustration of Algorithm 12, suppose that p = {[A, 1], [B, 2]} and r = {[A,

ALGORITHM 12: Record Leakage using Constant Weights

input : the records r, p
output: the record leakage $L(r, p)$

```
1  L ← 0;
2  for b ∈ p do
3      Y ← (1.0);
4      for a ∈ r do
5          Z ← ();
6          if a.l = b.l ∧ a.v = b.v then
7              continue to next loop;
8          Z.Add(Y.Get(0) × p(a, r));
9          for x = 0, ..., |Y| − 1 do
10             Z.Add(Y.Get(x) × (1 − p(a, r)) + Y.Get(x + 1) × p(a, r));
11         Z.Add(Y.Get(|Y| − 1) × (1 − p(a, r)));
12         Y ← Z;
13     for x = 0, ..., |Y| − 1 do
14         L ← L + 2 × p(b, r) × Y.Get(x) / (|p| + |Y| − x);
15 return L;
```

1, 0.1], [B, 3, 0.2]}. Also say that all the attribute weights have a value of 1. We first assign b = [A, 1] in Step 2 and set $Z = (0.2, (1 - 0.2)) = (0.2, 0.8)$ in Steps 8–11. We then set Y = (0.2, 0.8) in Step 12 and continue to Steps 13–14 where we compute L = $2 \times 0.1 \times \frac{0.2}{2+2-1+1} + 2 \times 0.1 \times \frac{0.8}{2+2-2+1}$ = $\frac{19}{300}$. Next, we set b = [B, 2] in Step 2 and set $Z = (0.1 \times 0.2, 0.1 \times (1-0.2) + (1-0.1) \times 0.2, (1-0.1) \times (1-0.2)) = (0.02, 0.26, 0.72)$ in Steps 8–10. However, we add 0 to L in Steps 13–14 because $p(b, r) = 0$. Hence, we return L = $\frac{19}{300}$. Notice that we have the same result as computing $L(p, r)$ using the brute-force naïve approach: $0.1 \times 0.2 \times \frac{2 \times 1/2 \times 1/2}{1/2+1/2} + 0.1 \times 0.8 \times \frac{2 \times 1 \times 1/2}{1+1/2} + 0.9 \times 0.2 \times 0 + 0.9 \times 0.8 \times 0$ = $\frac{19}{300}$.

The complexity of the algorithm above is $O(|p| \times |r|^2)$ time because for each record in p, we dynamically construct the coefficients of t^X, which takes $O(|r|^2)$ time.

6.4.2 Approximation using Arbitrary Weights

We now show how to compute an approximation of $L(r, p)$ efficiently where the weights can be assigned different values. We can use the Taylor series of $F(X)$ about the point $E[X]$ to approximate $F(X)$ as follows.

$$\begin{aligned} F(X) &= F(E[X]) + \frac{F'(E[X])}{1!} \times (X - E[X]) \\ &\quad + \frac{F''(E[X])}{2!} \times (X - E[X])^2 + \ldots \end{aligned}$$

If we only take the second order approximation (i.e., the Taylor series visible in the above equation) and compute the expected value of $F(X)$, we have

$$E[F(X)] \approx F(E[X]) + \frac{F''(E[X])}{2!} \times Var[X]$$

Hence, we can derive the following approximation by starting from equation (3) and setting $Y = \sum_{a \in \bar{r} \setminus \{b\}} w_{a.l}$.

$$\begin{aligned} L(r,p) &= 2 \times \sum_{b \in p} p(b,r) \times E\left[\frac{w_{b.l}}{Y + w_{b.l} + \sum_{a \in p} w_{a.l}}\right] \\ &\quad (\text{where } Y = \sum_{a \in \bar{r} \setminus \{b\}} w_{a.l}) \\ &\approx 2 \times \sum_{b \in p} p(b,r) \times (\frac{w_{b.l}}{E[Y] + w_{b.l} + \sum_{a \in p} w_{a.l}} + \\ &\quad \frac{w_{b.l}}{(E[Y] + w_{b.l} + \sum_{a \in p} w_{a.l})^3} \times Var[Y]) \end{aligned}$$

We can also compute the expected value and variance of Y as follows.

$$\begin{aligned} E[Y] &= \sum_{a \in z} w_{a.l} \times a.c \\ Var[Y] &= \sum_{a \in z} w_{a.l}^2 \times a.c - (w_{a.l} \times a.c)^2 \end{aligned}$$

where $z = \{c | c \in r \wedge (c.l \neq b.l \vee c.v \neq b.v)\}$.

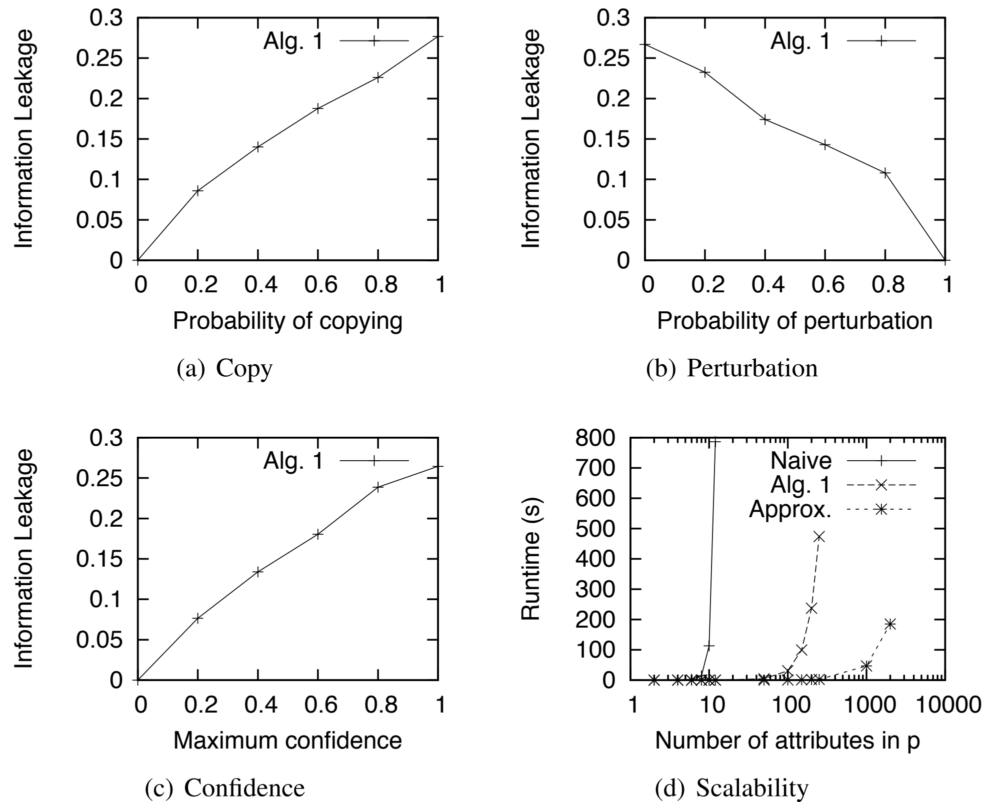
Par.	**Description**	**Basic**
n	Size of the gold standard p	100
$\|R\|$	Number of records to generate	10,000
p_c	Probability of copying attribute from p to r	0.5
p_p	Probability of perturbing a copied attribute	0.5
p_b	Probability of adding bogus attribute to r	0.5
m	Maximum confidence value	0.5
w	Weights are constant (C) or random (R)	C

Table 6.4: Parameters for Data Generation

Hence, an approximation of the record leakage can be computed in $O(|p| \times |r|)$ time. One can extend the Taylor series above to produce an even more accurate solution. However, we show in Section 6.5.2 that our approximation based on the second order series is already quite accurate.

6.5 Experiments

We run experiments on synthetic data in order to observe trends and to study the scalability of our algorithms. Table 6.4 shows the configuration used. We first generate the reference record p by creating a set of n random attributes. We then generate a record $r \in R$ by iterating over each attribute in p and copying it with a probability p_c. However, each time there is a copy, we perturb the attribute with probability p_p into a new attribute. In addition, for each attribute in p we also add a new bogus attribute to r with probability p_b. The confidence value for each attribute generated was a random number between 0 and m, the maximum possible confidence. If $w = C$, then we set all the weights to 1, and if $w = R$, we randomly generated random real numbers between 0 and 1 for the weights. We repeated the generation of a record $|R|$ times. The last column of Table 6.4 shows the basic values of the parameters. Our base case does not represent any particular application or scenario; it is simply a convenient starting point from which to explore a wide range of parameter settings. Our algorithms were implemented in Java, and our experiments were run in memory on a 2.4GHz Intel(R) Core 2 processor with 4 GB of RAM.



(a) Copy

(b) Perturbation

(c) Confidence

(d) Scalability

Figure 6.3: Trends and Scalability

6.5.1 Trends

We plot the information leakage while varying the parameters p_c, p_p, and m. Any parameter that was not varied was set to its basic value in Table 6.4. Figure 6.3(a) shows the leakage when varying p_c from 0 to 1. As p_c increases, more of p's attributes are copied to r, increasing the recall and thus the information leakage as well. Figure 6.3(b) shows the leakage when varying p_p from 0 to 1. This time, the more frequent the perturbation of an attributed being copied, the lower the precision and thus the information leakage becomes as well. Finally, Figure 6.3(c) shows the leakage when varying m from 0 to 1. As the average confidence increases, there are two competing factors that determine the information leakage: the higher confidence of correct information increases the leakage while the higher confidence of incorrect information decreases the leakage. In our setting, the correct information dominates and leakage increases as confidence increases.

n	p_c	p_p	b	m	w	**Exact**	**Approx.**
100	0.5	0.5	0.5	0.5	C	0.1740179	0.1740178
200	0.5	0.5	0.5	0.5	C	0.1374662	0.1374661
100	1.0	0.5	0.5	0.5	C	0.2752435	0.2752431
100	0.5	1.0	0.5	0.5	C	0.0	0.0
100	0.5	0.5	1.0	0.5	C	0.1581138	0.1581135
100	0.5	0.5	0.5	1.0	C	0.2625473	0.2625469
100	0.5	0.5	0.5	0.5	R	0.4047125	0.4046881

Table 6.5: Information Leakage Comparison

6.5.2 Accuracy of Approximate Algorithm

We now evaluate the accuracy of the approximate algorithm in Section 6.4.2. Table 6.5 shows the leakage values for Algorithm 12 and the approximation algorithm while varying the parameters n, p_c, p_p, b, m, and w. If w = C, we generated $|R| = 10,000$ records with constant weights and ran Algorithm 12 to compute the exact leakage. If w = R, we can only compute the exact leakage with the naïve algorithm (which is not scalable as shown in Section 6.5.3). Thus, we limited ourselves to records with only 10 attributes ($|p|$ = 10), and gave each attribute a random weight ranging from 0 to 1. As we can see in Table 6.5, in all scenarios the exact and approximate leakage values are nearly identical, with a maximum error rate of 0.006%. We conclude that our approximate algorithm is highly accurate.

6.5.3 Runtime Performance

We compare the scalability of Algorithm 12 and the approximate algorithm against the naïve implementation defined in the beginning of Section 6.5. We varied the parameter n and used constant weights while using the basic values for the other parameters as defined in Table 6.4. Figure 6.3(d) shows that the naïve algorithm only handle up to 12 attributes. In comparison, Algorithm 12 scales to 250 attributes, and the approximation algorithm scales to more than 2,000 attributes, demonstrating the scalability of our algorithms.

6.6 Related Work

Many works have proposed privacy schemes [94, 68, 84, 32, 113]. The k-anonymity [94, 93] model guarantees that linkage attacks on certain attributes cannot succeed. Subsequent work such as the l-diversity [68] and t-closeness [66] models has improved the k-anonymity model. Rastogi et al. [84] shows the tradeoff between privacy and utility in data publishing in the context of maintaining the accuracy of counting queries. A recent line of work [32, 33] studies differential privacy, which ensures that a removal or addition of a single database item does not (substantially) affect the outcome of any analysis on the database. In comparison, our information leakage measure focuses on quantifying the privacy of an individual against a given database and has the following features. First, we assume that some of our data is already public (i.e., out of our control) and that there can be a wide range of scenarios where we need to measure privacy. Second, our information leakage reflects the four different factors of privacy: the correctness and completeness of the leaked database, the adversary's confidence on the database, and the adversary's analysis on the database.

A closely-related framework to ours is P4P [2], which seeks to contain illegitimate use of personal information that has already been released to an external (possibly adversarial) entity. For different types of information, general-purpose mechanisms are proposed to retain control of the data. More recently, a startup called ReputationDefender [85] has started using disinformation techniques for managing the reputation of individuals focusing on improving search engine results (e.g., adding to the web positive or neutral information about its customers by either creating new web pages or by multiplying links to existing ones). The focus of ReputationDefender is to make one's correct information clearly visible. Hence, the disinformation is being used to maximize the information leakage. TrackMeNot [98] is a browser extension that helps protect web searchers from surveillance and data-profiling by search engines using noise and obfuscation. Finally, ICorrect [56] is a web site that allows one to clarify his/her misinformation on the Web. We believe that the work above show a clear need for using information leakage as a measure of privacy.

The information theoretic metric of entropy [86] is often used in the context of communication privacy and quantifies the amount of information an attacker is missing in verifying

a hypothesis within a confidence interval. In comparison, the information leakage model focuses on capturing the intuitive notion of leakage: correctness, completeness, and the adversary confidence. In addition, our model incorporates the data analysis operations of the adversary and the costs for performing the operations.

Information retrieval [70] searches for relevant information within documents. Many different measures for evaluating the performance of information retrieval have been proposed. The notions of precision were first proposed by Kent et al.[64]. The F measure was introduced by van Rijsbergen [99]. Information leakage adopts these measures in a privacy setting. In addition, our measure reflects the adversary confidence and data analysis. Compared to probabilistic information retrieval [70] where documents are probabilistically ranked, our work probabilistically computes the information leakage itself using possible worlds semantics.

6.7 Conclusion

We have proposed a framework using information leakage as a measure for data privacy. In many applications, an important observation is that privacy is no longer an all-or-nothing concept because one's data may inevitably become public through various interactions (e.g., buying a product from a vendor online). Our information leakage measure reflects four important factors of privacy: the correctness and completeness of the leaked data, the adversary's confidence on the data, and the adversary's data analysis. The adversary may perform ER for his data analysis. The better the adversary performs ER, the more information is leaked. We have compared our information leakage model with the k-anonymity and l-diversity models. We have described several challenges in managing information leakage that can be posed by our framework. We have proposed efficient algorithms for computing the exact and approximate values of information leakage. Finally, we have shown through extensive experiments on synthetic data that the information leakage measure indeed captures the important factors of privacy, and that our information leakage algorithms can scale to large data.

Chapter 7

Disinformation Techniques for Entity Resolution

In Chapter 6, we studied the problem of quantifying information leakage where we measured how much one's personal information can be exposed to the public. The information loss can become more serious if the adversary performs ER to gather and piece together the person's information. While the quantification of information leakage is important to understand the danger of exposing information, we would also like a way to manage the amount of information leakage in order to protect one's privacy.

In this chapter, we thus study how to actually lower information leakage. We study a specific technique called *disinformation*, which is a well-known strategy used to perturb known information by adding false information. A classic example is the Normandy landing during World War II, where the Allied forces used disinformation to make the Germans believe an attack was imminent on Pas de Calais instead of Normandy. As a result, the German forces were concentrated in Pas de Calais, which made the Normandy landing one of the turning points in the war.

Disinformation can be viewed as a tool for safeguarding sensitive information that for one or another reason may have "leaked". That is, an adversary may obtain some sensitive information, e.g., preparations for the real Normandy invasion. The disinformation (Calais invasion) causes the adversary either to disregard the true information, or at least to have less confidence in it. Thus, in a way the sensitive information is protected.

Disinformation and this type of information protection are closely related to entity resolution. In the invasion example, resolution was done by humans, combining evidence records (e.g., intercepted messages, shipping activity, etc.) to form the adversary's best guess for the invasion. In the absence of disinformation, the better the adversary is at ER, the better he will be at "connecting the dots" and learning one's sensitive information. Disinformation is a direct attack on the ER process, confusing its results, and protecting information. While we only studied the quantification of information leakage in Chapter 6, we now study how disinformation can be used to lower information leakage.

To present a more modern example, consider the way life insurers are predicting the life spans of their customers by piecing together health-related personal information on the Web [100]. Here the customers cannot prevent the sensitive information from leaking, as they need to give it out piecemeal to purchase items, interact with their friends, get jobs, etc. However, disinformation could be used to protect sensitive information by preventing the ER algorithm used by the insurance companies from identifying with certainty the customer's say habits or genes. As another example, people search engines like Spock.com [92] resolve hundreds of millions of records crawled from the Web to create one profile per person. Again, it is very hard to remove what is already on the Web, but it is possible to confuse the ER algorithm. For instance, by creating a fake Web page with person X's name and person Y's address and phone number, we may cause the ER algorithm to combine person X and Y into a single output profile, thus muddling what is published.

We are definitely *not* advocating that everyone should generate disinformation (and as we will discuss, there are also drawbacks to generating disinformation). However, we do believe that anyone using ER needs to understand disinformation attacks. That is, if we now take the role of the "adversary", and we have a choice of ER algorithms, we should prefer the one that is more robust against disinformation, i.e., the one that is less easily fooled by disinformation. In this chapter we will present a model for disinformation attacks, study the implications for ER, and will illustrate robustness comparisons among ER algorithms.

Motivating Example To further motivate our work, let us consider yet another example, in simplified fashion. Say that a camera manufacturer called Cakon is working on its latest camera called the C300X. Cakon needs to keep its new camera secret, for once customers

Record	Model	Pixels (M)	Price (K)
r	"C300X"	30	8
s	"C300"	20	7
t	"C200"	10	5
u	"C100"	10	1
v	"C100"	10	1

Table 7.1: Camera Rumor Records

know with certainty the details, they are much less likely to buy the older models (and the new model will not be available for purchase for months). Furthermore, Cakon does not want its competitors to know exactly what they are releasing. However, information about the camera will leak to the public by early testers, component suppliers, or insiders eager to brag. Indeed, there are multiple rumor Web sites that specialize in gathering leaks about upcoming cameras, manually doing ER, and predicting the new models. As in our previous example, Cakon is unable to eliminate leaked information about the C300X, but it may be able to generate disinformation, e.g., by sending out prototypes with different specifications to testers, or by asking a printer to design a pamphlet with incorrect details.

To be more concrete, say a rumors site (the adversary in this case) has collected five rumor records r, s, t, u, and v as shown in Table 7.1. Each record provides the anticipated camera model, number of megapixels (MP), and price. Say the ER algorithm identifies records that refer to the same real-world entity (camera) by computing the similarity in model names, MP counts, and prices. In this case, say all cameras look dissimilar except for u and v. Thus, the ER result is $E_1 = \{\{r\}, \{s\}, \{t\}, \{u, v\}\}$. Here, we use curly brackets to cluster the records that refer to the same entity.

We call the camera manufacturer the *agent*, and say its sensitive information is record $\{r\}$. That is, the new camera model is C300X, with 30M pixels and will sell for 8K dollars. The agent can reduce what is known about the C300X by generating a record that would make the adversary confuse the clusters $\{r\}$ and $\{s\}$. Generating the disinformation record would involve creating a model number that is similar to both C300X and C300 and then taking some realistic number of pixels between 20M and 30M and a price between 7K and 8K dollars. Suppose that the agent has generated the disinformation record d_1: $\{$[Model, "C300X"], [Pixels, 20], [Price, 7]$\}$. As a result, when the adversary runs ER, d_1 may match with r because they have the same model name. The ER algorithm can conceptually merge

the two records into $\langle r, d_1 \rangle$: $\{$[Model, "C300X"], [Pixels, 20], [Pixels, 30], [Price, 7], [Price, 8]$\}$. Now $\langle r, d_1 \rangle$ and s are now similar and might match with each other because they have the same number of pixels and price. As a result, $\langle r, d_1 \rangle$ and s may merge to produce the new ER result $E_2 = \{\{r, s, d_1\}, \{t\}, \{u, v\}\}$. While r and s were considered different entities in E_1, they are now considered the same entity in E_2 due to the disinformation record d_1.

As a result of the disinformation, the adversary is confused about the upcoming C300X: Will it have 20M or 30M pixels? Will it sell for 7K or 8K dollars? With all the uncertainty, the adversary may be even less confident that the C300X is a real upcoming product. (In Section 7.4.1 we define a concrete metric for confusion.)

The agent can further increase confusion by causing the target cluster to merge with even more clusters. For example, if the agent generates disinformation record d_2 = $\{$[Model, "C300"], [Pixels, 10], [Price, 5]$\}$ to Table 7.1, then the ER result may now consider the C300 and C200 to be the same model as well, leading to an ER result of $E_3 = \{\{r, s, d_1, d_2, t\}, \{u, v\}\}$ where r, s, and t have all merged together.

Our example has illustrated the effects of disinformation, but disinformation also has a cost for the agent. The disinformation must be "believable" so as discussed earlier it may require creating Web sites or prototype cameras. Thus, one of the problems we will address is how to maximize the "confusion" around an entity as much as possible using a total cost for creating the disinformation records within a fixed budget.

As mentioned earlier, one of the main uses of our model and work is as a framework for evaluating the *robustness* of ER algorithms. Note that in our example, whether r and s and d merge is very dependent on the ER algorithm in use. Some ER algorithms may be more susceptible to merging unrelated records while others may be more robust to a disinformation attack. The robustness may also depend on the comparison threshold used for matching records. For example, an ER algorithm with a more relaxed comparison threshold is more likely to mistakenly merge records compared to the same ER algorithm with a stricter comparison threshold. We will use our disinformation techniques to evaluate how sensitive each ER algorithm and threshold combination is to various attacks.

In summary, the main contributions of this chapter are:

- We formalize the notion of disinformation in an ER setting. We also define a disinformation optimization problem that maximizes confusion for a given disinformation budget. We analyze the hardness of this problem and propose a desirable ER property that makes disinformation more effective (Section 7.1).
- We propose efficient exact and approximate algorithms for a restricted version of the disinformation problem. We then propose heuristics for the general disinformation problem (Section 7.2).
- We propose several techniques for generating the values of disinformation records based on the values of existing records (Section 7.3).
- We experiment on synthetic and real data to demonstrate the effectiveness of disinformation and compare the robustness of existing ER algorithms (Section 7.4).

7.1 Framework

In this section, we formalize ER and the disinformation problem. We use the ER model from Section 6.1. We then introduce a pairwise approach for evaluating the cost of merging clusters in a partition of records. Next, we define the benefit obtained by merging clusters of records and develop a global strategy for generating disinformation. Finally, we identify a desirable property of ER algorithms that makes our pairwise approach effective.

7.1.1 ER Model

We use the ER model from Section 6.1. We assume the ER algorithm used by the adversary is known to the agent, but cannot be modified. This assumption is common where one must guess the sophistication and compute power of an adversary. For instance, Cakon may know all the possible websites containing rumors of Cakon products and may have an idea on how an adversary might piece together the rumors from the websites. (In Section 7.4.1 we discuss what happens when the agent does not know the adversary's ER algorithm.) In addition, we assume the agent cannot delete or modify records in the database R.

7.1.2 Pairwise Approach for Merging Clusters

To generate disinformation, we first take a pairwise approach of analyzing the costs of merging clusters. We assume that inducing the merge of two clusters c_i and c_j has a well-defined cost function $D(c_i, c_j)$ that is positive and commutative. The cost function measures the agent's effort to generate disinformation records that would merge c_i and c_j. In addition, we assume a function $PLAN(c_i, c_j)$ that specifies the steps for actually generating the disinformation records in order to merge c_i and c_j. The definitions of the two functions depend on the ER algorithm and the records as we illustrate below.

The cost function D can reflect the amount of disinformation that needs to be generated. For example, suppose that all the records are in a Euclidean space, and the ER algorithm always clusters records that are within a Euclidean distance of 1. If there are two singleton clusters c_1 and c_2 that have a Euclidean distance of 10, then we need to generate at least 9 records between c_1 and c_2 that have a distance of 1 between each other and with c_1 and c_2. If the cost of creating the records is estimated as the number of disinformation records that need to be created, then $D(c_1, c_2) = 9$ and $PLAN(c_1, c_2)$ provides the steps for actually creating the 9 disinformation records.

A more sophisticated cost function may also reflect the effort needed to create the specific disinformation values. For example, creating a new public camera record would require some person to post a rumor about the camera on a public website or blog, and the effort may vary depending on the contents of the rumor. A rumor about a camera with unrealistically-high specs may actually damage the reputation of Cakon and can be viewed as a costly disinformation value to create. The result of $PLAN(c_i, c_j)$ may now include instructions for logging into the website and posting the rumor.

In the case where there is no way to induce the merge of c_i and c_j (e.g., due to the ER algorithm or restrictions in the records that can be generated), then the distance $D(c_i, c_j)$ is given as ∞. In general, the more different c_i and c_j are, the costlier it is to generate the disinformation needed to merge the clusters.

For the optimization problem we define next, we assume that the merging costs for different pairs of clusters are independent of each other. That is, the value of $D(c_i, c_j)$ is fixed and not affected by whether other clusters were merged. For example, if $D(c_1, c_2) = 5$

and $D(c_1, c_3) = 4$, then the cost of merging c_1 and c_2 is always 5 regardless of whether we will merge c_1 and c_3. In reality, however, the costs may not be independent because the merging of multiple pairs of clusters may be affected by the same disinformation. For instance, if the disinformation record d is in the middle of the clusters c_1 and c_2 as well as the clusters c_3 and c_4, then by adding d, we may end up merging c_1 and c_2 as well as c_3 and c_4 at the same time. Hence, once c_1 and c_2 are merged, the merging cost of c_3 and c_4 reduces to 0. Note that when we evaluate our disinformation strategies in Section 7.4, we do *not* enforce this assumption for the actual ER algorithms. In Section 7.4.2, we show that even without the independence assumption, our techniques work well in practice.

7.1.3 Disinformation Problem

We consider the problem of maximizing the "confusion" of one entity e, which we call the *target entity*. Given an ER algorithm E and a database R, we call the cluster $c_0 \in E(R)$ that represents the information of e the *target cluster*. Intuitively, by merging other clusters in $E(R)$ to the target cluster, we can dilute the information in the target cluster and thus increase the confusion. In our motivating example, the camera company Cakon was increasing the confusion on the target entity C300X by merging the C300 cluster $\{s\}$ to the target cluster $\{r\}$. If there are multiple clusters that represent e, we choose the cluster that "best" represents e and set it as the target cluster c_0. For example, we could define the best cluster as the one that contains the largest number of records that refer to e.

The confusion of an entity is an application-specific measure. For example, we can define the confusion of e as the number of incorrect attributes of e minus the number of correct attributes of e where we count duplicate attributes. The amount of confusion we gain whenever we merge a cluster $c_i \in E(R)$ with the target cluster c_0 can be captured as the *benefit* of c_i, which is computed as $N(c_i)$ using a benefit function N. In our example above, we can define the benefit of c_i to be the number of incorrect attributes in c_i about e. Suppose that e can be represented as the record r = $\{$[Model, "C300X"], [Pixels, 30]$\}$. Then a cluster c containing the records s = $\{$[Model, "C300"], [Pixels, 20]$\}$ and t = $\{$[Model, "C200"], [Pixels, 20]$\}$ has one correct attribute (i.e., [Model, "C300X"]) and three incorrect attributes (i.e., one [Model, "C200"] and two [Pixels, 20]'s). As a result, the benefit of c

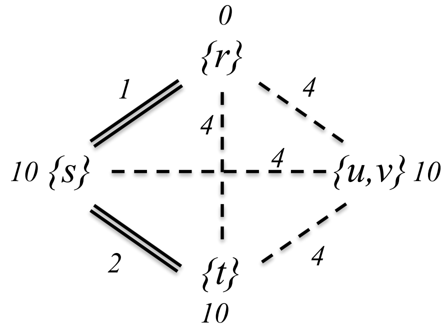


Figure 7.1: Cost Graph

is 3. As a default, we always define the benefit $N(c_0)$ to be 0 because c_0 does not need to merge with itself. In Section 7.4.1 we define a concrete metric for confusion.

Given our knowledge on the ER algorithm E, database R, cost function D, and the benefit function N, we now define an optimization problem of producing the best set of pairwise cluster merges that can maximize the total benefit while using a total cost for merging clusters within a fixed budget. We first draw an undirected cost graph G among the clusters in $E(R)$ where each edge between the clusters c_i and c_j (denoted as c_i–c_j) has a weight of $D(c_i, c_j)$. We denote the set of vertices in G as $G.V$ and the set of edges as $G.E$. For example, suppose that $E(R)$ = $\{\{r\}$, $\{s\}$, $\{t\}$, $\{u, v\}\}$ and the target cluster c_0 = $\{r\}$. Also suppose that $D(\{r\}, \{s\})$ = 1, $D(\{s\}, \{t\})$ = 2, and the rest of the edges have the weight 4. In this example, we also assume that the benefits for all clusters have the value 10, except for c_0, which has a benefit of 0. The resulting cost graph G is shown in Figure 7.1 (ignore the double lines for now).

We view any subtree J in G that has the target cluster c_0 as its root a *disinformation plan* of the entity e that specifies which pairs of clusters should be merged together through disinformation. Just like in G, we denote the set of vertices in J as $J.V$ and the set of edges in J as $J.E$. The cost of merging the clusters connected by J is then $\sum_{(c_i,c_j)\in J.E} D(c_i, c_j)$. Continuing our example above, suppose the subtree J of G connects the three clusters $\{r\}$, $\{s\}$, and $\{t\}$ with the two edges $\{r\}$–$\{s\}$ and $\{s\}$–$\{t\}$. Here, the plan is to add disinformation records between $\{r\}$ and $\{s\}$, and between $\{s\}$ and $\{t\}$ to merge the three clusters. As a result, the total merging cost of J is 1 + 2 = 3, and the total benefit obtained is 0 + 10 + 10 = 20. Figure 7.1 depicts the plan J by drawing double lines for the edges in J.

If the subtree J instead contained the edges $\{r\}$–$\{s\}$ and $\{r\}$–$\{t\}$, then the total merging cost would be 1 + 4 = 5 (but the benefit would be the same, i.e., 20).

Given a budget B that limits the total cost of generating disinformation, we define the optimal disinformation plan of e as follows.

Definition 7.1.1. *Given a cost graph G, a target cluster c_0, a cost function D, a benefit function N, and a cost budget B, the* optimal disinformation plan *J is the subtree of G that contains c_0 and has the maximum total benefit $\sum_{c_i \in J.V} N(c_i)$ subject to $\sum_{(c_i,c_j) \in J.E} D(c_i, c_j) \leq B$.*

Using the cost graph in Figure 7.1, suppose that $c_0 = \{r\}$ and the cost budget $B = 3$. As a result, the subtree J with the largest benefit connects the clusters $\{r\}$, $\{s\}$, and $\{t\}$ with the edges $\{r\}$–$\{s\}$ and $\{s\}$–$\{t\}$ and has a total benefit of 0 + 10 + 10 = 20 and a total merging cost of $D(\{r\}, \{s\}) + D(\{s\}, \{t\})$ = 1 + 2 = 3 $\leq B$. Merging $\{u, v\}$ to c_0 will require a total merging cost of 4, which exceeds B.

A disinformation plan provides a guideline for creating disinformation. Since we assume that all the merging costs are independent of each other, a disinformation plan satisfying Definition 7.1.1 does not necessarily lead to an optimal disinformation in the case where the costs are not independent. However, the independence assumption allows us to efficiently find out which clusters should be merged in order to increase the confusion significantly. In Section 7.4 we will study the effectiveness of disinformation plans based on the independence assumption in scenarios where the merging costs are not independent.

In general, the total benefit of the merged clusters may not be expressible as a linear sum of fixed benefits and may depend on the specific combination of clusters. The new problem can be stated by replacing the sum $\sum_{c_i \in J.V} N(c_i)$ in Definition 7.1.1 with some general function $F(J.V)$ that reflects the total benefit. While this generalization may capture more notions of confusion, there is less opportunity for an efficient computation of the optimal plan.

We now show that the disinformation problem is NP-hard in the strong sense [39], which means that the problem remains NP-hard even when all of its numerical parameters are bounded by a polynomial in the length of the input. In addition, it is proven that a problem that is NP-hard in the strong sense has no fully polynomial-time approximation

scheme unless P = NP.

Proposition 7.1.2. *Finding the optimal disinformation plan (Definition 7.1.1) is* NP*-hard in the strong sense.*

Proof. The decision version of the disinformation problem is to determine whether there exists a disinformation plan J that has a total benefit larger or equal to a given budget B while satisfying the constraints in Definition 7.1.1. We show that the decision version of the disinformation problem is NP-complete in the strong sense, which means that the problem remains NP-complete even when all of its numerical parameters are bounded by a polynomial in the length of the input. First, the decision version of the disinformation problem is in NP because we can verify any solution by comparing the total benefits and weights with the budgets in linear time. Next, we reduce the decision version of the Set Cover problem, which is known to be NP-complete in the strong sense, to the decision version of the disinformation problem. The Set Cover problem has a finite set U, a family C of subsets of U, and $k \in \mathbb{N}$. The question is whether there is a cover D of U in C with at most k sets. Let $U = \{a_1, \ldots, a_n\}$ and $C = \{s_1, \ldots, s_m\}$. We can construct an instance of the decision version of the disinformation problem as follows. We first create a vertex v_0 for c_0 with benefit 0. We then create m vertices $x_1, \ldots, x_m$ with benefit 0, each of which has an edge from v_0 with weight 1. Finally, we create n vertices $y_1, \ldots, y_n$ with benefit 1. Also, each vertex y_i has an edge from each vertex x_j with a weight of $k + 1$ if $a_i \in s_j$. As a result, there is a cover of U in C with at most k sets if and only if the graph G has a subtree J that contains v_0 such that the total benefit of J is at least n and the total weight of J is at most $k + n \times (k + 1)$. Hence, the decision version of the disinformation problem is NP-complete in the strong sense, which makes the optimization version in Definition 7.1.1 NP-hard in the strong sense. □

Given that the disinformation problem is NP-hard in the strong sense, we now consider a more restricted version of the disinformation problem where we only consider disinformation plans that have heights of at most h. Here, we define the height of a tree as the length of the longest path from the root to the deepest node in the tree. For example, if a tree has a root node and two child nodes, the height is 1. We can prove that even if h = 2, the disinformation problem is still NP-hard in the strong sense. However, if h = 1

where all the clusters other than c_0 can only be directly connected to c_0 in a disinformation plan, the disinformation problem is NP-hard in the weak sense [39] where there exists a pseudo-polynomial algorithm that returns the optimal disinformation plan.

Proposition 7.1.3. *Finding the optimal disinformation plan (Definition 7.1.1) with $h = 1$ is* NP-*hard in the weak sense.*

Proof. We first prove that the disinformation problem with h = 1 is NP-hard by showing a reduction from the 0–1 Knapsack optimization problem, which is known to be NP-hard in the weak sense. Suppose there are n items $i_1, \ldots, i_n$ with the weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$. Given a capacity C, the 0–1 Knapsack problem tries to find the subset of items that have a total weight within C, but with a maximal total value. We can reduce this problem to a disinformation problem by first creating a cluster c_0 with a benefit of $N(c_0)$ = 0. We then create for each item i_j a cluster c_j where $D(c_j, c_0)$ = w_j and $N(c_j)$ = v_j. For any subtree J in G, the merge cost is $\sum_{c_i \in J.V \setminus \{c_0\}} w_i$ while the total benefit is $\sum_{c_i \in J.V \setminus \{c_0\}} v_i$. Hence, if we find the optimal disinformation S, the optimal solution for the Knapsack problem is $\{i_j | c_j \in J.V \setminus \{c_0\}\}$. Next, we prove that the disinformation problem with h = 1 is NP-hard in the weak sense by reducing it to the 0–1 Knapsack problem. For any instance of the restricted disinformation problem with h = 1, we can create an instance of the 0–1 Knapsack problem by creating for each cluster $c_j \neq c_0$ an item i_j that has the value v_j = $N(c_j)$ and weight w_j = $D(c_j, c_0)$. Since the 0–1 Knapsack problem is NP-hard in the weak sense, the disinformation problem with h = 1 is NP-hard in the weak sense as well. □

The disinformation problem with h = 1 is interesting because it captures the natural strategy of comparing the target entity e with one other entity at a time, making it a practical approach for disinformation. In Section 7.2, we show there are an exact pseudo-polynomial algorithm and an approximate polynomial-time algorithm for the h = 1 problem. In Section 7.4, we show that disinformation plans with h = 1 perform just as good as general disinformation plans in terms of maximizing the confusion of e while taking much less time to generate.

7.1.4 Monotonicity

We now define a property of an ER algorithm that makes our disinformation techniques more effective.

Definition 7.1.4. *An ER algorithm E is* monotonic *if for any database R and disinformation record d, $\forall c_i \in E(R)$, $\exists c_j \in E(R \cup \{d\})$ where $c_i \subseteq c_j$.*

For example, say that the ER algorithm E is monotonic and $E(R) = \{\{r, s\}, \{t\}\}$. Then if we add a disinformation record d and compute $E(R \cup \{d\})$, the records r and s can never split. Thus a possible ER result would be $\{\{r, s, d\}, \{t\}\}$, but not $\{\{r, d\}, \{s\}, \{t\}\}$.

The monotonicity property is helpful in the agent's point of view because we do not have to worry about the ER algorithm splitting any clusters when we are trying to merge two clusters. As a result, the analysis of the cost graph is accurate, and the agent can better predict how the ER result would change if we add disinformation records according to the optimal disinformation plan.

In this chapter, we use the SN algorithm (see Section 2.2.2) and the HC_S algorithm (see Section 2.3.2) for evaluating our disinformation techniques. We show which ER algorithms satisfy the monotonicity property.

Proposition 7.1.5. *The HC_S algorithm is monotonic, but the SN algorithm is not monotonic.*

Proof. We first prove that the HC_S algorithm is monotonic. According to the algorithm, two records r and s merge with each other if there exists a sequence of records $[r_1(= r), \ldots, r_n(= s)]$ where for each pair (r_i, r_{i+1}) in the path, $D(r_i, r_{i+1}) \leq T$. Thus, even if we add a new record d to R, the records r and s that used to merge will have the same sequence of records that can merge them together. We now prove that the SN algorithm is not monotonic by providing a counter example. Suppose the database is $R = \{r, s\}$ where r and s match with each other. If the window size W is 2 and the sorted list of records is [r, s], the SN algorithm will compare r and s in the same window and cluster them together. Say that we add a disinformation record d that does not match with either r or s, but has a key value between those of r and s. As a result, the SN algorithm will not be able to compare r and s within the same window and thus will not cluster them together. □

Notice that if the window size is at least $|R|$ (i.e., the total number of records), then the SN algorithm does satisfies monotonicity because the algorithm reduces to a pairwise comparison of all records followed by a transitive closure.

The monotonicity property is desirable because the disinformation we generate (see Section 7.3 for details) is more likely to merge clusters according to the disinformation plan without any clusters splitting in the process.

7.2 Planning Algorithms

We start by proposing an algorithm that returns an optimal disinformation plan (Definition 7.1.1) where h = 1. Restricting h to 1 gives us the insight for solving the general problem later on. We propose a pseudo-polynomial algorithm that uses dynamic programming and runs in $O(|G.V| \times B)$ time, which is polynomial to the numerical value of the budget B, but still exponential to the length of the binary representation of B. We assume that B is an integer and that all the edges in the cost graph G have integer values. Next, we propose a 2-approximate greedy algorithm that runs in $O(|G.V| \times log(|G.V|))$ time. Finally, we propose two heuristics for the general disinformation problem based on the first two algorithms for the restricted problem.

7.2.1 Exact Algorithm for 1-Level Plans

Algorithm 13 is an exact algorithm that uses dynamic programming to solve the disinformation problem where h = 1. This algorithm is similar to a dynamic algorithm used to solve the 0–1 Knapsack problem. Given the cost graph G, the root node $c_0 \in G.V$, the cost function D, the benefit function N, and the budget B, we first assign sequential ids starting from 1 to the vertices other than c_0 in G. Each subproblem in $\{(i, t)| \; i = 0, \ldots, |G.V| - 1$ and $t = 0, \ldots, B\}$ is defined as solving the disinformation problem for a subgraph of G that contains all the vertices up to the id i along with the edges among those vertices while using the cost budget t. We use a 2-dimensional array m where $m[i, t]$ contains the maximum benefit for each subproblem (i, t). In addition, we store the clusters in the optimal disinformation plan for each subproblem in the array s. After running the algorithm, the

ALGORITHM 13: Exact algorithm for 1-level plans

```
input  : the cost graph G, the root node c_0, the cost function D, the benefit function N, the cost
         budget B, the array of the optimal benefits m, the array of the optimal disinformation
         plans s
output: the optimal disinformation plan J
1  for t = 0, ..., B do
2      m[0, t] = 0;
3      s[0, t] = {c_0};
4  for i = 0, ..., |G.V| − 1 do
5      m[i, 0] = 0;
6      s[i, 0] = {c_0};
7  for i = 1, ..., |G.V| − 1 do
8      for t = 1, ..., B do
9          if D(c_0, c_i) ≤ t ∧ m[i − 1, t − D(c_0, c_i)] + N(c_i) > m[i − 1, t] then
10             m[i, t] = m[i − 1, t − D(c_0, c_i)] + N(c_i);
11             s[i, t] = s[i − 1, t − D(c_0, c_i)] ∪ {c_i};
12         else
13             m[i, t] = m[i − 1, t];
14             s[i, t] = s[i − 1, t];
15 J.V ← s[|G.V| − 1, B];
16 J.E ← {c_0–c_i | c_i ∈ s[|G.V| − 1, B] \ {c_0}};
17 return J;
```

optimal disinformation plan J has a total benefit of $m[|G.V| - 1, B]$.

To illustrate Algorithm 13, suppose we have a cost graph G that contains the vertices c_0, c_1, c_2 and edges with the weights $D(c_0, c_1)$ = 1, $D(c_0, c_2)$ = 2, and $D(c_1, c_2)$ = ∞. Also, suppose that $N(c_1) = N(c_2) = 1$, and the cost budget B = 1. In Steps 1–6, we initialize the arrays m and s when either the first or second index is 0. We then solve the remaining subproblems in Steps 7–14. In Steps 7–8, we set i = 1 and t = 1. In Step 9, $D(c_0, c_1) = 1 \leq t = 1$, so we check the second condition. Since $m[i-1, t-D(c_0, c_i)] + N(c_1)$ = $m[0, 0] + 1 = 1 > m[i-1, t] = m[0, 1] = 0$, we set $m[1, 1] = 1$ and $s[1, 1]$ = $\{c_0, c_1\}$. We then continue to Steps 7–8 and set i = 2 and t = 1. In Step 9, $D(c_0, c_2) = 2 > t = 1$, so we set $m[2, 1] = m[1, 1] = 1$ and $s[2, 1] = \{c_0, c_1\}$. Finally, we continue to Steps 15–17 and construct the optimal disinformation plan J by setting $J.V$ = $s[2, 1]$ = $\{c_0, c_1\}$ and $J.E$ = $\{c_0–c_1\}$ and then return J. The optimal benefit is $m[2, 1]$ = 1.

Proposition 7.2.1. *Algorithm 13 generates the optimal disinformation plan with h = 1.*

Proof. We prove by induction. In the base case, if either i or t is 0, then the only possible disinformation plan is $\{c_0\}$. Now suppose neither i nor t are 0. We need to decide whether to add c_i to the disinformation plan or not. As long as $D(c_0, c_i) > t$, we can compute the optimal benefit including c_i by adding the optimal benefit of the subproblem $(i, t - D(c_0, c_i))$ and $N(c_i)$. The only other option is to compute the optimal benefit of the subproblem ($i-1$, t). The larger benefit is clearly the optimal benefit for the subproblem (i, t). Hence, we can derive the optimal benefit and disinformation plan for all subproblems. □

Proposition 7.2.2. *The time complexity of Algorithm 13 is $O(|G.V| \times B)$, and the space complexity is $O(|G.V|^2 \times T)$.*

Proof. The time complexity of Algorithm 13 is $O(|G.V| \times B)$ because the double loop in Steps 7–14 iterates $(|G.V| - 1) \times B$ times. The space complexity of Algorithm 13 is $O(|G.V|^2 \times T)$ because we need to store the optimal disinformation plan for each entry in s. Although not shown in the algorithm, the space complexity can be reduced to $O(|G.V| \times B)$ if we only store the incremental changes in the disinformation plan for each subproblem. □

7.2.2 Approximate Algorithm for 1-Level Plans

We now propose a 2-approximate greedy algorithm that runs in polynomial time. The algorithm is similar to a 2-approximation algorithm that solves the 0–1 Knapsack problem. We first add c_0 to the disinformation plan. We then select the clusters where $D(c_0, c_i) \leq B$ and sort them by the benefit-per-cost ratio $\frac{N(c_i)}{D(c_0,c_i)}$ in decreasing order into the list $[c'_1, \ldots, c'_n]$. We then iterate through the sorted list of clusters and add each cluster to the disinformation plan J until the current total cost exceeds B. Suppose that we have added the sequence of clusters $[c'_1, \ldots, c'_k]$ where $k \leq n$. If $k = n$ or $\sum_{i=1,\ldots,k} N(c_i) > N(c_{k+1})$, we return the disinformation plan J where $J.V$ = $\{c_0, c'_1 \ldots, c'_k\}$ and $J.E$ = $\{c_0–c'_i | i = 1, \ldots, k\}$. Otherwise, we return the plan J where $J.V$ = $\{c_0, c'_{k+1}\}$ and $J.E$ = $\{c_0–c'_{k+1}\}$.

For example, suppose we have a cost graph G that contains the vertices c_0, c_1, c_2, and c_3 with edges that have the weights $D(c_0, c_1)$ = 1, $D(c_0, c_2)$ = 2, $D(c_0, c_3)$ = 3, and $D(c_0, c_4)$

= 6. (The other weights are not needed to solve the problem.) Also say that the benefits are $N(c_1)$ = 3, $N(c_2)$ = 6, $N(c_3)$ = 6, $N(c_4)$ = 12, and the budget B = 5. We first sort the clusters other than c_0 that have a cost $D(c_0, c_i) \leq B$ by their $\frac{N(c_i)}{D(c_0,c_i)}$ values in decreasing order. The benefit-per-cost ratios of c_1, c_2, c_3, c_4 are 3, 3, 2, 2, respectively. Since c_4 cannot be in the list because $D(c_0, c_4) = 6 > B = 5$, we obtain the sorted list [c_1, c_2, c_3]. We then scan the sorted list and add each cluster to the disinformation plan J until the total cost exceeds B. Since $N(c_1)$ + $N(c_2)$ = 9 $>$ $N(c_3)$ = 6, we have $J.V$ = $\{c_0, c_1, c_2\}$ and $J.E$ = $\{c_0$–c_1, c_0–$c_2\}$ with a total benefit of 3 + 6 = 9. The optimal solution turns out to be $J.V$ = $\{c_0, c_2, c_3\}$ and $J.E$ = $\{c_0$–c_2, c_0–$c_3\}$ with a benefit of 6 + 6 = 12, demonstrating that the greedy algorithm is not an optimal algorithm.

Proposition 7.2.3. *The greedy algorithm generates a 2-approximate optimal disinformation plan with h = 1.*

Proof. Let d = $\sum_{j=1...k} D(c_0, c_j)$ and b = $\sum_{j=1...k} N(c_j)$. Since d + $D(c_0, c_{k+1}) > B$ and all the clusters are sorted by their $\frac{N(c_i)}{D(c_0,c_i)}$ ratios, we conclude that the optimal benefit OPT $< b + N(c_{k+1})$. The greedy algorithm returns a total benefit of $max\{b, N(c_{k+1})\}$ when $k < n$ (if k = n, all the clusters are included in the plan, so the greedy algorithm is optimal). Hence, OPT $< b + N(c_{k+1}) \leq 2 \times max\{b, N(c_{k+1})\}$, which makes the solution of the greedy algorithm 2-approximate. □

Proposition 7.2.4. *The time complexity of the greedy algorithm is $O(|G.V|^2 \times log(|G.V|))$, and the space complexity is $O(|G.V|)$.*

Proof. Since the greedy algorithm first needs to sort the clusters in $G.V$, it takes $O(|G.V|^2 \times log(|G.V|))$ time to run. The space complexity is $O(|G.V|)$ since we only need to store the sorted information of clusters. □

7.2.3 Heuristics for General Plans

Since the general disinformation problem (Definition 7.1.1) is NP-hard in the strong sense, there is no exact pseudo-polynomial algorithm or approximate polynomial algorithm for the problem. Instead, we propose two heuristics that extend the algorithms in Sections 7.2.1 and 7.2.2 to produce disinformation plans with no restriction in the heights.

Algorithm 14 (called EG) is a heuristic that repeatedly runs Algorithm 13 for constructing each level of the disinformation plan. As a result, Algorithm 14 always returns a disinformation plan that is at least as good as the 1-level plan generated by Algorithm 13. To illustrate Algorithm 14, suppose we are using the cost graph G in Figure 7.1 and set B = 3. In Steps 1–2, we initialize the disinformation plan J by setting $J.V$ = $\{\{r\}\}$ and $J.E$ = $\{\}$. In Steps 3–5, we set c = $\{r\}$, G' = G, and B' = 3. In Step 7, we run Algorithm 13 to derive the best 1-level plan for G' and B'. In our example, the result is J' where $J'.V$ = $\{\{r\}$, $\{s\}\}$ and $J'.E$ = $\{\{r\}$–$\{s\}\}$. Since $|J'.V| = 2 > 1$ in Step 8, we continue to Steps 10–20 where we merge the clusters $\{r\}$ and $\{s\}$ within G' and update the edges accordingly. In Step 10, we update the budget B' to $3 - 1 = 2$. In Steps 11–12, we update J by setting $J.E$ to $\{\{r\}$–$\{s\}\}$ and $J.V$ to $\{\{r\}$, $\{s\}\}$. In Step 13, we updated c to the newly merged cluster $\{r, s\}$. In Steps 14–15, we update $G'.V$ to $\{\{r, s\}$, $\{t\}$, $\{u, v\}\}$. In Step 16, we remove the unnecessary edges in G' by setting $G'.E$ to $\{\{t\}$–$\{u, v\}\}$. In Steps 17–20, we add the new edges connecting c to the vertices in G' and update the cost function D accordingly. As a result, $G'.E$ becomes $\{\{t\}$–$\{u, v\}$, $\{r, s\}$–$\{t\}$, $\{r, s\}$–$\{u, v\}\}$ and the following weights are updated: $D(\{r, s\}$, $\{t\})$ = $D(\{t\}$, $\{r, s\})$ = $min\{D(\{r\}$, $\{t\})$, $D(\{s\}$, $\{t\})\}$ = 2 and $D(\{r, s\}$, $\{u, v\})$ = $D(\{u, v\}$, $\{r, s\})$ = $min\{D(\{r\}$, $\{u, v\})$, $D(\{s\}$, $\{u, v\})\}$ = 4. We now repeat the loop in Steps 6–20 with the updated B', G', and c values. This time, we merge $\{t\}$ to $\{r, s\}$ and get the results T' = 0, $G'.V$ = $\{\{r, s, t\}$, $\{u, v\}\}$, $G'.E$ = $\{\{r, s, t\}$–$\{u, v\}\}$, and c = $\{r, s, t\}$. In the next loop, $\{u, v\}$ cannot merge with $\{r, s, t\}$ because B' = 0, so we break the while loop at Step 9 and return J in Step 21 where $J.V$ = $\{\{r\}$, $\{s\}$, $\{t\}\}$ and $J.E$ = $\{\{r\}$–$\{s\}$, $\{s\}$–$\{t\}\}$.

Proposition 7.2.5. *The time complexity of Algorithm 14 is* $O(|G.V|^2 \times B + |G.V|^3)$*, and the space complexity is* $O(|G.V|^2 \times B)$.

Proof. Algorithm 14 has a time complexity of $O(|G.V|^2 \times log(|G.V|))$ because each time a cluster with the highest benefit per cost ratio is merged to c_0, we re-sort the list of remaining clusters. The space complexity is $O(|G.V|)$ because we only need to store a sorted list of clusters in G. □

Our second heuristic (called AG) extends the greedy algorithm in Section 7.2.2. Again, we first sort the clusters other than c_0 that have a cost $D(c_0, c_i) \leq B$ by their $\frac{N(c_i)}{D(c_0,c_i)}$ values

ALGORITHM 14: Heuristic using Algorithm 13 for general plans

input : the budget B, the cost graph G, the root node c_0, the cost function D, and the benefit function N

output: a disinformation plan J

```
1  J.V ← {c0};
2  J.E ← {};
3  c ← c0;
4  G' ← G;
5  B' ← B;
6  while B' > 0 do
7      J' ← Alg. 13(B', G', D, N, c, [], []);
8      if |J'.V| = 1 then
9          break;
10     B' ← B' − Σ_{(ci,cj)∈J'.E} D(ci, cj);
11     J.E ← J.E ∪ {ci–cj | ci ∈ J'.V \ {c} and cj = argmin_{d∈J.V} D(d, ci)};
12     J.V ← J.V ∪ (J'.V \ {c});
13     c ← ⋃_{d∈J.V} d;
14     G'.V ← G'.V \ J'.V;
15     G'.V ← G'.V ∪ {c};
16     G'.E ← G'.E \ {(ci, cj) | (ci, cj) ∈ G'.E ∧ (ci ∈ J'.V ∨ cj ∈ J'.V})};
17     for d ∈ G'.V \ {c} do
18         G'.E ← G'.E ∪ {c–d};
19         D(d, c) ← min_{e∈J'.V} D(d, e);
20         D(c, d) ← D(d, c);
21 return J;
```

in decreasing order. The algorithm then only merges the cluster with the highest benefit-per-cost ratio to the closest cluster in the current plan and updates the edges and the budget just like in the EG algorithm. We also update the disinformation plan J by setting $J.E$ = $J.E \cup \{c_i\text{–}c_j | c_j = argmin_{c \in J.V} D(c, c_i)\}$ and then $J.V$ = $J.V \cup \{c_i\}$. (The ordering of the updates is important.) We repeat the process of sorting the remaining clusters and merging the best one with the closest cluster in the plan until no cluster can be merged without costing more than the budget.

To illustrate AG, suppose we again use the cost graph G in Figure 7.1 and set B = 3. We first sort the clusters by their benefit per cost ratio. As a result, we get the sorted list [$\{s\}$, $\{t\}$] where $\{u, v\}$ is not in the sorted list because its cost 4 already

exceeds the budget B. We then merge $\{s\}$ with $\{r\}$ and create the edges $\{r,s\}$–$\{t\}$ and $\{r,s\}$–$\{u,v\}$ with the weights $D(\{r,s\},\{t\}) = D(\{t\},\{r,s\}) = min\{2,4\} = 2$ and $D(\{r,s\},\{u,v\}) = D(\{u,v\},\{r,s\}) = min\{4,4\} = 4$, respectively. We then re-sort the remaining clusters according to their benefit per cost ratio. This time, we get the sorted list $[\{t\}]$ where $\{u,v\}$ again has a cost of 4, which exceeds the current budget 2. We then merge $\{t\}$ with $\{r,s\}$ and create the edge $\{r,s,t\}$–$\{u,v\}$ with the weight $D(\{r,s,t\},\{u,v\}) = D(\{u,v\},\{r,s,t\}) = 4$. Since no cluster can now merge with $\{r,s,t\}$, we terminate and return the plan J where $J.V = \{\{r\},\{s\},\{t\}\}$ and $J.E = \{\{r\}$–$\{s\},\{s\}$–$\{t\}\}$.

Proposition 7.2.6. *The time complexity of the AG algorithm is $O(|G.V|^2 \times log(|G.V|))$, and the space complexity is $O(|G.V|)$.*

Proof. The AG algorithm has a time complexity of $O(|G.V|^2 \times log(|G.V|))$ because each time a cluster with the highest benefit per cost ratio is merged to c_0, we re-sort the list of remaining clusters. The space complexity is $O(|G.V|)$ because we only need to store a sorted list of clusters in G. □

7.3 Creating New Records

In this section, we discuss how to create new records for disinformation based on existing records. At first, we assume that the records are in a Euclidean space. We then propose various strategies for generating disinformation when the records are not in a Euclidean space.

7.3.1 Euclidean Space

Suppose the agent is inducing a merge between two clusters c_i and c_j in a Euclidean space by creating disinformation records in between. One method is to find the centroids r_i and r_j of c_i and c_j, respectively, by averaging the values of the records for each cluster, and then creating new records on the straight line connecting r_i and r_j. For example, if there are two clusters c_1: {{[X, 20], [Y, 7]}} and c_2: {{[X, 30], [Y, 8]}, {[X, 50], [Y, 8]}}, then the agent first generates the centroids r_1: {[X, 20], [Y, 7]} and r_2: {[X, 40], [Y, 8]}. If

the agent wants to generate a point exactly in the middle of r_1 and r_2 according to the Euclidean space, she can create the record t: $\{$[X, 30], [Y, 7.5]$\}$ by averaging the values for each attribute. If generating one disinformation record is not enough, the agent can further generate disinformation records that are between r_1 and t and between r_2 and t. In our example, the agent can create the disinformation records $\{$[X, 25], [Y, 7.25]$\}$ and $\{$[X, 35], [Y, 7.75]$\}$. Hence, the agent can easily create disinformation records based on existing values in a Euclidean space.

7.3.2 Non-Euclidean Space

Suppose that the clusters to merge – c_i and c_j – are in a non-Euclidean space, but the agent can still compute the pairwise distances between records. (An ER algorithm computes distances or similarities between records.) Again, the idea is to find the records r_i and r_j that represent c_i and c_j, respectively, and then create disinformation records between r_i and r_j. Since the two clusters are not in a Euclidean space, however, the agent can no longer compute the centroids of the clusters. An alternative way to choose r_i and r_j is to identify the clustroids of c_i and c_j based on the pairwise distances between records. Intuitively, a clustroid of a cluster c is the record $r \in c$ that is "closest" to the other points in c. For example, r may have the smallest average distance to the other points in c. Thus if the cluster is c: $\{\{$[X, "abc"]$\}$, $\{$[X, "bcd"]$\}$, $\{$[X, "cde"]$\}\}$, and the distance between two records is computed by taking the edit distance of their string values, the clustroid is $\{$[X, "bcd"]$\}$. The agent can also choose the records $r_i \in c_i$ and $r_j \in c_j$ that are closest to (or furthest from) each other.

The agent can create disinformation records between r_i and r_j by mapping them into more restricted spaces (e.g., a Euclidean space). For each attribute, say that the agent wants to find the middle point v_m of two values v_1 and v_2 such that the distances $d(v_1, v_m)$ and $d(v_2, v_m)$ are approximately the same in the non-Euclidean space. We assume there is a mapping function M for each attribute of a record that converts the values v_1 and v_2 into the values u_1 and u_2 in the restricted space. Once the agent operates in the restricted space, she can more easily find the mid-point u_m where the distance $d'(u_1, u_m)$ is the same as

$d'(u_2, u_m)$. The agent can then use an inverse function M^{-1} to map u_m back into a non-Euclidean value, which becomes the desired midpoint v_m.

Suppose the agent is using a Euclidean space as the restricted space, and there are two records containing strings: r: {[X, "C300X"]} and s: {[X, "C300"]}. The agent can convert each string into a real number that reflects the string's rank in the alphabetically sorted list of all possible strings. Here, we assume that strings are matched based on their closeness in alphabetical order and not by other measures such as edit distance. Once the agent computes the average value of M("C300X") and M("C300") and converts that value back into a string using M^{-1}, she obtains the string "C300LMM...", so the new record is {[X, "C300LMM..."]}.

The agent can also create disinformation records in a non-Euclidean space using a total order of values in a dictionary as the restricted space. This mapping is useful if the agent wants to make sure the values in a disinformation record are realistic. In our example above, the agent created the camera model "C300LMM...". However, if the adversary is knowledgeable in camera products, she could easily distinguish the disinformation records from the real ones. The solution is to only use values in a dictionary that contains a sequence of all the possible realistic values in the non-Euclidean space. For instance, if the dictionary is all the valid camera models in alphabetical order, then the model that is closest to "C300LMM..." could be "C300S", so the agent can create the disinformation record {[X, "C300S"]}.

If the values of existing records cannot be mapped to a total ordering of values in a dictionary, the agent can use a partial ordering of values that form a lattice instead. For example, say that the agent has two records containing sets: r: {[X, {1, 2}]} and s: {[X, {2, 3}]}. Also say the agent uses the Jaccard similarity when measuring the distances between sets. As a result, the similarity between r and s is $\frac{1}{3}$. Since sets of elements form a lattice, the agent can assign the common ancestor of the two sets (i.e., the union of the two sets {1, 2, 3}) as the value of the new record. The resulting disinformation record {[X, {1, 2, 3}]} has a distance of $\frac{2}{3}$ from each record.

Finally, if the agent cannot create new values at all, she can still create new records by only using the values in existing records. For example, if r = {[X, "C300"], [Y, 20]} and s = {[X, "C300X"], [Y, 30]}, then she can create the disinformation record {[X, "C300"], [Y,

30]}.

In general, the more restrictive the mapping space is, the costlier it is to create disinformation records.

7.4 Experiments

We evaluate the disinformation planning algorithms in Section 7.2 on synthetic data (Section 7.4.1) and then on real data (Section 7.4.2). We compare the robustness of the SN and HC_S algorithms. Our algorithms were implemented in Java, and our experiments were run in memory on a 2.4GHz Intel(R) Core 2 processor with 4 GB of RAM.

7.4.1 Synthetic Data Experiments

We first evaluate our disinformation techniques using synthetic data. The main advantage of synthetic data is that they are much easier to generate for different scenarios and provide more insights into the operation of our disinformation planning algorithms.

In general, there are two types of attributes in records: attributes used for matching records and ones that contain additional properties. For our synthetic data, we only create attributes needed for record matching and do not model the additional properties. We consider a scenario where records in a non-Euclidean space are converted to records in a Euclidean space using a mapping function M (see Section 7.3.2). As a result, all the converted records contain real numbers in their attributes. We then run ER and generate the disinformation records in the Euclidean space. The disinformation records could then be converted back into the non-Euclidean space using the inverse mapping function M^{-1}. We do not actually use the functions M and M^{-1}, but directly generate the mapped synthetic records in the Euclidean space.

Table 7.2 shows the parameters used for generating the synthetic database R and the default values for the parameters. There are s entities in the dataset that are distributed on a d-dimensional Euclidean space. For each dimension, we randomly assign the values in the list $[0, i, 2 \times i, \ldots, (s-1) \times i]$ to the s entities. As a result, any two entities have a distance of at least i from each other for any dimension. For each entity e, the data set

Par.	Description	Val.
Data Generation		
s	Number of entities	100
u	Avg. number of duplicate records per entity	10
f	Zipfian exponent number of # duplicates	1.0
d	Number of attributes (dimensions) per record	2
i	Minimum value difference between entities	50
v	Maximum deviation of value per entity	50
g	Zipfian exponent number of deviation	1.0
Match Rule		
t	Record comparison threshold	50

Table 7.2: Parameters for generating synthetic data

contains an average of u records that represent that entity, where the number of duplicates form a Zipfian distribution with an exponent of f. Each record r generated for the entity e contains d attributes. For each attribute a, r contains a value selected from a Zipfian distribution with an exponent of g within the range of $[x, x + v]$ where x is the a value of e and v is the maximum deviation of a duplicate record's value from its entity value.

ER Algorithms We experiment on the HC_S and SN algorithms. The HC_S algorithm repeatedly merges the closest clusters together until the closest-cluster distance exceeds the comparison threshold t = 50. The default value for t was set to be equal to the minimum value difference parameter i. When comparing two records r and s across clusters, the match function computes the Euclidean distance between the two records and checks if the distance is within t. For example, if $r = \{[v_1, 1], [v_2, 1]\}$ and $s = \{[v_1, 2], [v_2, 3]\}$, then the Euclidean distance is $\sqrt{(2-1)^2 + (3-1)^2} = \sqrt{5}$, which is smaller than t = 50. The SN algorithm sorts the records according to their first dimension value (of course, there are other ways to sort the records) and then uses a sliding window of size W to compare the records using a Boolean match function that returns true if the Euclidean distance of two records is within t and false otherwise. The smaller the window size W, the fewer records are compared. However, a window size that is too small will prevent SN from properly resolving the records. In our experiments, we set a window size of W = 20 so that SN was efficient and yet had nearly identical ER results as the HC_S algorithm when resolving R.

Confusion Metric We define a confusion metric C for a cluster c. In the motivating example in the beginning of this chapter, the adversary was confused into whether the records in the cluster $c = \{r, s\}$ (excluding the disinformation record) represented the same camera model C300X. However, the correct information of the target entity e = C300X was $\{r\}$. We first define the precision Pr of c as the fraction of non-disinformation records in c that refer to e. In our example, $Pr = \frac{1}{2}$. We also define the recall Re of c as the fraction of non-disinformation records that refer to e that are also found in c. Since there is only one record that refers to the C300X, $Re = \frac{1}{1}$. The F_1 score [70] represents the overall accuracy of the information in c and is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$. In our example, the F_1 score of c is $\frac{2 \times \frac{1}{2} \times \frac{1}{1}}{\frac{1}{2} + \frac{1}{1}} = \frac{2}{3}$. Finally, we define the confusion C of c as $1 - F_1$ where we capture the notion that the lower the accuracy, the higher the confusion the adversary has on c. In our example, the confusion of c is $C(c) = 1 - \frac{2}{3} = \frac{1}{3}$.

Target Entity and Cluster Before generating the disinformation, we choose one entity as the target entity e and then choose the cluster in the agent's ER result that "best" represents e as the target cluster c_0. There may be several clusters that contain records of e when the ER algorithm does not properly cluster the records that refer to e. In this case, we set the cluster with the lowest confusion as the target cluster. For example, suppose that the agent's ER result of R is $\{\{r_1, r_2, s_1\}, \{r_3, s_2\}\}$ where each record r_i refers to the entity e_1 and each record s_i refers to the entity e_2. If e_1 is our target entity, the confusion values of the two clusters are $1 - \frac{2 \times 2/3 \times 2/3}{2/3 + 2/3} = \frac{1}{3}$ and $1 - \frac{2 \times 1/2 \times 1/3}{1/2 + 1/3} = \frac{3}{5}$, respectively. Since $\frac{1}{3} < \frac{3}{5}$, we set $\{r_1, r_2, s_1\}$ as the target cluster c_0.

As a default, we choose the entity with the largest number of duplicates to be the target entity e. According to our data generation method, there is only one entity that has the most duplicates because of the Zipfian distribution of the number of duplicates per entity. Notice that we are using a worst-case scenario where the many duplicates of e makes it difficult to dilute e's information by merging clusters.

Benefit and Cost Functions We define the benefit function N to return the size $|c|$ of each cluster c. If we use the plan J for generating disinformation, we obtain a total benefit of $\sum_{c \in J.V} |c|$ and a confusion of $C(c_0 \cup \bigcup_{c \in J.V} c)$. While maximizing $\sum_{c \in J.V} |c|$ does not

Algorithm	Description
$E2$	Exact algorithm for 1-level plans
EG	Heuristic extending $E2$ for general plans
$A2$	Greedy algorithm for 1-level plans
AG	Heuristic extending $A2$ for general plans

Table 7.3: Disinformation Plan Algorithms

necessarily maximize $C(c_0 \cup \bigcup_{c \in J.V} c)$, we will see that we can still obtain a high confusion in practice. In the special case where the recall Re of the target cluster c_0 is 1, we can show that maximizing $C(c_0 \cup \bigcup_{c \in J.V} c)$ is in fact equivalent to maximizing $\sum_{c \in J.V} |c|$.

We define the cost function D to return the number of disinformation records that need to be created when merging two clusters. For example, if we need to generate two disinformation records d_1 and d_2 to merge the clusters $\{r\}$ and $\{s\}$, then $D(\{r\}, \{s\}) = 2$. Notice that the budget B thus specifies the maximum number of disinformation records that can be generated.

Disinformation Generation When creating disinformation records to merge the two clusters c_i and c_j, we first measure the distance between the centroids of the clusters. We then create disinformation records along the straight line in the Euclidean space connecting the two centroids with an interval of at most t so that any two consecutive records along the line are guaranteed to match with each other. For example, if $c_1 = \{r : \{[v, 1]\}, s : \{[v, 3]\}\}$ and if $c_2 = \{t : \{[v, 7]\}\}$, then the centroid of c_1 is $\{[v, 2]\}$, and the centroid of c_2 is $\{[v, 7]\}$. If the distance threshold $t = 2$, the merging cost is $\lceil \frac{7-2}{2} \rceil - 1 = 2$, and we can create the two disinformation records $\{[v, 4]\}$ and $\{[v, 6]\}$. In our experiments, we use the four disinformation planning algorithms defined in Section 7.2, which are summarized in Table 7.3.

ER Algorithm Robustness

We compare the robustness of the HC_S and SN algorithms against the $E2$ planning algorithm. (Using any other planning algorithm produces similar results.) We vary the budget B from 100 to 400 records and see the increase in confusion as we generate more disinformation records. Since we choose the target entity as the one with the largest number

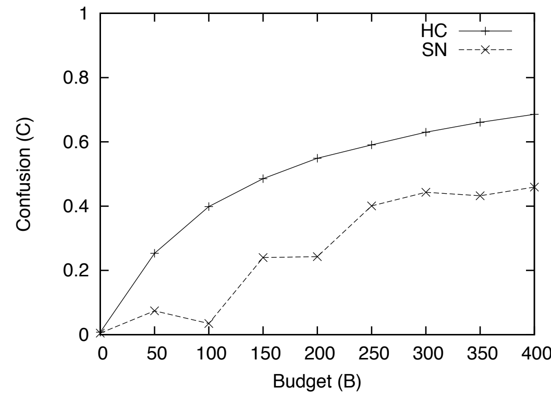


Figure 7.2: Robustness of the HC_S and SN algorithms

of duplicates, it takes many disinformation records to significantly increase the confusion. For target entities with fewer duplicates, the increase of confusion is much more rapid (see Section 7.4.1).

Figure 7.2 shows that the overall confusion results for the SN algorithm are lower than those of the HC_S algorithm. Initially, the ER results without the disinformation were nearly the same where the SN algorithm produced 105 clusters with the largest cluster of size 195 while the HC_S algorithm produced 104 clusters with the largest cluster of size 196. However, as we add disinformation records, the SN algorithm shows a much slower increase in confusion, demonstrating that it is more robust to disinformation than the HC_S algorithm. The main reason is that HC_S satisfies monotonicity, so clusters are guaranteed to merge by adding disinformation whereas the SN algorithm may not properly merge the same clusters despite the disinformation.

Figure 7.3 compares the four planning algorithms using the SN algorithm. We can observe in the figure that the EG, $E2$, and $A2$ algorithms have similar confusion results. Interestingly, the AG algorithm performs consistently worse than the other three algorithms when the budget exceeds 100. The reason is that the AG algorithm was generating disinformation plans with large heights (e.g., the optimal plan when B = 200 had a height of 8), but the SN algorithm was not able to merge all the clusters connected by the plan due to the limited sliding window size. For example, even if two clusters c_1 and c_2 were connected with a straight line of disinformation records, the records of some other cluster c_3

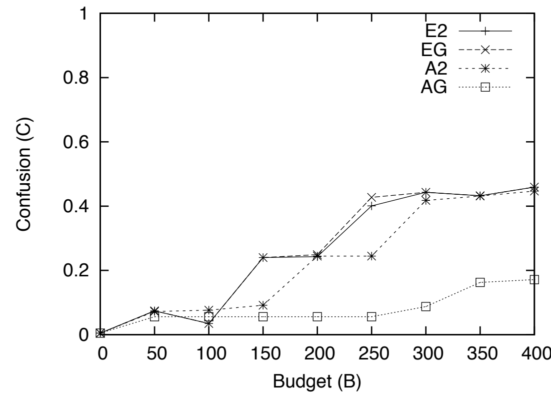


Figure 7.3: Comparison of disinformation algorithms

were preventing some of the records connecting c_1 and c_2 from being compared within the same sliding window.

Another observation is that the confusion plots do not necessarily increase as the budget increases. For example, the confusion of the $A2$ algorithm decreases when B increases from 50 to 100. Again, the reason is that the disinformation was not merging clusters as planned due to the random intervention of other existing clusters. The frequency of failing to merge clusters strongly depends on the data and sliding window size. That is, if we were to use a database other than R, then the AG algorithm could have different confusion results compared to the one shown in the graph. We conclude that the 1-level planning algorithms can actually perform better than the general planning algorithms when using the SN algorithm.

To see the relation between the sliding window size of the SN algorithm and the increase of confusion, we vary the window size from 10 to 50 and observe the confusion of the SN algorithm against the $E2$ disinformation planning algorithm using a budget of B = 200 records. As the window size increases, the confusion increases because the disinformation records have a higher chance of merging the clusters they connect.

We now test the robustness of the HC_S algorithm against the four disinformation planning algorithms. We vary the budget B from 100–400 records and see the increase in confusion as we generate more disinformation records. Figure 7.5 shows that all four planning algorithms have smoothly increasing plots where the confusion initially increases rapidly

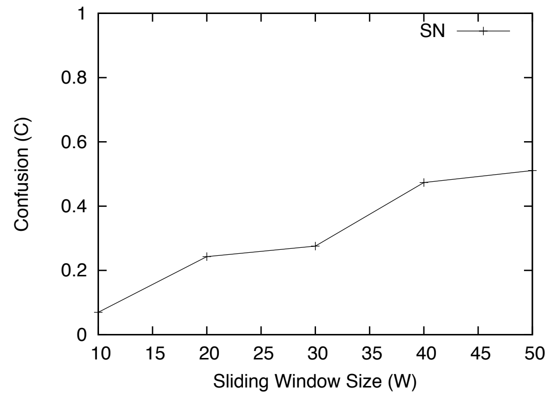


Figure 7.4: Window size impact on confusion for SN

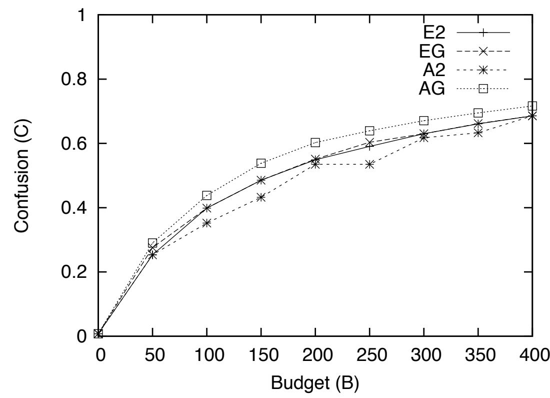


Figure 7.5: Robustness of the HC_S algorithm

and then slows down as merging large clusters to the target cluster becomes more difficult. The EG algorithm performs slightly better than the $E2$ algorithm while the $A2$ algorithm performs slightly worse than the $E2$ algorithm. The AG algorithm consistently outperforms the three other algorithms by up to a confusion difference of 0.11. Nonetheless, the results suggest that the 1-level plan algorithms have confusion performances comparable to the general plan algorithms using the HC_S algorithm.

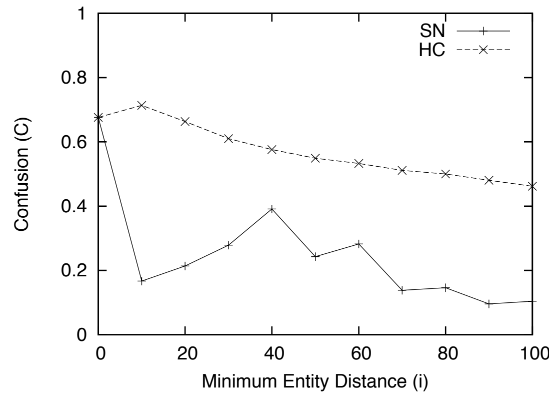


Figure 7.6: Entity distance impact on confusion

Entity Distance Impact

We investigate how the distances among entities influence the confusion results. Figure 7.6 shows how the accuracies of the SN and HC_S algorithms change depending on the minimum value difference i between entities using a budget of B = 200 records. The closer the entities are with each other (i.e., as i decreases), the more likely the ER algorithm will mistakenly merge different clusters, which leads to a higher confusion. The HC_S algorithm plot clearly shows this trend. The only exception is when i decreases from 10 to 0. The confusion happens to slightly decrease because some of the records that were newly merged with the target cluster were actually correct records that referred to e. The SN algorithm plot becomes increasingly unpredictable as i decreases. The reason is that when merging two clusters with disinformation, there is a higher chance for other clusters to randomly interfere with the disinformation.

Universality of Disinformation

In practice, the agent may not be able to tell which ER algorithm the adversary will use on her database. Hence, it is important for our disinformation techniques to be universal in a sense that the disinformation records generated from the agent's ER algorithm should increase the confusion of the target entity even if the adversary uses any other ER algorithm. We claim that, as long as the ER algorithm used for generating the disinformation "correctly" clusters the records in the database, the optimal disinformation generated by

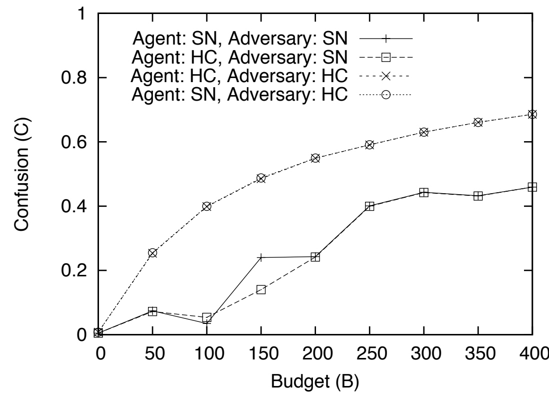


Figure 7.7: Universal disinformation

using the agent's ER algorithm are indeed applicable when the adversary uses a different ER algorithm.

Figure 7.7 shows the results of using the disinformation generated when the agent assumes the SN (HC_S) algorithm while the adversary actually uses the HC_S (SN) algorithm. We observe that there is almost no change in the confusion results compared to when the agent and adversary use the same ER algorithms. The reason is that the SN and HC_S algorithms identified nearly the same entities when resolving R, so the disinformation that was generated were nearly the same as well.

In a worst-case scenario, the ER algorithms of the agent and adversary may produce very different ER results for R, leading to different disinformation results as well. However, the different ER results means that one (or both) of the ER algorithms must have incorrectly resolved R. Suppose that the agent's ER algorithm clustered the records correctly and the disinformation was generated using that ER algorithm. Then although the disinformation may not significantly increase the confusion of the adversary's (incorrect) ER algorithm, the adversary's ER algorithm produced a high confusion in the first place, so it is natural that the disinformation cannot further increase the confusion. Thus, as long as we generate the disinformation records from correct ER results, the records can be universally applied to any other correctly running ER algorithm.

Table 7.4: Decrease in confusion (%) with sampling

Sampling Rate	Budget							
	50	100	150	200	250	300	350	400
20	15.6	14.1	12.1	13.4	9.6	9.5	10.6	8.2
40	7	10.3	11.8	10.5	9.8	10.2	9.2	8.3
60	3.5	5.4	5	7.2	6.9	6.6	7	7.1
80	0	3.9	3.5	3.4	2.8	2.2	2.6	3

Partial Knowledge

Until now, we have assumed the agent to have a complete knowledge of the database R. In reality, the agent may not know all the information in the public. For example, Cakon may not know every single rumor of its new camera model on the Web. Hence, we investigate how the agent only having a partial knowledge of the database influences the confusion results. We first compute the ER result of the HC_S algorithm on R and select the target cluster. We then take a random sample of the clusters in the ER result (without the target cluster) and add them to the partial information. We then generate the disinformation records based on the target cluster and partial information.

Table 7.4 shows the decrease in confusion (%) relative to the confusion using the full information without sampling. We vary the sampling rate from 20 to 80%. As the sampling rate goes up, the closer the confusion values are to those of the full information. For example, if we use a budget of 200 records, a sampling rate of 20% decreases the full-information confusion by 13.4% while a sampling rate of 80% only decreases the confusion by 3.4%. Nevertheless, we conclude that the disinformation generated from partial information is still effective.

Restricted Values

We explore how restrictions in creating values can influence the confusion results. When creating disinformation records, one may need to create values that are realistic to the adversary. For example, when creating fake camera models, the agent may only be able to use a valid model name. In our experiments, we simulate this restriction by only allowing an attribute value of a disinformation record to be multiples of an integer, which we call the

Table 7.5: Decrease in confusion (%) with restrictions

Granularity	Budget							
	50	100	150	200	250	300	350	400
20	13.1	-2.3	3.8	1.6	3.5	2.3	2.7	2.3
40	26.7	15.8	9.9	8.7	6.8	5.8	5.3	5.2

granularity. If the granularity is 2, then the disinformation record $\{[v_1, 0], [v_2, 2]\}$ is valid, but $\{[v_1, 0], [v_2, 3]\}$ is not valid.

When generating disinformation records that connect two centroids r and s, we start from one of the centroids (say r) and create the disinformation record d that satisfies the granularity constraint and has the shortest Euclidean distance to s while being within a Euclidean distance of the comparison threshold t from r. We then repeat the same search starting from d and continue creating disinformation records until the current disinformation record is within a Euclidean distance of t from s. For example, suppose that r = $\{[v_1, 0], [v_2, 0]\}$ and s = $\{[v_1, 5], [v_2, 3]\}$. Suppose that we only allow the attribute values of disinformation records to be multiples of 2. If the comparison threshold $t = 2\sqrt{2}$ and we start from r, then we create the disinformation record $d_1 = \{[v_1, 2], [v_2, 2]\}$, which is the closest record to s that has a distance within $2\sqrt{2}$ from r. We then create the next disinformation record $d_2 = \{[v_1, 4], [v_2, 2]\}$. Alternatively, we could have created the record $\{[v_1, 4], [v_2, 4]\}$. Since d_2 is within a distance of $\sqrt{2}$ from s, we return the two records $\{d_1, d_2\}$ as the final disinformation. Table 7.5 shows the difference in confusion as we increase the granularity from 20 to 40 when we use the HC_S algorithm. Any granularity larger than the comparison threshold t = 50 is not feasible because there is no way to create two disinformation records that match with each other. As the granularity increases, the confusion values decrease as well for the same budget because it takes more disinformation records to merge clusters. When B = 100, setting the granularity to 20 temporarily results in a higher confusion than the result without restrictions because there were relatively more disinformation records that were being generated, and these records were merging more clusters to the target cluster beyond what was planned. We conclude that the disinformation generated with value restrictions is still effective unless the restrictions make it absolutely infeasible to create disinformation.

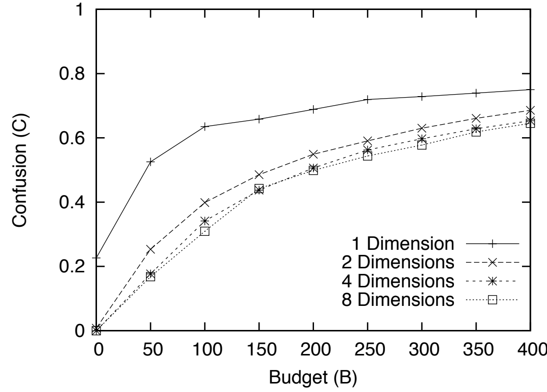


Figure 7.8: Number of dimensions impact on confusion

Number of Dimensions

In practice, records may contain more than two attributes and thus be in high-dimensional spaces. To see how our disinformation planning algorithms work for high-dimensional data, we increase the number of dimensions d of the synthetic data. The other parameters in Table 7.2 were set to their default values. We also increased the comparison threshold t in proportion to $\sqrt{d}$. For example, if t was the threshold used in a 2-dimension space, we use the threshold $\frac{\sqrt{3}}{\sqrt{2}}t$ in a 3-dimensional space.

Figure 7.8 shows the confusion results when we use the HC_S algorithm and increase the number of dimensions from 1 to 8. As a result, the confusion values decrease as d increases, but converge for large dimensions. The reason that the confusion values decrease is because the average distance between entities increases slightly faster than the comparison threshold t. As a result, fewer clusters are merged together, so the confusion decreases. For example, suppose that we create three entities in an n-dimensional space. If n = 1, then according to our construction method, the three entities have the values 0, i, and $2i$. The average entity distance is always $\frac{4i}{3}$. Now if we create three entities on a 2-dimensional space, we can show that the average entity distance for any assignment of entities is at least $\sqrt{2}\frac{4i}{3}$. For example, one possible assignment of the values could be (0, $2i$), (i, 0), and ($2i$, i). In this case, the average entity distance is $\frac{(2\sqrt{5}+\sqrt{2})i}{3}$, which is larger than $\sqrt{2}\frac{4i}{3}$. Since t only increases by $\sqrt{2}$, the entities are now relatively further away from each other.

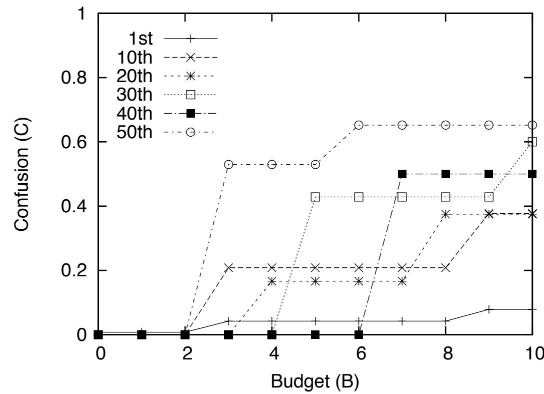


Figure 7.9: Entities with fewer duplicates

Target Entities with Fewer Duplicates

In this section, we consider target entities that have fewer duplicates and observe how their confusion values increase against disinformation. The fewer the duplicates, the more rapidly the confusion increases as a result of merging clusters. For example, suppose that Cakon has made an official announcement of a new camera model. With a lot of press coverage (i.e., there are many duplicate records about the model), it is hard to confuse the adversary of this information even with many false rumors. However, if Cakon has not made any announcements, and there are only speculations about the new model (i.e., there are few duplicate records), then it is much easier to confuse the adversary by adding just a few false rumors.

Figure 7.9 shows the confusion results when we use the entities with the k-th most duplicates as the target entities where k varied from 1 to 50. (Recall there is a total of s = 100 entities.) For each entity, we measure its confusion against disinformation generated by the $E2$ algorithm using a budget B of at most 10. The other parameters in Table 7.2 were set to their default values. As a result, the entities with fewer duplicates tend to have a more rapid increase in confusion against the same budget. For example, we only need to generate 3 disinformation records to increase the confusion of the entity with the 50-th largest number of duplicates to 0.53. Our results show that it is easier to confuse the adversary on entities with fewer duplicates.

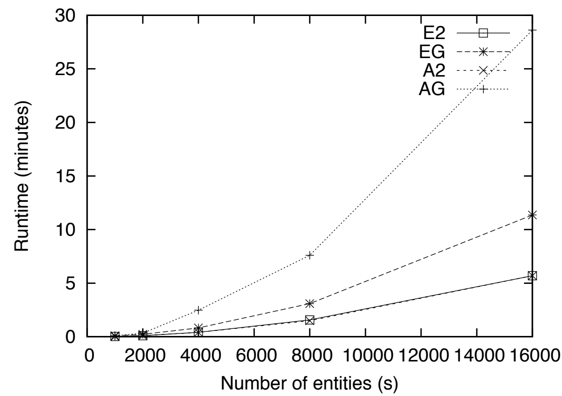


Figure 7.10: Scalability of disinformation generation

Scalability

We now test the scalability of our techniques on large numbers of entities. Figure 7.10 shows the runtime increase of the planning algorithms as the number of entities increases from 1,000 to 16,000. The target entity and other parameters in Table 7.2 were set to their default values. We used the SN algorithm with a window size of 20 as the ER algorithm and set the budget B to 200. As a result, the EG and AG algorithms are 2 to 6x slower than the $E2$ and $A2$ algorithms because of the time to generate the general plans. All the runtimes increase in a quadratic fashion against the number of entities. We conclude that the 1-level plans are practical because they are just as effective as the general plans and can also be generated more efficiently.

7.4.2 Real Data Experiments

We evaluate our disinformation techniques on real data to see how disinformation works in two domains where records are not necessarily in a Euclidean space.

Hotel data results

Suppose that a celebrity wants to hold an event in a secret location without letting the public know. She might want to confuse the adversary by creating false information about locations. Using this scenario, we experimented on a hotel database where the hotel records

simulate possible locations for the secret event. The hotel data was provided by Yahoo! Travel where tens of thousands of records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users. Each hotel record contains a name, street address, city, state, zip code, and latitude/longitude coordinates. We experimented on a random subset of 5,000 hotel records located in the United States.

Resolving hotel records involves the comparison of multiple non-Euclidean values. In particular, the match function B_H compares two hotel records and first checks if two records differ by their state+city combinations, zip codes, or latitude+longitude combinations. If there is no difference, then B_H computes the string similarities between the names and street addresses of the two records using the Jaro distance [111], which ranges from 0 to 1 and is higher for closer strings. If the two hotel names have a Jaro distance of at least 0.7 while the two street addresses have a distance of 0.95, B_H returns true. Or if the two records have the exact same phone numbers, B_H also returns true. Otherwise, B_H returns false. We use the HC_S algorithm for resolving records while using a Boolean match function as the distance function. That is, two records have a distance of 0 if they match according to the match function and 1 otherwise. We set the comparison threshold to be 0.5.

Creating the disinformation now involves the generation of non-Euclidean values as well. When inducing a merge between two clusters c_i and c_j with disinformation, we first choose the records $r \in c_i$ and $s \in c_j$ that require the fewest disinformation records to merge r and s together. We then create a series of disinformation records between r and s where the names and street addresses first resemble r and then gradually resemble s. For the attributes other than the name and street address, we simply add the union of the values to all the disinformation records.

When creating the names and street addresses for the disinformation records between r and s, it is hard to directly generate the strings using the Jaro distance on the non-Euclidean space. Instead, we use a mapping function M (see Section 7.3.2) that converts the records into a partial order space where strings are related to each other based on character inserts, deletes, and updates. During the conversion, M does not actually change the strings themselves. In addition, we implicitly use an identity function as the inverted mapping function M^{-1}.

To generate the names and street addresses, we first compute the Levenshtein distance l_n between the names of r and s, which is the minimum number of character inserts, deletes, and updates to convert r's name to s's name. In addition, we set the maximum possible number of edits d_n between consecutive names. We then set d's name as the string that has an edit distance of $min\{l_n, d_n\}$ from r's name and $l_n - min\{l_n, d_n\}$ from s's name. Similarly, we create d's street address using the Levenshtein distance l_a between the street addresses of r and s and the maximum possible number of edits d_a between consecutive street addresses. For example, suppose that r = $\{$[name, "Hyatt"], [street address, "111 Main"], [phone, 123]$\}$ and s = $\{$[name, "Hilton"], [street address, "222 Main"], [phone, 456]$\}$. The Levenshtein distances for the names and street addresses are l_n = 4 and l_a = 2, respectively. If we set d_n = 2 and d_a = 1, we can create the disinformation record d = $\{\{$[name, "Hyaton"], [street address, "122 Main"], [phone, 123], [phone, 456]$\}$. In general, we need to generate $max\{\lceil \frac{l_n}{d_n} \rceil - 1, \lceil \frac{l_a}{d_a} \rceil - 1, 0\}$ disinformation records. In our experiments, we set d_n = 37 and d_a = 1.

We set the target entity e_h for the hotel data to be the one with the largest number of duplicates, which is a worst-case scenario where increasing the confusion of e_h is difficult. The maximum number of duplicates per entity turns out to be 3 because the hotel data was collected from only a few data sources that did not contain duplicates within themselves.

In Figure 7.11, we evaluate the four disinformation planning algorithms on the hotel records. Since the target cluster only had a size of 3, the confusion of the target entity was sensitive to even a few records merging with the target cluster. For example, using 10 disinformation records, the confusion of the target entity increased to 0.4. The four planning algorithms produce identical confusion results as the budget increases because the cluster sizes were very uniform, so there was little incentive to use multi-level plans so that "far away" clusters would merge with the target cluster. The results show that, even if we generate disinformation on a partial order space, we were still able to significantly increase the confusion for an adversary ER algorithm that uses the Jaro distance for comparing the names and street addresses of hotels.

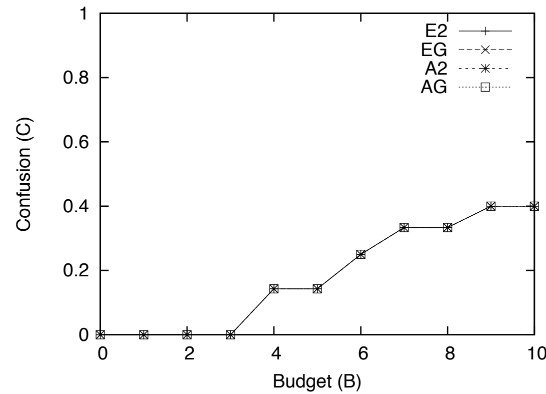


Figure 7.11: Hotel data confusion results

Shopping data results

We also experimented on a comparison shopping database that simulates our Cakon scenario in the beginning of this chapter where there are various rumors of items on the Web, and the agent wants to hide the information about a specific product by introducing disinformation. The shopping data was provided by Yahoo! Shopping where millions of records arrive on a regular basis from different online stores and must be resolved before they are used to answer customer queries. Each record contains attributes including the title, price, and category of an item. We experimented on a random subset of 5,000 shopping records that had the string "iPod" in their titles.

The match function B_S compares two shopping records and checks if the two records have similar titles, prices, and categories. When comparing the titles, B_S checks if the Jaro distance is at least 0.85. When comparing the prices, B_S checks if the smaller price is within 33% of the larger price. For the categories, B_S performs an equality check. If one of the three values are not similar enough, then B_S returns false. Again, we use the HC_S algorithm for resolving records while using a Boolean match function as the distance function.

When creating the first disinformation record d between two shopping records r and s, we first make sure r's price $r.p$ is higher than or equal to s's price $s.p$. If not, we swap r and s. We then create a new title as we do for hotel names based on the Levenshtein distance l_t between the titles of r and s and the maximum possible number of edits d_t

between consecutive titles. We also create a new price that is $X\%$ smaller than $r.p$ unless $s.p$ is already within $X\%$ of $r.p$. Finally, we union the categories of r and s and add them to d. For example, suppose that r = $\{$[title, "iPod Pink"], [price, 100], [category, electronics]$\}$ and s = $\{$[title, "iPod Purple"], [price, 20], [category, mp3]$\}$. If d_t = 2 and X = 50, then we can create the disinformation records d_1 = $\{$[title, "iPod Pinkle"], [price, 50], [category, electronics], [category, mp3]$\}$ and d_2 = $\{$[title, "iPod Pirple"], [price, 25], [category, electronics], [category, mp3]$\}$. In general, we need to generate $max\{\lceil \frac{l_t}{d_t} \rceil - 1, \lceil log_X \frac{r.p}{s.p} \rceil - 1, 0\}$ disinformation records. For our experiments, we set d_t = 1 and X = 50.

For the shopping data, we unfortunately did not have a gold standard, so we could not select the target entity based on its number of true duplicates. Instead, we simply assumed that the initial ER result of the shopping data (without any disinformation) was the gold standard. That is, each cluster in the ER result is assumed to contain the exact set of records that refer to a certain entity. We set the target entity e_s for the shopping data to be the one represented by the 10th largest cluster in the ER result, which had a size of 20.

The titles of the shopping records are on average shorter than the names and street addresses of the hotel data, so it was more difficult to create disinformation records that guaranteed the merge of two clusters. In fact, there were cases where two disinformation records could not merge even if they had titles that differed by only one character edit because the titles still did not have a Jaro distance that exceeded the comparison threshold. As a result, the disinformation records occasionally failed to merge clusters, which is illustrated by the decrease of confusion in Figure 7.12 when the budget increases from 10 to 11. However, the confusion of the target entity eventually increases to high values for larger budgets as shown in the figure. All the four planning algorithms show near-identical performances.

In conclusion, we have shown that our disinformation techniques are effective for two real-world applications where the comparisons of records is sophisticated and involves multiple types of non-Euclidean data. In addition, the 1-level plans perform just as well as the general plans regardless of the application.

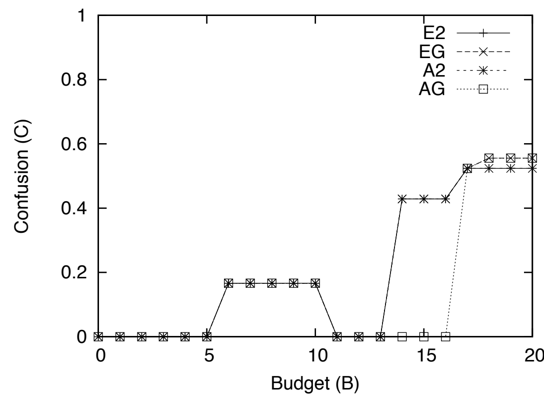


Figure 7.12: Shopping data confusion results

7.5 Related Work

Most ER work in the literature focuses on improving the ER quality or scalability. In contrast, our approach is to dilute the information of ER results by adding disinformation records. Our techniques can be useful when sensitive information has leaked to the public and cannot be deleted.

The problem of managing sensitive information in the public has been addressed in several works. The P4P framework [2] seeks to contain illegitimate use of personal information that has already been released to an adversary. For different types of information, general-purpose mechanisms are proposed to retain control of the data. Measures based on ER [105] have been proposed to quantify the amount of sensitive information that has been released to the public. Reference [63] defines the leakage of information in a general data mining context and provides detection and prevention techniques for leakage. In comparison, our work models the adversary as an ER operator and maximizes the confusion of the target entity.

A recent line of work uses disinformation for managing sensitive information in the public. Reference [80] uses disinformation while distributing data to detect if any information has leaked and to tell who was the culprit. A startup called Reputation.com [85] uses disinformation techniques for managing the reputation of individuals on the Web. For instance, Reputation.com suppresses negative information of individuals in search engine results by creating new web pages or by multiplying links to existing ones. TrackMeNot

[98] is a browser extension that helps protect web searchers from surveillance and data-profiling by search engines using noise and obfuscation. In comparison, our work uses disinformation against an ER algorithm to increase the confusion of the target entity.

Clustering techniques that are robust against noise have been studied extensively in the past [11, 114]. Most of these work proposes clustering algorithms that find the right clusters in the presence of unnecessary noise. In contrast, we take an opposite approach where our goal is to intentionally confuse the ER algorithm for the target entity as much as possible. The disinformation records we generate can thus be viewed as an extreme case of noise where the ER algorithm is forced to produce incorrect results.

7.6 Conclusion

Disinformation is an effective strategy for an agent to prevent an adversary from piecing together sensitive information. We have formalized the disinformation problem by modeling the adversary as an Entity Resolution process and proposed efficient algorithms for generating disinformation that induces the target cluster to merge with other clusters. Our experiments on synthetic data show that the optimal disinformation can significantly increase the confusion of the target entity, especially if the ER algorithm satisfies monotonicity. We have shown that the optimal disinformation generated from correct ER results can be applied when the adversary uses a different (but correct) ER algorithm. Our disinformation techniques perform reasonably well even with partial information on the ER results. We have compared the scalability of our disinformation planning algorithms and have shown that 1-level plans can be generated quickly while having confusion results comparable to those of general plans. Finally, we have demonstrated with real data that our disinformation techniques are effective even when the records are not in a Euclidean space and the match functions are complex.

Chapter 8

Conclusion

8.1 Summary

With data analytics, data is integrated from different sources and analyzed in order to discover useful information. With the unprecedented explosion of data around us, there are new opportunities for discovering useful information on a large scale. At the same time, the sheer amount of data poses non-trivial challenges for combining and analyzing the data.

In this thesis, we studied two closely-related problems within analytics: data integration and data privacy. For both problems, we used a fundamental operation called entity resolution (ER), which identifies and merges records that refer to the same real-world entity. In data integration, ER plays a key role in avoiding duplicates when combining data as well as in consolidating data on the same entity. The flip side of data integration is the danger of one's personal information being exposed to the public. Hence, in data privacy, ER becomes an operation that the adversary uses to discover more information.

We first proposed scalable ER techniques that can be used on large datasets. Our techniques are general because they consider either the entire ER process or the functions for matching and merging records as a black-box operation and can thus be used in a wide range of ER applications. In Chapter 2, we proposed a pay-as-you-go approach for ER where given a limit in resources (e.g., work, runtime) we attempt to make the maximum progress possible. We introduced the novel concept of hints, which can guide an ER algorithm to focus on resolving the more likely matching records first. Our techniques are

effective when there are either too many records to resolve within a reasonable amount of time or when there is a time limit (e.g., in real-time systems). We proposed three types of hints as well as various methods for ER algorithms to use these hints. Our experimental results evaluated the overhead of constructing hints as well as the runtime benefits for using hints.

We also considered several new functionalities for ER that have not been studied in the past. ER may not be a one-time process, but is constantly improved as the data, schema, and application are better understood. Hence in Chapter 3, we addressed the problem of keeping the ER result up-to-date when the ER logic evolves frequently. A naïve approach that reruns ER from scratch may not be tolerable for resolving large datasets. We showed when and how we could instead exploit previous "materialized" ER results to save redundant work with evolved logic.

Next, we addressed the problem of running ER on multiple types of data. Often, records of different types (e.g. authors, publications, institutions, venues) are involved in resolution, and resolving one type of records can impact the resolution of other types of records. Thus in Chapter 4, we proposed a flexible, modular resolution framework where existing ER algorithms developed for a given record type can be plugged in and used in concert with other ER algorithms for resolving multiple types of records. Our approach also made it possible to run ER on subsets of similar records at a time, important when the full data is too large to resolve together. We studied the scheduling and coordination of the individual ER algorithms in order to resolve the full data set. We evaluated our joint ER techniques on synthetic and real data and showed the scalability of our approach.

Finally, we studied the problem of ER with negative rules. In practice, ER results may contain inconsistencies, either due to mistakes by the match and merge functions or changes in the application semantics. Hence in Chapter 5, we formalized the problem of Entity Resolution with inconsistencies (ER-N). The unary and binary negative rules we introduced capture "sanity checks" written by domain specialists who are different from the ones writing the match and merge functions used for resolution. We provided two algorithms that return an ER-N solution based on guidance from a "solver" domain expert. The GNR algorithm is a generic way to solve ER-N while the ENR algorithm makes the GNR algorithm efficient by exploiting additional properties for the match, merge, and negative

rules.

Our ER work in data integration inspired us to study the problem of data privacy as well. As more sensitive data gets exposed to a variety of merchants, health care providers, employers, social sites and so on, there is a higher chance that an adversary can use ER to piece together information, leading to even more loss of privacy.

We first studied in Chapter 6 the problem of quantifying information leakage. Our information leakage measure reflects four important factors of privacy: the correctness and completeness of the leaked data, the adversary's confidence on the data, and the adversary's data analysis. We compared our information leakage model with existing privacy models in the literature and proposed efficient algorithms for computing the exact and approximate values of information leakage. We demonstrated that our information leakage measure captures the main factors of privacy and can be computed on large data.

We also studied the problem of managing information leakage. In Chapter 7, we proposed disinformation techniques for lowering information leakage. We formalized the disinformation problem by modeling the adversary as an ER process and proposed efficient algorithms for generating disinformation that induces the target cluster to merge with other clusters. Our experiments show that our disinformation techniques can significantly increase the confusion of a target entity. We showed that disinformation generated from correct ER results can be universally applied to other (correct) ER algorithms as well. Our techniques also work in the presence of partial information and can scale to large datasets. In general, our disinformation techniques can be used as a framework for evaluating ER robustness.

8.2 Future Work

In this thesis, we have made several contributions to data integration and data privacy using entity resolution. Our problems are by no means solved, and there are many interesting avenues for future work.

Finding the Best Hint In Chapter 2, we studied three types of hints and showed how ER algorithms can benefit from one of these hints. However, we had to manually figure out

which type of hint was good for which ER algorithm. It will be useful to perform a more formal analysis of different types of hints and provide general guidance for constructing and updating the best hint for any given ER algorithm.

Incremental ER on Data In Chapter 3, we focused on incrementally updating an ER result when the match rule changes frequently. Another important problem is updating an ER result when new records are added to the dataset. For example, search engines perform ER on their data and are also constantly crawling new information from the Web. As a result, the new records need to be efficiently resolved against the existing records. We have already defined the General Incremental property (see Definition 3.1.6) that characterizes ER algorithms that support incremental resolution. An interesting future step is to extend our evolving rules framework so that ER results can be efficiently updated both for new rules and new records.

Information Leakage on Real Applications In Chapter 6, we defined an information leakage measure that was evaluated on synthetic data. The next step is to evaluate the measure on real-world ER applications. Compared to other measures that assume privacy to be an all-or-nothing concept, our leakage measure can capture fine-grained notions of privacy. For example, our measure can quantify the additional leakage of a person's information in the presence of more sophisticated ER algorithms.

Robustness of ER Algorithms In Chapter 7, we presented preliminary results on evaluating the robustness of two ER algorithms against disinformation. We also demonstrated how the monotonicity property (Definition 7.1.4) prevents a cluster from splitting unnecessarily. An interesting future step is to survey all the ER algorithms in the literature and see which ER algorithms are more likely to make mistakes when we create more noise. We can then identify the key properties (like monotonicity) of robust ER algorithms.

Specializing on Data Types In Chapter 1.1, we assumed our datasets contain records of any format. As a result, our general ER techniques may not perform as well as ER algorithms specialized for certain data types. For example, running ER on graph data can be

optimized further if we exploit the structure and statistics of the graph. It will be interesting to see how our general ER techniques can scale better by tailoring them to different formats of data.

Combining Scalability with Privacy Chapters 2–5 covered scalable and general techniques for ER while Chapters 6 and 7 studied privacy against ER. A natural extension is to combine the two goals and provide scalable ER techniques that also minimize unnecessary information leakage. For example, we may be performing ER on data from multiple companies that are unwilling to share all of their confidential data with others. The more strict the companies are on sharing information, the harder it is to scale ER. Hence, an interesting question is to find a "middle point" between scalability and privacy.

Beyond Machines Throughout this thesis, we have assumed that ER is performed by machines. With the advent of platforms for human computation [4, 26], it is now possible to augment challenging computation with the wisdom of the crowd. Compared to machines, humans are better at solving difficult AI problems such as labeling images, digitizing books, and transcribing audios. Within ER, humans can help in various ER stages. For example, if we are resolving records using a given ER algorithm, then the humans can help compare the records. Or if we are verifying an ER result, then the humans can check if the records were merged correctly. On the other hand, humans may be slow, expensive, and error prone. Accommodating this human behavior is the key challenge in using humans for ER.

Bibliography

[1] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003.

[2] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Nina Mishra, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, Jennifer Widom, and Ying Xu. Vision paper: Enabling privacy for the paranoids. In *VLDB*, pages 708–719, 2004.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[4] Amazon mechanical turk. https://www.mturk.com.

[5] Rohit Ananthakrishna, Surajit Chaudhuri, and Venkatesh Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. of VLDB*, pages 586–597, 2002.

[6] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[7] Arvind Arasu, Christopher Ré, and Dan Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, pages 952–963, 2009.

[8] Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *Proc. of ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[9] Omar Benjelloun, Hector Garcia-Molina, Hideki Kawai, Tait E. Larson, David Menestrina, and Suthipong Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *ICDCS*, 2007.

[10] Omar Benjelloun, Hector Garcia-Molina, David Menestrina, Qi Su, Steven E. Whang, and Jennifer Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.

[11] Michael Berthold and David J. Hand, editors. *Intelligent Data Analysis: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1999.

[12] Indrajit Bhattacharya and Lise Getoor. Iterative record linkage for cleaning and integration. In *DMKD*, 2004.

[13] Indrajit Bhattacharya and Lise Getoor. Relational clustering for multi-type entity resolution. In *MRDM '05: Proceedings of the 4th international workshop on Multi-relational mining*, pages 3–12, New York, NY, USA, 2005. ACM Press.

[14] Indrajit Bhattacharya and Lise Getoor. A latent dirichlet model for unsupervised entity resolution. In *SDM*, 2006.

[15] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *TKDD*, 1(1), 2007.

[16] Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD*, pages 39–48, 2003.

[17] Philip Bohannon, Michael Flaster, Wenfei Fan, and Rajeev Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD Conference*, pages 143–154, 2005.

[18] Peter Brucker. *Scheduling algorithms (4. ed.)*. Springer, 2004.

[19] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. In *STOC*, pages 626–635, 1997.

[20] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

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

[22] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *ICDE*, pages 190–200, 1995.

[23] Surajit Chaudhuri, Anish Das Sarma, Venkatesh Ganti, and Raghav Kaushik. Leveraging aggregate constraints for deduplication. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 437–448, New York, NY, USA, 2007. ACM Press.

[24] Jan Chomicki and Jerzy Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. In *Inconsistency Tolerance*, pages 119–150, 2005.

[25] William W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Trans. Inf. Syst.*, 18(3):288–321, 2000.

[26] Crowdflower. http://crowdflower.com.

[27] Aron Culotta and Andrew Mccallum. A conditional model of deduplication for multi-type relational data. Technical report, Univ. of Massachusetts, 2005.

[28] Aron Culotta and Andrew McCallum. Joint deduplication of multiple record types in relational data. In *CIKM*, pages 257–258, 2005.

[29] AnHai Doan, Ying Lu, Yoonkyong Lee, and Jiawei Han. Object matching for information integration: A profiler-based approach. In *IIWeb*, pages 53–58, 2003.

[30] AnHai Doan, Ying Lu, Yoonkyong Lee, and Jiawei Han. Profile-based object matching for information integration. *IEEE Intelligent Systems*, 18(5):54–59, 2003.

[31] Xin Dong, Alon Y. Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD Conference*, pages 85–96, 2005.

[32] Cynthia Dwork. Differential privacy. In *ICALP (2)*, pages 1–12, 2006.

[33] Cynthia Dwork. Differential privacy: A survey of results. In *TAMC*, pages 1–19, 2008.

[34] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[35] Kapali P. Eswaran and Donald D. Chamberlin. Functional specifications of subsystem for database integrity. In *VLDB*, pages 48–68, 1975.

[36] Ivan P. Fellegi and David Holt. A systematic approach to automatic edit and imputation. *Journal of the American Statistical Association*, 71(353):17–35, 1976.

[37] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[38] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census data repair: a challenging application of disjunctive logic programming. In *LPAR '01: Proceedings of the Artificial Intelligence on Logic for Programming*, pages 561–578, London, UK, 2001. Springer-Verlag.

[39] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[40] Michael R. Garey, David S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1:117–129, 1976.

[41] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Palo Alto, CA, 1988.

[42] Matthew L. Ginsberg. *Readings in Nonmonotonic Reasoning*. Morgan Kaufmann, Los Altos, CA, 1987.

[43] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[44] Jonathan Goldstein and Per-Åke Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342, 2001.

[45] J. C. Gower and G. J. S. Ross. Minimum spanning trees and single linkage cluster analysis. *Applied Statistics*, 18(1):54–64, 1969.

[46] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[47] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[48] Lifang Gu, Rohan Baxter, Deanne Vickers, and Chris Rainsford. Record linkage: Current practice and future directions. Technical Report 03/83, CSIRO Mathematical and Information Sciences, 2003.

[49] Laura M. Haas, Martin Hentschel, Donald Kossmann, and Renée J. Miller. Schema and data: A holistic approach to mapping, resolution and fusion in information integration. In *ER*, pages 27–40, 2009.

[50] Michael Hammer and Dennis McLeod. Semantic integrity in a relational data base system. In *VLDB*, pages 25–47, 1975.

[51] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *SIGMOD Conference*, pages 205–216, 1996.

[52] Oktie Hassanzadeh, Fei Chiang, Renée J. Miller, and Hyun Chul Lee. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009.

[53] Oktie Hassanzadeh and Renée J. Miller. Creating probabilistic databases from duplicated data. *VLDB J.*, 18(5):1141–1166, 2009.

[54] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In *SIGMOD Conference*, pages 127–138, 1995.

[55] Mauricio A. Hernández and Salvatore J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2(1):9–37, 1998.

[56] ICorrect. http://http://www.icorrect.com/.

[57] Piotr Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001.

[58] Anil K. Jain. Data clustering: 50 years beyond k-means. In *ECML/PKDD (1)*, pages 3–4, 2008.

[59] Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[60] Matthew A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[61] Shawn R. Jeffery, Michael J. Franklin, and Alon Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD Conference*, pages 847–860, 2008.

[62] Liang Jin, Chen Li, and Sharad Mehrotra. Efficient record linkage in large data sets. In *Proc. of Intl. Conf. on Database Systems for Advanced Applications*, pages 137–, 2003.

[63] Shachar Kaufman, Saharon Rosset, and Claudia Perlich. Leakage in data mining: formulation, detection, and avoidance. In *KDD*, pages 556–563, 2011.

[64] Allen Kent, Madeline M. Berry, Fred U. Luehrs, Jr., and J. W. Perry. Machine literature searching VIII. Operational criteria for designing information retrieval systems. *American Documentation*, 6(2):93–101, 1955.

[65] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1):484–493, 2010.

[66] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE*, pages 106–115, 2007.

[67] Libsvm. http://www.csie.ntu.edu.tw/ cjlin/libsvm/.

[68] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE*, page 24, 2006.

[69] Jayant Madhavan, Shirley Cohen, Xin Luna Dong, Alon Y. Halevy, Shawn R. Jeffery, David Ko, and Cong Yu. Web-scale data integration: You can afford to pay as you go. In *CIDR*, pages 342–350, 2007.

[70] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[71] Andrew K. McCallum, Kamal Nigam, and Lyle Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of KDD*, pages 169–178, Boston, MA, 2000.

[72] David Menestrina, Omar Benjelloun, and Hector Garcia-Molina. Generic Entity Resolution with Data Confidences. In *First Int'l VLDB Workshop on Clean Databases*, Seoul, Korea, 2006.

[73] David Menestrina, Steven E. Whang, and Hector Garcia-Molina. Evaluating entity resolution results. *PVLDB*, 3(1):208–219, 2010.

[74] Alvaro E. Monge and Charles Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *DMKD*, pages 23–29, 1997.

[75] Amihai Motro and Philipp Anokhin. Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. *Information Fusion*, 7(2):176–196, 2006.

[76] New York Times. How to muddy your tracks on the internet, 2012.

[77] Howard B. Newcombe and James M. Kennedy. Record linkage: making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5(11):563–566, 1962.

[78] Howard B. Newcombe, James M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.

[79] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, San Francisco, CA, 1998.

[80] Panagiotis Papadimitriou and Hector Garcia-Molina. Data leakage detection. *IEEE Trans. Knowl. Data Eng.*, 23(1):51–63, 2011.

[81] Parag and Pedro Domingos. Multi-relational record linkage. In *In Proceedings of the KDD-2004 Workshop on Multi-Relational Data Mining*, pages 31–48, 2004.

[82] Hoifung Poon and Pedro Domingos. Joint inference in information extraction. In *AAAI*, pages 913–918, 2007.

[83] Vibhor Rastogi, Nilesh N. Dalvi, and Minos N. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011.

[84] Vibhor Rastogi, Sungho Hong, and Dan Suciu. The boundary between privacy and utility in data publishing. In *VLDB*, pages 531–542, 2007.

[85] Reputation.com. http://www.reputation.com.

[86] Fazlollah M. Reza. *An Introduction to Information Theory*. Dover Publications, September 1994.

[87] Sunita Sarawagi and Anuradha Bhamidipaty. Interactive deduplication using active learning. In *KDD*, pages 269–278, 2002.

[88] Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. Extensible and similarity-based grouping for data integration. In *ICDE*, page 277, 2002.

[89] Warren Shen, Xin Li, and AnHai Doan. Constraint-based entity matching. In *AAAI*, pages 862–867, 2005.

[90] Parag Singla and Pedro Domingos. Entity resolution with markov logic. In *ICDM*, pages 572–582, 2006.

[91] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[92] Spock. http://spock.com.

[93] Latanya Sweeney. Achieving k-anonymity privacy protection using generalization and suppression. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):571–588, 2002.

[94] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[95] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[96] Robert E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Inf.*, 6:171–185, 1976.

[97] Sheila Tejada, Craig A. Knoblock, and Steven Minton. Learning object identification rules for information integration. *Information Systems Journal*, 26(8):635–656, 2001.

[98] TrackMeNot. http://cs.nyu.edu/trackmenot.

[99] Cornelis Joost van Rijsbergen. *Information Retrieval*. Butterworths, London, 2nd edition, 1979.

[100] Wall Street Journal. Insurers test data profiles to identify risky clients, 2011.

[101] Melanie Weis and Felix Naumann. Detecting duplicates in complex xml data. In *ICDE*, page 109, 2006.

[102] Steven E. Whang, Omar Benjelloun, and Hector Garcia-Molina. Generic entity resolution with negative rules. *VLDB J.*, 18(6):1261–1277, 2009.

[103] Steven E. Whang and Hector Garcia-Molina. Entity resolution with evolving rules. *PVLDB*, 3(1):1326–1337, 2010.

[104] Steven E. Whang and Hector Garcia-Molina. Disinformation techniques for entity resolution. Technical report, Stanford University, 2011.

[105] Steven E. Whang and Hector Garcia-Molina. Managing information leakage. In *CIDR*, pages 79–84, 2011.

[106] Steven E. Whang and Hector Garcia-Molina. A model for quantifying information leakage. Technical report, Stanford University, 2011.

[107] Steven E. Whang and Hector Garcia-Molina. Joint entity resolution. In *ICDE*, 2012.

[108] Steven E. Whang, David Marmaros, and Hector Garcia-Molina. Pay-as-you-go entity resolution. *IEEE Trans. Knowl. Data Eng.*, 2012.

[109] Steven E. Whang, David Menestrina, Georgia Koutrika, Martin Theobald, and Hector Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD Conference*, pages 219–232, 2009.

[110] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

[111] William Winkler. Overview of record linkage and current research directions. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 2006.

[112] William E. Winkler. State of statistical data editing and current research problems. In *UN/ECE Work Session on Statistical Data Editing, Working Paper n.29*, pages 2–4, 1999.

[113] Xiaokui Xiao and Yufei Tao. Personalized privacy preservation. In *SIGMOD Conference*, pages 229–240, 2006.

[114] Rui Xu and Donald Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[115] William Yancey. Bigmatch: A program for extracting probable matches from a large file for record linkage. Technical report, US Bureau of the Census, 2002.

[116] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer, 2005.