MATCHING AND UNIFYING RECORDS IN A DISTRIBUTED SYSTEM

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

David Menestrina
February 2010

© Copyright by David Menestrina 2010
All Rights Reserved

ban comic sans

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.

Jeffrey D. Ullman

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

Entity resolution (ER) is the process of identifying records in a database that refer to the same real world entity. We consider various aspects of the entity resolution problem:

- We present multiple methods for efficiently distributing the ER workload across multiple processors and provide guidelines on when to use each method.

- We explore the use of blocking on multiple criteria at once to reduce the runtime of processing, while minimizing a loss of accuracy. We propose an efficient technique for performing multiple blocking when the full record set is too large to fit in memory at once and must therefore be stored on disk.

- We present new locality sensitive hashing schemes based on minhash for new data types, including maps, sets with weighted values, and composite data types. These hashing schemes can be used in conjunction with blocking techniques for entity resolution.

- We describe ER on data with confidences and provide efficient methods of handling the confidences, introducing the concepts of confidence thresholds and domination.

- We explore how the results of ER algorithms are evaluated against a gold standard result, and propose a new, configurable distance measure that

provides an intuitive method of adapting the measure to any given ER application.

Acknowledgements

First and foremost, I would like to thank Hector Garcia-Molina for his incredible help in his role as my advisor. Hector has an overwhelmingly positive attitude, and he has deftly guided me in the right direction without ever responding negatively towards my ideas. Many times, I have gone into his office with results that I found disappointing, only to be met with excitement and optimism from Hector. Without Hector's support, I just would not have made it through the program. I am very happy to have him as both an advisor and a friend.

Jeff Ullman and Jennifer Widom served as the two other members of my reading committee, and I appreciate their hard work in giving me helpful comments. I would also like to thank Andreas Paepcke for providing a friendly face during my defense, and Mark Musen for serving as the chair of my defense committee.

Next, I would like to acknowledge the feedback I received from Rajeev Motwani on the work that would become Chapter 4. His passing is certainly a great loss to Stanford.

The staff of the Computer Science department have all been extremely helpful. I would like to specifically recognize Marianne Siroker, Jam Kiattinant, and Kathi DiTommaso for their help with all things administrative.

In the Ph.D. program, collaborations among colleagues are encouraged, and I am pleased to have had the opportunity to collaborate with many great minds. Of these, I'd first like to thank Steven Whang, whose dedication and hard work inspired me to become a better researcher. Many of my collaborations

have resulted in publications, so I'd like to thank the rest of my coauthors: Omar Benjelloun, Heng Gong, Hideki Kawai, Georgia Koutrika, Tait Eliott Larson, Qi Su, Sutthipong Thavisomboon, and Martin Theobald. Finally, I would like to thank the rest of the members of the Stanford InfoLab, including Parag Agrawal, Ioannis Antonellis, Anish Das Sarma, Zoltán Gyöngyi, Paul Heymann, Robert Ikeda, Andy Kacsmar, Jawed Karim, Bobji Mungamuru, Raghotham Murthy, Panagiotis Papadimitriou, Aditya Parameswaran, Eldar Sadikov, Petros Venetis, Gary Wesley, and Gio Wiederhold.

While the toughest part of the Ph.D. program lies in developing the work that forms the thesis, the second-largest hurdle is passing the qualifying examinations. I was lucky enough to be part of a study group full of smart people. Studying for the quals would have been more difficult (and far more boring) without Tassos Argyros, Brian Carlstrom, Martin Casado, Michael Kassoff, Austen McDonald, and Mayur Naik. Before the quals, the most difficult part of getting a graduate degree is actually getting into the school of your choice. I would like to thank the three people who wrote letters of recommendation for my application to Stanford: Kai Li, Sylvia Perez, and Larry Peterson. I would also like to acknowledge Evan Greenberg for encouraging me to apply to Stanford and offering good advice throughout my career as a student.

Next, I'd like to thank my family for their support over these years: my father, Leo Menestrina; my mother, Martha Menestrina; and my sister, Lisa Menestrina. But that's not all. I have a large family full of friendly, supportive people, and I'll do my best to mention them all here: Mary Mahoney, George Sharpe, Alison Sharpe, Orr Sharpe, Eran Sharpe, Ronni Sharpe, Kai Sharpe, Arlyn Sharpe, Kevin Weiler, Jay Weiler, Zak Weiler, Bob Sharpe, Ellie Sharpe, Jeff Sharpe, Richie Sharpe, Bob Kreiser, Jeanette Kreiser, Deborah Kreiser-Francis, Mike Francis, Evelyn Francis, Julia Francis, Josh Kreiser, Jess Kreiser, Shayna Kreiser, Larry Sharpe, Blanche Sharpe, Allan Conan, Renee Conan-Tadmor, Ilan Tadmor, Mayaan Schoeman, Michael Schoeman, Tain Schoeman, Liam Schoeman, Mark Rehr, Paula Rehr, Scott Rehr, Beth Rehr, Gabriel Rehr, Elliot Pierce, and Vivian Pierce.

While at Stanford, I took up swing dancing, a new hobby that led to the formation of many wonderful friendships. Although dance was a distraction from school, it was a much-needed distraction from work-induced stress. Therefore, I am very grateful to have met so many great people in the swing dancing community. They are far too numerous to list them all, but to name a few, I'd like to thank Rachael Eisner, Nicole Farar, Ashleigh Golden, Carla Heiney, Nina Joshi, Jean Ma, Frankie Manning, Bromley Palamountain, Sarah Price, Joe Rabinoff, Yon Richner, Grace Shen, David Shilane, Jacqueline Tay, Kim Tran, Ellen Vuong, Tai Woltmann, and Kylie Woodard.

Finally, I would like to express my appreciation to the rest of my friends for their support, including Mike Bates, Matt Childerston, Alex Chou, Rusty Coleman, Christine Fessler, Evan Greenberg, Dao Huynh, John Truscott Reese, Haakon Ringberg, Tony Santos, Florian Simatos, Terry Tang [80], James Van Zoeren, Curt Wright, Dave and Rachel Weldon, Bei Wu, Jen and Alvin Yan, Hanson Zhou, and Joanna Zurada.

Contents

Abstract v

Acknowledgements vii

1 Introduction 1

1.1 Entity Resolution 1

1.2 Model 3

1.3 Pairwise, Iterative Approach 4

1.3.1 Domination 6

1.3.2 Defining Pairwise ER 6

1.4 Correctness and Comprehensiveness 7

1.5 Related Work 8

1.5.1 Clustering-Based ER 9

1.6 Problem Statement 10

2 Distributed Swoosh 13

2.1 Introduction 13

2.1.1 Overview of Our Approach 14

2.2 Preliminaries 17

2.3 Distributed ER 18

2.3.1 Distribution Primitives 19

2.3.2 The D-Swoosh Algorithm 20

2.4 Choosing scope and resp 24

2.4.1 Strategies Without Domain Knowledge 24

2.4.2 Strategies With Domain Knowledge 30

2.4.3 "Tuning" scope and resp 34
2.5 Experiments 35
2.5.1 Experimental Setting 36
2.5.2 No Domain Knowledge 37
2.5.3 With Domain Knowledge 41
2.5.4 Scalability 46
2.6 Related Work 48
2.7 Conclusion 49

3 Entity Resolution with Iterative Blocking 51
3.1 Introduction 51
3.2 ER Model 54
3.3 Blocking Model 55
3.4 Simple Blocking 56
3.5 Iterative Blocking 57
3.5.1 Lego Algorithm 60
3.6 On-Disk Iterative Blocking 61
3.6.1 Segments 61
3.6.2 Merge Log 63
3.6.3 The Duplo Algorithm 64
3.6.4 Segment Queue Policy 68
3.7 Experimental Evaluation 68
3.7.1 Accuracy 70
3.7.2 Lego Runtime 73
3.7.3 Duplo Performance 74
3.7.4 Other Datasets 78
3.7.5 Other CER Algorithms 79
3.8 Related Work 81
3.9 Conclusion 82

4 LSH and Minhash 85
4.1 Introduction 85

4.2 Preliminaries 86
4.2.1 LSH 86
4.2.2 Minhash 87
4.3 Map Minhash 90
4.4 Sets with Weighted Values 93
4.4.1 Integer Weights 96
4.4.2 Real Weights 99
4.4.3 Limitation: Weighted Sets 108
4.5 Composed Data Types 109
4.5.1 Arithmetic Mean 111
4.5.2 Geometric Mean 112
4.6 Experimental results 116
4.7 Related Work 121
4.8 Conclusion 123

5 Entity Resolution with Confidences 125

5.1 Introduction 125
5.2 Model 127
5.3 Generic Entity Resolution with Confidences 128
5.4 Koosh 132
5.5 Domination 136
5.5.1 Algorithm Koosh-ND 138
5.6 The Packages Algorithm 139
5.6.1 Phase 1 141
5.6.2 Phase 2 144
5.6.3 Example 146
5.6.4 Packages-ND 146
5.7 Thresholds 147
5.7.1 Algorithms Koosh-T and Koosh-TND 148
5.7.2 Packages-T and Packages-TND 149
5.8 Experiments 150

5.9 Related Work 156
5.10 Conclusion 157

6 Evaluating Entity Resolution Results **159**
6.1 Introduction 159
6.2 Evaluating ER Results 162
6.3 Existing Measures 164
6.3.1 Pairwise Comparison 165
6.3.2 Cluster-level Comparison 166
6.3.3 Basic Merge Distance 167
6.3.4 Variation of Information 169
6.4 Generalized Merge Distance 169
6.4.1 Operation Order Independence 177
6.4.2 Merge Precision and Recall 181
6.4.3 Relationship to Other Measures 182
6.5 Computing Measures 189
6.6 Computing Merge Distance 191
6.6.1 General Algorithm 191
6.6.2 Slice Algorithm 191
6.7 Experiments 194
6.8 Conclusion 196

7 Conclusion **199**
7.1 Summary 199
7.2 Future Work 201

Bibliography **203**

List of Tables

6.1 Comparing two ER results 161
6.2 Example path 171

List of Figures

2.1 A sample run of Entity Resolution 14
2.2 One possible scope function 15
2.3 Scope for the majority scheme 27
2.4 Processor arrangement in the grid scheme 28
2.5 Aggregated computation cost for strategies without domain knowledge 38
2.6 Runtime; no domain knowledge 39
2.7 Storage cost; no domain knowledge 41
2.8 Communication cost; no domain knowledge 42
2.9 Communication cost for domain knowledge schemes 43
2.10 Aggregated computation cost for domain knowledge schemes . . 44
2.11 Runtime for domain knowledge schemes 45
2.12 Storage for domain knowledge schemes 46
2.13 Runtime evolution with dataset size (10 processors) 47

3.1 Customer records 52

3.2 After initial distribution . . . 59
3.3 Intermediate result after first iteration . . . 59
3.4 Intermediate result after second iteration . . . 60
3.5 Assigning blocks to segments . . . 62
3.6 Initial segments of Duplo . . . 65
3.7 Processing segments with the Duplo algorithm . . . 65
3.8 Final state of Duplo . . . 67
3.9 Average block size impact on accuracy, 50K records . . . 72
3.10 Number of Blocking Criteria impact on accuracy, 50K records . 73
3.11 Average block size impact on runtime, 50K records . . . 74
3.12 Runtimes for different segment queue strategies, 1M records . 76
3.13 Scalability, 2M records . . . 77
3.14 Runtime needed to achieve given accuracy, 2M records . . . 78
3.15 Runtimes on the hotel dataset, 1.7M records . . . 79
3.16 Accuracy for the ME algorithm, 50K records . . . 80
3.17 Scalability for the ME algorithm, 2M records . . . 81

4.1 A function f . . . 89
4.2 Runtime of Weighted Schemes vs. Weight Values . . . 118
4.3 Measured hash matches vs. Weighted Jaccard similarity . . . 121

5.1 Thresholds vs. Matches . . . 152
5.2 Thresholds vs. Merges . . . 153
5.3 Selectivity vs. Comparisons . . . 154
5.4 Effects of removing dominated records . . . 155

6.1 A precedence graph. . . . 172
6.2 Precedence graph for the transformed path. . . . 173
6.3 A merge with an edge to a split is rewritten into a split with an edge to a merge. . . . 174
6.4 Scalability . . . 195

Chapter 1

Introduction

1.1 Entity Resolution

Entity resolution (ER) is the identification of records in a database that refer to the same real-world entity. The process of entity resolution is required in most information-integration tasks.

Consider, for example, when two large companies merge and wish to combine their customer databases. It is highly likely that the two companies shared many customers. It is highly unlikely, however, that the shared customers are referred to in precisely the same way. Suppose both companies have Oracle Corporation as a customer. Company A may simply call this customer "Oracle", whereas Company B may call it "Oracle Corporation". Or, perhaps Company B did business with many individual branches of Oracle. In that case, Company B may have separate accounts for their dealings with "Oracle, Redwood Shores, CA" and "Oracle Corp, Bethesda, MD". Performing entity resolution on these databases would entail determining what the real world entities are, and which records in each database refer to each real world entity.

The main task of entity resolution is finding matching records. However, there still remains the question of what to do with matching records once they are discovered. For some applications, it may suffice to simply retain

the records in their original form, with the added knowledge of which records refer to the same entity. For the application of merging customer databases, however, we may want to combine all the records for a given entity into a single composite record that represents the entity. As we will discuss later, combining records can have an effect on the process of discovering matches as well.

The entity resolution problem has the deceptive appearance of simplicity. It is clear to humans that in the above example, "Oracle" and "Oracle Corporation" refer to the same company. However, it is unreasonable to expect humans to solve this problem manually on large databases. Computers, on the other hand, have the computational power necessary to handle large databases, but they lack the human intelligence that allows us to link up the Oracle records, yet might cause us to think twice before linking "Apple" and "Apple Records". Further, entity resolution is fundamentally an $O(n^2)$ process, since, in the absence of heuristics, every record must be compared against every other record. So while computers can handle larger databases than humans, they too will have trouble quickly performing entity resolution on very large databases.

Entity resolution is not only difficult. It is also an important problem, and has applications in many fields. Comparison-shopping websites such as Yahoo! Shopping attempt to present the user with the lowest price for any given item. Such a website must necessarily aggregate price information feeds from many different vendors and identify the distinct items for sale. Another example would be a news aggregation website such as Google News. A news aggregation site must present to the user a small subset of the available news items. Presenting two different articles that cover the same story would be a waste of space. Further, knowing which articles cover the same story allows the implementation of a feature for the user to see more articles covering a particular story of interest. Finally, entity resolution has applications in law enforcement. For example, a crime-fighting agency may have a long list of known evildoers, and may want to match it up against other databases. The

applications are indeed widespread. Perhaps because the entity resolution problem has been encountered in many diverse fields, it has come to be known under many different names, including merge-purge, reference reconciliation, and record linkage.

1.2 Model

In entity resolution, our task is to determine:

1. the number of real-world entities in the input data, and
2. which pieces of information in the input data refer to the same real-world entity.

We define the entity resolution problem formally as follows. An ER problem consists of a set I of N input records. The structure of a record is not important. Records may be tuples, XML fragments, or even multimedia data types such as images or videos. For the purposes of demonstration, we will generally represent a record as a collection of fields with values. We do assume, however, that each record refers to one real-world entity.

The result of an ER algorithm is a *partition* of I. A partition of a set S is a set of sets $\{s_1, s_2, s_3, \ldots, s_n\}$ such that:

1. $\bigcup_i s_i = S$,
2. for all pairs s_i, s_j, $s_i \cap s_j = \varnothing$, and
3. for all sets s_i, $s_i \neq \varnothing$.

We refer to each s_i as a cluster. Although clusters are sets, we will denote them using $\langle , \rangle$ to distinguish them from sets that are not to be understood as members of a partition. As an example, the partition $\{\langle a, b, c\rangle, \langle d, e\rangle\}$ identifies two real-world entities, with records a, b, c representing the first, and d, e representing the second.

Some ER algorithms return the resulting partitions directly; others go one step further and merge the records within a cluster into a single representative record. (In addition to the representative record, these algorithms can

also return the identities of the original records that form the cluster, i.e., the lineage.)

Note that entity resolution is naturally a clustering problem, as the input records are partitioned into clusters. In Section 1.5.1, we briefly review clustering algorithms, but for now, observe that not all clustering algorithms make good ER algorithms. For example, some clustering algorithms take as input the desired number of clusters, which is not reasonable for ER, since we generally do not know in advance the number of real-world entities represented in the input data.

1.3 Pairwise, Iterative Approach

Some of the work in this thesis (specifically, the work of Chapters 3 and 6) can work in conjunction with any ER algorithm. However, in Chapters 2 and 5 we focus on particular ER algorithms, and in this section we provide an overview of the *pairwise, iterative* approach they follow. (The pairwise, iterative approach to entity resolution is fully described in [11].)

In the pairwise approach, we consider one pair of records at a time and make a decision whether or not they refer to the same real-world entity. There are many techniques for making this decision, so we abstract the decision into a Boolean black-box *match function*. The match function, $M(r, s)$, returns true if records r and s refer to the same real-world entity, and false otherwise. For shorthand, we will often use the $\approx$ symbol to indicate that two records match (i.e. $r \approx s \Leftrightarrow M(r, s)$).

Suppose that we have a simple match function that declares a match when the names have a “decent” level of similarity and at least one other field is an exact match. Now consider the following records:

r_1 = { name: Garcia-Molina, job: Professor }
r_2 = { name: Hector, phone: 650-555-1212, job: Professor }
r_3 = { name: Hector García Molina, phone: 650-555-1212 }

Records r_1 and r_2 match exactly on their job fields, but the names are not similar at all. So our match function would decide that $r_1 \not\approx r_2$. Comparing further, we find that $r_1 \not\approx r_3$ as well. The names have a decent amount of similarity, but no other field is an exact match. Finally, when we compare r_2 with r_3, we find that they indeed are a match, since the records share some similarity in the name field, and match exactly on phone number.

A simple pairwise approach to entity resolution might therefore conclude that r_2 and r_3 refer to the same person, but r_1 refers to a different person. Note, however, that if we combine the information from r_2 and r_3, we might obtain a record that blends the attributes from the two records, such as:

> r_{23} = { name: Hector García Molina, phone: 650-555-1212, job: Professor }

Now, record $r_{23} \approx r_1$, even though neither of the original records matched with r_1. This example motivates the iterative approach: once we find matching records, we can combine them and look for more matches. We therefore invoke one more abstraction: a *merge function*. The merge function $\mu(r, s)$ returns a single record that combines the information in records r and s. The merge function is defined whenever $r \approx s$. We can then perform pairwise, iterative entity resolution on a collection of records using the match function to find matching records, and the merge function to combine them. As with the match function, we employ a shorthand to represent the merge of two records: $\mu(r, s) = \langle r, s \rangle$. This notation introduces some ambiguity with the use of angle brackets to denote clusters, however it will be clear from context which meaning is intended.

Since the match and merge functions are black-box functions, a completely general entity resolution algorithm would not assume anything about their operation. However, the functions *may* satisfy some useful properties, and if they do, we can exploit these properties to perform entity resolution more efficiently. Benjelloun et al. [11] describe four such properties, called the ICAR properties. We repeat the definitions of these properties here, as the

chapters in this thesis will refer to them later.

- *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 $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.
- *Reflexivity/Idempotence:* $\forall r$, $r \approx r$ and $\langle r, r \rangle = r$.
- *Merge 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 $r_3 \approx r_4$.
- *Merge associativity:* $\forall r_1, r_2, r_3$ such that both $\langle r_1, \langle r_2, r_3 \rangle \rangle$ and $\langle \langle r_1, r_2 \rangle, r_3 \rangle$ exist, $\langle r_1, \langle r_2, r_3 \rangle \rangle = \langle \langle r_1, r_2 \rangle, r_3 \rangle$.

1.3.1 Domination

During entity resolution, we may generate merged records that do not carry additional information, relative to other records. In our example, record r_{23} has all of the information contained in r_2 and r_3. Once r_{23} is found, there is no longer any need for records r_2 and r_3. Further, should record r_{23} later be merged with r_1, then both of these two records may no longer be necessary as well. We call the unnecessary records *dominated*.

Benjelloun et al. [11] fully discuss this concept and note that any partial order on records may be used as a definition of domination. In the case where all ICAR properties hold, however, the authors recommend the use of a specific form of domination called merge domination.

Definition 1.3.1. *Given two records,* r_1 *and* r_2*, we say that* r_1 is merge dominated by r_2*, denoted* $r_1 \leq r_2$ *if* $\langle r_1, r_2 \rangle = r_2$.

When the ICAR properties do not hold, other domination orders may be more appropriate.

1.3.2 Defining Pairwise ER

We are now ready to define the result of entity resolution within the pairwise context.

Definition 1.3.2. *Given a set of input records* R, *an* entity resolution *of* R *is a set* S *of records such that:*

- *Any record in* S *is derived (through merges) from records in* R*;*
- *Any record that can be derived from* R *is either in* S*, or is dominated by a record in* S*;*
- *No two records in* S *match, and no record in* S *is dominated by any other.*

Benjelloun et al. [11] show that when the ICAR properties hold, the entity resolution of R is unique. We can therefore denote the entity resolution of R as ER(R).

1.4 Correctness and Comprehensiveness

Another question related to entity resolution is what the "correct" result is. We define two separate concepts dealing with this issue.

The first is a practical concept we simply call *correctness*. It is ultimately humans who will be the consumers of the result of an entity resolution process. Therefore, it is reasonable to define the correct result in terms of human input. We can define the correct answer to be the result of putting a group of human experts to the task of solving a given entity resolution task. Generally, researchers test their entity resolution algorithms on datasets that come with a human-generated gold standard for a subset of the data. They can then run their algorithms on the same subset and decide how close their results are to the human-generated answer. It is generally assumed that if an algorithm delivers good results when run on a subset of the data, it will perform comparably on the entire data set.

The second concept we define is called *comprehensiveness*. Definition 1.3.2 gives us the result of an exhaustive ER algorithm using the pairwise approach. If we assume that the match and merge functions are decent, then we can consider ER(R) as the gold standard. Comprehensiveness is therefore a measure of how close an algorithm's result is to ER(R).

Both correctness and comprehensiveness give us a gold standard that we can use to evaluate the quality of ER results. There are many techniques for putting a number on the closeness of a result to the gold standard, and a complete discussion of this topic is presented in Chapter 6.

1.5 Related Work

The chapters of this thesis cover different aspects of the central theme of entity resolution. Therefore, each chapter will have its own related work section that covers the related work specific to that chapter. There is, however, a great body of work on entity resolution in general, which we will cover here.

The earliest reference to this problem that we have found is the work of Newcombe et al. [69, 70] on the problem of "record linkage" in gathering statistics from medical records. The problem was formalized by Fellegi and Sunter [37] in 1969. Hernández and Stolfo [46] have described the sorted-neighborhood method for the "merge/purge" problem. Numerous surveys have also been published on the latest techniques in entity resolution [90, 41, 91, 36].

While our work takes a generic approach, abstracting away the pairwise decision process into a black-box match function, numerous works are related to the construction of a good match function. A common component of match functions for text data is edit distance, which was first proposed by Sellers [76]. Edit distance has many variants, and Winkler [94] has described a good variant specific to the application of entity resolution. The technique of q-grams [24] is also a useful component. Cohen [27] has used TF-IDF as part of a match function.

Since the best match function to use may be difficult to find, many works consider the use of machine-learning techniques to entity resolution. Bilenko et al. [17] demonstrate how machine learning may allow the match function

to be refined as entity resolution proceeds. Similarly, Sarawagi and Bhamidipaty [74] used active machine learning for entity resolution, allowing a human expert to assist with training. Verykios et al. [84] have applied Bayes networks to the task, and Singla and Domingos [77] applied the techniques of conditional random fields to allow decisions made for one pair of records to have an effect on the decision made for other pairs.

1.5.1 Clustering-Based ER

Since entity resolution is a clustering problem, it is natural to approach the problem with the use of existing clustering techniques. First, we mention that the well-established k-means [60] and CURE [43] techniques can only be used in the case where records can be mapped onto a Euclidean space. Records often contain strings, which are challenging to map onto a Euclidean space.

One important characteristic of a clustering algorithm is its "decision locality": can the decision of whether two records belong in the same cluster be made only by looking at the two records, or does it require more global information? The pairwise, iterative approach we use in Chapters 2 and 5 uses local decisions: the match function only needs two records to decide if the records go into the same cluster and should be merged. Dong et al. [34] also use a pairwise approach that uses only local decisions for the merging of records.

Of course, entity resolution is not only done with local merge decisions. In hierarchical clustering [53], used with entity resolution by Bhattacharya et al. [16, 15], the closest pair of clusters is merged in each iteration—a global decision. Partition-based clustering techniques [6, 25] define desirable properties on clusters and then search for partitions that maximize the degree to which these properties are satisfied. Again, partition-based techniques make global decisions when choosing candidate partitions.

As a rule of thumb, we would expect clustering techniques that use global

decisions to outperform the pairwise approach in terms of correctness, whereas the pairwise approach generally allows optimizations that improve the runtime of obtaining the ER result.

The algorithms presented in Chapter 3 work in conjunction with any ER algorithm, so any clustering technique can be plugged directly into those algorithms. Chapter 4 presents locality-sensitive hash function families that could come to the assistance of many global clustering algorithms. The clustering technique presented by Chaudhuri et al. [25], for example, makes use of an index for finding the nearest neighbors of a given record. This index could easily be implemented using LSH with the minhash extensions of Chapter 4. Chapter 6 presents measures for evaluating the results of entity resolution algorithms. Since these measures are independent of the method used to generate an ER result, they apply to the results of any ER algorithm.

1.6 Problem Statement

In summary, ER is both a difficult and important problem, and it is encountered in many fields. A fundamental problem is that ER is inherently expensive: simply applying the match function to all record pairs is an $O(n^2)$ computation. Generally, this complexity creates a tradeoff between performance and accuracy. This thesis will discuss various aspects of this problem. We will now continue to discuss the contents of each chapter individually.

Chapter 2 presents the D-Swoosh algorithm, or more precisely, the D-Swoosh framework for performing entity resolution on multiple processing nodes. The D-Swoosh framework distributes the workload using two functions: scope, which determines the nodes that a given record should be sent to, and resp, which determines the node that is responsible for performing the comparison of a given pair of records. If the scope and resp functions satisfy certain properties, then D-Swoosh will return the correct entity resolution result. The speed at which the algorithm arrives at the result, however,

may depend other properties of these two functions. This chapter presents the D-Swoosh framework along with a few scope and resp functions, and then compares the performance of D-Swoosh with these functions on various metrics.

Chapter 3 explores the application of blocking, a well known technique for reducing the runtime of entity resolution. Blocking is the division of the input records into buckets, using the assumption that records that fall into different buckets do not refer to the same entity. Blocking is relatively simple to implement for a single blocking criterion, but the algorithms in this chapter all support blocking on multiple criteria. Since blocking is especially useful on large data sets, this chapter proposes a disk-based algorithm that efficiently computes entity resolution using blocking even under the assumption that not all records can fit in memory at once.

Chapter 4 focuses on minhash and locality sensitive hashing. Locality sensitive hashing is a technique for finding similar data items, and as such is very useful in an entity resolution context (most of the blocking schemes from Chapter 3 use a family of locality sensitive hash functions defined by minhash). One drawback of locality sensitive hashing is that it relies upon the existence of a family of locality sensitive hash functions that has a close relationship with the similarity measure being used. Minhash, for example, defines a family of locality sensitive hash functions work well when data items are being compared using the Jaccard similarity of sets. Locality sensitive hashing cannot help if such hash functions do not exist for a specific application's similarity measure. This chapter expands the set of known LSH function families, providing LSH families for data types beyond sets and vectors. Specifically, this chapter extends minhash to define LSH function families for sets with weighted values, maps, and data types composed of other LSH compatible data types.

Chapter 5 extends the entity resolution problem to data with confidences. Confidence values on data can be introduced either via the base records (where some sources may be trusted more than others), or via the uncertainty of the

match and merge process. This chapter defines the result of entity resolution on data with confidences, presents various algorithms for computing the result, and explores crucial techniques for speeding up the processing of entity resolution on such data: pruning records below a confidence threshold, and pruning "dominated" records.

Chapter 6 explores methods for evaluating the results of an entity resolution algorithm. A standard technique for evaluating ER results is to compare an algorithm's results to a gold standard. There are, however, many (often conflicting!) ways to evaluate the distance from a result to the gold standard. Although one measure is more popular than the rest, there is no clear reason for it to be used as a standard for all applications. This chapter proposes a new measure (called merge distance) for evaluating entity resolution results. Merge distance is configurable via two functions, which can be used to create a measure tailor-made for a particular application. This chapter also shows that two existing ER measures are in fact special cases of merge distance.

Finally, in Chapter 7, we summarize the results of this thesis and discuss future directions in solutions to the entity resolution problem.

Chapter 2

Distributed Swoosh

2.1 Introduction

There are generally two ways to deal with the extremely high computational load of ER: One is to "partition" the problem using semantic knowledge and the other is to exploit multiple processors. For example, in a comparison-shopping application, we may have a collection of records representing products, and each record may have a field indicating the category of the product. We can divide our product records by category (CDs, MP3 players, cameras, etc.), and then only try to match records in the same category (possibly with some overlap between partitions, *e.g.*, if products have multiple categories). Here we are using the knowledge that records in different categories will never match. Of course, in some applications we may not have such knowledge, or we may not be willing to assume that the categorization is perfect. But even with semantic knowledge, there may still be many comparisons to perform because the resulting categories or buckets are still relatively large. Thus, we believe that the second option, parallelism, will be essential in many applications. In this chapter we focus on techniques for distributing the ER process over multiple processors, both for the case where we have no semantic knowledge, and the case where we have ways of partitioning the problem.

2.1.1 Overview of Our Approach

To motivate our approach, and to illustrate some of the challenges faced, consider the following simple example. Say we have 6 input records, r_0 through r_5. Figure 2.1 shows one possible ER outcome for this example, assuming we run on a single processor. After comparing the input records, we discover that r_0 and r_3 match, so they are merged into r_6. The new record then matches with r_4; the resulting merged record is r_8. In this example, r_2 and r_5 also merge, into record r_7. The final answer is $\{r_1, r_7, r_8\}$, where none of these records match any further.

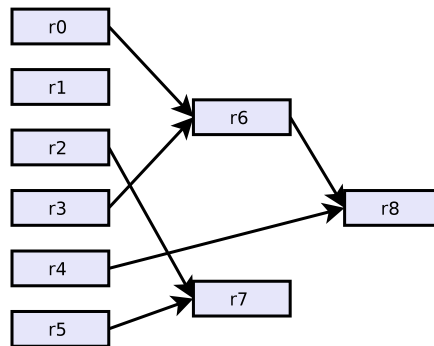

Figure 2.1: A sample run of Entity Resolution

Say we want to distribute this ER processing over three processors, P_0, P_1 and P_2. The problem is analogous to performing a distributed self-join in a database system: we wish to distribute the records to the processors so that the comparisons are done in parallel. However, there are two important differences with joins:

- With a join, the match function is simple and known, so we can exploit this knowledge. For example, if the common join attribute is X, we can distribute records with $X < 10$ to one processor, those with $X \geq 10$ to another, and comparisons will be localized to one processor. With ER, we may not know anything about the match function, so we need to develop mechanisms that work in general.

- With ER, unlike joins, there is a "feedback" loop where merged records need to be re-distributed and compared to other records.

To address these issues, we use a general distribution function we call *scope*. That is, scope(r_i) gives us the set of processors to which r_i will be sent. For example, Figure 2.2 shows one possible scope function. Note that this scope function can be expressed by scope(r_i) = $\{P_j, P_k\}$ where $j = i \bmod 3$, and $k = (i + 1) \bmod 3$.

P_0	P_1	P_2
r_0	r_0	
	r_1	r_1
r_2		r_2
r_3	r_3	
	r_4	r_4
r_5		r_5
r_6	r_6	
	r_7	r_7
r_8		r_8

Figure 2.2: One possible scope function

Notice that for every pair of records, there is at least one processor that has both records. For example, the pair r_0, r_2 is at P_0, and the pair r_0, r_3 is at P_0 and P_1. Thus, if each processor executes all match comparisons on records in its scope, we will not miss any comparisons. Our algorithm will proceed by distributing the input records to processors using the scope function. Then each processor will apply the match function to the pairs of records it has.

Of course, with this scheme we may perform redundant comparisons, e.g., in our case, the r_0, r_3 comparisons would be performed by both P_0 and P_1. To avoid redundant comparisons we also introduce a *responsible* function (denoted resp): processor P_k will apply the match function to r_i and r_j only if it is in the scope of both records, and resp(P_k, r_i, r_j) is true. Intuitively, the responsible function needs to compute the set of processors that will have both

r_i and r_j, and then deterministically choose one of them, without overloading some processors. Since the responsible function typically performs simple arithmetic (or more basic) operations, it is much cheaper to compute than the match function, and hence the saving can be significant. In Sections 2.4.1 and 2.4.2, we provide several instances of scope and resp functions.

When a processor finds a match among its records, it computes the merged record and distributes it so that the new record is compared against all existing records. Thus, scope and responsible functions should work with merged records, and guarantee that no matches are missed. However, even with a good responsible function, it will be impossible to avoid all redundant work due to the concurrent execution of related comparisons. To illustrate this unavoidable redundancy, suppose that three records r_i, r_j and r_k all match pairwise. Say that processor P_0 is responsible for the r_i, r_j comparison, processor P_1 for the r_i, r_k pair, and P_2 for the r_j, r_k pair. Each processor in parallel will discover a match and will merge its pair of records. The three resulting records, r_{ij}, r_{ik} and r_{jk} are distributed, and compared again, and each pair of the new records could be the responsibility of a different processor. Each of these processors will then independently compute the same merged record r_{ijk} (recall that merge order is not important). Thus, r_{ijk} is generated three times. Our algorithm will eventually remove the duplicate copies, but we have performed more work than necessary to get r_{ijk}. Depending on the timing of events, some of the redundant work may be avoided, but there is always the potential for redundancy. As a matter of fact, adding processors beyond some limit may be counterproductive, as it may lead to more redundant work and more communication overhead. One of our goals here is to empirically study these issues, and to evaluate how much redundant work may be done in practice.

If we have semantic knowledge, we may develop scope and responsible functions that reduce the number of comparisons, analogous to what is done with joins. For instance, if product records can only match if their "price" attribute is numerically close, we can design a scope function that assigns records to processors by price, so a processor only gets records whose price is close.

We discuss scope and resp functions that exploit semantic knowledge in Section 2.4.2.
In summary, the contributions we make are:

- We define the problem of distributed entity resolution, for the case where black-box match and merge functions are used (Section 2.3).
- We present a generic, distributed algorithm, D-Swoosh, that runs ER on multiple processors using scope and responsible functions, and computes the correct answer (Section 2.3.2).
- We suggest a variety of scope and responsible functions, some for scenarios with no semantic knowledge, others that exploit knowledge that often arises in ER applications (Section 2.4).
- We present metrics to evaluate distributed ER, and conduct a detailed evaluation of D-Swoosh and the various scope and responsibility functions on a 15 processor testbed, using comparison-shopping data provided by Yahoo! (Section 2.5). Our results show that the ER process can take advantage of parallelism with great benefit.

Related work is presented in Section 2.6. Section 2.7 is our conclusion.

The contents of this chapter are joint work with a team of researchers: Benjelloun, Garcia-Molina, Gong, Kawai, Larson, Menestrina, and Thavisomboon. This team published this work in ICDCS 2007 [12]. The author of this thesis played a significant role in all aspects of the work, including the development of algorithms, coding, and experimentation.

2.2 Preliminaries

Section 1.3 describes the pairwise ER model that we use in this chapter, including the four ICAR properties. Since we assume all four ICAR properties, we will be using merge domination as the domination order for the remainder of the chapter. We will therefore shorten the name and simply refer to it as "domination".

The work in this chapter is based on the R-Swoosh algorithm presented by Benjelloun et al. [11]. R-Swoosh is an optimal algorithm for performing entity resolution on a single processor when the ICAR properties hold. R-Swoosh incrementally processes the input records, while maintaining a set R' of (so far) nondominated, nonmatching records. R-Swoosh performs a merge as soon as a pair of matching records is found, puts the resulting record back into the input set R, and deletes the pair of matching records immediately.

To illustrate the operation of R-Swoosh, consider a run on the records from the example in Section 2.1.1, supposing the records are processed in the order of their identifiers. We take r_0 and compare it against records in R'. Since R' is initially empty, there are no matches with r_0, so r_0 is moved to R'. Next, r_1 is compared against records in R', i.e., against r_0. In our example (Figure 2.1), there are no matches, so r_1 is also moved to R'. Record r_2 is moved to R' in a similar fashion. When r_3 is compared against R', we discover that r_0 and r_3 match. Therefore, $r_6 = \langle r_0, r_3 \rangle$ is added to R, and r_0 and r_3 are removed from R' and R, respectively. Next, r_4 is processed and added to R', r_5 is processed generating $r_7 = \langle r_2, r_5 \rangle$, which is added into R; r_6 is processed generating $r_8 = \langle r_4, r_6 \rangle$ into R. The remaining R records are processed and moved to R'. At the end, R' holds $ER(R) = \{r_1, r_7, r_8\}$.

2.3 Distributed ER

In this section, we extend the R-Swoosh sequential algorithm for generic ER to be distributed, in order to run in parallel on multiple processors. We start by defining the primitives needed to capture distribution, then present the algorithm.

2.3.1 Distribution Primitives

Our p processors are $\mathcal{P} = \{P_0, \ldots, P_{p-1}\}$. As mentioned in Section 2.1, we introduce a "scope" function to distribute records across processors and a "responsible" predicate to reduce redundant work, by deciding which processors are responsible for which comparisons. These are defined formally as follows.

Definition 2.3.1. *A* scope function *is a function from $\mathcal{R}$, the domain of records to $2^{\mathcal{P}}$ that assigns to each record* r *a subset* scope(r) *of the processors. A* responsible predicate *is a Boolean predicate over $\mathcal{P} \times \mathcal{R} \times \mathcal{R}$. When* $resp(P_i, r, r') =$ `true`*, we say that processor* P_i *is responsible for the pair of records* (r, r')*. Scope and resp satisfy the following "coverage" property.*
Coverage property: *For any pair of matching records* r, r'*, there exists at least one processor* P_i *such that* $P_i \in scope(r) \cap scope(r')$ *and* $resp(P_i, r, r') =$ *true.*

Interestingly, the coverage property is related to the distributed mutual exclusion problem [32]. We can think of record r as a process that "locks" the processors in its scope. Since r′ also locks its scope, and since $scope(r) \cap scope(r')$ is not empty, then r and r′ are mutually excluded. Thus, requiring scopes to intersect (so every pair of matching records is compared) is equivalent to requiring the locks to conflict on at least one processor. Any coterie [39] (roughly, a set of groups such that any two groups share at least one element) used for mutual exclusion can hence be used for guaranteeing coverage in our context. However, we only want coteries that distribute the workload evenly, since we are doing expensive record comparisons and not locking.

Distributed Computing Model

We make the standard assumptions about our distributed computing model: First, we assume that no messages exchanged between processors are lost.

Second, we assume that messages sent from one processor to another are received and processed in the same order as they are sent (this can be achieved easily by numbering the messages). Finally, we assume that processors are able to use an existing distributed termination detection technique to detect that they are all in an idle state (see, e.g., [22]).

2.3.2 The D-Swoosh Algorithm

We are now ready to give our general algorithm for distributed ER. Algorithms 2.1 and 2.2 together asynchronously run a variant of the R-Swoosh algorithm at each processor P_i. Initially, each P_i receives the records in its scope. Each P_i maintains a set R'_i of nondominated, nonmatching records. The processor also keeps a set D_i of the records it knows have been deleted.

There are two types of messages sent between peers in D-Swoosh: add record and delete record. We will represent a message instructing a processor to add a record r with the notation +(r). Likewise, a message instructing a processor to delete a record r will be written as −(r).

When a new (added) record r is received by P_i, it is successively compared to every record r' in R'_i, provided P_i is responsible for that comparison. If a match is found, the matching records are merged immediately, and messages are sent to the relevant processors (identified through the scope function) instructing them to add or delete records. If no match is found, then r is added to R'_i.

Note that if the new merged record $\langle r, r' \rangle$ is identical to either r or r', then not all the messages are sent out. Furthermore, if $\langle r, r' \rangle$ is equal to r, the record we are processing, we continue processing r. If no match is found with any of the R'_i records, r is added to R'_i. The algorithm terminates when all processors have resolved all the records in their scope, and the messages they sent have been received and processed. The final answer is the union of the R'_i sets at the processors.

Consider running D-Swoosh on our 6 record example dataset. Recall that

our example scope function (Figure 2.2) assigns to P_0 records $\{r_0, r_2, r_3, r_5\}$, while P_1 gets $\{r_0, r_1, r_3, r_4\}$ and P_2 gets $\{r_1, r_2, r_4, r_5\}$. Let us focus on the actions at P_0. Say P_0 is responsible for the r_0, r_3 comparison, but not the r_2, r_5 one. When P_0 merges r_0, r_3 into r_6, it sends $-(r_0)$, $-(r_3)$ messages to itself and P_1 (the processors that have these records in their scope). It also sends $+(r_6)$ to the processors in scope(r_6), i.e., P_0 and P_1.

In the meantime, P_0 will be getting messages from other processors. For example, the processor responsible for the pair r_2, r_5 will send to P_0 messages $-(r_2)$, $-(r_5)$ when those records merge. As the messages arrive at P_0, they are processed. For instance, when the $-(r_2)$ message arrives at P_0, record r_2 is removed from R'_0, if it is there. Note that r_2 may not be in R'_0, e.g., the initial $+(r_2)$ message may not have arrived. This is why the $-(r_2)$ message is "remembered" in the set D_0: when r_2 finally arrives, it will be ignored because r_2 is in D_0. Records r_6 and r_4 will be merged at processor P_1 to form r_8, $+(r_8)$ will be sent to processors r_0 and r_2.

Input: A set R of records
Output: A set R′ of records

Initialization:
$\forall P_i$, $R'_i \leftarrow \emptyset$
$\forall P_i$, $D_i \leftarrow \emptyset$
$\forall r \in R$ **send(**$+(r), P_i$**)** to every processor P_i in $S(r)$

Termination:
Detect that all processes are idle
return $\cup_{P_i} R'_i$

Algorithm 2.1: D-Swoosh Initialization and Termination

Theorem 2.3.2. *Given a set of records* R*, the D-Swoosh algorithm terminates and computes* ER(R).

Proof. Benjelloun et al. [11] show that, given a set of records R, ER(R) is computed by any maximal derivation sequence of R. A derivation sequence is a

```
on receive(+(r), P_i):
  if r not in (R'_i ∪ D_i) then
    for all records r' in R'_i such that resp(P_i, r, r') do
      if r ≈ r' then
        r'' ← ⟨r, r'⟩
        if r'' = r then
          ∀P_j ∈ scope(r'), send(−(r'), P_j)
        else
          if r'' ≠ r' then
            ∀P_j ∈ scope(r'), send(−(r'), P_j)
            ∀P_j ∈ scope(r''), send(+(r''), P_j)
          end if
          ∀P_j ∈ scope(r), send(−(r), P_j)
          exit +(r) handler
        end if
      end if
    end for
    add r to R'_i
  end if

on receive(−(r), P_i):
  remove r from R'_i if present
  add r to D_i
```

Algorithm 2.2: D-Swoosh Message Processing

sequence of merge steps (addition of a merged record) and purge steps (deletion of a dominated record). A derivation sequence is maximal if no additional merge or purge steps can be performed.

To prove the correctness of D-Swoosh, we show that it computes a maximal derivation sequence of the set $I = W \cup \bigcup_{P_i} R'_i$, where W is the set of all records pending processing at any of the processors. Since I is initially equal to R and $W = \varnothing$ when the algorithm terminates, this will effectively establish the correctness of D-Swoosh.

We first prove that each event handled by a processor corresponds either to a merge or a purge step on I, or keeps I unchanged. We then prove that the derivation sequence is maximal, and that the algorithm terminates.

When processor P_i handles a $+(r)$ message, r is either added to R'_i or a $-(r)$ message is sent to one or more processors. In both cases, r remains in I. If r matches a record r' in R'_i, the two records are merged into r'', and $+(r'')$ is sent to one or more processors (hence added to W). In case r'' was not present in I before (e.g., at some other processor), the algorithm has performed a merge step: a record obtained by merging two records previously present in I is added to I.

When processor P_i handles a $-(r)$ message, r is removed from R'_i, hence r is removed from I if no other processor still has r or is waiting to process r. It is easy to see that $-(r)$ messages are only sent when r is known to be dominated by another record in I. Therefore, if r is effectively removed from I, the algorithm has performed a purge step.

D-Swoosh performs a sequence of merge and purge steps on I, and therefore computes a correct derivation sequence of I. We must now show that the derivation sequence is maximal, i.e., once I stops evolving, no further merge or purge step is possible. We will then show that the algorithm terminates. Observe that for any record r, all messages relevant to r ($+(r)$ or $-(r)$) are sent exactly to the set of processors in scope(r). Therefore the only processors to ever manipulate r are precisely those in scope(r).

Suppose that I does not evolve, but a purge step is possible, say record r_1 is dominated by record r_2. By the coverage property r_1 and r_2 are handled by at least one processor P_i. Processor P_i must send $-(r_1)$ to all processors in scope(r_1). After all processors have handled this message, r_1 is not in W. Moreover, r_1 is not present in any of the R'_i sets, and cannot reappear because it was added to the D_i sets. Therefore, the purge step would effectively have been performed, again a contradiction.

Suppose now that I does not evolve anymore, and a merge step is still possible, say between records r_1 and r_2. We have already shown that no purge steps are possible, so we can assume that r_1 and r_2 are not dominated by any other records in I. Therefore a $+(r_1)$ (resp. $+(r_2)$) message is received at some point by all the processors in scope(r_1) (resp. scope(r_2)). Again, by

the coverage property, one of the processors in the intersection of these scopes (say, P_i) is responsible for the pair (r_1, r_2). Upon receiving the first of these two records (say, r_1), P_i adds r_1 to R_i, because it is not dominated. When P_i receives r_2, it compares it with r_1 and performs the merge step, a contradiction.

No merge or purge step is possible on the final state of I, hence the algorithm computes a maximal derivation sequence. We now show that the algorithm terminates. Observe that after the algorithm has computed the maximal derivation sequence, no two records may match. Indeed, if two records match, then a merge or purge step would necessarily follow, a contradiction. As a consequence, no more "send" messages are exchanged by processors. Any record in W disappears from there once it has been processed by all the processors in its scope. W eventually becomes empty, and all processors become idle, hence the algorithm terminates. □

2.4 Choosing scope and resp

We now consider particular scope and resp functions to distribute the ER work among a set of processors. We start with strategies that are always applicable because they do not make any extra assumptions, then turn to strategies that exploit existing semantic knowledge.

2.4.1 Strategies Without Domain Knowledge

In this section, we consider strategies in which we do not have any a priori domain knowledge about records, and therefore must consider all possible matches between records. Our goal will be to distribute the work evenly across processors, while trying to reduce communications and redundant computations. We present three schemes: full replication, majority (which generalizes the scheme in Figure 2.2) and grid. We do not prove it here, but it is

not difficult to see that the scope and resp functions given in Tables 1, 2, and 3 satisfy the coverage property of Definition 2.3.1.

Table 1 Full Replication Scheme

Scope of r_i:
  Cardinality: p
  for $l := 0$ to $p - 1$, processor P_l is in scope(r_i).
Records r_i, r_j at:
  Number of overlap processors: $m = p$;
  for $l := 0$ to $m - 1$, processor P_l in scope of both r_i and r_j.
Resp(P_k, r_i, r_j) = true if $k = \min(i, j) \bmod p$.

Full Replication Scheme

In the full replications scheme, records are sent to all p processors. The scheme is defined in Table 1, using the same style that will be used for the other schemes. To select the processor that is responsible for comparing records r_i and r_j, we select the smallest of the two indexes, i or j, and use it to identify the processor. If record identifiers are evenly distributed, this scheme ensures that each processor has a similar number of pairs to compare.

The full replication scheme clearly has a high storage overhead, so our next two schemes will reduce storage costs. However, as shown by our experiments (Section 2.5), full replication has benefits under some of the other metrics we consider.

Majority Scheme

The majority scheme (Table 2) generalizes the scope and resp functions we used in Section 2.1, from 3 to an arbitrary number p of processors. The essential idea is that if any record has a scope that consists of more than half (i.e., a majority) of the processors, then any pair of records will share at least one common processor in their scopes. We also generalize the resp

Table 2 Majority Scheme

Notation:
  Number of buckets: B;
  Total number of processors: $p = B$;
  Record r_i hashes to bucket x; record r_j to bucket y;
  Assume $x \geq y$.
Scope of r_i:
  Cardinality: $\lfloor p/2 \rfloor + 1$;
  for $k := x$ to $x + \lfloor p/2 \rfloor$ (modulo p),
    processor P_k is in scope(r_i).
Records r_i, r_j at:
  Case (a): $x - y < p/2$;
    Distance: $d = (x - y) \bmod p$;
    Number of overlap processors: $m = (\lfloor p/2 \rfloor + 1) - d$;
    for $k := x$ to $x + m - 1$ (mod p),
      processor P_k is in scope of both r_i and r_j;
    $Resp(P_k, r_i, r_j)$ = true if $k = x + (\min(i, j)) \bmod m$).
  Case (b): $x - y > p/2$;
    Reverse x and y in equations for case (a).
  Case (c): $x - y = p/2$;
    Number of overlap processors: $m = 2$;
    Processors P_x and P_y are in scope of both r_i and r_j;
    $Resp(P_k, r_i, r_j)$ = true if
      ($\min(i, j)$ is even and $k = x$) OR
      ($\min(i, j)$ is odd and $k = y$).

function such that any pair of records is the responsibility of exactly one processor. With the majority scheme, every processor receives and stores about half of the records in the dataset.

Figure 2.3 illustrates the majority scheme with p = 7 processors. The records are partitioned into 7 buckets, b_0 through b_6, using hashing or modulo arithmetic. Each bucket b_x is distributed to 4 processors, starting with processor x. Additional records would follow the same pattern. For example, if r_{11} falls into bucket b_4, the record is sent to P_4, P_5, P_6, and P_0. Similarly, we see that processor P_2 holds buckets b_0, b_1, b_2 and b_6.

As shown in Table 2, there are three cases to consider in determining where the scopes of two records intersect. For instance, consider two records

Table 3 Grid Scheme

Notation:
  Number of buckets: B;
  Total number of processors: $p = B(B + 1)/2$;
  If $i \geq j$, processor $P_{i,j}$ is processor P_k,
    where $k = i + jp - j(j + 1)/2$;
  If $i < j$, processor $P_{i,j}$ is processor $P_{j,i}$;
  Record r_i hashes to bucket x; record r_j to bucket y.
Scope of r_i:
  Cardinality: B;
  for $k := 0$ to $B - 1$, processor $P_{x,k}$ is in scope(r_i).
Records r_i, r_j at:
  Number of overlap processors:
    If $x \neq y$ then $m = 1$ else $m = B$;
  If $x \neq y$ then processor $P_{x,y}$ is in scope of both r_i and r_j;
  If $x = y$ then scope(r_i) = scope(r_j).
Resp(P_k, r_i, r_j) = true if $P_k = P_{x,y}$.

in buckets b_1 and b_3 respectively. This situation corresponds to the first case since $x - y$ (i.e., the difference in bucket indexes) is $3 - 1$, which is less than half the processors. Here, the scopes overlap at $m = 2$ processors: P_3 and P_4. The resp function then uses the smallest record identifier to select one of these two processors.

P_0	P_1	P_2	P_3	P_4	P_5	P_6
b_0	b_0	b_0	b_0			
	b_1	b_1	b_1	b_1		
		b_2	b_2	b_2	b_2	
			b_3	b_3	b_3	b_3
b_4				b_4	b_4	b_4
b_5	b_5				b_5	b_5
b_6	b_6	b_6				b_6

Figure 2.3: Scope for the majority scheme

Grid Scheme

With the grid scheme (Table 3), we again divide records into B disjoint buckets. For this scheme, we require that the number of processors be $B(B+1)/2$, which corresponds to the number of elements in half a matrix of size B, including the diagonal. For $B = 7$ buckets, we need 28 processors, which are arranged as illustrated in Figure 2.4.

	b_0	b_1	b_2	b_3	b_4	b_5	b_6
b_0	P_0						
b_1	P_1	P_7					
b_2	P_2	P_8	P_{13}				
b_3	P_3	P_9	P_{14}	P_{18}			
b_4	P_4	P_{10}	P_{15}	P_{19}	P_{22}		
b_5	P_5	P_{11}	P_{16}	P_{20}	P_{23}	P_{25}	
b_6	P_6	P_{12}	P_{17}	P_{21}	P_{24}	P_{26}	P_{27}

Figure 2.4: Processor arrangement in the grid scheme

For the scope, every record in bucket j is sent to the processors that appear on the jth line and column of the (half) matrix. For instance, in our example, records in bucket 3 are sent to processors $\{P_3, P_9, P_{14}, P_{18}, P_{19}, P_{20}, P_{21}\}$. Observe that processors on the diagonal get records from exactly one bucket, while processors in other positions receive records from two buckets. For the resp function, the processor at the intersection of the ith and j^{th} column is responsible for the pairs of records r, r' such that one of them is in bucket i and the other in bucket j. Hence, processors on the diagonal are responsible for comparing all the records in their scope, while other processors are only responsible for the pairs of records in their scope which belong to different buckets.

To illustrate, in our example, processor P_{18} has all the records of bucket 3 in its scope, and is responsible for comparing all of them pairwise. Processor P_{16} is at the intersection of line 5 and column 2 in the matrix, and therefore has all records of both buckets 2 and 5, but is only responsible for comparing the pairs belonging to different buckets.

Note incidentally that with the grid scheme, the D-Swoosh answer can be computed by taking the union of the records held by the processors on the diagonal only. This operation does not even require looking for duplicate records, as these sets are disjoint.

While the full replication, majority, and grid schemes all efficiently distribute the workload, they have different initial storage costs. In the grid scheme, for large B, the number of processors $p = B(B+1)/2$ approaches $B^2/2$, so $B = \sqrt{2p}$. Each of n input records is copied to B processors, so the total initial storage cost is $n\sqrt{2p}$ (compared to np for the full replication scheme, and $n(\lfloor p/2 \rfloor + 1)$ for the majority scheme).

We can show a lower bound on the initial storage costs of any scheme that efficiently distributes the workload (measured in number of comparisons). Any scheme must be able to handle the case where no records match. In this case, there are $n^2/2$ comparisons that must be performed among the initial records. Since the workload is distributed evenly, there will be $n^2/2p$ comparisons performed at each processor. For a processor to perform c distinct comparisons, it must have about $\sqrt{2c}$ distinct records. Therefore, each processor requires $\sqrt{n^2/p} = n/\sqrt{p}$ records. Since there are p processors, the total initial storage cost is at least $n\sqrt{p}$. Thus, we see that the grid is within a factor of $\sqrt{2}$ of the optimal scheme for the initial storage cost.

Cost of the resp Function

In Section 2.1, we argued that calling the responsible function is inexpensive. The functions we have presented here are relatively cheap since they only involve simple arithmetic. Furthermore, for the grid scheme, the resp function can be checked simply by keeping the records at a processor in two areas. In particular, for non-diagonal processors (say, at position (i, j)), the set R' can be kept as two disjoint sets $R'|_i$ and $R'|_j$ holding the records that belong to each of the buckets respectively. When a new record arrives, it need only be compared to the records in the opposite bucket. For processors on the

diagonal, resp is true for all pairs of records in the scope of the processor (which are the only records the processor gets to process), and hence can be skipped.

2.4.2 Strategies With Domain Knowledge

We now turn to strategies that use domain knowledge to distribute ER computations across multiple processors. Essentially, domain knowledge provides *necessary conditions* for pairs of records to match. In our example comparison-shopping application, we may know that products may match only if they are in related categories, or if their prices are not too far apart. The key assumption is that checking necessary conditions (e.g., whether two products are in the same category) is much cheaper than invoking the full match function. With a single processor, the cost of ER can already be greatly reduced by only comparing pairs of records that satisfy the necessary conditions. This technique is widely used in a sequential setting, and commonly referred to as "blocking" in the ER literature [9].

With multiple processors, domain knowledge can be even more beneficial because groups of records that are known in advance not to match can be processed independently (and in parallel) by different processors. For instance, one processor can match the DVD records, another the cameras, and so on, assuming that merged records remain in their same category.

Groups

To exploit domain knowledge, we divide the records into G semantically meaningful *g*roups, $g_0, g_1, ...g_G$. Each record r (in the original dataset, or derived through merges) belongs to one or more groups. (Note that the buckets used in Section 2.4.1 had no semantic meaning, and records could only be in one bucket.)

Figure 4 summarizes the scope and resp functions we use with groups. As we can see, the scheme is very simple: each group is assigned to one processor,

Table 4 Groups Scheme

Notation:
  Number of groups: G;
  Total number of processors: $p = G$;
  Processor P_k gets records in group g_k;
  Record r_i is in set of groups X; record r_j is in set Y.
Scope of r_i:
  Cardinality: $|X|$;
  for all $g_k \in X$, processor P_k is in scope(r_i).
Records r_i, r_j at:
  Number of overlap processors: $m = |X \cap Y|$;
  for $g_k \in X \cap Y$, processor P_k is in scope of both r_i and r_j;
Resp(P_k, r_i, r_j) = true if $g_k = l^{th}$ group in $X \cap Y$,
    where $l = (\min(i, j)) \bmod m$.

and one processor gets only one group. With buckets (Section 2.4.1), the scope function ensured that any pair of records r_i, r_j would show up at some processor for comparison. With groups, there is no such guarantee: r_i and r_j will be at a common processor only if they happen to be in a common group. Hence, now it is the burden of the semantic group assignment function to ensure that if there is any chance that r_i and r_j may match, then they should be assigned to some common group. This property can be expressed as follows.

Group property: *If two records (initial or merged)* r_i *and* r_j *may match, they must be assigned to at least one common group.*

The above group property ensures that the coverage property of Definition 2.3.1 holds (in spite of the trivial scope function), so D-Swoosh correctly computes ER(R). We now show how groups that satisfy this property can be derived for common forms of domain knowledge: value equality, hierarchies and linear ordering.

Value Equality

A common situation in ER is when pairs of matching records share a common value for some attribute, which we call the "equality" attribute. For instance, product records may all have a "category" associated with them, and two records may only match if they are in the same category. For flexibility, records can have multiple values for the equality attribute (*e.g.*, a product may be a sporting good and a clothing item). In this case, pairs of records match if they share at least one value for the equality attribute, in the spirit of canopies [62]. A merged record inherits the equality values of its source records (to ensure representativity holds).

One possible strategy for value equality is to have one group per value, e.g., one group for cameras, one for DVDs, and so on. However, in some applications we may have many category values, and hence we may need too many groups/processors. Instead, a *partition* of the values can be used to determine the groups. For instance, cameras, MP3 players, etc. can go into one partition, while shoes, shirts, jackets, etc. can go into another. The group property continues to hold, as products in the same category continue to be in the same group. The partition can either be arbitrary, or semantically meaningful as in our example. A semantic partition is advantageous if it increases the chances that a multi-attribute record falls within one group. For instance, with a semantic partition that includes cameras and telephones, a camera phone will be in one group; with an arbitrary partition, the camera phone record may have to be sent to two processors.

Hierarchies

Another common form of domain knowledge is when the values of some attribute, called the "hierarchy" attribute, are related through a hierarchy. For example, vehicles may be cars or trucks; Cars may be sedans, hatchbacks, station-wagons, etc.; Trucks may be pick-up, vans, etc. A pair of records may

only match if one hierarchy attribute is a descendant of the other. For instance, a sedan may match a car, but not a truck. When two records match, the merged record inherits the most general of the hierarchy values (again, to ensure representativity). (If a record has no value for the hierarchy attribute, we can assign it the root value.)

To handle hierarchies, we create one group g_t for each term t of the hierarchy. A record with value t gets assigned to group g_t, as well as to all groups g_s where s is a descendant of t in the hierarchy. To illustrate, a car record will belong to the car and sedan groups; a sedan record will only belong to the sedan group.

Linear Ordering

A last form of domain knowledge we consider here is a linear ordering of what we call a "window" attribute. In this scenario, two records may match only if their values are within some window δ. For instance, product records may have a price attribute, and records whose prices differ by more than $50 cannot represent the same product. The window size may also be relative to the values, e.g., one price must be within some percentage of the other, or may be determined based on the density of the records on the window attribute, e.g., a product record may only match with the n cheaper or more expensive records. Records may also have multiple values for the window attribute, in which case they may only match if at least one of the values for each of the records satisfy the window condition.

We form groups by dividing the range of window values into p partitions (recall that p is the number of processors and of groups). Say the ith partition has lower bound L_i (inclusive) and upper bound L_{i+1}. Also, let δ be the window size, i.e., two matching records must have window attribute values distant by less than or equal to δ. A record r belongs to group g_i if one of the r values for its window attribute is in the interval $[L_i, L_{i+1} + \delta)$.

Notice that if δ is small relative to the partition sizes and records values

for the window attribute are evenly distributed, only a "few" records that happen to have values close to the boundaries are replicated. For instance, if the boundaries are at prices 0, 1000, 2000, ... dollars and δ is $50, a record with price $1049 will be in two groups, but a record with price $1050 (to $1949) will not be replicated. As δ grows, more records are replicated, into possibly more than two groups. For instance, if δ is 2000, a record with price $2500 is in three groups. Thus, there is an interesting interplay between the application parameters (range of values, window size) and the number of processors we may have available or we may need to scale-up performance.

2.4.3 "Tuning" scope and resp

So far we have presented scope and responsible functions that try to distribute records and responsibility "uniformly" across processors. However, in some cases that may not be the best approach. For example, consider three records r_1, r_2 and r_3 that all match pairwise, and will eventually be merged into record r_{123}. In general, each pair of initial matches will be discovered by different processors, e.g., P_1 will generate $\langle r_1, r_2\rangle = r_{12}$, P_2 will generate $\langle r_1, r_3\rangle = r_{13}$, and P_3 will generate $\langle r_2, r_3\rangle = r_{23}$. The new records will then be merged at other processors, e.g., P_4 will find $\langle r_{12}, r_{13}\rangle = r_{123}$, P_5 will find $\langle r_{12}, r_{23}\rangle = r_{123}$, P_6 will find $\langle r_{23}, r_{13}\rangle = r_{123}$. Of course, some of these processors may be the same, and such coincidences improve performance. For instance, if $P_4 = P_5$, one useless generation of r_{123} is avoided. Similarly, if $P_1 = P_2 = P_4 = P_5$, then r_{123} is found quickly by P_1 without any message delays. (The faster P_1 propagates $+(r_{123})$, the more unnecessary comparisons we can avoid.)

Our scope and resp functions differ in how they distribute records, and hence the number of "lucky coincidences" like the ones illustrated above varies. As a matter of fact, in our experiments we will see that a scheme like the full replication can do less work than Majority because merges are disseminated faster and there is less redundant work, even though the full replication

is making more copies of records than Majority is.

If we change how records are assigned to buckets, or if we change the responsible function a bit, we may even be able to increase the chances that the lucky coincidences happen more often. For example, say a merged record is assigned to buckets using the smallest identifier in its lineage. In our example, r_{12} and r_{13} would fall in the same bucket as r_1. Since the scope of these three records would be identical, this mapping increases the chances that $P_4 = P_1$ and $P_5 = P_1$. We can increase the chances further by not selecting the responsible processor randomly. For instance, say processor P_k is responsible for r_i, r_j if it is the processor with the smallest identifier that is in both the scope of r_i and of r_j. With this additional change, it is certain that $P_4 = P_1$ (P_1 is the processor with the smallest identifier in the scope of r_{12}, and hence it is also the smallest processor in the intersection of the scopes of r_{12} and r_{13}) and that $P_5 = P_1$. Thus, we expect that merges will be discovered "faster", without as many message exchanges. On the other hand, the load may not be uniformly distributed (e.g., the processor with the smallest identifier will tend to get more work), so each alternative needs to be carefully evaluated.

When semantic knowledge is available, we can again modify the scope and responsible functions to try to discover merges with fewer message delays. For example, consider distance proximity, and some records that have close values for their window attribute and lie close to a partition boundary. In this case, it makes sense to have one processor out of the ones that share that overlap region be responsible for comparisons, instead of having all the processors share the load.

2.5 Experiments

We implemented the D-Swoosh algorithm, and the various choices of scope and resp functions discussed in the previous sections, and conducted extensive experiments on subsets of a comparison shopping dataset from Yahoo! In this

section, we describe our implementation and experimental setting, and report the main findings of our experiments.

2.5.1 Experimental Setting

We ran our experiments on subsets of a comparison shopping dataset provided by Yahoo!. The dataset consists of product records sent by merchants to appear on the Yahoo! shopping site. It is about 6Gb in size, and contains a very large array of products from many different vendors. Such a large dataset must be (and is in practice) divided into groups for processing. For our experiments we extracted a subset of about 138,000 records consisting of up to 50,000 records for each of four keywords: "iPod", "mp3", "book", and "dvd". From this subset, we randomly selected subsets varying in size, starting with a size of 5,000 records.

Each of these smaller datasets represents a possible set of records that must be resolved by one or more processors. If we are modeling a scenario with no semantic knowledge, then all records in the dataset must be compared. If we are considering a scenario where semantic knowledge is available, the database can be further divided by category.

Our match function considers two attributes of the product records, the title and the price. Titles are compared using the Jaro-Winkler similarity measure [90], to which a threshold t is applied to get a yes/no answer. Prices are compared based on their numerical distance, and match if one is within some percentage α of the other. Two records match if both their titles and price match. We experimented with different t and α values, and selected ones that worked reasonably well with this data ($t = 0.95$, $\alpha = 0.33$). In one of our experiments (see below) we vary t to model different applications where more or fewer records would match. To merge pairs of records, we simply took the union of distinct values for each attribute.

We implemented the D-Swoosh algorithm in Java 1.5, using TCP connections for peer-to-peer message passing. To avoid stalling due to TCP flow control,

sending and receiving messages were separated into different threads. For our experiments we ran the code on up to 15 approximately identical machines, all connected via Ethernet. Each machine had a Pentium 4 2.8GHz CPU with hyperthreading enabled, and 2GB of RAM. We ran the code under the Sun Java VM 1.5.0_05-b05, on a GNU/Linux system using kernel 2.6.11 with SMP enabled.

To evaluate the performance of D-Swoosh, we use the following metrics, which can be precisely evaluated in our environment.

- Runtime: The total amount of time to distribute the records to the peers, compute the result, and detect the termination of the algorithm. This metric captures the raw performance of the algorithm under the specified conditions.
- Aggregated computation: The main cost of ER is the pairwise comparisons of records, so we count the overall number of comparisons performed by all processors. When multiple processors are used, this metric captures the overhead of parallelism.
- Communication: We count the total number of bytes sent across the network by the processors, and use this as a metric of the overall communication cost. We focus on the communication cost generated by the actual ER process, and do not count the initial distribution of records to processors, nor the gathering of the result after termination.
- Storage: We measure the maximum number of records in the R'_i set of each processor during the execution, and sum across all processors to get the overall storage cost.

2.5.2 No Domain Knowledge

We start by studying the performance of the schemes we presented in Section 2.4.1, which do not assume any form of domain knowledge. We ran D-Swoosh on a 10,000 record dataset, generated as described in Section 2.5.1. All the schemes compute the same ER result, which contains 9,715 records.

This number indicates that relatively few record matches were found—a fact we attribute to the random sampling, which can greatly reduce the percentage of duplicate records in a set. We compared the Full Replication, Majority, and Grid schemes, while varying the number of processors from 1 to 15.

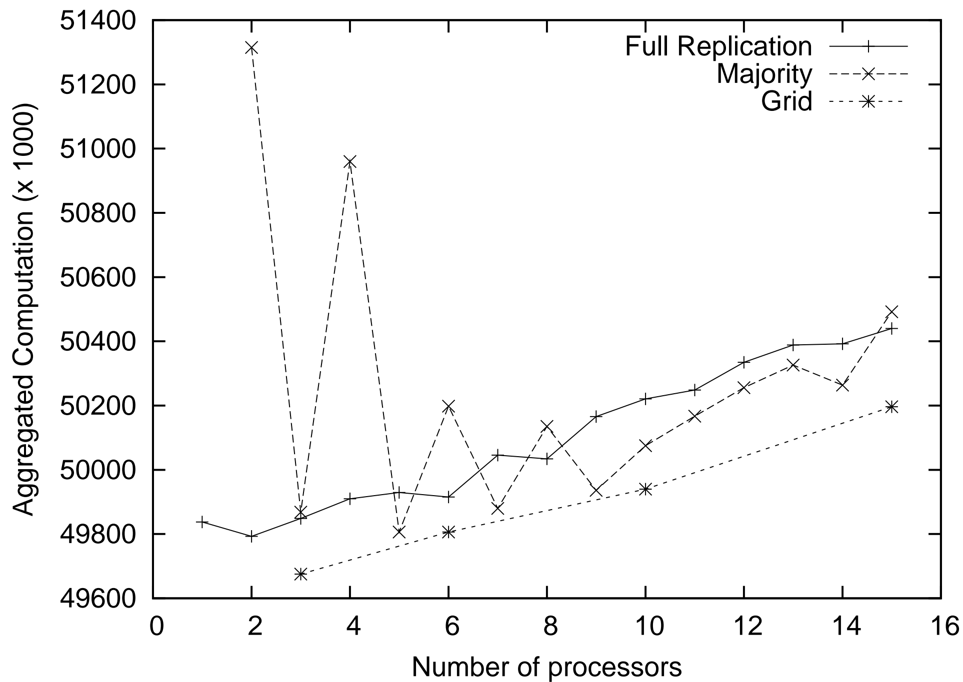

Figure 2.5: Aggregated computation cost for strategies without domain knowledge

Figure 2.5 gives the aggregated computation as a function of the number of processors, for each of the schemes. Note that the (approximately) 50 million comparisons observed for one processor corresponds to the number of comparisons performed by the sequential R-Swoosh algorithm. A few of the schemes perform slightly fewer comparisons. This fact is surprising, considering that the sequential algorithm is optimal. However, the optimality of the sequential algorithm refers to the expected number of computations when all orderings of the input are considered, not on one given ordering. Distributing the records across different processors can result in a more "lucky" ordering that reduces the number of comparisons.

The vertical scale in Figure 2.5 starts at 49.6 million comparisons, so the good news is that the extra work as we distribute the load is not excessive. In the worst case we perform roughly 2% extra comparisons. As the number of processors grows, each of the schemes starts performing redundant comparisons, even though each unique comparison is the responsibility of one and only one processor. Intuitively, the sequential R-Swoosh algorithm is efficient because it is able to perform merges and deletions as early as possible. This benefit is gradually lost as records are spread out randomly across processors, as discussed in Section 2.4.3. In the figure we see that the different distribution schemes generate similar workloads, and that surprisingly the Full Replication scheme does very well (with a higher storage cost, see below).

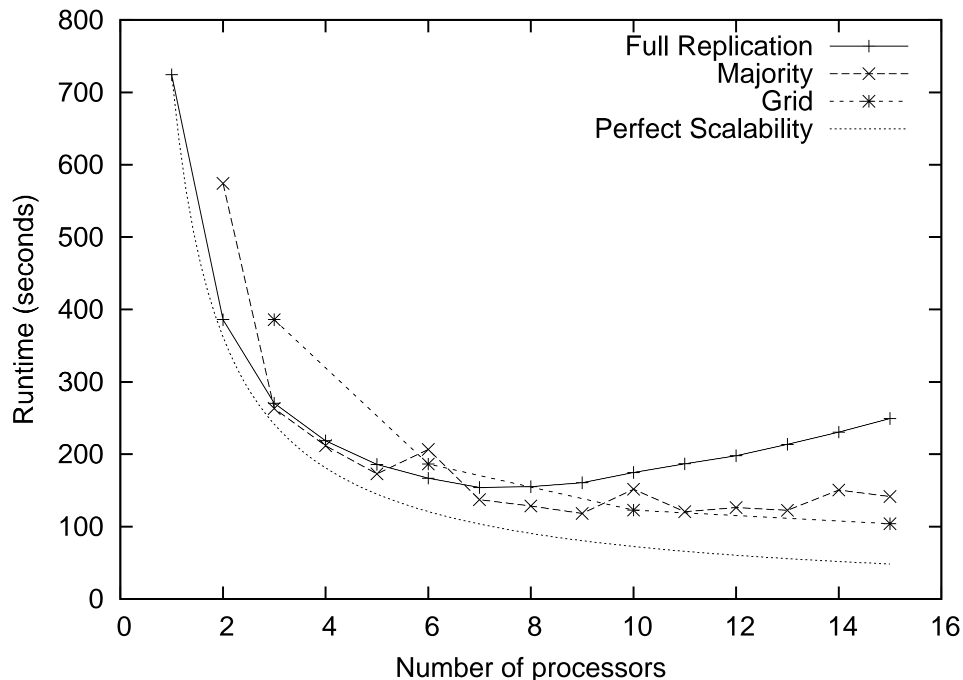

Figure 2.6: Runtime; no domain knowledge

We now consider runtime, which captures the true benefit of parallelism obtained from distributing ER across multiple processors. As shown by Figure 2.6, all schemes benefit from having more processors by reducing the computational cost for each processor, and therefore for the ER computation

overall. The runtime of a single processor goes down from around 700 seconds for the sequential case, to about 150 seconds with 15 processors. The lowest line in the graph suggests how an algorithm would perform if it had perfect scalability. That is, if the runtime of the algorithm were simply the runtime for a single processor divided by the total number of processors. The Majority and Grid schemes both stay quite close to this line up through 15 processors.

The Full Replication scheme shows the most improvement for small numbers of processors, but as expected, the scheme does not scale well to larger numbers of processors, as the increase in network bandwidth begins to have an effect on the runtime. The irregularity of the Majority scheme is due to the fact that the scheme works better for odd numbers of processors than for even numbers. For low numbers of processors, the Grid scheme is slightly less efficient than the two others because the processors on the diagonal of the grid perform less work than the others. This difference evens out as the number of processors increases, hence the runtime of the Grid scheme fares better with larger numbers of processors.

Figure 2.7 gives the storage cost for each scheme as a function of the number of processors. The Full Replication strategy sends every record to every processor, so its storage cost grows linearly. In the Majority scheme, every record is sent to a majority of the processors, so the storage cost also grows linearly, but only stepping up on transitions from odd to even numbers of processors. The Grid scheme performs much better, as the storage cost grows only proportionally to $\sqrt{p}$. It is in fact the only scheme where the storage requirements *per processor* decreases arbitrarily as the number of processors increases.

Figure 2.8 gives the total number of bytes sent during ER, as a function of the number of processors, for each of our schemes. Intuitively, there are two factors that impact communication costs. More record replication implies more communication, as merged records have to be sent to more processors. On the other hand, as we have discussed, replication may cause merges

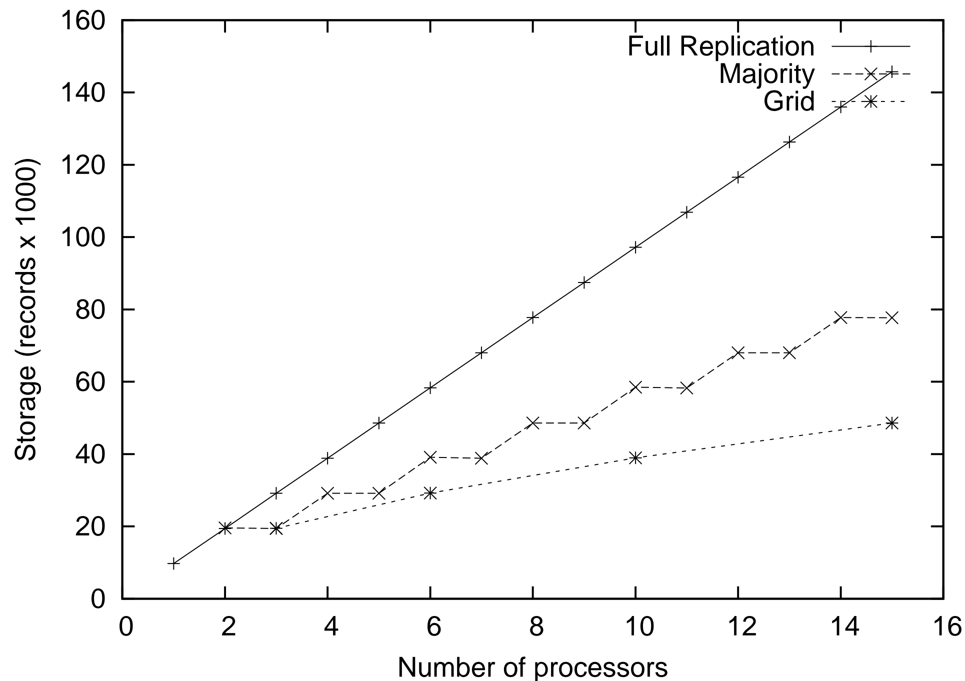

Figure 2.7: Storage cost; no domain knowledge

to occur earlier, which reduces redundant work and unnecessary communications. Our experiments show, however, that this second factor does not play a significant role in the communication cost. The communication cost for the algorithms appears strictly proportional to the amount of record replication.

To summarize the results of this section, we find that parallelism is very effective for increasing the performance of the ER process. Our three strategies all perform quite well, however Majority seems to be the best selection for a small number of processors, and Grid is best for larger number of processors.

2.5.3 With Domain Knowledge

We now consider schemes with domain knowledge, and focus here on Linear Ordering on the price and Value Equality on a product category field. For these experiments, we used the 10,000 record data set created as described

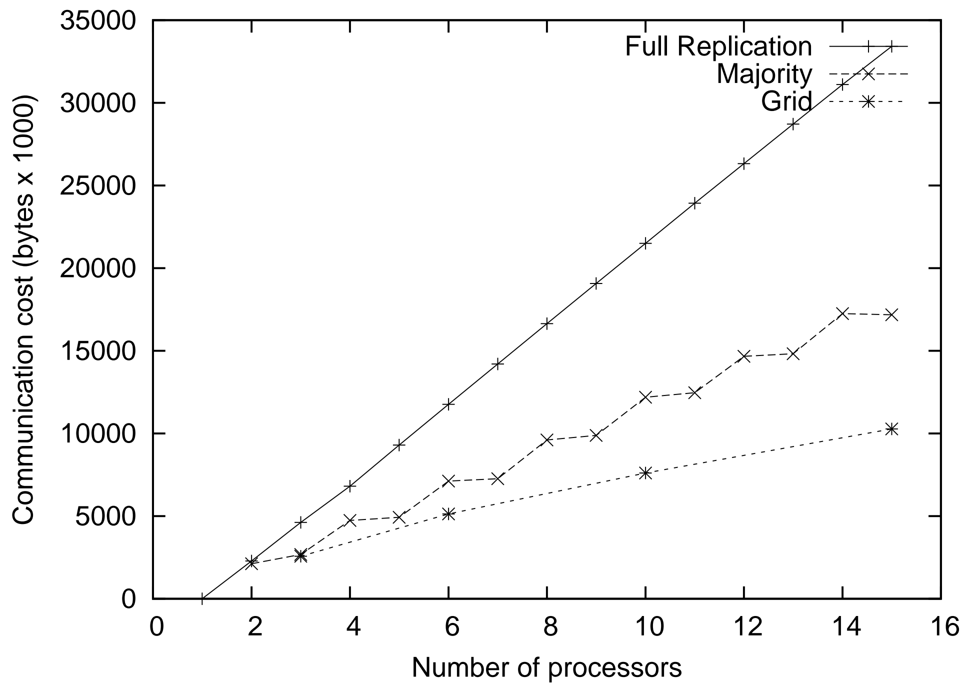

Figure 2.8: Communication cost; no domain knowledge

in Section 2.5.1. These records fall into a total of 48 categories. (A product is only in one category in our input data.)

For the Linear Ordering scheme we partition the total price range into p groups corresponding to intervals of equal size. Since the items vary in price from \$0 to \$200, we settle on a value of \$2 for the window size. For Value Equality, we partition at random (but as evenly as possible) the 48 categories into p groups. Note that some groups may get one more category than another group, and that some processors may get more popular categories. The same is true with Linear Ordering: some price partitions may have many more records than others. Note that these approaches may miss results that would have been found in the schemes that do not exploit semantic knowledge. Indeed, in our experiments, the result sizes for these schemes were generally 10 to 50 records larger due to missed matches.

Figure 2.9 illustrates the communication costs incurred by both strategies, with the Grid line included for comparison. Because the category of records

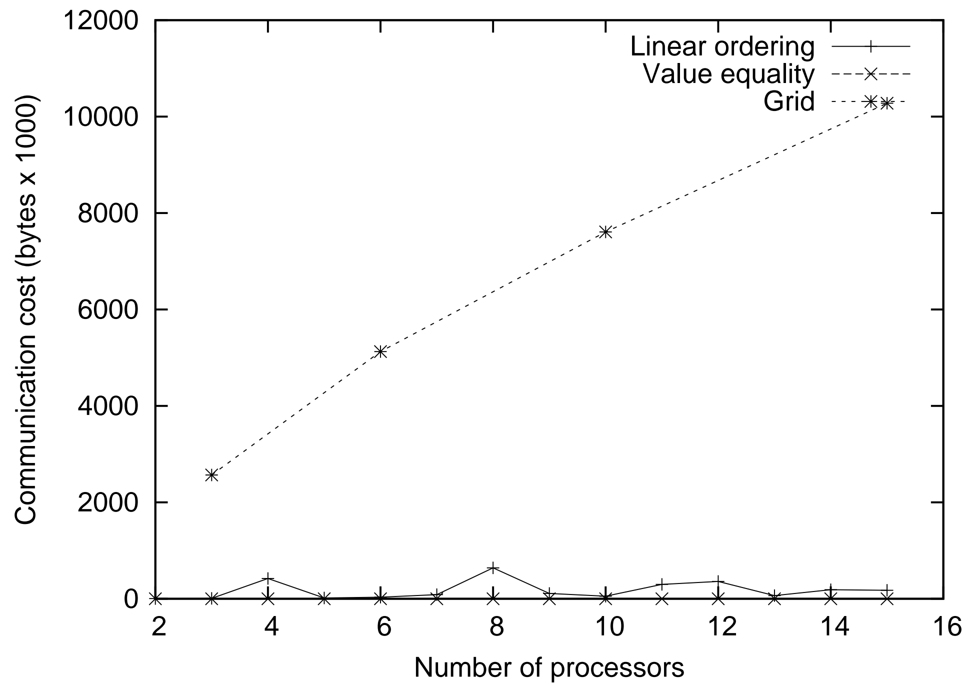

Figure 2.9: Communication cost for domain knowledge schemes

is preserved through merges, groups remain disjoint, and the Value Equality scheme has no communication cost. Linear ordering does result in some communication, but it is relatively small and constant up through 15 processors. We would expect an increase in communication as the number of processors goes higher, since the constant window size becomes larger relative to the shrinking bucket size. However, this effect appears to be negligible in our experiments.

In Figure 2.10, we see the aggregated computation cost for our two strategies, as compared with the Grid scheme. The aggregated computation for the Value Equality scheme is essentially constant with respect to the number of processors, as the scheme simply spreads buckets across the extra processors, without requiring more comparisons. In the Linear Ordering scheme, on the other hand, increasing the number of processors reduces the size of each bucket, so the number of comparisons decreases as the number of processors increases. This graph shows some promise for the domain knowledge schemes,

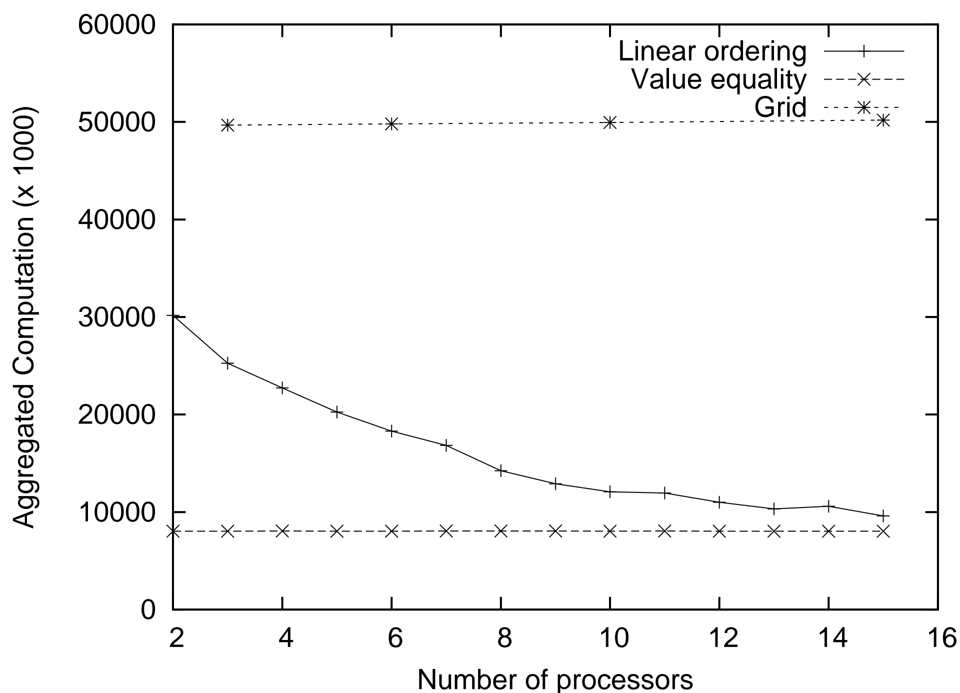

Figure 2.10: Aggregated computation cost for domain knowledge schemes

as their aggregated computation is always lower than that of the Grid scheme.

Figure 2.11 gives the runtime for both strategies as a function of the number of processors. Again, we included the Grid line from Figure 2.6 for comparison. The results shown are highly surprising. Not only do the two algorithms poorly utilize the extra processors, but they also fail to perform better than the Grid scheme, which makes no use of domain knowledge. Ironically, making use of extra information seems to have hurt, rather than helped, our performance.

The main reason for this behavior is that processing load is not evenly distributed: although there were 48 categories, one of those categories contained over one third of the records in the data set. In our experiments with Value Equality, all of the processors consistently ended up waiting for the processor with the huge category to complete. For the Linear Ordering scheme as well, there is a high concentration of products in some small price ranges, so extra processors are not as efficiently used as they are in the Grid scheme.

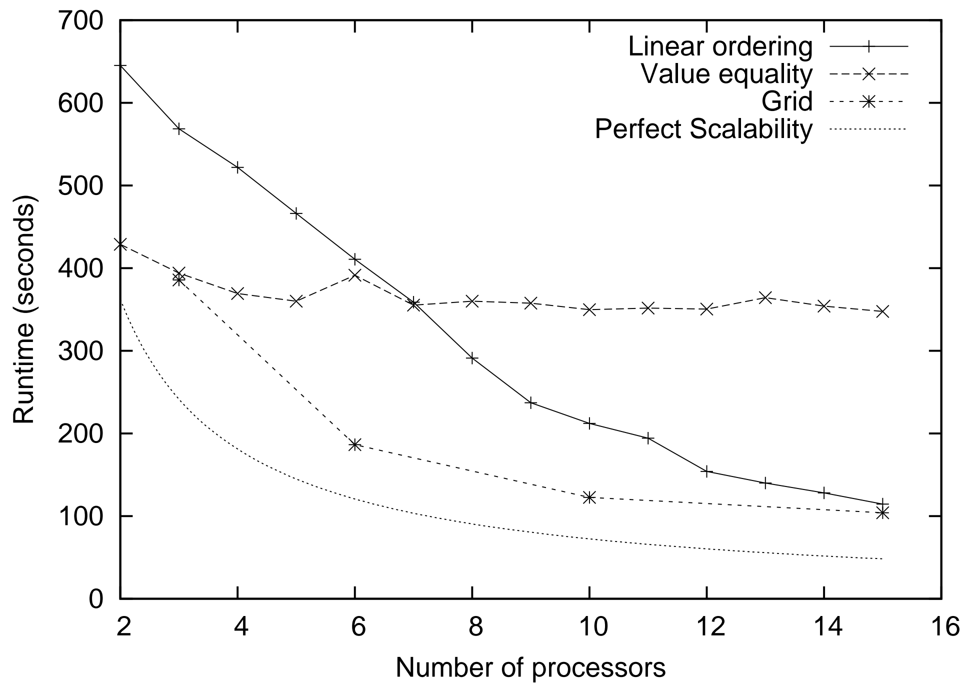

Figure 2.11: Runtime for domain knowledge schemes

Figure 2.12 shows the storage costs for both strategies, again as a function of the number of processors. The storage cost for the Grid scheme is much greater than that of these two schemes, so we have omitted the Grid scheme in the graph to allow the details of the other two lines to be shown. The storage cost of the Value Equality scheme is essentially constant, as groups do not overlap. For the Linear Ordering, it grows roughly linearly with the number of processors. The growth is not exactly linear because the amount of extra replication depends on whether the partition boundaries happen to fall in an area densely populated with records.

Our results show that fully exploiting domain knowledge may be tricky. One solution may be to adjust the partitions: if value distributions are stable, and we know the number of processors in advance, we may analyze the data and determine good semantic partitions that even out the load. Another possibility is to use a hybrid scheme, e.g., using the Grid scheme to distribute the work of heavily loaded processors onto multiple processors. But if runtime is the

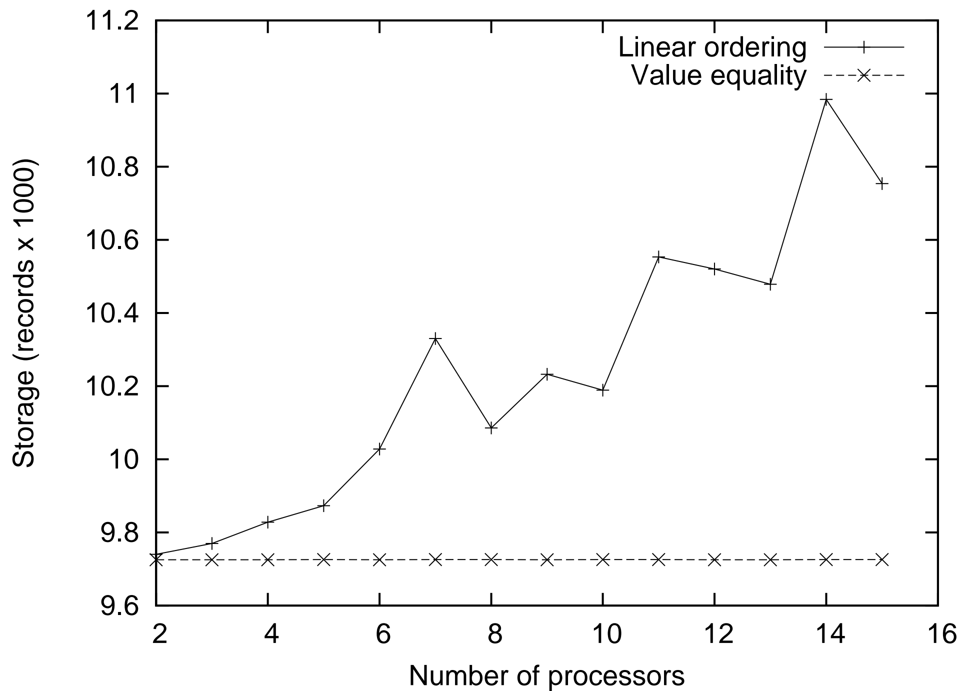

Figure 2.12: Storage for domain knowledge schemes

critical metric, it may be simpler to just use the “dumb” Grid scheme, which does extremely well.

2.5.4 Scalability

To assess the scalability of our algorithms, we compared performance on input datasets of different sizes. We started by generating a 138,000-record set as described in Section 2.5.1. We then generated 5,000, 10,000, 20,000, 40,000, and 80,000-record sets by randomly removing records from the first set. Thus, roughly the same *fraction* of records appears in each category (and price range) in all four datasets

In Figure 2.13 we show the runtime as a function of the size of the initial dataset. The curve labeled “sequential” is for one processor; the other curves are for a 10-processor system using Grid, Linear Ordering and Value Equality schemes. We stopped testing at the 40,000-record point to save time, but

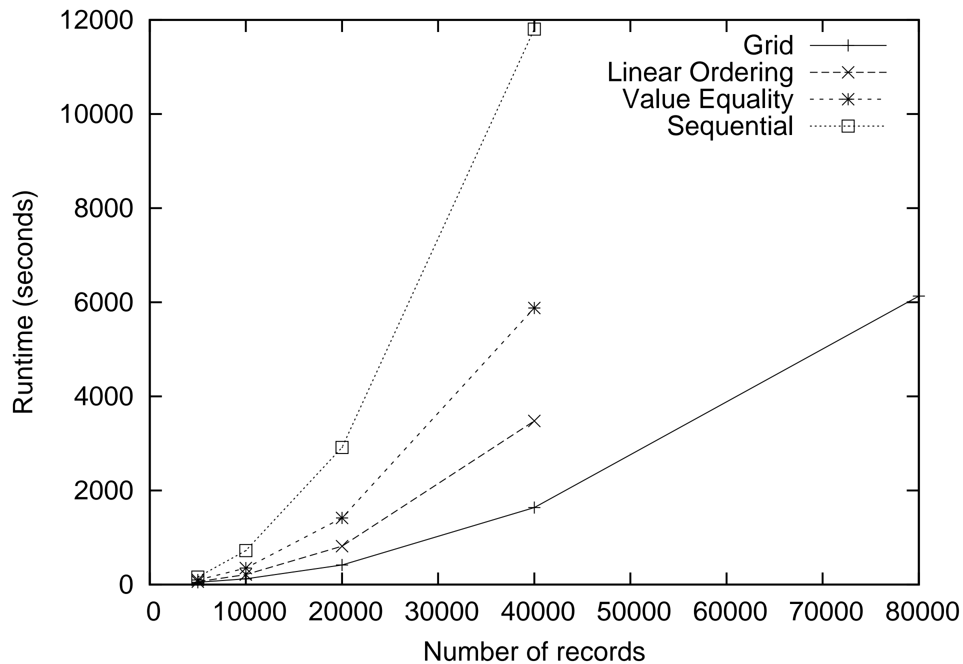

Figure 2.13: Runtime evolution with dataset size (10 processors)

the Grid scheme was fast enough to do a final test with 80,000 records. For all algorithms, the cost grows quadratically with the number of records, and this characteristic is inherent to ER. However, this evolution is much slower for parallel strategies than for the sequential one, and the gain from using parallelism increases with the size of the dataset. Here, for 20,000 records, the Grid scheme beats the sequential scheme by more than a factor of 6.

As mentioned in Section 2.5.2, random sampling can greatly reduce the percentage of duplicate records in a dataset. Therefore, there are more merges in the larger datasets. Merges reduce the number of comparisons required to complete processing, so our algorithms will run faster with a greater number of merges. If our datasets had a consistent percentage of duplicate records, we would therefore expect the curves in this graph to be steeper.

Finally, note that the *relative* ordering of the schemes in Figure 2.13 remains fixed. This fact suggests that our conclusions regarding the strengths and weaknesses of the schemes will hold for datasets larger than what we

were able to consider in our study.

2.6 Related Work

Beyond the work on entity resolution in general laid out in Section 1.5, we identify the following areas of related work.

"Blocking" techniques use domain knowledge to prune the space of comparisons when performing ER. Canopies [62] construct overlapping subsets of the records similarly to our value equality scheme, and the sorted neighborhoods of [46] is very similar to our linear ordering scheme. An overview of such blocking methods can be found in [9]. With the exception of [46], these approaches assume a single processor. [46]'s "band join" parallelizes linear ordering like we do, but does not generalize to other forms of domain knowledge, and does not consider merging records.

In the distributed computing literature, we mentioned the connection of our "coverage condition", which guarantees the correctness of D-Swoosh to the distributed mutual exclusion problem [32], and schemes based on coteries [39]. Our majority scheme is based on the well-known principle of a quorum [82]. The grid scheme is a variant of Maekawa's construction in [58]. An important difference is that coteries designed for mutual exclusion try to maximize the number of operating nodes [71] or their availability [50]. By contrast, the coteries used in our strategies distribute work across processors, and are optimal when they minimize computation and communication.

Recently, Bilenko et al. considered the ER problem on a comparison shopping dataset from Google [17]. Their focus is on continuously learning the appropriate match function for records, and is therefore complementary to our work, which focuses on executing matches and merges efficiently.

2.7 Conclusion

We presented several schemes for distributed ER, both exploiting and not exploiting domain knowledge. Among the schemes that do not exploit domain knowledge, we found that for few processors, the full replication scheme works quite well, but for true scalability for large numbers of processors, the grid scheme scales the best. Surprisingly, we discovered that simply minimizing the number of record replicas is not enough: schemes that make more copies may do better because they speed up the discovery of matching records. We also discovered that exploiting domain knowledge is tricky, as it requires a good distribution of records across semantic groups, and may involve a trade-off between good performance and lowered recall (missed answer records).

Chapter 3

Entity Resolution with Iterative Blocking

3.1 Introduction

In Chapter 1 we mentioned that there are two ways to improve the performance of entity resolution. The first, parallelism, was explored in Chapter 2. Parallelizing the process of entity resolution has the benefit of reducing the runtime without affecting the ER result. But parallelism comes with a cost in dollars—purchasing new hardware or compute time on a large cluster can be very expensive. Further, any parallel algorithm has its scalability limits, and at some point, the improvement in runtime by adding new processors is not worth the money spent on those processors. Therefore, we must turn to the second method of improving performance: weakening correctness.

As discussed in Chapter 1, correctness is often defined as the result that would be obtained by a human expert. For most entity resolution applications, however, users accept the fact that even the best algorithms will give different results from those that a human would generate. Since there is no expectation of 100% accuracy in a computer-generated entity resolution result, it is reasonable to sacrifice some degree of correctness if we can significantly speed up the process of entity resolution. In this chapter we will consider the

Record	Name	Address(zip)	Email
r	John Doe	02139	jdoe@yahoo.com
s	John Doe	94305	
t	John Foe	94305	jdoe@yahoo.com
u	Bobbie Brown	12345	bob@google.com
v	Bob Brown	12345	bob@google.com

Figure 3.1: Customer records

technique of "blocking" to speed up the processing of entity resolution.

We first explored the concept of blocking in Section 2.4.2. The idea of blocking is to put the records into buckets (or "blocks") so that any two records that match are likely to be placed into the same block. By performing entity resolution only within blocks, we can greatly reduce the processing time.

Consider the example data in Figure 3.1 consisting of five records referring to two distinct real world entities. Let us assume we have a match function that detects a match between the following pairs of records: u, v; r, s; r, t; and s, t.

To use blocking on this data, we might decide to have one block per unique ZIP code. This would result in three blocks: one block with r, another with s, t, and a third block with u, v. We would then perform ER on the blocks individually. The first block contains only one record, so no comparisons need be performed. ER on the second block will discover that records s and t match, and ER on the third block will find the match between u and v. The final result of ER with blocking by ZIP code will then be $\{\langle r\rangle, \langle s, t\rangle, \langle u, v\rangle\}$. We note that record r has erroneously been found not to match with any other record. However, the result was generated quickly, requiring only two comparisons.

To solve problem of missing matches due to blocking, it is common to use multiple blocking criteria. If, for example, we ran a second pass on the same data above, blocking by first letter of last name, then we would obtain three new blocks: one containing r, s, another containing t alone, and a third block containing u, v. This second pass would indeed find a match between records r and s. We would also rediscover the match between records u and v. In this

case, by combining the results of the two runs, we can arrive at the correct solution. In general, however, there is no guarantee of 100% accuracy when blocking techniques are used.

This simple approach to using multiple blocking criteria is effective, but it can be improved. First, note that records r and s refer to the same entity, but they only share a single common attribute (Name). A different match function than the one we have been using might declare a match only if there is high similarity on at least two fields. This new match function would not identify r and s as matching records unless s were first merged with t. In this case, neither of the two blocking runs would detect the match between these two records. Only if the results from one blocking run are carried over to the second run would this match be found.

A second issue with this simple approach is that redundant comparisons are performed. In both blocking runs, the records u and v are compared and found to match. Again, if the results from the first run were carried over to the second, this comparison could be avoided.

Therefore, in this chapter, we consider the application of an iterative model to multiple blocking algorithms. Rather than processing each block as an independent operation, we will carry the results from previously processed blocks into the new blocks. Rather than processing each block once, we will iterate over the set of blocks multiple times, each time carrying forward new information in the hopes of discovering new matches. As we shall see, iterative blocking can improve the accuracy of the ER result while simultaneously speeding up the processing by eliminating redundant comparisons.

The contributions of this chapter are the following:

- We describe our framework for using blocking in the processing of entity resolution (Sections 3.2 and 3.3).
- We present in-memory entity resolution algorithms using both iterative and non-iterative multiple blocking (Sections 3.4 and 3.5).
- We present Duplo, an efficient implementation of on-disk iterative blocking for large datasets that cannot fit in main memory (Section 3.6).

- We perform experiments comparing both in-memory and on-disk iterative blocking algorithms with their non-iterative counterparts, demonstrating that iterative blocking allows improvement on both accuracy and runtime performance simultaneously (Section 3.7).

Section 3.8 covers related work and in Section 3.9 we describe our conclusions.

The work in this chapter has been published by Whang, Menestrina, and Garcia-Molina [87] in SIGMOD 2009. This chapter presents a simplification of the work presented there, focusing more heavily on the Duplo on-disk algorithm that was primarily the work of the author of this thesis. The experiments are primarily the work of Whang, though we present them here to justify both the iterative blocking approach and the implementation of Duplo.

3.2 ER Model

In this chapter, we use the model of the entity resolution problem defined in Section 1.2. Briefly, the output of an entity resolution algorithm is a partition of the input records.

Since we may want to run ER on the output of a previous resolution (as we shall see in Section 3.5), we generalize our model slightly so that the input itself may be a partition. We now say that both the ER input and output are partitions of the base records. We assume that ER never undoes the groupings performed earlier. In our example, we then view the initial input as the partition $\{\langle r\rangle, \langle s\rangle, \langle t\rangle, \langle u\rangle, \langle v\rangle\}$. If we run a second ER on the first output $\{\langle r,s,t\rangle, \langle u,v\rangle\}$, we may obtain the partition $\{\langle r,s,t,u,v\rangle\}$. In this case, $\langle r,s,t\rangle$ contains enough information to combine with $\langle u,v\rangle$.

Since the clusters in an input partition are analogous to records, we may refer to clusters of records as either clusters or records, interchangeably. Thus, cluster $\langle r,s,t\rangle$ is an input record to the second ER above. For simplicity, we omit the cluster brackets for singleton clusters. For example, we write

$\{\langle r, s, t\rangle, \langle u\rangle\}$ as $\{\langle r, s, t\rangle, u\}$.

Any ER algorithm that fits in this model can be plugged into the blocking algorithms discussed in this chapter. To distinguish the algorithm that performs the ER within blocks from the higher level blocking algorithms, we will refer to the within-block algorithm as the "core ER algorithm", or CER for short.

3.3 Blocking Model

Our algorithms make use of a *set of blocks* B. The number of blocks is determined at runtime and can change dynamically. Each block is associated with a unique key. Keys might be, for example, ZIP codes, first letters of last names, or hash values. Blocking schemes may be "heterogeneous" in that multiple different types of keys are associated with blocks, e.g., integers for ZIP codes and characters for letters extracted from names.

We apply functions to records to determine to which blocks they are assigned. We call such a function a *single blocking criterion*. Given a base record r, a single blocking criterion SC(r) yields a set K of one or more keys. Record r is assigned to each block associated with a key $k \in K$; if there is no block yet with key k, one is created. We will use the notation B[k] to refer to the block associated with key k. An SC function is only defined on base records, but we extend the notation such that when SC is invoked on a cluster c, it returns the keys from all of the base records in c. Formally, for a cluster $c = \langle r_1, r_2, \ldots, r_m\rangle$, we define $SC(c) = \bigcup_i SC(r_i)$.

As discussed in Section 3.1, our goal is to make use of multiple blocking criteria to increase the accuracy of the ER result. All of our algorithms, therefore, will take in a set of single blocking criteria. We call this set a *multiple blocking criterion* and refer to it as MC. The elements of the MC set are the single blocking criteria SC_1, SC_2, ..., SC_N. Now each single blocking criterion SC_j generates its own set of blocks, which we will refer to as B_j.

Therefore, $B_j[k]$ refers to the block of records from criterion SC_j associated with key k.

For convenience, we will sometimes treat MC as a function that returns the set of all keys returned by the underlying single blocking criteria. Formally, let $MC(r) = \bigcup_j SC_j(r)$ for any record or cluster r. Note that, while a single blocking criterion typically produces a specific type of key value (e.g., ZIP codes), the set returned by MC is a union of the keys from different criteria and may therefore be heterogeneous.

3.4 Simple Blocking

In the introductory example, we described a simple method of using (non-iterative) blocking. We now provide a full description of this method.

For a single blocking criterion, we use the criterion to divide the base records into blocks. We then invoke CER on each block, adding the results to the output set. We repeat this process for each single blocking criterion in our multiple blocking criterion. The pseudocode for this method is shown in Algorithm 3.1.

We note that the result of Algorithm 3.1 is not necessarily a partition of the base records. Returning to our introductory example, we note that the result of this algorithm will have both clusters $\langle r, s\rangle$ and $\langle s, t\rangle$. To properly convert this result to a partition of base records, we should merge overlapping clusters together (in the example, merge $\langle r, s\rangle$ and $\langle s, t\rangle$ to $\langle r, s, t\rangle$). This merging is called the "connected-component" operation, and we will refer to Blocking followed by the connected-component operation as Blocking-CC.

We also point out that the Blocking algorithm is easy to adapt to the case where the entire set of base records does not fit in main memory at once. In this case, the B_i arrays may be written to disk and then, assuming blocks are small enough to fit in memory, each block may be read in one at a time as CER is invoked on it.

```
Input:    R: the set of base records
          MC: the multiple blocking criterion
          CER: the core entity resolution algorithm
Output:   a set of clusters of base records
 1: output ← ∅
 2: for all SC_i ∈ MC do
 3:    initialize B_i, an array of empty blocks
 4:    // Distribute the records into the blocks
 5:    for all r ∈ R do
 6:       for all blockid ∈ SC_i(r) do
 7:          add r to B_i[blockid]
 8:       end for
 9:    end for
10:    // run CER on each block
11:    for all block ∈ B_i do
12:       output ← output ∪ CER(block)
13:    end for
14: end for
```

Algorithm 3.1: Simple Blocking

3.5 Iterative Blocking

We now proceed to describe a basic algorithm for iterative blocking. Algorithm 3.2 shows the pseudocode for this algorithm, which we will walk through.

In iterative blocking, we begin—as in simple blocking—by assigning the records to blocks (Lines 1–9). We then enter a loop that processes blocks with CER (Line 15) and then updates all blocks with the newly discovered matches (Lines 18–29). The loop completes when no new clusters have been found (Line 33) and the union of all blocks is returned (Line 34).

To illustrate this algorithm, we will step through the processing of the data in our introductory example from Figure 3.1.

The first step is to distribute all the base records into blocks as shown in Figure 3.2. Each single blocking criterion SC yields three blocks on the input data, and we have arbitrarily assigned these blocks with the numbers 1, 2, and 3. For example, the records s and t are located together in $B_1[2]$, which is the

```
Input:   R: the set of base records
         MC: the multiple blocking criterion
         CER: the core entity resolution algorithm
Output:  a partition of the base records
 1: // Assign records to blocks
 2: for all SCi ∈ MC do
 3:   initialize Bi, an array of empty blocks
 4:   for all r ∈ R do
 5:     for all blockid ∈ SCi(r) do
 6:       add r to Bi[blockid]
 7:     end for
 8:   end for
 9: end for
10: // Process blocks until no new clusters formed
11: repeat
12:   morework ← false
13:   for all SCi ∈ MC do
14:     for all block ∈ Bi do
15:       result ← CER(b)
16:       if result ≠ block then
17:         morework ← true
18:         // Update blocks with new information
19:         for all c ∈ (result – block) ∪ (block – result) do
20:           for all SCj ∈ MC do
21:             for all blockid ∈ SCj(c) do
22:               if c ∈ result then
23:                 add c to Bj[blockid]
24:               else
25:                 remove c from Bj[blockid]
26:               end if
27:             end for
28:           end for
29:         end for
30:       end if
31:     end for
32:   end for
33: until morework = false
34: return ⋃i,j Bi[j]
```

Algorithm 3.2: Iterative Blocking

second block of the first blocking criterion SC_1.

Criterion	$B_i[1]$	$B_i[2]$	$B_i[3]$
SC_1	r	s, t	u, v
SC_2	r, s	t	u, v

Figure 3.2: After initial distribution

Next, we start processing all the blocks. Suppose that we repeatedly process blocks in the order of increasing subscript order (i.e., $B_1[1]$,$B_1[2]$,...,$B_2[3]$). The first block that has matching records is $B_1[3]$ where the CER algorithm (Line 12) will combine u and v into the cluster $\langle u, v\rangle$. Recall that a blocking criterion assigns each cluster to the same blocks as the base records it comprises. Therefore, the $\langle u, v\rangle$ cluster is distributed to all the blocks containing either u or v and is thus assigned to $B_2[3]$ (and $B_1[3]$, of course). Line 25 removes the base records u and v from the very same blocks.

We next process (Line 12) $B_2[1]$ which contains the matching records r and s. This time, we distribute the resulting cluster $\langle r, s\rangle$ to $B_1[1]$, $B_1[2]$, and $B_2[1]$, and then remove r and s. The next block $B_2[2]$ does not produce any record matches. Then CER is run on $B_2[3]$, which now only contains the cluster $\langle u, v\rangle$, so CER does not generate any record matches. The intermediate result after this first iteration is shown in Figure 3.3.

Criterion	$B_i[1]$	$B_i[2]$	$B_i[3]$
SC_1	$\langle r, s\rangle$	t, $\langle r, s\rangle$	$\langle u, v\rangle$
SC_2	$\langle r, s\rangle$	t	$\langle u, v\rangle$

Figure 3.3: Intermediate result after first iteration

We now repeat the entire loop again. Block $B_1[1]$ does not generate any new records after the CER algorithm is applied. Block $B_1[2]$, however, generates the new record $\langle r, s, t\rangle$ by merging $\langle r, s\rangle$ and t. Cluster $\langle r, s, t\rangle$ is then distributed to $B_1[1]$, $B_2[1]$, and $B_2[2]$, while cluster $\langle r, s\rangle$ and record t are removed from those blocks. Subsequent blocks $B_1[3]$, $B_2[1]$, $B_{2,2}$, and $B_2[3]$ do not generate record merges. Hence, the intermediate result after the second iteration is shown in Figure 3.4.

Criterion	$B_i[1]$	$B_i[2]$	$B_i[3]$
SC_1	$\langle r, s, t \rangle$	$\langle r, s, t \rangle$	$\langle u, v \rangle$
SC_2	$\langle r, s, t \rangle$	$\langle r, s, t \rangle$	$\langle u, v \rangle$

Figure 3.4: Intermediate result after second iteration

As we can see, the second iteration has fully resolved all blocks. However, new clusters were created during the second iteration, so the algorithm is not yet complete. One more pass must be done on each block to see if new merges are found. None of the six blocks will produce new merges, and then the loop will terminate. We then union the records of any blocking criterion (SC_1 or SC_2) to get the final answer $\{\langle r, s, t \rangle, \langle u, v \rangle\}$. In total, we have processed 18 blocks (6 for each of the three iterations) to derive the final solution.

Since the iterative blocking algorithm automatically replaces records with the largest cluster found that contains them, the clusters in the final result can only overlap if they are equal. That is, if one block contains $\langle r, s \rangle$ at the termination of the algorithm, then no block will contain the overlapping record $\langle s, t \rangle$. Therefore, we do not need to perform a connected-component operation on the clusters in the result.

3.5.1 Lego Algorithm

Algorithm 3.2 is a simple implementation of iterative blocking, but is not too efficient. Consider that in the final iteration, block $B_1[2]$ has not received any new records since it was last processed. In fact, the same is true of all of the blocks other than $B_1[1]$. Therefore, only $B_1[1]$ needs to be reprocessed. This is one of the techniques used in the Lego algorithm, described by Whang et al. [87] to improve the performance of iterative blocking. We do not present the Lego algorithm here, but we will use it in our experiments to justify the iterative blocking framework.

3.6 On-Disk Iterative Blocking

When the input data set is too large to fit in main memory, we can only perform entity resolutions on subsets of the input data at a time. Simple blocking matches perfectly with on-disk processing for this reason. Iterative blocking, however, requires the careful optimization of I/O costs.

There are two main types of I/O costs involved in on-disk iterative blocking. First, blocks must be shuttled from disk to memory and back, which clearly involves I/O. To make this cost as efficient as possible, we introduce the concept of a "segment" that incorporates multiple blocks. We discuss segments in the next subsection.

The second type of I/O comes from the application of ER results from one block to subsequent blocks. Updating the blocks on disk as new merges are found would involve slow random I/Os. In addition these I/Os could be unnecessary, as updating a block may involve rewriting unaffected portions of the block. To manage this type of I/O, we make use of a "merge log" that stores every merge performed sequentially on disk.

After introducing these two building blocks, we will describe the actual Duplo algorithm for on-disk iterative blocking.

3.6.1 Segments

In order to maximize our use of main memory and ensure that blocks are read from disk with long stretches of sequential I/O, we use the concept of a "segment". A segment is a group of blocks stored together on disk. In the Duplo algorithm, we transfer a block between memory and disk only as part of an entire segment. For notation, segment $s_{i,j}$ refers to the jth segment for criterion SC_i. An example of segmentation is shown in Figure 3.5. The column labeled $s_{-,1}$ contains the first segment for each blocking criterion, and the column labeled $s_{-,2}$ contains the second segment for each blocking criterion. In this example, blocks $B_1[1]$ and $B_1[3]$ have been assigned to the same segment

$s_{1,1}$.

Criterion	$s_{-,1}$	$s_{-,2}$
SC_1	$B_1[1], B_1[3]$	$B_1[2]$
SC_2	$B_2[1]$	$B_2[2], B_2[3]$

Figure 3.5: Assigning blocks to segments

The allocation of records into segments is performed as follows. Segments are of fixed size S_S bytes, and we select S_S based on the amount of main memory available to the algorithm for the storage of records. For each single blocking criterion SC_i, we allocate a fixed number of segments on disk. The number of segments N_S is chosen by considering the size of the data S_D. Clearly, $N_S \geq S_D/S_S$. Since the contents of each segment may not be exactly equal in size, segments could overflow if N_S were chosen too close to S_D/S_S. We do not discuss the handling of overflow here. We assume that the probability of overflowing a segment is negligible.

To summarize the allocation so far, each single blocking criterion SC_i is allocated N_S segments of size S_S each. Now, for each record r we compute $SC_i(r)$ to determine the blocks it belongs in. We use simple hashing to determine which segment each block belongs in, and we write r into each of the designated segments. Note that the records are written into segments as they are considered, and therefore the records within a segment may not be segregated by block. When Duplo reads a segment into memory, one of its initial tasks is to separate the records into their respective blocks.

A greedy method of reducing disk I/Os is to ensure that all blocks in a segment are processed as completely as possible before they are written back to disk. Suppose that blocks $b_1 = \{r, s\}$ and $b_2 = \{s, t\}$ are both in the same segment. If we process b_1 first, we may find $\langle r, s\rangle$ and carry it over to b_2 to find $\langle r, s, t\rangle$. The discovery of $\langle r, s, t\rangle$ requires b_1 to be reprocessed, and an in-memory algorithm such as Lego would simply add b_1 to the end of a queue of blocks to be processed. For an on-disk algorithm, however, delaying the reprocessing of b_1 until later will force us to incur more I/O costs to read the

segment back into memory later. Therefore, in this example, Duplo will continue iteratively processing the blocks in memory until both contain $\langle r, s, t \rangle$. At this point, Duplo will write the segment back to disk and load another segment into memory for processing.

3.6.2 Merge Log

As discussed earlier, it is impractical for a disk-based algorithm to immediately update data in blocks on disk as new pairs of matching clusters are found. We instead use an on-disk data structure called a merge log to keep track of the cluster merges performed by the algorithm. Whenever we read a segment into memory, we "preprocess" all of the records in that segment by performing a sequential scan of the merge log. By scanning through the merge log, we can update each cluster c in memory to the most recent merged cluster that contains all the records in c.

The structure of the merge log is a simple sequence of "update" operations. When clusters c_1 and c_2 are merged into cluster c_3, we write $c_1 \rightarrow c_3$, $c_2 \rightarrow c_3$ into the merge log. We write the full records into the log, rather than simple record ids. If c_3 later merges with c_4 to make c_5, then we write $c_3 \rightarrow c_5$, $c_4 \rightarrow c_5$ into the merge log. Note that at no point do we update previously written merge log entries. For example, although c_1 eventually merges into c_5, we do not change the initial entry in the log to $c_1 \rightarrow c_5$. The contents of the merge log after these two merges would simple be the sequence: $c_1 \rightarrow c_3$, $c_2 \rightarrow c_3$, $c_3 \rightarrow c_5$, $c_4 \rightarrow c_5$.

Continuing our example, suppose that we read in a segment with a block that contains c_2. By scanning through the merge log, we would find $c_2 \rightarrow c_3$ and replace c_2 with c_3 in the block in memory. We would then continue and find $c_3 \rightarrow c_5$ and replace c_3 with c_5, thus bringing the block to be processed completely up to date with the latest information.

The use of the merge log is very I/O efficient due to the use of sequential I/O. Although entries may be written into the log in small chunks, the fact

that they are written sequentially allows us to convert the writes into larger I/Os by using a buffer.

Since the merge log only grows as processing continues, reading the entire merge log for every segment processed would get more and more expensive. To combat this, we make use of a form of timestamps. The Duplo algorithm maintains a counter in memory of the number of segments read in to memory. Whenever a segment has completed processing and all of the resulting merge log entries have been written to disk, the counter is updated and the current disk location of the end of the log is added to a simple in-memory index. The index contains a map from counter values to disk locations. Whenever a processed segment is written back to disk, we include the current value of the counter as a timestamp. In this way, when the segment is read off the disk to be processed again, we can use the timestamp to skip over the parts of the merge log that this segment has already seen.

We note another potential optimization from the use of the merge log. Suppose that in processing a segment containing a large number of clusters, we only find a few matches, taking up only a few kilobytes in the merge log. Rather than writing out the updated segment to disk, it may make sense to take a "lazy" strategy. We can simply discard the segment's data from memory. When the time comes to process that segment again, we will read the old version of segment off of disk and the algorithm will automatically read from the merge log at the proper location to bring the segment completely up to date. The version of the Duplo algorithm we use in this chapter does not include this optimization, but it could be a useful improvement.

3.6.3 The Duplo Algorithm

The Duplo algorithm (Algorithm 3.3) uses segments and the merge log to scale iterative blocking. The idea of Duplo is to process "N blocks at a time", assuming a segment contains N blocks on average. Hence, the iterative blocking is now done in two levels: processing a segment and then processing the blocks

within that segment.

We maintain two queues and two merge logs for managing the segments and the blocks within each segment. In order to process segments, we use a *segment queue* Q_1 that determines which segment to process next and a "global" merge log L_1 that keeps track of all the record merges done until now. Next, in order to process the blocks of a single segment, we use a *block queue* Q_2 that determines which block to process next and a "local" merge log L_2 that keeps track of the record merges done within the current segment. Merge log L_1 is located on disk and is accessed once for each segment processed while L_2 is in memory and is accessed once for each block processed.

To illustrate the Duplo algorithm, suppose that we create the segments of Figure 3.5. The actual contents of the segments are shown in Figure 3.6 (i.e., each segment contains a union of records of its blocks). We show how each segment is processed in Figure 3.7.

Criterion	$s_{-,1}$	$s_{-,2}$
SC_1	r, u, v	s, t
SC_2	r, s	t, u, v

Figure 3.6: Initial segments of Duplo

Initially, all four segments are placed in the segment queue Q_1 (i.e., Q_1 ={$s_{1,1}$, $s_{1,2}$, $s_{2,1}$, $s_{2,2}$}). When we first process segment $s_{1,1}$, we divide the records into

Step	**Segment**	**Records before** CER	**Records after** CER	Q_1 **after** CER
1	$s_{1,1}$	r, u, v	r, ⟨u, v⟩	{$s_{1,2}$,$s_{2,1}$,$s_{2,2}$}
2	$s_{1,2}$	s, t	s, t	{$s_{2,1}$,$s_{2,2}$}
3	$s_{2,1}$	r, s	⟨r, s⟩	{$s_{2,2}$, $s_{1,1}$, $s_{1,2}$}
4	$s_{2,2}$	t, ⟨u, v⟩	t, ⟨u, v⟩	{$s_{1,1}$, $s_{1,2}$}
5	$s_{1,1}$	⟨r, s⟩, ⟨u, v⟩	⟨r, s⟩, ⟨u, v⟩	{$s_{1,2}$}
6	$s_{1,2}$	⟨r, s⟩, t	⟨r, s, t⟩	{$s_{1,1}$,$s_{2,1}$,$s_{2,2}$}
7	$s_{1,1}$	⟨r, s, t⟩,⟨u, v⟩	⟨r, s, t⟩,⟨u, v⟩	{$s_{2,1}$,$s_{2,2}$}
8	$s_{2,1}$	⟨r, s, t⟩	⟨r, s, t⟩	{$s_{2,2}$}
9	$s_{2,2}$	⟨r, s, t⟩, ⟨u, v⟩	⟨r, s, t⟩, ⟨u, v⟩	{}

Figure 3.7: Processing segments with the Duplo algorithm

```
Input:    R: the set of base records
          MC: the multiple blocking criterion
          CER: the core entity resolution algorithm
Output:   a partition of the base records
 1: L1 ← ∅ // Disk merge log for segments
 2: Q1 ← ∅ // Segment queue
 3: Create segments
 4: Push all segments into Q1
 5: while Q1 is not empty do
 6:   s ← Q1.Pop()
 7:   update s with new merges from L1
 8:   L2 ← ∅ // In-memory merge log for blocks
 9:   Q2 ← ∅ // Block queue
10:   Push all blocks in s into Q2
11:   while Q2 is not empty do
12:     b ← Q2.Pop()
13:     update b with new merges from L2
14:     b_out ← CER(b)
15:     Add to L1, L2 the new merges in b_out
16:     for all r ∈ b_out − b do
17:       for all b′ ∈ MC(r) do
18:         s′ ← BlockToSegment(b′) // Returns the segment where b′ is
19:         if s = s′ then // Hit the same segment
20:           if b ≠ b′ and b′ ∉ Q2 then
21:             Q2.Push(b′)
22:           end if
23:         else // Hit a different segment
24:           if s′ ∉ Q1 then
25:             Q1.Push(s′)
26:           end if
27:         end if
28:       end for
29:     end for
30:   end while
31:   Write s back to disk
32: end while
33: J ←{Records in R that were never merged}
34: J ← J ∪ {Records in L1 that are not contained by any other record in L1}
35: return J
```

Algorithm 3.3: The Duplo Algorithm

the two blocks $B_1[1]$ ={r} and $B_1[3]$ ={u, v}, and process them with the CER algorithm. Since u and v merge into $\langle u, v\rangle$, the merge logs L_1 and L_2 are both set to $\{u \rightarrow \langle u, v\rangle, v \rightarrow \langle u, v\rangle\}$. Segment $s_{2,2}$ would be affected by $\langle u, v\rangle$, but we do not insert the segment into Q_1 because Q_1 already contains $s_{2,2}$. We then continue with processing segment $s_{1,2}$, which does not have matching records. When we process segment $s_{2,1}$, we extract $B_2[2]$ ={r, s} and merge r and s into $\langle r, s\rangle$. The global merge log L_1 is then updated (by reading and applying the merge log) to $\{u \rightarrow \langle u, v\rangle, v \rightarrow \langle u, v\rangle, r \rightarrow \langle r, s\rangle, s \rightarrow \langle r, s\rangle\}$ while L_2 is updated to $\{r \rightarrow \langle r, s\rangle, s \rightarrow \langle r, s\rangle\}$. (Notice that L_1 and L_2 are now different because L_2 only keeps track of the local merges within the current segment.) Since $\langle r, s\rangle$ affects the segments $s_{1,1}$ and $s_{1,2}$, we push $s_{1,1}$ and $s_{1,2}$ back into Q_1 (i.e., Q_1 =$\{s_{2,2}, s_{1,1}, s_{1,2}\}$). The next segment $s_{2,2}$ is updated to $\{t, \langle u, v\rangle\}$, but does not generate any record matches. Similarly, the next segment $s_{1,1}$ is updated to $\{\langle r, s\rangle, \langle u, v\rangle\}$, but does not generate record matches. We then process $s_{1,2}$. This time, after we update the segment to $\{\langle r, s\rangle, t\}$ and extract the block $B_1[2]$ =$\{\langle r, s\rangle, t\}$, we merge $\langle r, s\rangle$ and t into $\langle r, s, t\rangle$. The global merge log L_1 is updated to $\{u \rightarrow \langle u, v\rangle, v \rightarrow \langle u, v\rangle, r \rightarrow \langle r, s\rangle, s \rightarrow \langle r, s\rangle, \langle r, s\rangle \rightarrow \langle r, s, t\rangle, t \rightarrow \langle r, s, t\rangle\}$ while L_2 is updated to $\{\langle r, s\rangle \rightarrow \langle r, s, t\rangle, t \rightarrow \langle r, s, t\rangle\}$. Since $\langle r, s, t\rangle$ affects all four segments, we insert $s_{1,1}, s_{1,2}$, and $s_{2,1}$ back into Q_1. After processing all the segments in Q_1, we arrive at the final state of Figure 3.8.

Criterion	$s_{-,1}$	$s_{-,2}$
SC_1	$\langle r, s, t\rangle, \langle u, v\rangle$	$\langle r, s, t\rangle$
SC_2	$\langle r, s, t\rangle$	$\langle r, s, t\rangle, \langle u, v\rangle$

Figure 3.8: Final state of Duplo

As with simple iterative blocking, the algorithm is guaranteed to result in non-overlapping clusters, and therefore a connected-component operation is not necessary for postprocessing the results.

3.6.4 Segment Queue Policy

The policy for determining which segment to process from Q_1 significantly affects the runtime of Duplo. The following list shows various policies Q_1 might have.

- *First-Come-First-Serve (FCFS):* Processes the segments in the order they were pushed into the segment queue.
- *Hits:* Sorts the segments in decreasing number of "hits" they receive from newly merged records of other segments. Segments with the highest number of hits are processed first. Initially, all segments have an infinite number of hits. The hit count for a segment is set to zero after the segment is processed.
- *Random:* Randomly processes segments.
- *Inverse Hits:* Sorts the segments in increasing number of hits, so that the segments with the fewest hits are processed first.

Later in Section 3.7.3, we compare the policies and show which policy makes Duplo efficient.

3.7 Experimental Evaluation

In this section, we evaluate iterative blocking on real datasets and show how iterative blocking outperforms blocking both in accuracy and runtime. 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. We also used a raw disk without a file system for storing the blocks in order to avoid caching and accurately measure the I/O cost.

As mentioned in Section 3.1, the experiments in this section were carried out by Whang [87]. Whang also implemented the basic blocking and Lego algorithms, but we present the experimental results for those algorithms here in order to justify the iterative blocking approach. The implementation of Duplo was primarily the work of the author of this thesis.

Real Dataset The comparison-shopping dataset we used is the same dataset used in Chapter 2. The dataset 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 set of 2 million records randomly chosen from the entire dataset. We also experimented on a hotel dataset provided by Yahoo! Travel (see Section 3.7.4) 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.

CER **Algorithm** For our evaluation we used two different CER algorithms. Our primary CER algorithm is the R-Swoosh algorithm of Benjelloun et al. [11], and the main body of our results were generated with this algorithm. However, to illustrate that our Lego and Duplo algorithms can be used with any CER algorithm, in Section 3.7.5 we study a version of the Monge and Elkan algorithm [67], and show how the accuracy-performance tradeoffs vary. For both algorithms we used the products and travel datasets.

R-Swoosh uses a Boolean pairwise match function to compare records and a pairwise merge function to merge two records that match into a composite record. The match function for the shopping data compares the title, price, and category values of two records. For the hotel dataset, we compared the names and addresses of hotels.

Blocking Criteria We used minhash signatures [51] for distributing the records into blocks. A minhash signature is used to estimate the Jaccard similarity between two strings (i.e., the portion of n-grams of the strings shared). A complete description of minhash is given in Chapter 4. Records with the same minhash signature are assigned to the same block. For our datasets, we extracted 3-grams from the titles of the shopping records. Similarly, we extracted 3-grams from the names of the hotel records. Throughout our

experiments, we did not vary the n-gram length and fixed it to 3. We then generated a minhash signature that is an array of integers, where each integer is generated by applying a random hash function to the 3-gram set of the record. We can produce several minhash signatures of a record (one for each blocking criterion) by using different sets of random hash functions.

The advantage of using minhash signatures is that we can easily adjust the number of blocking criteria and signature length to produce reasonable accuracy and performance. Although there are many other ways to produce blocking criteria (e.g., manually creating blocking criteria instead of using minhash signatures), we believe our approach is ideal in showing the trends of accuracy and performance against different "qualities" of blocking criteria.

Metrics We used accuracy and runtime metrics to evaluate iterative blocking. To evaluate accuracy, we compared our algorithm results with a "gold standard", which is the result of running CER on the entire dataset (i.e., CER(R)). Note that we did *not* measure the correctness of the CER algorithm itself, but instead determined the comprehensiveness: how "close" the blocking or iterative blocking results are to the exhaustive result.

We used the Pairwise F_1 metric to evaluate the accuracy of the results of our algorithms. Suppose that the gold standard G contains the set of record pairs that match for the exhaustive solution while set S contains the matching pairs for one of our algorithms. Then the precision PairPrecision is $\frac{|G \cap S|}{|S|}$ while the recall PairRecall is $\frac{|G \cap S|}{|G|}$. Using PairPrecision and PairRecall, we compute Pairwise F_1, which is defined as $\frac{2 \times \text{PairPrecision} \times \text{PairRecall}}{\text{PairPrecision} + \text{PairRecall}}$. See Chapter 6 for details on the Pairwise F_1 metric.

For runtime, we measured the wall-clock runtime for each algorithm.

3.7.1 Accuracy

We compare the accuracy (Pairwise F_1) of the Lego algorithm with the following techniques:

- *Blocking:* Runs CER on each block separately. The answer is produced by simply collecting the final records from all the blocks.
- *Blocking-CC:* Runs CER on each block, and then performs a connected-component operation on all the records. For example, if the final records $\langle r, s \rangle$ and $\langle s, t \rangle$ are in different blocks, they are merged to $\langle r, s, t \rangle$.

We use relatively small datasets of 50,000 records in order to be able to compare our results with the gold standard, which takes a long time to produce. The accuracy and runtime values are the average of five test results on distinct random subsets of the larger 2 million record set.

Varying the Average Block Size In Figure 3.9, we compare the accuracy results of Lego, Blocking, and Blocking-CC using different average block sizes. Varying the minhash signature length affects the average block size. In our experiments, we varied the minhash signature length from 14 to 1 integers (the shorter the minhash signature, the larger the average block size becomes) and fixed the number of blocking criteria to 5. We plotted against the average block size (i.e., the average number of records in a block), which is the average size of all the blocks produced by MC. When the average block size increases, we find more record matches, but also perform more record comparisons.

As the average block size increases, Lego shows the most rapid increase in accuracy (see Figure 3.9). For example, when the average block size is about 13, Lego has 88% accuracy while Blocking has 76% accuracy. Blocking-CC has a higher accuracy than Blocking, since the connected-component operation yields many more correct matches. However, since the connected-component operation does not check with the match function, some of these matches are incorrect. The Lego algorithm only combines two clusters with the permission of the match function and therefore does not yield false positives. Further, the Lego algorithm finds matches through iteration that Blocking-CC might not. Therefore, the Lego algorithm has greater accuracy than Blocking-CC. Note that, for the largest average block size 32, Blocking has almost the

same accuracy as Lego because enough records are compared with each other. However, increasing the average block size also increases the runtime as we shall see in Section 3.7.2.

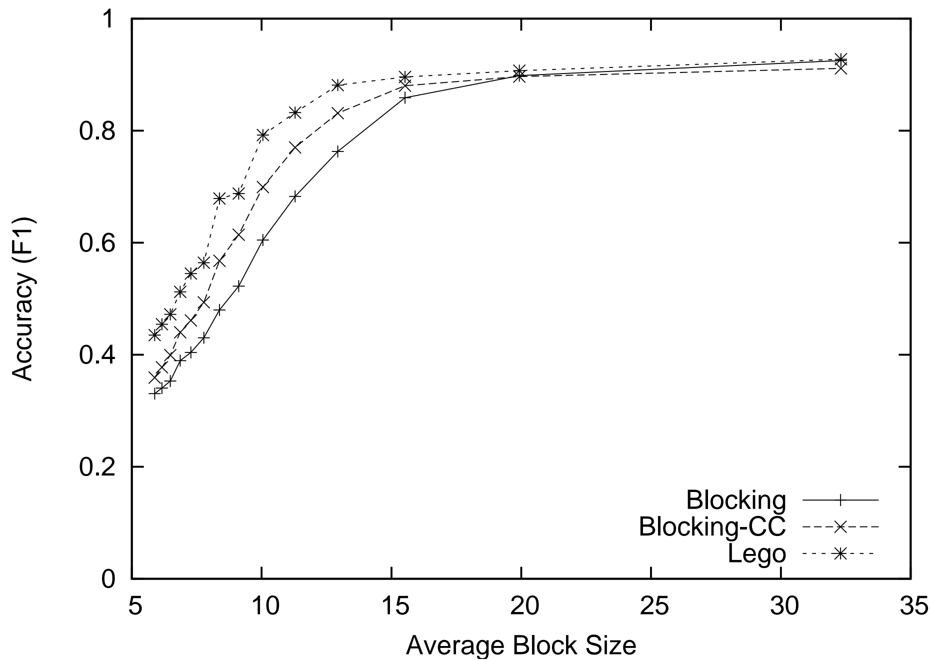

Figure 3.9: Average block size impact on accuracy, 50K records

Varying the Number of Blocking Criteria We also compared the accuracies of Lego, Blocking, and Blocking-CC while fixing the minhash signature length to 10 (making the average block size 7.3, the fifth smallest block size in Figure 3.9) and increasing the number of blocking criteria from 1 to 15 as shown in Figure 3.10. As the number of blocking criteria increases, the accuracy of Lego increases faster than the other approaches, since Lego benefits both from the extra matches found within the new blocks as well as the extra matches found due to iteration across blocks. When the number of blocking criteria is 15, Lego achieves 78% accuracy while Blocking achieves 55% accuracy. All the algorithms will eventually achieve high accuracy as the number of blocking criteria increases further, but Lego can achieve a high accuracy using

far fewer blocking criteria than the other approaches.

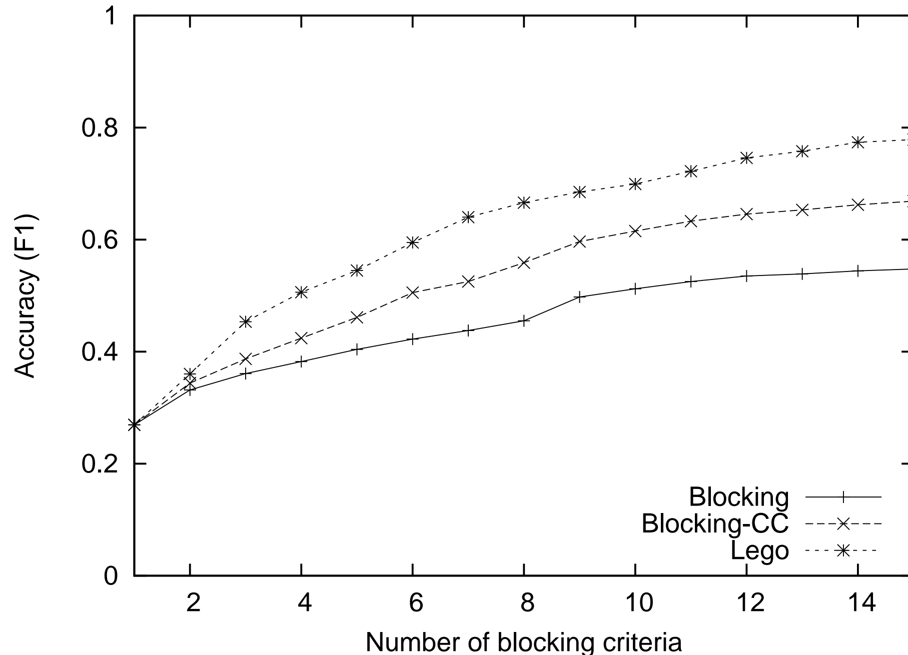

Figure 3.10: Number of Blocking Criteria impact on accuracy, 50K records

3.7.2 Lego Runtime

We compare the runtimes of Lego, Blocking, and Blocking-CC with the gold standard (i.e., running ER on the entire dataset). In all cases, all data was memory resident. Figure 3.11 shows the runtimes (on a log scale) while fixing the number of blocking criteria to 5 and decreasing the minhash signature length from 14 to 1 integers to generate different average block sizes. The points of Figure 3.11 that have an average block size of 7.3 (using a signature length of 10) correspond to the points of Figure 3.10 that use 5 blocking criteria. The runtime for the gold standard is 1641 seconds while the runtime for Lego ranges from 10 to 289 seconds.

Lego also has a comparable performance with Blocking. Lego is actually faster than Blocking for the average block sizes of 16 and 20 because the

time saved by avoiding redundant comparisons is larger than the additional time needed to iteratively process blocks. To illustrate how Lego can be faster than Blocking, suppose that two blocks b_1 and b_2 contain the same two matching records r and s. While Blocking compares r and s twice, Lego can process b_1 and reflect the merged record $\langle r, s \rangle$ on b_2, saving the next record comparison. Finally, Blocking-CC is slower than Lego and Blocking because of the additional overhead of connecting components.

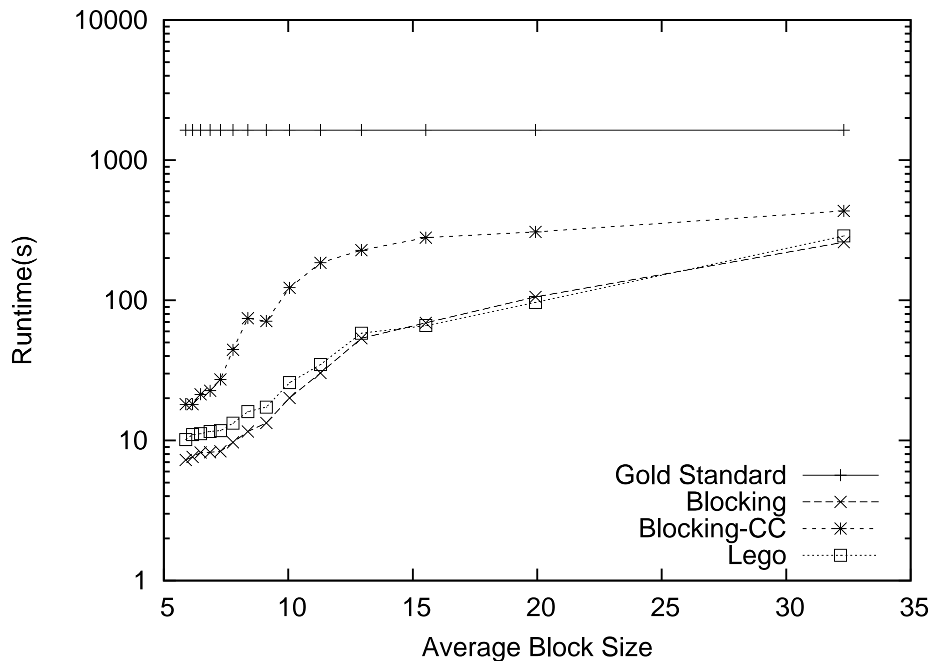

Figure 3.11: Average block size impact on runtime, 50K records

We omit the plot for the runtime versus number of blocking criteria because it shows a similar trend as Figure 3.11.

3.7.3 Duplo Performance

We now compare the performances of the Duplo algorithm and Blocking on large datasets that may not fit in memory. We adapt the Blocking algorithm to make use of the disk as described in Section 3.4. For the Duplo algorithm,

we now use segments and the merge log to iteratively process blocks from the disk. We also test on a variant of Duplo (called Duplo Memory), which uses segments but not the global merge log L_1 (i.e., Duplo Memory keeps an in-memory hash table for managing maximal records just like in Lego). While having a hash table in memory enhances the performance of Duplo, we must trade off that gain against the required larger amount of memory.

Before comparing the three techniques, we first evaluate several segment queue policies that determine which segment to process from Q_1 first. Once we have found the best segment queue policy, we then compare the scalability results of Duplo, Duplo Memory, and Blocking. We show that both Duplo and Duplo Memory outperform Blocking when producing accurate ER results of large datasets. Throughout the experiments for Duplo and Duplo Memory, we use 160 segments per blocking criterion for 5 blocking criteria and allocate 30MB[1] of disk space per segment.

Segment Queue Policy The segment queue policy significantly affects the number of segments that need to be processed as well as the overall runtime. We compare the four policies mentioned in Section 3.6.3: Hits, FCFS, Random, and Inverse Hits. Figure 3.12 shows the runtime results of using the four policies on 1 million random shopping records where the minhash signature length is 10.

Both Hits and FCFS have excellent performance, with Hits being slightly faster than FCFS. The Hits policy works well because segments that have many hits are likely to generate many record merges each time they are processed. The FCFS policy also works well because segments stay in the queue as long as possible and thus tend to generate more record merges when they are processed. The Inverse Hits policy is the worst strategy and is even slower than the Random policy because segments that are least likely to generate

[1]While a segment size of 30MB may seem small relative to the amount of main memory available, additional memory is needed to process the records in the segment by the CER algorithm and the local merge log L_2.

record matches are processed first, resulting in repeated processing of the same segments. Having a good segment queue strategy is thus very important for efficiency. For the rest of our experiments for Duplo and Duplo Memory, we use Hits as our default policy.

Strategy	Runtime (hrs)
Hits	2.0
FCFS	2.1
Random	7.5
Inverse Hits	11.7

Figure 3.12: Runtimes for different segment queue strategies, 1M records

Scalability Using the Hits policy, we compare the scalability results of Duplo, Duplo Memory, and Blocking. Figure 3.13 shows the result of running the three algorithms on 0.5 to 2 million random shopping records with a minhash signature length of 4 (the average block sizes being 75, 130, 232 for 0.5, 1, 2 million records, respectively). We set the maximum java heap size to 3.5G for Duplo Memory and 1.5G for Duplo in order to demonstrate that Duplo uses a small amount of memory (Duplo Memory actually slows down significantly when using 1.5G of memory). Both Duplo and Duplo Memory scale better than Blocking (by 15% and 43% for 2 million records, respectively) because Duplo and Duplo Memory save processing time by reflecting the ER results of blocks to other blocks. Duplo Memory is 33% faster than Duplo for 2 million records. The runtime improvements will of course depend on the number of blocking criteria and average block size used.

Figure 3.14 shows the runtimes needed for Duplo, Duplo Memory, and Blocking to achieve certain accuracy results on 2 million records. We generated the figure by running a series of scenarios with different minhash signatures. For each scenario we obtained an accuracy and runtime pair, and plotted it in Figure 3.14. We could not directly calculate the accuracy because the dataset was too large to compute the gold standard using the CER algorithm. Instead, we exploited the fact that the accuracy is only dependent on the number of

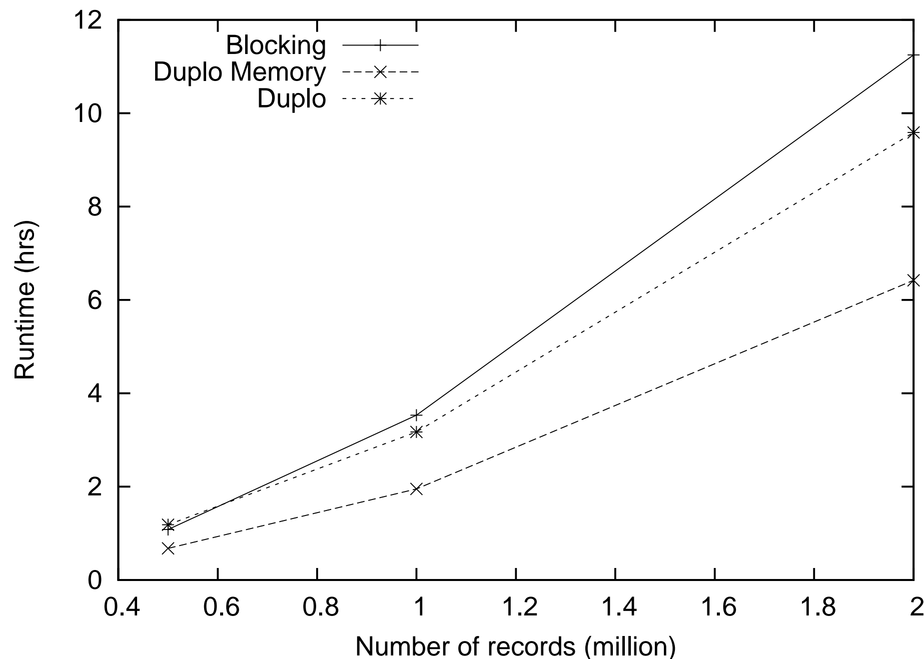

Figure 3.13: Scalability, 2M records

blocking criteria and minhash signature length, and not on the size of the random dataset. Hence, by using the same number of blocking criteria and minhash signature length, we could use the accuracy results on 50,000 records in Figure 3.9 to estimate the accuracy results on 2 million records. We can see in the figure that both Duplo and Duplo Memory significantly outperform Blocking for highly accurate results. For example, Duplo Memory and Duplo takes 14 and 17 hours to achieve 91% accuracy while Blocking takes 26 hours to achieve 90% accuracy.

In summary, Duplo and Duplo Memory outperform Blocking when producing accurate ER results on large datasets. The key idea is that ER results of blocks are reflected on other blocks, saving a lot of processing time. Although Duplo Memory is faster than Duplo, it requires a large amount of memory to use the in-memory hash table. Duplo should be used instead of Duplo Memory when the main memory size is too small to hold the hash table.

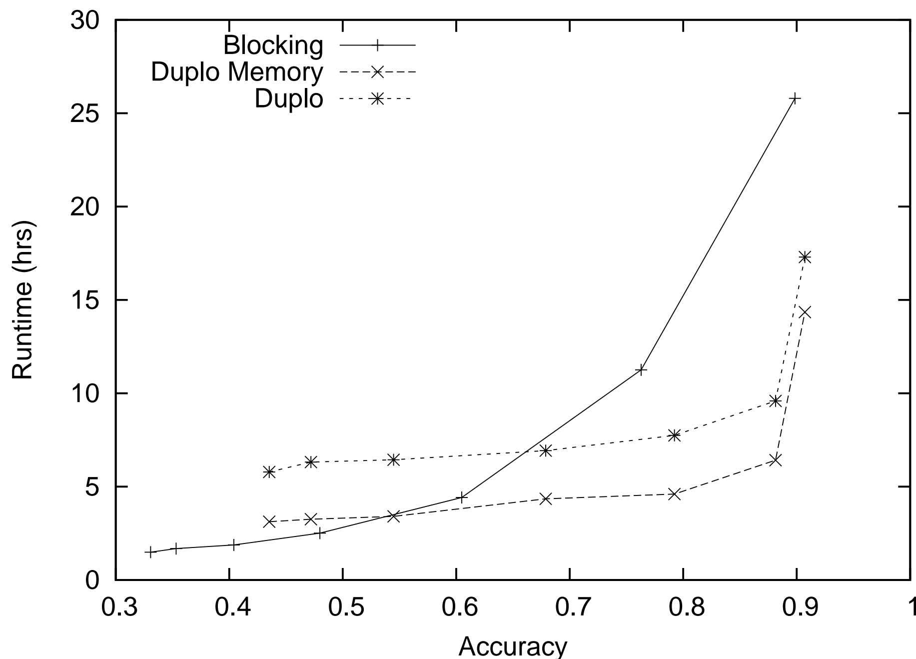

Figure 3.14: Runtime needed to achieve given accuracy, 2M records

3.7.4 Other Datasets

In our shopping application, there are many iterative record matches where merged records generate additional record matches. However, in other applications there may be fewer iterative matches. For example, suppose that there are two blocks b_1 ={r, s} and b_2 ={s, t}. If r and s merge into $\langle r, s\rangle$ in b_1, but never match with t, then we cannot generate any new record matches by reflecting $\langle r, s\rangle$ to b_2. In order to investigate how iterative blocking works for different types of datasets, we also tested on a hotel dataset provided by Yahoo! Travel. The records in the hotel dataset mostly come from two data sources that do not have duplicates within themselves. As a result, we rarely have more than two records matching with each other.

In this case, iterative blocking does not significantly improve accuracy over blocking. However, its blocks still have a performance gain because ER processing in one block saves processing time for subsequent blocks. Figure 3.15

shows the runtimes for Duplo Memory and Blocking on hotel records. The hotel dataset was only 27,049 records, so we generated larger datasets by simply replicating the same dataset 16, 32, and 64 times (resulting in 0.43, 0.86, 1.73 million records, respectively). Duplo Memory improves Blocking by 48% for 1.73 million records. We omit the data for Duplo because replicating the data was producing too many record merges, significantly increasing the time to read the merge log for each segment. In practice, the number of merging records is much smaller.

Algorithm	**Runtimes for 0.43M / 0.86M / 1.73M records (hrs)**
Blocking	0.27 / 0.73 / 2.27
Duplo Memory	0.18 / 0.41 / 1.18

Figure 3.15: Runtimes on the hotel dataset, 1.7M records

3.7.5 Other CER Algorithms

In this section, we experiment with a CER algorithm based on Monge and Elkan [67] (we call it the ME algorithm) where records are sorted using an application-specific key and then clustered with a sequential scan. Each cluster has a representative record that can be compared with other records. The representative record contains values of recently added records to the cluster, but only those that were "far away enough" from the representative record when added to the cluster. During the sequential scan, a priority queue is used to contain the most recently updated clusters. (While the size of the priority queue in [67] is constant, we use a size proportional to the number of records processed.) Each new record is compared to all the clusters in the priority queue. If there is a matching cluster, the record is combined with the cluster, and the cluster is moved to the head of the priority queue. If there is no match, a new singleton cluster is created and pushed into the head of the priority queue. The final result is a set of clusters.

We used the Yahoo! Shopping dataset and compared the records with the same match function used in R-Swoosh. We also used the same blocking criteria that uses minhash signatures generated from the titles of the records. Throughout our experiments, we used 5 blocking criteria and a minhash signature length of 3.

Figure 3.16 shows that Lego has a higher accuracy than Blocking or Blocking-CC for 50,000 shopping records. To correctly measure the accuracy values, we constructed the gold standard by repeatedly running the ME algorithm on the entire dataset until the clusters no longer merged. Although both Lego and Blocking-CC have lower precision values than Blocking, they have much higher recall and accuracy values.

Algorithm	Precision	Recall	Accuracy
Lego	0.93	0.74	0.82
Blocking	0.99	0.38	0.55
Blocking-CC	0.93	0.69	0.79

Figure 3.16: Accuracy for the ME algorithm, 50K records

Figure 3.17 shows that Duplo Memory is 42∼76% slower than Blocking while Duplo Disk is 101∼109% slower than Blocking for 0.5 to 2 million shopping records (the average block sizes being 165, 309, 577 for 0.5, 1, 2 million records, respectively). This result contrasts with that of Figure 3.13 where Blocking is the slowest algorithm. The reason is that ME was not finding all the matching records for each block (notice the low recall for Blocking in Figure 3.16) because the shopping data required iterative comparisons to be completely resolved. As a result, we could not exploit the ER results of previously processed blocks as much as we did before in Figure 3.13. In addition, more blocks had to be iteratively processed until there were no matching records in any of the blocks. Hence, the ME algorithm illustrates the case where we are trading off runtime (i.e., the overhead of managing the blocks and merge logs) for better accuracy.

In summary, both Duplo Memory and Duplo Disk are better than Blocking in accuracy, but are worse in runtime when the ME algorithm is used as the CER

Algorithm	Runtimes for 0.5M / 1M / 2M records (hrs)
Blocking	0.83 / 2.95 / 10.46
Duplo Memory	1.18 / 4.75 / 18.37
Duplo Disk	1.67 / 6.02 / 21.9

Figure 3.17: Scalability for the ME algorithm, 2M records

algorithm. It is important to understand that we are not comparing the performances of R-Swoosh and ME through these experiments. ME is designed to be an efficient algorithm for data that can easily be clustered (which is not the case here) while R-Swoosh is a more exhaustive algorithm that can find subtle record matches. The key observations here are that any CER algorithm can be plugged into the iterative blocking framework and that there is a different runtime/accuracy tradeoff when using the ME algorithm.

3.8 Related Work

Various blocking techniques that focus on ER accuracy have been proposed. Early works [70, 68] manually generate blocking criteria based on the characteristics of the data. Learning techniques [18, 66] produce blocking criteria based on training data so that the number of matching record pairs found is maximized while the number of non-matching record pairs in the same block is minimized. Estimation techniques [93, 47] can be used to estimate the number of matching record pairs missed by each blocking criterion. Most of the works above use several blocking criteria to increase the chance of similar records to be compared within the same block. Our iterative blocking technique can use any blocking criteria produced from the works above as its MC function. Gu and Baxter [42] propose “adaptive filtering”, a technique for combatting the problem of large blocks by pruning comparisons unlikely to match within large blocks. Adaptive filtering may be used in conjunction with the techniques presented here.

Another line of research is blocking techniques that focus on ER performance [10]. The Canopy Clustering [62] technique uses a cheap, approximate distance measure to efficiently divide the data into overlapping subsets called canopies and then processes each canopy using exact distances. The notion of canopies maps directly to our notion of blocks, and our work takes the extra step of exploiting the ER results of previously processed blocks. The Sorted Neighborhood method [45] sorts records based on a sorting key and moves a fixed-sized window sequentially over the sorted records. Several passes can be done using different sorting keys. A transitive closure is then performed on all the matching record pairs. While the Sorted Neighborhood method can be seen as a form of blocking by considering each window as a block, it does not exploit the ER results of blocks in different blocking criteria.

To the best of our knowledge, the only work that takes a generic approach in efficiently processing blocks is the BigMatch [96] algorithm. BigMatch compares a large file of records on disk with a smaller file of records in memory without having to sort the larger file. For each record from the larger file, BigMatch uses the blocking criteria to search all the records in the smaller file that are in the same block for at least one blocking criteria. In contrast, iterative blocking is designed to resolve a single large set of records and has the additional feature of iteratively processing blocks.

As mentioned in Section 3.1, the work presented in this chapter is a simplification of the formalized interactive blocking framework described by Whang et al. [87].

3.9 Conclusion

In this chapter, we have presented an iterative blocking framework that works in conjunction with any core entity resolution algorithm to generate approximate ER results with greatly improved performance. Iterative blocking is

distinguished from standard blocking techniques in that it carries over information discovered in the processing of one block into all subsequent blocks.

We have presented the Duplo on-disk algorithm for use with large datasets for which only a subset of the records may fit in memory. We also presented four queueing strategies for the processing of blocks in the Duplo algorithm and identified the best strategy by experiment.

We presented experimental results comparing iterative blocking with standard blocking techniques. Iterative blocking can beat standard techniques on two fronts. Iterative blocking improves accuracy by reprocessing blocks in light of new merges found in other blocks. Performance is also improved by eliminating redundant comparisons already performed in other blocks. Our experimental results confirm that iterative blocking can offer both improvements at the same time.

Chapter 4

LSH and Minhash

4.1 Introduction

As discussed in Chapter 3, entity resolution can benefit from the use of blocking. While blocking is a powerful technique, it requires a good set of blocking criteria to divide the records into independent blocks. The experiments of Chapter 3 used a method called minhash in this capacity. Minhash is one of a number of methods that define a family of hash functions for grouping data items according to their similarity. Such a method is generally referred to as a *locality-sensitive* hash function family.

The minhash method groups sets into buckets such that similar pairs of sets (as defined by the Jaccard coefficient) are likely to end up in the same bucket. To use minhash in entity resolution, we are required to convert each record into a representative set, relying on the assumption that the Jaccard coefficient of these sets is a good estimate of the similarity of the original records. For example, in Chapter 3, we created these representative sets by shingling the string value of an attribute of each record.

If the blocking criteria we used more closely modeled the similarity of records, we would expect the accuracy of the blocking result to increase. We can turn to many known locality-sensitive hash function families for different

data types and similarity measures. Clearly minhash [21] works with the Jaccard coefficient of sets, but we also have simhash [23] for cosine similarity, and [4] describes locality-sensitive hash function families for Hamming and Euclidean distances. In this chapter, we will explore how to extend the basic minhash technique for data types and similarity measures not yet considered.

The contributions of this chapter are the following:

- Section 4.3 describes MapMinhash, a modification to minhash suitable for use on map data structures.
- In Section 4.4, we show how minhash may be modified for use with sets with weighted values, using the weighted Jaccard coefficient as a similarity measure.
- Section 4.5 shows how minhash may be applied to complex data types whose individual components have data types with corresponding locality-sensitive hash function families.
- In Section 4.6, we present experimental results comparing the techniques of Section 4.4 to the only other known technique for weighted sets.

We introduce our preliminaries in Section 4.2 and conclude in Section 4.8.

4.2 Preliminaries

4.2.1 LSH

In this section, we precisely define the property required of a locality-sensitive hash function family. We also briefly introduce the LSH algorithm, which is the standard algorithm used in conjunction with locality-sensitive hash function families.

Locality-sensitive hash function families must satisfy what we call the LSH property.[1]

[1] This property is defined by Charikar [23]. It is notably different from the one described

Property 4.2.1. *We say that a family of hash functions* H *satisfies the LSH property for a similarity measure* sim *if for a randomly selected* $h \in H$*, for any values* x *and* y *in the domain of* sim, $Pr[h(x) = h(y)] = sim(x, y)$

The LSH algorithm is to select $l \times m$ hash functions from a family H of hash functions. For each item, we use these functions to generate m signatures, each with l components (with each component being the result of invoking a hash function on the given item). The item is then added to a bucket for each of its signatures. After each item has been added to its respective buckets, the algorithm considers each bucket independently. For each bucket, the algorithm compares all of the contents against one another to find items of high similarity. Generally, we may be interested in finding near neighbors above a certain threshold of similarity. By properly selecting the parameters l and m, this algorithm can target any similarity threshold, while simultaneously reducing false negatives (missed matches) and false positives (dissimilar records ending up in the same bucket). See [4] for a detailed description of the algorithm and its applications.

4.2.2 Minhash

Before detailing our modifications to minhash, we will briefly review minhash itself. Minhash defines a family of hash functions on sets that satisfies the LSH property for the Jaccard similarity metric. That is, for sets R and S, a random hash function in the family satisfies the LSH property:

$$Pr[h(R), h(S)] = J(R, S)$$

Where $J(R, S)$ is the Jaccard coefficient for the two sets R and S:

$$J(R, S) = \frac{|R \cap S|}{|R \cup S|}$$

---

by Andoni and Indyk [4], but similar in spirit. We use this property for simplicity.

Minhash is based on permutations of possible set elements. We can assume that the sets to be compared are subsets of a universe U of possible set elements. Minhash defines its hash functions in reference to permutations of U. Specifically, for a permutation π, the minhash hash function h_π corresponding to π is defined as:

$$h_\pi(S) = \min_{x \in S} \pi(x)$$

(Here we use $\pi(x)$ to refer to the rank of element x in permutation π.) Choosing π randomly from the set of all possible permutations of U results in a direct relationship between this hashing scheme and the Jaccard coefficient [21].

As an example, suppose that U consists of the letters $\{a, b, c, d, e\}$. Let a randomly chosen permutation π be b, e, a, c, d. Then the minhash hash function corresponding to π would return 0 for any set containing b, 1 for any set containing e but not containing b, 2 for any set containing a but not containing b or e, and so on.

The hash functions in the minhash family run in linear time in the number of elements in the input set, so long as computing $\pi(x)$ is a constant-time operation. Unfortunately, it is difficult to choose a truly random permutation. Further, storage of a random permutation in a manner that allows quick computation of $\pi(x)$ is quite large. For example, storing a permutation as a sequential list of elements would offer constant-time access, but requires $O(|U|)$ storage.

Broder et al. discuss this issue in [21], proposing choosing permutations randomly from smaller subsets (families) of all possible permutations. The authors identify "min-wise independence" as the necessary property of these permutation families that allows the relationship between minhash and the Jaccard coefficient to be maintained. A family of permutations F is min-wise independent if for a randomly chosen permutation π, and any set $S \subset U$, $\Pr[\min_{s \in S} \pi(s) = \pi(x)] = 1/|S|$, for any element $x \in S$.

Broder also identifies more relaxed properties that may suffice to give an efficient approximation. Indyk [52] discovered a family of permutations

satisfying the relaxed property of "approximate min-wise independence". It seems, however, that the most common implementation of minhash uses a much simpler, though less theoretically pure, technique. We note that any function that maps elements in universe U to numerical values can be used to define an permutation of U. For example, consider the function f defined by the table in Figure 4.2.2.

x	f(x)
a	$\frac{1}{137}$
b	3.14
c	2.998
d	6.626
e	2.718

Figure 4.1: A function f

We can define a permutation on U by sorting based on the results of the function f. In our example, the permutation implied by f would be a, e, c, b, d.

For implementing minhash, it is most common to use a randomly chosen linear function: $f(x) = ax + b \mod P$, with P being a fixed prime greater than $|U|$, and $a \in [1, P), b \in [0, P)$ being randomly chosen integers. (Here, we are treating an element x as though it were a numeric value. If the items in U are not numeric, we can use standard hash functions to compute a numeric value for each element. In this case, U actually refers to the universe of possible hash values. We can assume that the probability of hash collisions is low enough to be negligible. We will continue to use items as numeric values under the understanding that they may be the results of standard hash functions of the actual elements.)

Strictly, a linear function f is not a permutation, as it does not return return the exact rank of x. However, if we define permutation π as the permutation implied by f, then the standard minhash function h_π is equivalent to a similarly defined h_f hash function:

$$h_f(S) = \min_{x \in S} f(x)$$

That is, $Pr[h_\pi(R) = h_\pi(S)] = Pr[h_f(R) = h_f(S)] = J(R, S)$. Note that, since prime $P > |U|$, f is a one-to-one function and will never return the same value for two different elements.

The linear function method is popular, most likely due to its efficiency and ease of implementation. While it does not satisfy even the relaxed property of approximate min-wise independence [21], it can be demonstrated empirically to work well in practice.

Now to summarize the minhash algorithm using the random linear function method. First, we choose a prime P that is larger than the size of the universe of possible elements in the sets to be considered. Then we choose random integers a and b. The minhash of a set S is then defined as:

$$h(S) = \min_{x \in S}(ax + b \mod P)$$

The result of this definition is that $Pr[h(R) = h(S)] = J(R, S)$, which allows us to use the LSH algorithm to identify similar sets.

Algorithm 4.1 shows this implementation of Minhash. The algorithm is written as a signature-generating algorithm. That is, rather than generating a single hash value, it generates a signature of l random hash values. Although an LSH signature-generating function makes use of random values, it is important that the random number generator be deterministic—generating the same random numbers on every invocation. To ensure that our function is deterministic, it makes use of a fixed value (called FIXED_SEED) to seed a pseudorandom number generator.

4.3 Map Minhash

A map (or dictionary) is a data type that stores mappings between keys and values. Maps are used widely, but to give a concrete example, we might use a map to store the ZIP codes of various cities: $\{Princeton : 08544, Stanford : 94305, Berkeley : 94704\}$. In this map, the key Stanford has value 94305.

```
Input:   S: the set to generate a signature for
         l: the desired length of the signature
Output:  an l-length signature for S
 1: Minhash(S, l)
 2: randstream ← new RandomStream(FIXED_SEED)
 3: for i = 1 to l do
 4:    a ← random integer from randstream in [1, P)
 5:    b ← random integer from randstream in [0, P)
 6:    // compute minimum of ax + b  mod P
 7:    minhash ← P
 8:    for all x ∈ S do
 9:       current ← ax + b  mod P
10:       if minhash > current then
11:          minhash ← current;
12:       end if
13:    end for
14:    sig[i] ← minhash
15: end for
16: return sig
```

Algorithm 4.1: Minhash Signature algorithm

Much like primary keys in a database, map keys must not appear more than once in any given map. Given this constraint, we note that any relation with a primary key can be modeled as a map. Conversely, maps can be seen as sets of tuples: our example tuple can be written equivalently as the set {(Princeton, 08544), (Stanford, 94305), (Berkeley, 94704)}. Given this representation, we can refer to the union of maps, the intersection of maps, and even the Jaccard coefficient of maps.

Since maps are equivalent to sets of tuples, we can apply basic minhash directly, and we immediately have an LSH family for the Jaccard coefficient of maps. Unfortunately, applying minhash directly leads to some surprising results. We demonstrate such a result with a simple example.

Consider the maps $M = \{a : 1, b : 2\}$ and $N = \{a : 1, b : 3\}$. Since M and N share two keys, and only match on one of them, we would expect a good similarity function to tell us that $sim(M, N) = 1/2$. The Jaccard coefficient,

however, is not such a metric!

The set representations of the two sets are $M = \{(a,1),(b,2)\}$ and $N = \{(a,1),(b,3)\}$. The intersection of the sets consists of the single tuple $(a,1)$, and the union is the three tuple set $\{(a,1),(b,2),(b,3)\}$. Therefore the Jaccard coefficient $J(M,N) = 1/3$. Maps with more items can have Jaccard coefficients that vary by as much of a factor of 2 from expectations using the logic above.

This problem stems from the fact that keys that appear in both maps with different values will show up twice in the union of the two maps, whereas keys with values that match will show up only once. This analysis suggests a simple modification to the Jaccard coefficient that we can use as a similarity measure for maps. Using the notation that keys(M) refers to the set of keys in map M, we can define a similarity metric for maps as follows:

$$\mathrm{sim}(M,N) = \frac{|M \cap N|}{|\mathrm{keys}(M) \cup \mathrm{keys}(N)|}$$

This measure modifies the Jaccard coefficient by dividing by the size of the union of keys in the maps, rather than the number of tuples in the union of the maps. This modification generally reduces the size of the denominator. However, the fact that maps have only one entry for a given key guarantees that $\mathrm{sim}(M,N) \leq 1$.

For us to use this more natural similarity measure in LSH, we must find a corresponding LSH family. Luckily, we can modify basic minhash to satisfy the LSH property for this new similarity measure.

We formally define the map minhash mmh_π of a map M with respect to a permutation π of all possible map keys.

Definition 4.3.1. *Let* k_m *be the element in* keys(M) *with the lowest rank in* π. *That is,* $\pi(k_m) = \min_{k \in \mathrm{keys}(M)} \pi(k)$. *Then* mmh_π *is defined as follows:*

$$\mathrm{mmh}_\pi(M) = (\pi(k_m), M(k_m))$$

That is, to compute the map minhash, we find the key k_m that has the lowest rank in π. The hash is then the rank of k_m followed by the value k_m takes on in M. Note that the map minhash is a pair, not a single value.

If we select π randomly from a min-wise independent family of permutations, then mmh satisfies the LSH property for our map similarity measure.

Theorem 4.3.2. *If permutation* π *is selected randomly from a family of min-wise independent permutations, then* $\Pr[\mathrm{mmh}_\pi(M) = \mathrm{mmh}_\pi(N)] = \frac{|M \cap N|}{|\mathrm{keys}(M) \cup \mathrm{keys}(N)|}$.

Proof. First, we consider the key $k^* \in \mathrm{keys}(M) \cup \mathrm{keys}(N)$ that has the lowest rank in π. That is, k^* is the element of $\mathrm{keys}(M) \cup \mathrm{keys}(N)$ such that $\pi(k^*) = \min_{k \in \mathrm{keys}(M) \cup \mathrm{keys}(N)} \pi(k)$. Since π was selected from a min-wise independent family, for any key k, $\Pr[k = k^*] = 1/|\mathrm{keys}(M) \cup \mathrm{keys}(N)|$. If $k = k^*$, then the probability that the hashes of M and N match is entirely determined by whether k appears in both M and N with the same value. If it does, the hashes match. If it does not, the hashes do not match. Therefore, the probability of matching hashes overall is $\Pr[k = k^*]$ multiplied by the number of matching key value pairs in M and N: $\frac{|M \cap N|}{|\mathrm{keys}(M) \cup \mathrm{keys}(N)|}$. □

Algorithm 4.2 details a signature-generating algorithm for Map Minhash. It is essentially identical to Algorithm 4.1, but it uses a new variable minv to keep track of the value associated with the key with minimum hash. The minv value is then used as the second part of a tuple composing a hash.

4.4 Sets with Weighted Values

In many circumstances, data can be represented as sets, but some items in the set are more important than others. The importance of set items can be represented by adding a weight value to each item. For example, if set S contains values a and b, but b is twice as important as a, we might represent S as the weighted set $\{a : 1, b : 2\}$. We note that the set R may also contain the same two values, but with a now being more important than b, resulting

```
Input:    M: the map to generate a signature for
          l: the desired length of the signature
Output:   an l-length signature for M
 1: MapMinhash(M, l)
 2: randstream ← new RandomStream(FIXED_SEED)
 3: for i = 1 to l do
 4:   a ← random integer from randstream in [1, P)
 5:   b ← random integer from randstream in [0, P)
 6:   minhash ← P
 7:   minv ← 0
 8:   for all (k → v) ∈ M do
 9:     current ← ak + b  mod P
10:     if minhash > current then
11:       minhash ← current;
12:       minv ← v;
13:     end if
14:   end for
15:   sig[i] ← (minhash, minv)
16: end for
17: return sig
```

Algorithm 4.2: Map Minhash Signature algorithm

in $R = \{a : 2, b : 1\}$. We will use the notation $S(x)$ to refer to the weight of element x in the weighted set S. For example, using the definition of S above, $S(b) = 2$.

The idea that a value may have different weights in different sets may be initially baffling, but the surprise can be resolved with a simple example. Suppose that we are modeling documents as the sets of words they contain. It is clear that some words are inherently more important to a document than others. Generally, we consider rare words (those with a high IDF value) as important: the word "interesting" appears in many documents, but the word "Jaccard" appears in relatively few. However, some words may be of particularly high importance due to the number of times they appear in a document (the TF value), or even their placement within a document (e.g. words appearing in the title of a document are more important). This latter effect allows

for words to have differing importance for different documents, and thus differing weights when modeling documents as sets.

For this type of data, the weighted Jaccard coefficient is a commonly used similarity measure. It is a natural generalization to the Jaccard coefficient, and is in fact equivalent for the case where all weights are equal. For the weighted Jaccard coefficient, we divide the sum of the weights in the intersection by the sum of the weights in the union. Formally, the weighted Jaccard coefficient is defined as:

$$wJ(R, S) = \frac{\sum_{x \in R \cap S} \min(R(x), S(x))}{\sum_{y \in R \cup S} \max(R(y), S(y))}$$

In [59], Manasse et al. developed a method of sampling from weighted sets that leads directly to a family of hash functions that satisfies the LSH property for the weighted Jaccard coefficient. The Manasse scheme, however, suffers from a few problems: it is relatively slow and it is complicated to implement. In our work, we have discovered a simpler, faster scheme, and at the time of this writing, it is not patented.

The benefits of this scheme do not come without a price, however. Unfortunately, the scheme we discovered does not apply to weighted sets in general. For our scheme to work properly, the weight for any particular set element must remain the same no matter which set it appears in. That is, each set element x has an *inherent* weight, which we write as $w(x)$. We refer to these sets as "sets with weighted values", to differentiate from the more general "weighted sets".

Is this a fatal flaw in the scheme we propose? Clearly, we believe the answer to be no. There are many situations in which sets with weighted values are an acceptable model. Users in a social network may be compared based on their sets of interests—sets which generally have no indication of their importance to a particular user. In the Netflix data set, we may model a user as the set of movies she has rated highly. In a tagging system, a document may be represented as its set of tags, as the number of times a document

has been labeled with a particular tag is not necessarily a valuable metric of the importance of a tag. In all of these situations, the weight of a set element can be usefully defined as the IDF value of the element: rare interests, rare movies, and rare tags are generally more telling than their more common counterparts.

With this justification in place, we continue to describe our LSH hash family for sets with weighted values, using the weighted Jaccard coefficient as the similarity measure. Note that since an element's weight is constant in all sets, our expression for the weighted Jaccard coefficient simplifies to the following:

$$wJ(R, S) = \frac{\sum_{x \in R \cap S} w(x)}{\sum_{y \in R \cup S} w(y)}$$

We begin by solving the problem for the specific case of integral weight values, using it as a simple first stepping stone towards solving the general problem of real-valued weights.

4.4.1 Integer Weights

An LSH family for weighted Jaccard with integer weights has been described briefly in [23]. Here we provide greater detail.

First, recall that minhash is based upon families of min-wise independent permutations. This idea is extended by allowing duplicate values in the random permutations we generate. We create a bag B of elements in the universe U. For each element $x \in U$, it appears $w(x)$ times in B. We can then use any min-wise independent permutation family to select a permutation π_B of the bag B. Since a single element can have multiple positions in a permutation of a bag, we can't treat π_B as a function anymore. Instead, treating it as a list, we can eliminate duplicates from π_B, leaving only the first occurrence of each element. This gives us a permutation of U, which we will call π_U. Now that duplicates have been removed, π_U can be treated as a function again. As an example, if we have elements a, b, and c with weights 1, 2, and 3, respectively,

then $B = \{a, b, b, c, c, c\}$ and π_B could be cacbcb. We can then remove duplicates from the permutation to obtain π_U = cab.

Given a permutation created in this way, the rest of minhash applies directly. A minhash value for set A is $\min_{x \in A} \pi_U(x)$ with $\pi(x)$ being the rank of element x permutation π_U.

As in minhash, we can sidestep the creation of these independent permutations by using the order implied by random linear functions. For weighted minhash, however, we may use a sequence of random linear functions to generate a single hash. Let w_{max} be the largest possible weight of any set element. Then we generate $\{a_1, a_2, \ldots, a_{w_{max}}\}$ and $\{b_1, b_2, \ldots, b_{w_{max}}\}$. Given these values, we can define the weighted minhash of a set S as:

$$wmh(S) = \min_{x \in S} \min_{i \in [1, w(x)]} (a_i x + b_i \mod P)$$

Algorithm 4.3 details an implementation of this scheme. We note one major difference between this scheme and Algorithm 4.1 (basic Minhash) in the use of randomness. The body of Algorithm 4.1's outer loop first generates the values a, b and uses these same values for computing the linear function of each set element. A naive implementation of the integer weight algorithm would generate the arrays $\{a_i\}$ and $\{b_i\}$ as the first step in the body of this outer loop. However, there may be no need to generate the entire array, so instead, we retrieve a single value from the random stream. We then use this value as a seed for a second random stream (called subrand in the code). In this way, we can access the $\{a_i\}$ and $\{b_i\}$ values as a stream, rather than precomputing both arrays. With the code written in this way, we also no longer have the requirement of knowing the maximum possible weight w_{max}.

Accessing the $\{a_i\}$ and $\{b_i\}$ values as a stream requires recomputing the random values each time they are used. Computing the next random value in a stream is a constant-time operation, so accessing the values as a stream does not change the complexity of our algorithm (i.e. it is still linear in the total weight of the input set). Our experiments have shown, however, that

```
Input:   S: the set to generate a signature for
         l: the desired length of the signature
Output:  an l-length signature for S
 1: IntWeightedMinhash(S, l)
 2: randstream ← new RandomStream(FIXED_SEED)
 3: for i = 1 to l do
 4:    minhash ← P
 5:    randseed ← random integer from randstream
 6:    for all x ∈ S do
 7:       subrand ← new RandomStream(randseed)
 8:       for j = 1 to w(x) do
 9:          a ← random integer from subrand in [1, P)
10:          b ← random integer from subrand in [0, P)
11:          current ← ax + b  mod P
12:          if minhash > current then
13:             minhash ← current;
14:          end if
15:       end for
16:    end for
17:    sig[i] ← minhash
18: end for
19: return sig
```

Algorithm 4.3: Weighted Minhash Signature algorithm (Integer Weights)

continually regenerating these random values is computationally expensive. We can work around this limitation by pre-computing the $\{a_i\}$ and $\{b_i\}$ arrays (one pair of arrays for each of the l components in a signature) and storing them in memory. Of course this optimization brings back the need for knowing w_{max} in advance.

This integer weights method is not one of the contributions of this chapter, so we present it without proof that it satisfies the LSH property for weighted Jaccard similarity. However, the proof follows from Proposition 4.4.1, proven in Section 4.4.2.

We do note here two limitations of this scheme. The first obvious limitation is that it can only handle integer weights. Rounding weights to the nearest integer may be acceptable for some applications, but not all. If rounding is

not acceptable, more precision can be added by multiplying all weights by a constant factor, and then rounding. However, this process results in larger weights, which leads to the second limitation: performance. In regular minhash, the creation of a hash is linear in the number of elements in the set. With the integer weighted minhash, the creation of a hash is linear in the sum of the weights of the elements in the set. With large weights, this process can be quite slow. For an application that uses IDF values as the weights, we can expect weights for infrequent terms to be *very* high.

4.4.2 Real Weights

The discovery of an algorithm for real valued weights falls out of considering a solution to the performance issue of the integer weight algorithm. In the integer weight algorithm, for an element x, we must generate $w(x)$ values and compute the minimum, which can be expensive if $w(x)$ is large. We note that the values are independent, as they are computed by independently chosen random linear functions. By turning to statistics, we can simulate the process of choosing the minimum of $w(x)$ independent random values in a more efficient manner.

Given a sequence of random variables $X_1, \ldots, X_n$, the kth smallest value, denoted $X_{(k)}$ is called the kth order statistic. The first order statistic is another name for the minimum of the values, that is, $X_{(1)} = \min X_i$. When the X_i variables are independent and each follow the uniform distribution, the order statistics follow a well known distribution called the beta distribution. Our algorithm for real valued weights is based on the idea of selecting a single value from a beta-distributed random variable, rather than selecting multiple uniformly distributed values and computing the minimum.

The intuition for this method relies on the beta distribution's relation to order statistics, which only applies to the integer weight case. However, the parameters of the beta distribution can have non-integer values, and real valued weights can easily be used in place of integers. In this section, we define a

method of generating weighted minhashes using the beta distribution, prove it to be theoretically sound for both integer and real valued weights, and finally we describe an efficient practical algorithm based on this theory.

The necessary property for minhash is min-wise independence, but for weighted minhash, we desire a different property, which we call *min-wise weight bias*. Formally, a method of selecting a random permutation π is min-wise weight biased if $\Pr[\min_{s \in S} \pi(s) = \pi(x)] = w(x) / \sum_{s \in S} w(s)$. Min-wise weight bias generalizes min-wise independence in the following sense: when all weights are set to 1, the method of selecting a random permutation from a min-wise independent family of permutations is a min-wise weight biased method.

Note that we apply the term min-wise weight bias to *methods* of selecting random permutations, rather than to families of permutations. One *could* use min-wise weight biased to describe a family of permutations such that selecting a permutation uniformly at random is a min-wise weight biased method. However, applying the term to the method allows us to select permutations from a family nonuniformly in order to obtain min-wise weight bias. To illustrate with an example, suppose that $U = \{a, b\}$. Then the family F^* of all permutations of U is $\{ab, ba\}$. Family F^* is min-wise independent, and if $w(a) = w(b) = 1$, then the method of selecting a permutation at random from F^* is a min-wise weight biased method. But suppose $w(a) = 1$ and $w(b) = 2$. In that case, we can still obtain min-wise weight bias out of U by selecting permutation ba with probability 2/3, and permutation ab with probability 1/3.

Proposition 4.4.1. *If a min-wise weight biased method is used to select a permutation* π *and we define* $h_\pi(S) = \min_{x \in S} \pi(x)$*, then* $\Pr[h_\pi(R) = h_\pi(S)] = wJ(R, S)$.

Proof. The proof is similar to the proof of basic minhash. Let m be the element of $R \cup S$ such that $\pi(m) = \min_{e \in R \cup S} \pi(e)$. If $m \in R$ and $m \in S$, then the hashes of R and S surely match, as both sets contain the element with minimum rank ($h_\pi(R) = h_\pi(S) = \pi(m)$). If m is in one set but not the other, then the hashes of R and S definitely do not match. Example: suppose $m \in R$ and $m \notin S$. Then

$h_\pi(R) = \pi(m)$, but $h_\pi(S) \neq \pi(m)$, because $m \notin S$. So $h_\pi(R) = h_\pi(S)$ if and only if $m \in R \cap S$.

Now, consider some element $x \in R \cup S$. What is the probability that $x = m$? By min-wise weight bias, $\Pr[x = m] = w(x)/\sum_{y \in R \cup S} w(y)$. In the case that $x = m$, we know from the above argument that the hashes of R and S match if and only if $x \in R \cap S$. So $\Pr[h_\pi(R) = h_\pi(S) | x = m] = 1$ if $x \in R \cap S$, and is 0 otherwise. Using these facts to compute the overall probability that $h_\pi(R) = h_\pi(S)$, we find that it is indeed $wJ(R, S)$:

$$\begin{aligned}
\Pr[h_\pi(R) = h_\pi(S)] &= \sum_{x \in R \cup S} \Pr[x = m] \Pr[h_\pi(R) = h_\pi(S) | x = m] \\
&= \sum_{x \in R \cap S} \Pr[x = m] \Pr[h_\pi(R) = h_\pi(S) | x = m] + \\
&\quad \sum_{x \in R \cup S - R \cap S} \Pr[x = m] \Pr[h_\pi(R) = h_\pi(S) | x = m] \\
&= \sum_{x \in R \cap S} \Pr[x = m] \times 1 + \sum_{x \in R \cup S - R \cap S} \Pr[x = m] \times 0 \\
&= \sum_{x \in R \cap S} \Pr[x = m] \\
&= \sum_{x \in R \cap S} \frac{w(x)}{\sum_{y \in R \cup S} w(y)} \\
&= \frac{\sum_{x \in R \cap S} w(x)}{\sum_{y \in R \cup S} w(y)} \\
&= wJ(R, S) \qquad \square
\end{aligned}$$

So now we need only search for a min-wise weight biased method of selecting a permutation of U. We propose one here based on the beta distribution.

First, we will discuss the basics of the beta distribution. All definitions and notation are derived from those used in [86]. The beta distribution is

represented by the symbol β, and it takes two parameters called α and β. Although the symbol β has two meanings, it should be clear from context which meaning is intended.

The beta distribution is defined in part with reference to two forms of the beta function. Specifically, the incomplete beta function B_x is defined as:

$$B_x(a,b) = \int_0^x u^{a-1}(1-u)^{b-1}du$$

for real numbers $a > 0$ and $b > 0$. The complete beta function B is defined as $B(a,b) = B_1(a,b)$, or:

$$B(a,b) = \int_0^1 u^{a-1}(1-u)^{b-1}du$$

A random variable distributed as $\beta(\alpha,\beta)$ has the following probability density function P and cumulative distribution function D for values of x in $[0,1]$:

$$P(x) = \frac{(1-x)^{\beta-1}x^{\alpha-1}}{B(\alpha,\beta)}$$

$$D(x) = \frac{B_x(\alpha,\beta)}{B(\alpha,\beta)}$$

For $x < 0$, $P(x) = D(x) = 0$. For $x > 1$, $P(x) = 0$ and $D(x) = 1$.

We now use these facts to prove some useful properties of the beta distribution, with the goal of proving that beta random variables can be used to select permutations with min-wise weight bias.

Lemma 4.4.2. *If independent random variables* X, Y *are distributed as* $\beta(1,\beta_x)$ *and* $\beta(1,\beta_y)$ *respectively, then* $\Pr[X < Y] = \beta_x/(\beta_x + \beta_y)$.

Proof. Let P_X be the probability density function for X, and let D_Y be the cumulative density function for Y. We can determine $\Pr[X < Y]$ by integrating $P_X(x)\Pr(x < Y)$ over all possible values of X. Knowing that $\Pr(x < Y) = 1 - D_Y(x)$, and knowing the expressions for P_X and D_Y, we can evaluate this integral, and come up with the result $\beta_x/(\beta_x + \beta_y)$. The math follows.

First, note that:

$$
\begin{aligned}
B(1,\beta) &= \int_0^1 (1-t)^{\beta-1}dt \\
&= -\frac{(1-t)^{\beta}}{\beta}\Bigg]_0^1 \\
&= \frac{1}{\beta}
\end{aligned}
$$

$$
\begin{aligned}
\Pr[X < Y] &= \int_0^1 P_X(x)(1 - D_Y(x))dx \\
&= \int_0^1 P_X(x)\left(1 - \frac{B_x(1,\beta_Y)}{B(1,\beta_Y)}\right)dx \\
&= \int_0^1 P_X(x)dx - \int_0^1 P_X(x)\frac{B_x(1,\beta_Y)}{B(1,\beta_Y)}dx \\
&= 1 - \int_0^1 P_X(x)\beta_Y B_x(1,\beta_Y)dx \\
&= 1 - \beta_Y \int_0^1 P_X(x)B_x(1,\beta_Y)dx \\
&= 1 - \beta_Y \int_0^1 \frac{1}{B(1,\beta_x)}x^{1-1}(1-x)^{\beta_x-1}B_x(1,\beta_Y)dx \\
&= 1 - \beta_Y \int_0^1 \beta_x(1-x)^{\beta_x-1}B_x(1,\beta_Y)dx \\
&= 1 - \beta_Y\beta_x \int_0^1 (1-x)^{\beta_x-1}B_x(1,\beta_Y)dx
\end{aligned}
$$

$$= 1 - \beta_y\beta_x \int_0^1 (1 - x)^{\beta_x - 1} \int_0^x (1 - t)^{\beta_y - 1} dt\, dx$$
$$= 1 - \beta_y\beta_x \int_0^1 (1 - x)^{\beta_x - 1} \left[-\frac{(1 - t)^{\beta_y}}{\beta_y} \right]_0^x dx$$
$$= 1 - \beta_y\beta_x \int_0^1 (1 - x)^{\beta_x - 1} \left(\frac{1}{\beta_y} - \frac{(1 - x)^{\beta_y}}{\beta_y} \right) dx$$
$$= 1 - \beta_x \int_0^1 (1 - x)^{\beta_x - 1} \left(1 - (1 - x)^{\beta_y} \right) dx$$
$$= 1 - \beta_x \int_0^1 ((1 - x)^{\beta_x - 1} - (1 - x)^{\beta_y + \beta_x - 1}) dx$$
$$= 1 - \beta_x \left(\int_0^1 (1 - x)^{\beta_x - 1} dx - \int_0^1 (1 - x)^{\beta_y + \beta_x - 1} dx \right)$$
$$= 1 - \beta_x \left(\left[-\frac{(1 - x)^{\beta_x}}{\beta_x} \right]_0^1 - \left[-\frac{(1 - x)^{\beta_y + \beta_x}}{\beta_y + \beta_x} \right]_0^1 \right)$$
$$= 1 - \beta_x \left(\frac{1}{\beta_x} - \frac{1}{\beta_y + \beta_x} \right)$$
$$= 1 - \left(1 - \frac{\beta_x}{\beta_y + \beta_x} \right)$$
$$= \frac{\beta_x}{\beta_x + \beta_y}$$

□

Lemma 4.4.3. *If independent random variables* X, Y *are distributed as* $\beta(1, \beta_x)$ *and* $\beta(1, \beta_y)$ *respectively, then random variable* $Z = \min(X, Y)$ *is distributed as* $\beta(1, \beta_x + \beta_y)$.

Proof. Random variable Z can take on a particular value z in two disjoint ways:

1. $X = z$ and $z \leq Y$
2. $Y = z$ and $z < X$

We can write an expression for the probability density function P_Z in terms of the the probability density functions and cumulative distribution functions P_X, P_Y, D_X, D_Y using the above two cases as a guide:

$$P_Z(z) = P_X(z)(1 - D_Y(z)) + P_Y(z)(1 - D_X(Z))$$

Using the known expressions for all of the functions in this expression, we can evaluate and find that:

$$P_Z(z) = \frac{(1-z)^{\beta_x+\beta_y-1}}{B(1, \beta_x + \beta_y)}$$

which is the probability density function for $\beta(1, \beta_x + \beta_y)$. The math follows.

First, note that:

$$\begin{aligned} B_z(1,\beta) &= \int_0^z (1-t)^{\beta-1} dt \\ &= -\frac{(1-t)^{\beta}}{\beta}\Bigg]_0^z \\ &= -\frac{(1-z)^{\beta}}{\beta} + \frac{1}{\beta} \\ &= \frac{1-(1-z)^{\beta}}{\beta} \end{aligned}$$

Now, computing $P_Z(z)$:

$$\begin{aligned} P_Z(z) &= P_X(z)(1 - D_Y(z)) + P_Y(z)(1 - D_X(Z)) \\ &= \frac{(1-z)^{\beta_x-1} z^{1-1}}{B(1,\beta_x)} \left(1 - \frac{B_z(1,\beta_y)}{B(1,\beta_y)}\right) + \frac{(1-z)^{\beta_y-1} z^{1-1}}{B(1,\beta_y)} \left(1 - \frac{B_z(1,\beta_x)}{B(1,\beta_x)}\right) \\ &= \beta_x(1-z)^{\beta_x-1}\left(1 - \beta_y B_z(1,\beta_y)\right) + \beta_y(1-z)^{\beta_y-1}(1 - \beta_x B_z(1,\beta_x)) \end{aligned}$$

$$
\begin{aligned}
&= \beta_x(1-z)^{\beta_x-1}\left(1-\beta_y\frac{1-(1-z)^{\beta_y}}{\beta_y}\right) \\
&\quad + \beta_y(1-z)^{\beta_y-1}\left(1-\beta_x\frac{1-(1-z)^{\beta_x}}{\beta_x}\right) \\
&= \beta_x(1-z)^{\beta_x-1}(1-z)^{\beta_y} + \beta_y(1-z)^{\beta_y-1}(1-z)^{\beta_x} \\
&= (\beta_x+\beta_y)(1-z)^{\beta_x+\beta_y-1} \\
&= \frac{(1-z)^{\beta_x+\beta_y-1}}{B(1,\beta_x+\beta_y)}
\end{aligned}
$$

So P_Z is exactly the probability density function of a variable distributed as $\beta(1, \beta_x + \beta_y)$. Thus, random variable $Z = \min(X, Y)$ is distributed as $\beta(1, \beta_x + \beta_y)$. □

Corollary 4.4.4. *If we are given a series of independent random variables* $X_1, \ldots, X_n$*, with each* X_i *distributed as* $\beta(1, \beta_i)$*, then random variable* $Z = \min(X_i)$ *is distributed as* $\beta(1, \sum \beta_i)$*.*

Proof. The case where $n = 2$ is simply a restatement of Lemma 4.4.3. The general case follows by induction. □

Now that we have proven some properties of the beta distribution, we can continue to prove our main theorem, which directly suggests a min-wise weight biased method for selecting a permutation.

Theorem 4.4.5. *If a collection of values* $\{v_1, v_2, \ldots, v_n\}$ *is drawn from a set of random variables* $\{V_1, V_2, \ldots, V_n\}$ *with each random variable* V_i *distributed as* $\beta(1, \beta_i)$*, then for a given* $v_x \in v_i$*,* $\Pr[\min_i(v_i) = v_x] = \beta_x / \sum \beta_i$*.*

Proof. We know that $\Pr[\min(v_i) = v_x] = \Pr[v_x < \min_{i \neq x}(v_i)]$. By Corollary 4.4.4, we have that $M = \min_{i \neq x}(v_i)$ is a random variable distributed as $\beta(1, \sum_{i \neq x} \beta_i)$. By Lemma 4.4.2, $\Pr[v_x < M] = \beta_x / (\beta_x + \sum_{i \neq x} \beta_i) = \beta_x / (\sum \beta_i)$. Therefore, $\Pr[\min(v_i) = v_x] = \beta_x / (\sum \beta_i)$. □

Theorem 4.4.5 suggests the following min-wise weight biased method of selecting a permutation. For each element x in U, we select a random value v_x

according to the distribution $\beta(1, w(x))$. We then define π as the order of the elements of U according to their v_x values. According to Theorem 4.4.5 for a set S, $\Pr[\min_{x \in S} \pi(x) = \pi(x)] = w(x) / \sum_{y \in S} w(y)$, and therefore this method is min-wise weight biased. Since the process for generating a permutation is entirely random, we can create as many permutations as we see fit. According to Proposition 4.4.1, we can use these permutations to generate hash functions that satisfy $\Pr[h(R) = h(S)] = wJ(R, S)$.

Unfortunately, if $|U|$ is large, then it can take a lot of memory to store these permutations. What we need is a compact and efficient method of computing values based on this scheme. We will discover such a method by returning to the use of random linear functions.

The function $f(x) = ax + b \mod P$ gives a random number from the discrete uniform distribution. By normalizing, we can approximate a value from the continuous uniform distribution, and then we can use the inverse CDF method to create a beta-distributed value. The inverse of the cumulative density function for $\beta(1, \beta)$ is $D^{-1}(x) = 1 - \sqrt[\beta]{1 - x}$. Noting that $1 - X$ has the same distribution as uniformly distributed X, we can simply generate a beta weighted value from uniformly distributed X by computing $1 - \sqrt[\beta]{X}$.

The inverse CDF method allows us to derive in the following hash function:

$$h(S) = \min_{x \in S} \left(1 - \sqrt[w(x)]{\frac{ax + b \mod P}{P - 1}} \right)$$

or equivalently:

$$h(S) = \max_{x \in S} \sqrt[w(x)]{\frac{ax + b \mod P}{P - 1}}$$

These functions are based upon the linear function technique commonly used in minhash. As briefly discussed in Section 4.2.2, the linear function technique for minhash is not min-wise independent, but it works very well in practice. In the same vein, we do not attempt to prove the correctness of the linear function technique for weighted minhash. Instead, we demonstrate

experimentally (in Section 4.6) that this function does indeed satisfy (or at least closely approximate) the desired property:

$$Pr[h(R) = h(S)] = wJ(R,S)$$

Algorithm 4.4 shows the pseudocode for an implementation of this scheme. It is identical to Algorithm 4.1 except for the computation of the linear function, which is now augmented with the computation of the w(x)th root of the function.

```
Input:   S: the set to generate a signature for
         l: the desired length of the signature
Output:  an l-length signature for S
 1: RealWeightedMinhash(S, l)
 2: randstream ← new RandomStream(FIXED_SEED)
 3: for i = 1 to l do
 4:   a ← random integer from randstream in [1, P)
 5:   b ← random integer from randstream in [0, P)
 6:   // compute minimum of 1 − w(x)√((ax+b mod P)/(P−1))
 7:   minhash ← P
 8:   for all x ∈ S do
 9:     current ← 1 − w(x)√((ax+b mod P)/(P−1))
10:     if minhash > current then
11:       minhash ← current;
12:     end if
13:   end for
14:   sig[i] ← minhash
15: end for
16: return sig
```

Algorithm 4.4: Weighted Minhash Signature algorithm (Real Weights)

4.4.3 Limitation: Weighted Sets

As mentioned earlier, our real weights scheme works perfectly when items in U have an inherent weight. However, it does not work for the case where an

element can appear in different sets with different weights. A simple example shows that general weighted sets are not supported by this scheme.

Suppose that weighted sets R and S are defined as follows: $R = \{x : 1.0\}$ and $S = \{x : 2.0\}$. The minimum weight of element x is 1, and the maximum weight is 2. Therefore, using the above definition, $wJ(R, S) = 1/2$. However, if we apply our weighted minhash technique from Section 4.4.2, then $h(R) = \frac{ax+b \mod P}{P-1}$ and $h(S) = \sqrt{\frac{ax+b \mod P}{P-1}}$. So h(R) only equals h(S) when $ax + b = P - 1$, an event that occurs rarely.

4.5 Composed Data Types

We now have good LSH hash families for maps, sets, and sets with weighted elements. However, many things cannot be modeled as a simple set or map. For example, if the values of a map are themselves maps (or sets), then we would like a method of comparing such maps considering that values may have varying shades of similarities (rather than black or white equality).

In this section, we consider data types that have no generally accepted similarity measure or LSH family, but are compositions of data types that do have an appropriate LSH family. As an example, consider a data type that models a grocery store as a set of fruits and a set of vegetables:

```
class GroceryStore
{
  Set fruits;
  Set vegetables;
}
```

Our notation for defining GroceryStores will be similar to the notation for defining maps, but we stress that these data structures are different from maps. As opposed to a map, a GroceryStore has a fixed schema with two fields. Further, in this section we are considering that the values of those fields may have degrees of similarity other than strict equality or inequality.

Let store s_1 = {fruits : {apples, oranges}, vegetables : {carrots, celery}}. That is, the fruits that s_1 sells are apples and oranges, and the vegetables sold there are carrots and celery. Similarly, suppose that another store s_2 = {fruits : {oranges}, vegetables : {celery, radishes}}. Based on the Jaccard coefficient of the fruit sets, the similarity of s_1 and s_2 is 1/2. Based on vegetables, however, the similarity is 1/3. What ought to be the similarity of the stores as a whole?

One simple method of finding a combined similarity is to notice that the fruit and vegetable components are both sets, so we could treat a store as the union of its fruit and vegetable sets. From there, we could simply use the Jaccard coefficient as our similarity measure, and just use basic minhash as in Section 4.2.2. One potential problem with this method is that it raises the importance of large sets: if most stores carry 100 fruits, while only carrying 10 vegetables, then vegetables will not be very important if everything is bundled together in one set. This bias could be desirable or undesirable, depending on the application. Regardless, treating the composed data type as a single set won't work in the general case. In the general case, we cannot rely on every component of a composed data structure being a set. We may have data structures that combine sets, maps, vectors, and others, each with their own similarity measures and LSH functions.

Taking a more "black-box" approach to the problem, we have a data structure with two components, which each have their own appropriate similarity measures. A simple method of combining the similarities is to use the mean of the two similarity values. For example, if the similarity of the fruit sets is f and the similarity of the vegetable sets is v, then we may consider the similarity of the two stores to be $(f + v)/2$. This is, of course the arithmetic mean. For some purposes, we may want to use the geometric mean instead. That is, we would define the similarity of the two stores as $\sqrt{fv}$. In this section, we detail methods of using either type of mean.

Let us define our composed data type as having fields $f_1, \ldots, f_n$. Each field f_i has a similarity measure sim_i, as well as a hash function family H_i that satisfies

the LSH property for sim_i. For each field, we will make use of a hash function $h_i \in H_i$ to generate a hash value for the field. We will use a dot notation to refer to the fields of a particular item (i.e. $s.f_1$ refers to the value of field f_1 in element s. Tying this notation to our example, the fields of a GroceryStore are f_1 = fruits and f_2 = vegetables. To keep things simple, we will use the Jaccard coefficient and basic minhash as the similarity and LSH functions for both fields. We'll use these concepts to construct a composed hash function h_c for the composed data type.

4.5.1 Arithmetic Mean

In this section, we demonstrate how to generate a family of hash functions that satisfies the LSH property with similarity defined as the arithmetic mean of the underlying field similarities. Formally, we define the similarity of two objects as $sim(x, y) = \sum sim_i(x.f_i, y.f_i)/n$. (Recall that n is the number of fields in the data type.)

To create a hash function that works with the arithmetic mean of the fields' similarities, we use a probabilistic method, which we demonstrate using our fruits and vegetables example. The first step in generating a hash function will be to flip a coin. If it's heads, we define $h_c(s) = h_1(s.fruits)$, so the hash is just the hash of the fruits field. If the flip comes up tails, we define $h_c(s) = h_2(s.vegetables)$, using only the vegetables field. Note that the coin flip is used to generate h_c. It is not computed inside h_c. That is, once h_c is defined according to the coin flip, it will always return the same value for a given input s. The function h_c will not randomly choose between the fruits and vegetables fields each time it is called.

If we have an unbiased coin flip, then the probability of hashing fruits or vegetables is 1/2. In the case that fruits are chosen, then the composed hash of two stores will match if and only if the fruit hashes match. Similarly if vegetables are chosen. These facts allow us to compute the probability that the composed hash of two stores matches. Specifically, $Pr[h_c(s_1) =$

$h_c(s_2)] = \frac{1}{2}sim_1(s_1.fruits, s_2.fruits) + \frac{1}{2}sim_2(s_1.vegetables, s_2.vegetables)$, which is the arithmetic mean of the two field similarity values. It is clear that this method can be extended to a weighted arithmetic mean simply by adding a bias to the coin flip.

```
Input:   S: the data structure to generate a signature for
         l: the desired length of the signature
         SigFunctions: an array of signature functions, one for each field
Output:  an l-length signature for S
 1: ArithmeticMeanHash(S, l, SigFunctions)
 2: randstream ← new RandomStream(FIXED_SEED)
 3: for all fields f_i do
 4:   sigs_i ← SigFunctions[i](S.f_i, l)
 5: end for
 6: for j = 1 to l do
 7:   f ← random integer from randstream in [1, n]
 8:   sig[j] ← sigs_f[j]
 9: end for
10: return sig
```

Algorithm 4.5: Arithmetic Mean Composed Signature algorithm

Algorithm 4.5 shows the pseudocode for this scheme. It takes in an array of signature-generating functions as input, which it uses to compute signatures on the individual fields of the data structure. It then generates an l-length signature by picking l random integers. The jth random integer determines which fields' hash to use for the jth component in the signature. We note that for each of the l signature components, this code generates a hash for each field, but then only makes use of one field. This extra work is inefficient, but we leave the code written this way for clarity.

4.5.2 Geometric Mean

For some applications, we may wish to define similarity as the geometric mean of the similarities of the individual fields. The geometric mean of a series of n numbers is the nth root of the product of the numbers. So we can define

the similarity of two objects as $sim_g(x, y) = \sqrt[n]{\prod_i sim_i(x.f_i, y.f_i)}$. Unfortunately, we have been unable to find a family of hash functions that satisfies the LSH property for this similarity metric. As we will explain, however, we may be able to use an LSH family for a different (but related) similarity measure in order to solve the problem of finding pairs of items with a high value for $sim_g(x, y)$.

Consider the related function $sim_p(x, y) = sim_g(x, y)^n = \prod_i sim_i(x.f_i, y.f_i)$. The function sim_p has the useful property that it increases as sim_g increases. Therefore, if we are looking for pairs where $sim_g(x, y)$ is greater than a threshold t, then we can translate our search by looking for pairs where $sim_p(x, y) > t'$, for a properly selected t'. By setting $t' = t^n$, we have the property that $sim_p(x, y) > t'$ if and only if $sim_g(x, y) > t$. The LSH algorithm can efficiently find pairs with similarity over a given threshold. Therefore, if we have a family of hash functions that satisfies the LSH property with similarity defined as $sim_p(x, y)$, then we can use the LSH algorithm to find pairs with $sim_p(x, y) > t^n$. All of the pairs found will also satisfy $sim_g(x, y) > t$. So let us now shift gears and look for a family that satisfies the LSH property for sim_p.

It turns out that an LSH family for sim_p is readily available. Starting with our example, let us define the composed hash function for a Grocery-Store s as returning a tuple: $h_c(s) = (h_1(s.fruits), h_2(s.vegetables))$. For two stores s_1 and s_2, the probability that $h_c(s_1) = h_c(s_2)$ is the probability that they match on both h_1 and h_2. Since h_1 and h_2 satisfy the LSH property for sim_1 and sim_2 respectively, the probability that both hashes match is $sim_1(s_1.fruits, s_2.fruits) \times sim_2(s_1.vegetables, s_2.vegetables)$.

Given this example, we can now formally define a composed hash function h_p for any generic data type. For each field f_i, select a hash function h_i from H_i. The composed hash function h_p is then the tuple:

$$h_p(x) = (h_1(x.f_1), h_2(x.f_2), \ldots, h_n(x.f_n))$$

We can use the definition of h_p above to generate a hash function family H_p of all possible functions h_p.

Theorem 4.5.1. *The hash function family* H_p *satisfies the LSH property for the similarity measure* $sim_p(x, y)$.

Proof. The proof follows simply from simple multiplication of the independent probabilities of individual components of the hash functions matching. □

We can therefore use the LSH algorithm to find pairs of items that have a geometric mean similarity over a threshold t. To do so, we use the hash family H_p to find pairs with a product similarity over the threshold t^n.

```
Input:   S: the data structure to generate a signature for
         l: the desired length of the signature
         SigFunctions: an array of signature functions, one for each field
Output:  an l-length signature for S
1: ProductHash(S, l, SigFunctions)
2: for all fields f_i do
3:    sigs_i ← SigFunctions[i](S.f_i, l)
4: end for
5: for j = 1 to l do
6:    sig[j] ← (sigs_1[j], sigs_2[j], . . . , sigs_n[j])
7: end for
8: return sig
```

Algorithm 4.6: Product Composed Signature algorithm (for Geometric Mean)

Algorithm 4.6 shows pseudocode for an implementation of this scheme. This function takes in an array of signature functions for computing the LSH signatures of the individual fields of the composed data structure. The code then computes the signatures of the individual fields and uses these signatures to build a signature for the entire structure.

We note that this scheme can be extended to weighted geometric mean (for integer weights) as follows. Let the function w(f) refer to the weight of a field f. The weighted geometric mean of the similarities of the fields is then defined as:

$$sim_{wg}(x, y) = \left(\prod sim_i(x.f_i, y.f_i)^{w(f_i)} \right)^{1/\sum w(f_i)}$$

Just as with regular geometric mean, we will not define an LSH family for sim_{wg} directly. Instead, we will define an LSH family for the related function sim_{wp}:

$$sim_{wp}(x, y) = \prod sim_i(x.f_i, y.f_i)^{w(f_i)}$$

We extend the technique used above simply by selecting a number of hash functions from each field's family equal to that field's desired weight. For example, if we wanted to consider fruits twice as important as vegetables, then we could define $h_c(s) = (h_{1,1}(s.fruits), h_{1,2}(s.fruits), h_2(s.vegetables))$, where $h_{1,1}$ and $h_{1,2}$ are both in H_1.

We can now formally define a composed hash function h_{wp} for the case with integer weights. For each field f_i, select $w(f_i)$ hash functions $h_{i,1}, \ldots, h_{i,w(f_i)}$ from H_i. The composed hash function h_p is then the tuple:

$$\begin{aligned} h_{wp}(x) = (\quad & h_{1,1}(x.f_1), \ldots, h_{1,w(f_1)}(x.f_1) \quad , \\ & h_{2,1}(x.f_2), \ldots, h_{2,w(f_2)}(x.f_2) \quad , \\ & \vdots \quad , \\ & h_{1,n}(x.f_n), \ldots, h_{1,w(f_n)}(x.f_n) \quad) \end{aligned}$$

We can use the definition of h_{wp} above to generate a hash function family H_{wp} of all possible functions h_{wp}.

Theorem 4.5.2. *The hash function family* H_{wp} *satisfies the LSH property for the similarity measure* $sim_{wp}(x, y)$.

Proof. The proof follows simply from simple multiplication of the independent probabilities of individual components of the hash functions matching. □

We can therefore use the LSH algorithm to find pairs of items that have a weighted geometric mean similarity over a threshold t. To do so, we use the hash family H_{wp} to find pairs with a product similarity over the threshold t^x where $x = \sum(w(f_i))$.

One limitation of this method (with or without weights) is that concatenating hashes to create a composed hash can result in large hash sizes. If an application (such as LSH) requires hashes to be stored in limited space (e.g. 32 bits), then we may use any standard hashing technique to compress the result of h_c into a more compact value, with negligible effect on the probability of a hash collision.

Another limitation of this method is that the efficiency of algorithms such as LSH may depend on the threshold used. Thus, searching for stores with threshold above t^2 may be much slower than searching for those above the (higher) threshold t. We point out this limitation, but do not give a solution. In the case that this issue becomes a serious limitation, we suggest considering the use of the arithmetic mean.

4.6 Experimental results

This chapter describes similarity hashing techniques for widely varying types of data. The techniques for hashing maps and composable data structures are proven correct and have predictable performance, so no experimental evaluation is necessary. The technique of Section 4.4.2 for hashing sets with weighted values, though based on proven theory, is only practically implemented as an approximation to the theory. As such, it requires evidence of correctness. Further, it is in competition with other schemes that can handle weighted sets, so we can use experiments to compare the performance of these schemes.

We evaluate our technique on the two metrics of correctness and performance:

1. *Correctness*—For our schemes to be useful, they must satisfy the LSH property for the proper similarity measure. We can experimentally verify this by generating item pairs, calculating their similarity, and then experimentally verifying that the probability their hashes match is equal

to their similarity.

2. *Performance*—Computing hash functions is often expensive, and therefore the performance of algorithms that make use of LSH hash functions may depend greatly on the time to compute individual hashes. We will measure the runtime of various hashing schemes as an indicator of the performance.

We start by evaluating the performance of the weighted minhash schemes described in Section 4.4. We implemented the integer weight scheme (of Section 4.4.1), the beta distribution based scheme (of Section 4.4.2), and the consistent weighted sampling scheme from Manasse et al. [59] in Java and compared performance numbers. We ran our performance experiments on a system with an Intel Core 2 Quad CPU Q6600 2.40GHz and 4GB of RAM, running Linux kernel 2.6.22 x86_64 and Sun's Java SDK version 1.6.0_03-b05.

To compare the schemes, our test program generates random sets and measures the time it takes to compute their hashes. Each set consists of 1000 random integer values, all values assigned with a fixed weight w. Since the schemes differ in how efficiently they deal with weighted values, we ran tests with different sets, each with a different value of w to observe the effects.

We varied w from 1.0 to 50.0 in increments of 0.25. Since one scheme only works with integer weights, the weights were rounded down to an integer for that scheme. For each value of w, we used each scheme to hash 100 random sets and then computed the average time per set. Figure 4.2 illustrates the results for $w \leq 20$.

The line labeled Iteration shows the runtime of the iteration-based integer weight scheme. The line shows the expected linearly increasing stair-step effect that we expect from a linear algorithm that deals only with integer weights. The line labeled Beta shows the performance of the real weights scheme based on the beta distribution. We expected the Iteration scheme to be faster for small values of w since it involves less complicated arithmetic. Surprisingly, the Beta scheme is faster than the Iteration scheme in

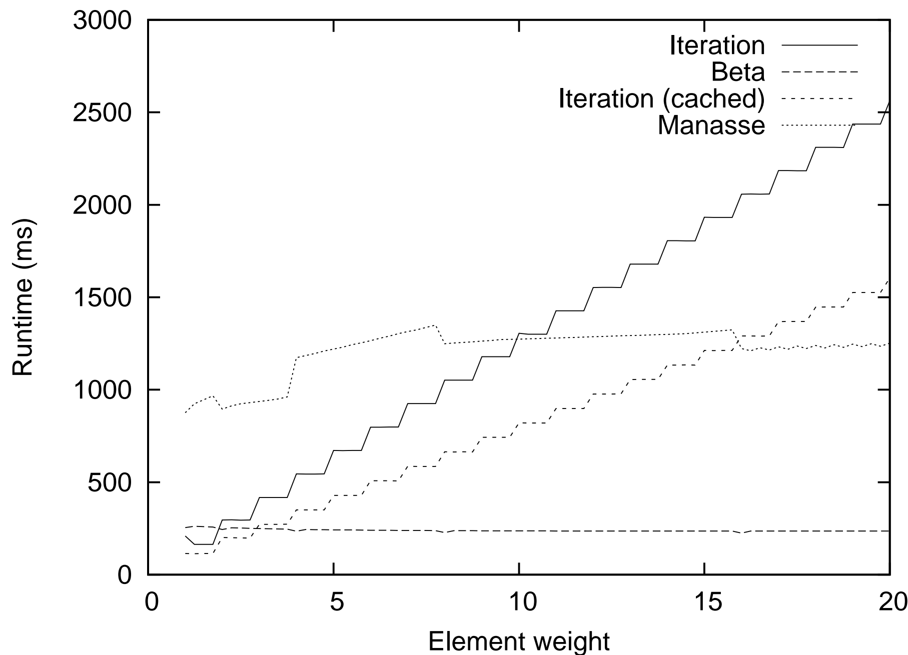

Figure 4.2: Runtime of Weighted Schemes vs. Weight Values

every case except where the weight w = 1 (when rounded down). Further investigation revealed that the recomputation of the random a and b values in Algorithm 4.3 was slowing down the Iteration scheme (relative to the Beta schemes need to only generate these values once in Algorithm 4.4). We decided to remove this extra overhead by precomputing the random values offline and storing them in an array in memory. The results of a run with this optimization are shown in the line labeled "Iteration (cached)". This optimization does indeed speed things up, but not too much relative to the Beta scheme. Even with this optimization, the Iteration scheme breaks even with the Beta scheme at about w = 3.0.

The Manasse scheme exhibits less predictable behavior, but we know it is expected constant time based on the analysis in [59]. Using the data shown, we can estimate that the Beta scheme is about 5 times faster than the Manasse scheme. We should point out that some of the optimizations in the Manasse scheme will cause it to be come more and more efficient as the size of the

set grows. (That is, the amount of time spent per set item will decrease on average as the size of the set increases.) We intentionally used a large set size (1000) to ensure that the Manasse scheme was able to make effective use of its optimizations.

The hawk-eyed reader may notice a very slight dip in the runtime of the Beta scheme at values 2, 4, and 8. The raw data also shows slight dips at 16 and 32. These values are powers of two, and the computations involved in the Beta scheme may be more efficient at these values. The fact that some values of w are particularly good for the Beta scheme raises the question of whether some values of w might be particularly bad for it—a case that might be missed by choosing weights evenly spaced across the number line. To ensure that we were not missing any such spikes, we ran experiments that generated uniform random values of w. We did not find any significant increases in the runtime of the Beta scheme, even with random values of w.

This experiment reveals that the Beta scheme's runtime is not dependent on the weight values, and is faster than the Iteration scheme whenever element weights are larger than 3. This fact makes a compelling case for use of the Beta scheme, as its support of real valued weights makes it more powerful than the Iteration scheme.

While the Beta scheme is based on proven theory, it is not an exact implementation of the theoretically proven scheme. Therefore, we ran an experiment to demonstrate the correctness of the technique. Our experiment computed signatures of length $l = 1000$ using the Beta scheme on randomly generated pairs of sets. We then compared the number of matches in the signatures to the number of matches we would expect based on the pair's actual weighted Jaccard coefficient.

As mentioned above, we generate pairs of sets. For each pair, we will call the two sets R and S. The sets are randomly generated according to the following strategy. First, we generate a random target similarity t between 0 and 1. Given the target t, we will attempt to create sets R and S such that $wJ(R, S) = t$. We then generate a random value w_x between 0 and 100, which

will be the total weight of elements in $R \cup S - R \cap S$. We first build R and S by randomly generating new set elements with random integer values and random real weights between 0 and 10. For each new set element, we flip a coin to decide if it should go into R or S. We end the process when the total weight of the two sets is $w_{total} > w_x - 10$, at which point we create a final new element with weight $w_x - w_{total}$ to increase the total weight of R and S to exactly w_x. We randomly add this element to either R or S.

Now we have two sets whose total weight is w_x, but they do not intersect. However, we can compute $w_a = w_x * t/(1-t)$ which is the total weight we desire in $R \cap S$. We build up a new set A with weight w_a in similar fashion. Set A is filled with random elements (not already in R or S) with weights between 0 and 10 until the total weight in A is w_a. Now we assign $R \leftarrow R \cup A$ and $S \leftarrow S \cup A$. Now $R \cap S$ has weight w_a, and $R \cup S$ has weight w_x. With simple arithmetic we get that $wJ(R, S) = t$.

We note that as a result of this scheme, low-similarity pairs generally have more items than high similarity pairs: since the weight of the intersection is fixed, we must generate a larger number of random elements outside of the intersection to force the similarity to be low. We do not believe this bias should have any effect on the results of our experiment.

Figure 4.3 shows the results of our experiments on 1000 random pairs. In this figure we plot a point for each pair of sets indicating the similarity of the sets on the horizontal axis, and the number of hash matches on the vertical axis. Since the signature length is 1000, we expect the number of matches at any similarity value s to be 1000s. We plotted a solid line indicating this expected result along with two dashed lines indicating a 99.7% confidence interval (3 standard deviations from expected according to the binomial distribution). With a confidence interval of 99.7%, we would expect about 3 points out of 1000 to fall outside the interval, and in fact we find exactly 3 points outside of the interval. A linear regression on these points yields a line (not shown) with slope 999.7 and intercept 0.42, quite close to the expected slope

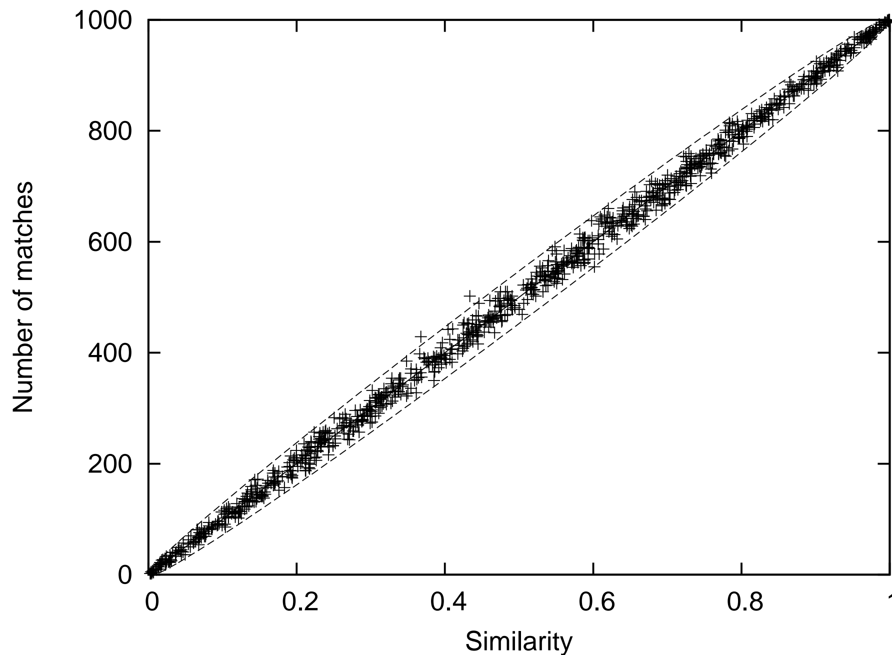

Figure 4.3: Measured hash matches vs. Weighted Jaccard similarity

of 1000 and intercept of 0. Given how closely these results meet our expectations, we conclude that the Beta scheme does in fact closely approximate the LSH property for weighted Jaccard similarity.

4.7 Related Work

The related work falls into three categories:

1. Works that can be used in conjunction with our schemes to find items of high similarity
2. Works that, in the same vein as this chapter, provide LSH hash functions for various data types and similarity measures
3. Works that describe alternative approaches to similarity search

The most obvious member of the first category is the LSH algorithm itself, described in [4]. Bawa et al. described a self-tuning LSH index in [8]. In addition, Bhagwat et al. describe a method of partitioning document indexes using hash functions satisfying the LSH property in [13]. All of these works can use any family of LSH functions, so they work hand in hand with the schemes presented in this chapter.

Clearly related, and a basis for all the schemes in this chapter is the work on Minhash itself. In [21], Broder et al. studied the technique of using min-wise independent permutations, as well as the linear functions technique used in this chapter. The authors left open the problem of developing an approximately min-wise independent family of hash functions where a function can be specified in a small number of bits. This problem was solved by Indyk, and the resulting techniques in [52] could be converted to support weights using the methods in Section 4.4.2.

Further, there are many papers that define LSH hash function families for similarity measures other than the Jaccard coefficient. The LSH survey paper [4] provides an LSH family for Hamming and Euclidean distances, and [23] defines LSH families for cosine similarity and the earth mover's distance. Reference [23] also has a section defining an LSH family for weighted sets based on rounding techniques. The technique in [23] does not easily lend itself to a practical setting, but Manasse et al. constructed a practical implementation in [59]. We compared the performance of the Manasse scheme to ours in this chapter.

Finally, there is great deal of work on similarity search and the near neighbor problem. Unfortunately, most of these techniques break down as the dimensionality of the data increases. A discussion of many such techniques (multiple key indexes, kd-trees, quad trees, and R-trees) is available in [40]. More recently, work on set-similarity join [26, 5] has revealed techniques for efficiently computing the exact answer to set-similarity join queries. Arasu provides a comparison of these techniques with LSH-based techniques in [5].

4.8 Conclusion

We have presented techniques for applying minhash to three new data types: maps, sets with weighted values, and composed data types. For each of these techniques, we demonstrated the correctness by proving that they satisfy the LSH property for their respective similarity measures.

For sets with weighted values, we identified the Manasse scheme, the only other known scheme that can handle sets with weighted values. The Manasse scheme supports the more general case of weighted sets, but our experimental results demonstrate that our scheme is about 5 times faster than the Manasse scheme. We therefore recommend the use of this scheme when values have consistent weights (as they might when using IDF weighting).

Chapter 5

Entity Resolution with Confidences

5.1 Introduction

Often, numerical confidences (or data quality) play a role in entity resolution. For instance, the input records may come from unreliable sources and have confidences or quality associated with them. The comparisons between records may also yield confidences that represent how likely it is that the records refer to the same real-world entity. Similarly, the merge process may introduce additional uncertainties, as it may not be certain how to combine the information from different records. In each application domain, the interpretation of the quality or confidence numbers may be different. For instance, a confidence number may represent a "belief" that a record faithfully reflects data from a real-world entity, or it may represent how "accurate" a record is.

Relatively little is known about how to efficiently deal with confidences in entity resolution. Specifically, confidences may make the ER process computationally more expensive, as compared to a scenario where confidences are not taken into account. For instance, without confidences, the order in which records are merged may be unimportant, and this property can be used to find efficient ER strategies. However, confidences may make order critical. For instance, say we merge r_1 to r_2 and then to r_3, giving us a record r_{123}. If r_1 and r_2 are "very similar", we may have a high confidence in the intermediate result,

which then gives us high confidence in r_{123}. However, say we merge r_1 to r_3 and then to r_2, giving us record r_{132}. In this case, r_1 and r_3 may not be "that similar", leading to a lower confidence r_{132}. Records r_{123} and r_{132} may even have the same attributes, but may have different confidences because they were derived differently. Thus, ER must consider many more potential derivations of composite records.

Our goal in this chapter is to explore ways to reduce the high computational costs of ER with confidences. We wish to achieve this goal without making too many assumptions about the confidence model and how confidences are computed when record are merged. Thus, we continue to use the generic *black-box model* for the functions that compare records, that merge records, and that compute new confidences. We will then postulate properties that these functions may have: if the properties hold, then efficient ER with confidences will be possible. If they do not hold, then one must run a more general version of ER (as we will detail here). Since we use generic match and merge functions, the algorithms we present can be used in many domains. All that is required is to check what properties the match and merge functions have, and then to select the appropriate algorithm.

The contributions of this chapter are the following:

- We define a generic framework for managing confidences during entity resolution (Sections 5.2 and 5.3).
- We present Koosh, an algorithm for resolution when confidences are involved (Section 5.4).
- We present three improvements over Koosh that can significantly reduce the amount of work during resolution: domination, packages and thresholds. We identify properties that must hold in order for these improvements to be achievable (Sections 5.5, 5.6, and 5.7).
- We evaluate the algorithms and quantify the potential performance gains (Section 5.8).

The work in this chapter has been published in CleanDB 2006 by Menestrina, Benjelloun, and Garcia-Molina [65].

5.2 Model

Each record r now has a *confidence*, which we refer to as r.$\mathcal{C}$. We will refer to the remaining non-confidence parts of the record as the *attributes* of a record, r.$\mathcal{A}$. We will not specify any particular representation for r.$\mathcal{A}$, but for illustration purposes, we can think of r.$\mathcal{A}$ as a set of label-value pairs. For example, the following record may represent a person:

0.7 [name : Fred, age :{45, 50}, zip : 94305]

In this example record, we write r.$\mathcal{C}$ (0.7 in this example) in front of the attributes.

Note that we are using a single number to represent the confidence of a record. We believe that single numbers (in the range 0 to 1) are the most common way to represent confidences in the ER process, but more general confidence models are possible. For example, a confidence could be a vector, stating the confidences in individual attributes. Similarly, the confidence could include lineage information explaining how the confidence was derived. However, these richer models make it harder for application programmers to develop merge functions (see below), so in practice, the applications we have seen all use a single number.

In this chapter, we continue to use the pairwise model defined in Section 1.3. One might expect it to be necessary to generalize our model to account for confidences. Specifically, one might assume that the pairwise match function should return a confidence value instead of a simple Boolean. While extending the model in this fashion may make sense, we find that the extension is unnecessary. Instead, we assume that the "closeness" of a match will be incorporated into how the merge function assigns confidence values to merged records.

The pairwise match and merge are generally not arbitrary functions, but have some properties that we can leverage to enable efficient entity resolution. Chapter 1 introduced the four ICAR properties, of which we will assume two for the remainder of this chapter. We restate these properties here:

- *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 $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.
- *Reflexivity/Idempotence:* $\forall r$, $r \approx r$ and $\langle r, r \rangle = r$.

We expect these properties to hold in almost all applications. Some readers may wonder if merging two identical records should really give the same record. For example, say the records represent two observations of some phenomenon. Then perhaps the merged record should have a higher confidence because there are two observations? The confidence would only be higher if the two records represent independent observations, not if they are identical. We assume that independent observations would differ in some way, e.g., in an attribute recording the time of observation. Thus, two *identical* records should really merge into the same record.

The properties of associativity and representativity generally do *not* hold when confidences are involved. As argued in Section 5.1, merging records in different orders may lead to different confidences. Similarly, when a record r_1 merges with r_2 forming r_3, we cannot discard r_1 and r_2, as they may have higher confidence that r_3 (see example below). Because we do not assume associativity and representativity, in this chapter we require as a starting point a more general algorithm than the Swoosh algorithm. This general algorithm is called Koosh (see Section 5.4).

5.3 Generic Entity Resolution with Confidences

Consider the following records:

$r_1 = 0.8$ [name : Alice, areacode : 202]

r_2 = 0.7 [name : Alice, phone : 555-1212]

The merge of the two records might be:

r_{12} = 0.56 [name : Alice, areacode : 202, phone : 555-1212]

In this case, the merged record has all of the information in r_1 and r_2, but with a lower confidence (the merge function appears to have multiplied the confidence values of r_1 and r_2). So dropping the original two records would lose information. That is, if we drop r_1 we would no longer know that we are quite confident (confidence = 0.8) that Alice's area code is 202. The new record r_3 connects Alice to area code 202 with lower confidence. Therefore, to be conservative, the result of an entity resolution algorithm must contain the original records as well as records derived through merges. This intuition is indeed captured in the definition of ER(R) in Section 1.3.2, as r_{12} does not dominate r_1 and r_2, so the base records will not be removed from the result. To start, we will explore the use of ER(R) when domination is left out of the picture. (Or more formally, the $\leq$ relation is false for all pairs.) To refer to this case unambiguously, we use the term GER.

The result ER(R) is known to be unique in the presence of the ICAR properties, but we do not assume all four ICAR properties in this chapter. To establish the uniqueness of GER(R), we need to introduce derivation trees for records. Note that a derivation tree is not a data structure that is used in our algorithms. It is instead a concept that we use as a tool for proving facts about our algorithms.

Definition 5.3.1. *A* derivation tree D *for record* r *is a binary tree whose nodes represent records. The root represents* r*, and for any node, its children represent the two records that merged to produce it (e.g., if* a *and* b *are children of* c*, then* $a \approx b$ *and* $\langle a, b \rangle = c$*). The leaves of* D*, denoted* L(D) *represent base records. The derivation tree of a base record contains only that record as the root.*

Intuitively, the derivation tree D explains how r was derived. Note incidentally that two nodes in D can represent the same record. For example, say

that a and b merge to produce d; a and c merge to yield e; and d and e merge into f. In this example two separate leaf nodes in D represent a.

Definition 5.3.2. *Given a base set of records* R*, a record* r *is* well-formed *if there exists a derivation tree* D *with root* r *such that* $L(D) \subseteq R$.

Proposition 5.3.3. *Let* r *be a well-formed record, and let* D *be its derivation tree* ($L(D) \subseteq R$)*. Then every internal node of* D *represents a well-formed record.*

Proof. Consider an internal node of D that represents a record s. The subtree of D with s at the root is a derivation tree for s. Since s has a derivation tree, it is a well-formed record by definition. □

Lemma 5.3.4. *Given a set* R*, every record in* GER(R) *is well-formed.*

Proof. Given a set S that satisfies the properties of Definition 1.3.2 and contains records that are not well-formed, one can show that removing the non-well-formed records from S yields a smaller set that still satisfies the properties of the definition. □

Theorem 5.3.5. *The solution to the ER problem is unique.*

Proof. Suppose that S_1 and S_2 are both solutions but different. Consider a record $r \in S_1$ that is not in S_2. Since r is well-formed (previous lemma) and in S_1, it has a derivation tree D with base records as leaves. Since S_2 is also an ER solution, then every internal record in D, as well as the root r, must be in S_2, a contradiction. Thus, every record in S_1 must be in S_2. We can similarly show the converse, and hence S_1 must equal S_2, a contradiction. □

Intuitively, GER(R) is the set of all records that can be derived from the records in R, or from records derived from them. A natural "brute-force" algorithm (BFA) for computing GER(R) is given in Algorithm 5.1. The following theorem states the correctness of BFA.

Theorem 5.3.6. *For any set of records* R *such that* GER(R) *is finite, BFA terminates and correctly computes* GER(R).

```
1: input: a set R of records
2: output: a set R′ of records, R′ = GER(R)
3: R′ ← R; N ← ∅
4: repeat
5:    R′ ← R′ ∪ N; N ← ∅
6:    for all pairs (r_i, r_j) of records in R′ do
7:       if r_i ≈ r_j then
8:          merged ← ⟨r, r′⟩
9:          if merged ∉ R′ then
10:            add merged to N
11:         end if
12:      end if
13:   end for
14: until N = ∅
15: return R′
```

Algorithm 5.1: The BFA algorithm for GER(R)

Proof. First, note that any record added to R′ has a well-formed derivation tree, so at any time in BFA, $R' \subseteq GER(R)$. From this fact we see that BFA must terminate when GER(R) is finite. (If BFA does not terminate, then N is non-empty at the end of every iteration (line 14). This means that at least one new element from GER(R) is added at each iteration, which is impossible if GER(R) is finite.)

Since we know that $R' \subseteq GER(R)$ when BFA terminates, we only need to show that $GER(R) \subseteq R'$ to prove that R′ = GER(R) when BFA terminates.

Thus, we next show that every record $r \in GER(R)$ is generated by BFA. Since $r \in GER(R)$, r has a well-formed derivation tree D.

We define the level of a node $x \in D$, $l(x)$, as follows: If x is a base record ($x \in R$), then $l(x) = 0$. Otherwise, $l(x) = 1 + \max(l(c_1), l(c_2))$, where c_1 and c_2 are the children of x in D.

Clearly, all D records at level 0 are in the initial Z set (Step 1 of BFA). All level 1 records will be generated in the first iteration of BFA. Similarly, all level 2 records will be added to Z in the second iteration, and so on. Thus, record r will be added in the jth iteration, where j is the level of r. □

```
1: input: a set R of records
2: output: a set R′ of records, R′ = GER(R)
3: R′ ← ∅
4: while R ≠ ∅ do
5:    r ← a record from R
6:    remove r from R
7:    for all r′ ∈ R′ do
8:       if r ≈ r′ then
9:          merged ← ⟨r, r′⟩
10:         if merged ∉ R ∪ R′ ∪ {r} then
11:            add merged to R
12:         end if
13:      end if
14:   end for
15:   add r to R′
16: end while
17: return R′
```

Algorithm 5.2: The Koosh algorithm for GER(R)

5.4 Koosh

A brute-force algorithm like BFA is inefficient, essentially because the results of match comparisons are forgotten after every iteration. As an example, suppose $R = \{r_1, r_2\}$, $r_1 \approx r_2$, and $\langle r_1, r_2 \rangle$ doesn't match anything. In the first round, BFA will compare r_1 with r_2, and merge them together, adding $\langle r_1, r_2 \rangle$ to the set. In the second round, r_1 will be compared with r_2 a second time, and then merged together again. This comparison is redundant. In data sets with more records, the number of redundant comparisons is even greater.

We give in Algorithm 5.2 the Koosh algorithm, which improves upon BFA by removing these redundant comparisons. The algorithm works by maintaining two sets. R is the set of records that have not been compared yet, and R' is a set of records that have all been compared with each other. The algorithm works by iteratively taking a record r out of R, comparing it to every record in R', and then adding it to R'. For each record r' that matched r, the record $\langle r, r' \rangle$ will be added to R.

Using our simple example, we illustrate the fact that redundant comparisons are eliminated. Initially, $R = \{r_1, r_2\}$ and $R' = \emptyset$. In the first iteration, r_1 is removed from R and compared against everything in R'. There is nothing in R', so there are no matches, and r_1 is added to R'. In the second iteration, r_2 is removed and compared with everything in R', which consists of r_1. Since $r_1 \approx r_2$, the two records are merged and $\langle r_1, r_2 \rangle$ is added to R. Record r_2 is added to R'. In the third iteration, $\langle r_1, r_2 \rangle$ is removed from R and compared against r_1 and r_2 in R'. Neither matches, so $\langle r_1, r_2 \rangle$ is added to R'. Now R is empty, and the algorithm terminates. In the above example, r_1 and r_2 were compared against each other only once, so the redundant comparison that occurred in BFA has been eliminated.

Before establishing the correctness of Koosh, we show the following lemmas.

Lemma 5.4.1. *At any point in the Koosh execution,* $R' \subseteq GER(R)$*. If* $GER(R)$ *is finite, Koosh terminates.*

Proof. The records in R' will either be base records, which are well-formed by definition, or they will be new records generated by the algorithm. Koosh only generates new records using the merge function, and therefore all new records will have a derivation tree. Therefore, all records in R' must be in $GER(R)$.

Suppose that Koosh does not terminate. At the end of every iteration (Line 15) a new record is added to R'. This record cannot already be in R', so R' grows by 1 at every iteration. Thus, R' will be infinite. Since $R' \subseteq GER(R)$, then $GER(R)$ must also be infinite, a contradiction. □

Lemma 5.4.2. *When Koosh terminates, all records in the initial set* R *are in* R'.

Proof. Each iteration of the loop removes one record from R and adds it to R'. Records are never removed from R in any other case. Koosh terminates when $R = \emptyset$. Therefore, when Koosh terminates, all records in the initial set R have been moved to R'. □

Lemma 5.4.3. *Whenever Koosh reaches line 4:*

1. *all pairs of distinct records in* R' *have been compared*
2. *if any pair matches, the merged record is in* R *or* R'.

Proof. We prove this lemma by induction. The base case will be before the first iteration, when $R' = \varnothing$, and therefore the condition holds. Each iteration removes one record r from R and adds it to R'. Directly before adding r to R', r is compared to all of the records in R'. This fact, in combination with the inductive hypothesis, guarantees us the first part of the condition. Furthermore, any record that matches r causes the creation of a merged record. The merged record is added to R unless it is already in R', or if it is the same record as r. In the first two cases, the merged record will end up in either R or R'. In the last case, the merged record is the same as r, and r gets added to R' at the end of the iteration. Therefore, all the new merged records will end up in R or R'. In combination with the inductive hypothesis, we can conclude that at the end of the iteration, all pairs of matching records in R' have their merged records in R or R'. Therefore we have proven the inductive step and can conclude the lemma holds after each iteration of the loop. □

The correctness of Koosh is established by the following theorem.

Theorem 5.4.4. *For any set of records* R *such that* $GER(R)$ *is finite, Koosh terminates and correctly computes* $GER(R)$.

Proof. By Lemma 5.4.1 we know that Koosh terminates, so let S be the set returned by Koosh. We know from Lemma 5.4.2 that $R \subseteq S$. When Koosh terminates, the R set is empty. Therefore, by Lemma 5.4.3, for all $r_1, r_2 \in S$, if $r_1 \approx r_2$ then $\langle r_1, r_2 \rangle \in S$. Finally, since all records in S are well-formed (Lemma 5.4.1), we know that S is the smallest set that has those two properties. Therefore, the conditions of Definition 1.3.2 are satisfied and $S = GER(R)$. □

Let us now consider the performance of Koosh. First, we observe the initial motivation for Koosh.

Lemma 5.4.5. *For any two records* r_1 *and* r_2*, Koosh will never compare them more than once.*

Proof. When a record r is removed from R, it has not been compared with any records. It is compared once with all of the records in R' and then placed into R'. After this, the r is only compared with records in R as they are removed and placed into R'. Since Koosh never moves records back into R, and it doesn't allow records to be added to R if they are already in R', this guarantees that r will never be compared with the same record twice. Therefore, Koosh will never compare any pair of records more than once. □

Theorem 5.4.6. *Koosh is optimal, in the sense that no algorithm that computes* GER(R) *makes fewer comparisons.*

Proof. Suppose there is an algorithm A that generates GER(R) but performs fewer comparisons than Koosh. Then there exist two records $r_1, r_2 \in GER(R)$ that Koosh compares, but A does not compare. Now we construct new match and merge functions. Functions match′ and merge′ are the same as the original functions unless the two records are r_1 and r_2. In this case, match′ returns true and merge′ returns a new record k that is not in GER(R) using the original match and merge functions.

It is clear that if match and merge satisfy our two properties of idempotence and commutativity, then match′ and merge′ also satisfy those properties. Using match′ and merge′, $k \in GER(R)$. But algorithm A never compares r_1 and r_2, so it cannot merge them to obtain k. Therefore, algorithm A does not generate GER(R). This is a contradiction, so no algorithm that generates GER(R) can perform fewer comparisons than Koosh. □

5.5 Domination

Even though Koosh is quite efficient, it is still very expensive, especially since the answer it must compute can be very large. In this section and the next two, we explore ways to tame this complexity by exploiting additional properties of the match and merge functions (Section 5.6), or by only computing a still-interesting subset of the answer (using thresholds, in Section 5.7, or the notion of domination, which we discuss here).

We first mentioned the concept of domination in Chapter 1. In that chapter, we mentioned that the four ICAR properties gave us the justification for the use of "merge domination" as a definition of domination. In this chapter, however, we only assume idempotence and commutativity. According to Benjelloun et al. [11], any partial order may be used for domination. Since we do not have all four ICAR properties to justify the choice of merge domination, one must choose a definition of domination based on intuitions specific to the case of entity resolution with confidences and the particular application context.

To motivate the use of domination in ER with confidences, consider the following records r_1 and r_2 that match and merge into r_3:

r_1 = 0.8 [name : Alice, areacode : 202]

r_2 = 0.7 [name : Alice, phone : 555-1212]

r_3 = 0.7 [name : Alice, areacode : 202, phone : 555-1212]

The resulting r_3 contains all of the attributes of r_2, and its confidence is the same. In this case it is natural to consider a "dominated" record like r_2 to be redundant and unnecessary. Thus, a user may only want the ER answer to contain only nondominated records.

Since we have not defined a structure for $r.\mathcal{A}$, we cannot provide a definition of domination suitable for all contexts. However, it is natural to consider confidences as a key condition in deciding whether one record dominates another. Specifically, we expect that if r is dominated by s, then $r.\mathcal{C} \leq s.\mathcal{C}$.

Again, to distinguish the result with a suitable domination order from GER(R), we use the term NER(R). Just like GER(R), NER(R) may be infinite. In the case

that GER(R) is finite, one way to compute NER(R) is to first compute GER(R) and then remove dominated records. This strategy does not save much effort, since we still have to compute GER(R). A significant performance improvement is to discard a dominated record as soon as it is found in the resolution process, on the assumption that a dominated record will never participate in the generation of a nondominated record. This assumption is stated formally as follows:

Domination Property: If $s \leq r$ and $s \approx x$ then $r \approx x$ and $\langle s, x \rangle \leq \langle r, x \rangle$.

This domination property may or may not hold in a given application. For instance, let us return to our r_1, r_2, r_3 example at the beginning of this section. Consider a fourth record r_4 = 0.9 [name : Alice, areacode : 717, phone : 555-1212, age : 20]. A particular match function may decide that r_4 does *not* match r_3 because the area codes are different, but r_4 and r_2 may match since this conflict does not exist with r_2. In this scenario, we cannot discard r_2 when we generate a record that dominates it (r_3), since r_2 can still play a role in some matches.

However, in applications where having more information in a record can never reduce its match chances, the domination property can hold and we can take advantage of it. If the domination property holds then we can throw away dominated records as we find them while computing NER(R). The following lemma establishes this fact.

Lemma 5.5.1. *If the domination property holds, then every record in* NER(R) *has a well-formed derivation tree with no dominated records.*

Proof. Consider $r \in$ NER(R). Suppose that r has a well-formed derivation tree D where one of the records x in D is dominated by a different record $x' \in$ GER(R). Consider a new derivation tree D′ which is a modified version of D: First, we replace x by x′ in D′. Next, we look at the parent of x in D, say $p = \langle x, x_1 \rangle$, and replace p by $p' = \langle x', x_1 \rangle$ in D′. Note that because of the domination property, $p \leq p'$. Also note that $p' \in$ GER(R) since it has a well-formed derivation tree.

In a similar fashion, we replace the parent of p, and so on, until we replace r by r', where $r \leq r'$, and $r' \in GER(R)$.

If $r \neq r'$, then r is dominated by r', a contradiction since $r \in NER(R)$. Therefore $r = r'$. The same process can be repeated, until all dominated records are eliminated from the derivation tree of r. □

5.5.1 Algorithm Koosh-ND

Both algorithms BFA and Koosh can be modified to eliminate dominated records early. There are several ways to remove dominated records, presenting a tradeoff between how many domination checks are performed and how early dominated records are dropped. For example, a simple way would be to test for domination of one record over another just before they are compared for matching. If a record is dominated, it is dropped. Since both algorithms end up comparing all records in the result set against one another, this would guarantee that all dominated records are removed. However, this scheme does not identify domination as early as possible. For instance, suppose that an input base record r_1 is dominated by another base record r_2. Record r_1 would not be eliminated until it is compared with r_2. In the meantime, r_1 could be unnecessarily compared to other records.

To catch domination earlier, we propose here a more advanced scheme. We will focus on modifying Koosh to obtain Koosh-ND; the changes to BFA are analogous. First, Koosh-ND begins by removing all dominated records from the input set. Second, within the body of the algorithm, whenever a new merged record m is created (line 10), the algorithm checks whether m is dominated by any record in R or R'. If so, then m is immediately discarded, before it is used for any unnecessary comparisons. Note that we do *not* check if m dominates any other records, as this check would be expensive in the inner loop of the algorithm. Finally, since we do not incrementally check if m dominates other records, we add a step at the end to remove all dominated records from the output set.

The correctness of Koosh-ND is established by the following theorem.

Theorem 5.5.2. *For any set of records* R *such that* NER(R) *is finite, Koosh-ND terminates and computes* NER(R).

Proof. We know that Koosh computes GER(R) correctly. Koosh-ND is the same as Koosh except that it sometimes discards records that are dominated by another record. Given Lemma 5.5.1, we know that we can remove dominated records early, so Koosh-ND computes a superset of NER(R). The final check removes all dominated records from the output set, guaranteeing that the result of Koosh-ND is equal to NER(R). □

5.6 The Packages Algorithm

In Section 5.3, we illustrated why ER with confidences is expensive on the records r_1 and r_2 that merged into r_3:
$r_1 = 0.8$ [name : Alice, areacode : 202],
$r_2 = 0.7$ [name : Alice, phone : 555-1212],
$r_3 = 0.56$ [name : Alice, areacode : 202, phone : 555-1212].

Recall that r_2 cannot be discarded essentially because it has a higher confidence than the resulting record r_3. However, notice that other than the confidence, r_3 contains more label-value pairs, and hence, if it were not for its higher confidence, r_2 would not be necessary. This observation leads us to consider a scenario where the records minus confidences can be resolved efficiently, and then we can add the confidence computations in a second phase.

In particular, let us assume that our merge function is "information preserving" in the following sense: When a record r merges with other records, the information carried by r's attributes is not lost. We formalize this notion of "information" by defining a relation "$\sqsubseteq$": $r \sqsubseteq s$ means that the attributes of s carry more information than those of r. We assume that this relation is transitive. Note that $r \sqsubseteq s$ and $s \sqsubseteq r$ does *not* imply that $r = s$; it only implies that $r.\mathcal{A}$ carries as much information as $s.\mathcal{A}$.

The property that merges are information preserving is formalized as follows:

Property P1: If $r \approx s$ then $r \sqsubseteq \langle r, s \rangle$ and $s \sqsubseteq \langle r, s \rangle$.
Property P2: If $s \sqsubseteq r$, $s \approx x$ and $r \approx x$, then $\langle s, x \rangle \sqsubseteq \langle r, x \rangle$

For example, a merge function that unions the attributes of records would have properties P1 and P2. Such functions are common in "intelligence gathering" applications, where one wishes to collect all information known about entities, even if contradictory. For instance, say two records report different passport numbers or different ages for a person. If the records merge (e.g., due to evidence in other attributes) such applications typically gather all the facts, since the person may be using fake passports reporting different ages.

Furthermore, we assume that adding information to a record does not change the outcome of match. In addition, we assume that the match function does not consider confidences in determining whether two records match. These characteristics are formalized by:

Property P3: If $s \sqsubseteq r$ and $s \approx x$, then $r \approx x$.

Having a match function that ignores confidences is not very constraining: If two records are unlikely to match due to low confidences, the merge function can still assign a low confidence to the resulting record to indicate it is unlikely. The second aspect of Property P3 rules out "negative evidence": adding information to a record cannot rule out a future match. However, negative information can still be handled by decreasing the confidence of the resulting record. For example, say that record r_3 above is compared to a record r_4 with area code "717". Suppose that in this application the differing area codes make it very unlikely the records match. Instead of saying the records do not match, we do combine them but assign a very low probability to the resulting record.

To recap, properties P1, P2 and P3 may not always hold, but we believe they do occur naturally in many applications, e.g., intelligence gathering ones. Furthermore, if the properties do not hold naturally, a user may wish to consider modifying the match and merge functions as described above, in order to get the performance gains that the properties yield, as explained next.

Note that Properties P1, P2, P3 imply representativity as defined in [11]. (This follows from Lemma 5.6.1, shown below.)

Algorithm 5.3 exploits these properties to perform ER more efficiently. It proceeds in two phases: a first phase bypasses confidences and groups records into disjoint packages. Because of the properties, this first phase can be done efficiently, and records that fall into different packages are known not to match. The second phase runs ER with confidences on each package separately. We next explain and justify each of these two phases.

5.6.1 Phase 1

In Phase 1, we may use any generic ER algorithm, such as those in [11] to resolve the base records, but with some additional bookkeeping. For example, when two base records r_1 and r_2 merge into r_3, we combine all three records together into a *package* p_3. The package p_3 contains two things: (i) a root $r(p_3)$ which in this case is r_3, and (ii) the base records $b(p_3) = \{r_1, r_2\}$.

Actually, base records can also be viewed as packages. For example, record r_2 can be treated as package p_2 with $r(p_2) = r_2$, $b(p_2) = \{r_2\}$. Thus, the algorithm starts with a set of packages, and we generalize our match and merge functions to operate on packages.

For instance, suppose we want to compare p_3 with a package p_4 containing only base record r_4. That is, $r(p_4) = r_4$ and $b(p_4) = \{r_4\}$. To compare the packages, we only compare their roots: That is, $M(p_3, p_4)$ is equivalent to $M(r(p_3), r(p_4))$, or in this example equivalent to $M(r_3, r_4)$. (We use the same symbol M for record and package matching.) Say these records do match, so we generate a new package p_5 with $r(p_5) = \langle r_3, r_4 \rangle$ and $b(p_5) = b(p_3) \cup b(p_4)$

```
 1: input: a set R of records
 2: output: a set R′ of records, R′ = GER(R)
 3: Define for Packages:
 4: match: p ≈ p′ iff r(p) ≈ r(p′)
 5: merge: ⟨p, p′⟩ = p″ :
 6:          with root: r(p″) = ⟨r(p), r(p′)⟩
 7:          and base: b(p″) = b(p) ∪ b(p′)
 8: Phase 1:
 9: P ← ∅
10: for all records rec in R do
11:    create package p:
12:       with root: r(p) = rec
13:       and base: b(p) = {rec}
14:    add p to P
15: end for
16: compute P′ = GER(P) (e.g., using Koosh) with the following modification:
    Whenever packages p, p′ are merged into p″, delete p and p′ immediately,
    then proceed.
       Phase 2:
17: R′ ← ∅
18: for all packages p ∈ P′ do
19:    compute Q = GER(b(p)) (e.g. using Koosh)
20:    add all records in Q to R′
21: end for
22: return R′
```

Algorithm 5.3: The Packages algorithm

$= \{r_1, r_2, r_4\}$.

The package p_5 represents not only the records in $b(p_5)$, but also any records that can be derived from them. That is, p_5 represents all records in $GER(b(p_5))$. For example, p_5 implicitly represents the record $\langle r_1, r_4 \rangle$, which may have a higher confidence than the root of p_5. Let us refer to the complete set of records represented by p_5 as $c(p_5)$, i.e., $c(p_5) = GER(b(p_5))$. Note that the package does not contain $c(p_5)$ explicitly, the set is just implied by the package.

The key property of a package p is that $r(p).\mathcal{A}$ carries at least as much information as the attributes of any record in $c(p)$; that is, for any $s \in c(p)$,

$s \sqsubseteq r(p)$. This property implies that any record u that does *not* match r(p), cannot match any record in c(p). We will now prove this result.

For the following proofs, we assume Properties P1, P2, P3. We denote by B(r) the set of base records that appear as leaves in a derivation tree that produces r.

Lemma 5.6.1. *Consider two records* r, s *with derivation trees, such that* $B(r) \subseteq B(s)$*. Then* $r \sqsubseteq s$*.*

Proof. We denote by D(x) the depth of the derivation tree of a record x. Base records have a depth of 0. The proof is by induction on the depth of record r.

Induction hypothesis: Given two records r, s such that $B(r) \subseteq B(s)$ and an integer n, $0 \le n$: If $D(r) \le n$ then $r \sqsubseteq s$.

Base: $n = 0$. In this case, r is a base record that belongs to the derivation tree of s. Following the path from s to r in this derivation tree, each record has less or equivalent information than its parent node (due to P1), and therefore, by transitivity of $\sqsubseteq$, we have that $r \sqsubseteq s$.

Induction step: We now show that if the hypothesis holds for $n = k$, then it also holds for $n = k + 1$.

If $D(r) \le k$, we can directly apply the induction hypothesis. The case to deal with is $D(r) = k+1$. Consider the children of r in its derivation tree. $r = \langle r_1, r_2 \rangle$. Clearly, $D(r_1) \le k$ and $D(r_2) \le k$. Since $B(r_1) \subseteq B(s)$ and $B(r_2) \subseteq B(s)$, we can apply the induction hypothesis to r_1 and r_2. It follows that $r_1 \sqsubseteq s$, and $r_2 \sqsubseteq s$. Applying P2 and P3, we can inject r_2 to obtain that $\langle r_1, r_2 \rangle \sqsubseteq \langle s, r_2 \rangle$. Similarly, we obtain that $\langle r_2, s \rangle \sqsubseteq \langle s, s \rangle$. By merge commutativity and idempotence, and since $\sqsubseteq$ is transitive, it follows that $r \sqsubseteq s$. □

Corollary 5.6.2. *If a record* u *matches a record* $s \in c(p)$*, then* $u \approx r(p)$*.*

Proof. The root r(p) always has a valid derivation tree using all records in b(p), i.e., $B(r(p)) = b(p)$. Since $s \in c(p)$, s has a valid derivation tree, and $B(s) \subseteq b(p)$. Thus, since $B(s) \subseteq B(r(p))$, by Lemma 5.6.1, $s \sqsubseteq r(p)$. By Property P3, if $u \approx s$, then $u \approx r(p)$. □

Theorem 5.6.3. *For any package* p*, if a record* u *does not match the root* r(p)*, then* u *does not match any record in* c(p)*.*

Proof. Suppose that u matches a record $s \in$ c(p). By Corollary 5.6.2, u ≈ r(p), a contradiction. □

This fact in turn saves us a lot of work! In our example, once we wrap up base records r_1, r_2 and r_4 into p_5, we do not have to involve them in any more comparisons. We only use r(p_5) for comparing against other packages. If p_5 matches some other package p_8 (i.e., the roots match), we merge the packages. Otherwise, p_5 and p_8 remain separate since they have nothing in common. That is, nothing in c(p_5) matches anything in c(p_8).

Corollary 5.6.4. *Consider two packages* p, q *that do not match, i.e.,* M(r(p), r(q)) *is false. Then no record* $s \in$ c(p) *matches any record* $t \in$ c(q)*.*

Proof. Suppose that there is an $s \in$ c(p) that matches a $t \in$ c(q). Using Corollary 5.6.2, we see that s matches r(q). Now, since r(q) matches an element in c(p), we can use the corollary a second time to show that r(q) must match r(p), which is a contradiction. □

As an aside, observe that r(p) has a confidence associated with it. However, this confidence is not meaningful. Since the match function does not use confidences, r(p).$\mathcal{C}$ is not really used. As a matter of fact, there can be a record in c(p) with the same attributes as r(p) but with higher confidence. (In our running example, the root of p_5 was computed as $\langle\langle r_1, r_2\rangle, r_4\rangle$, but this record may be different from $\langle\langle r_1, r_4\rangle, r_2\rangle$. The latter record, which is in c(p) may have a higher confidence than r(p).)

5.6.2 Phase 2

At the end of Phase 1, we have resolved all the base records into a set of independent packages. In Phase 2 we resolve the records in each package, now taking into account confidences. That is, for each package p we compute

GER(b(p)), using an algorithm like Koosh. Since none of the records from other packages can match a record in c(p), the GER(b(p)) computation is completely independent from the other computations. Thus, we save a very large number of comparisons in this phase where we must consider the different order in which records can merge to compute their confidences. The more packages that result from Phase 1, the finer we have partitioned the problem, and the more efficient Phase 2 will be.

Note that Phase 2 correctly computes GER(R) as long as Corollary 5.6.4 holds: records falling into one package never match records falling into another. Even if Phase 1 combined records that do not actually match (false positives), Phase 2 would still correctly compute GER(R). This fact opens up a wider range of situations in which the Packages algorithm could apply. Even if the match and merge functions do not satisfy Properties P1, P2, and P3, there could exist a more permissive match function, and a corresponding merge function that do satisfy the three properties. In this case, Phase 1 could be run using the more permissive match and merge functions, and Phase 2, using the original match and merge functions would still correctly compute GER(R). An alternative to Phase 2 is to leave the packages un-expanded, and report them to the user. To the end user, a package may be more understandable than a large number of records with different confidences. If the user finds an interesting package, then she may selectively expand it or query it. For example, say a user sees a package containing records related to a particular person. In the records, the user may observe attributes name : Alice, address : 123 Main, passport : 789, and may then query for the confidence of the record [name : Alice, address : 123 Main, passport : 789]. The answer to the query will tell the user how likely it is there is an entity with these three attributes.

5.6.3 Example

To illustrate the two phases of our packages algorithm, consider the following example. Symbols a, b, c, d represent label-value pairs. Records match only if they contain the a attribute. The merge function unions all attributes, and the resulting confidence is the minimum of the confidences of the two merging records times 0.5. Initially there are 4 records:

$r_1 = 0.9[a, b]$
$r_2 = 0.6[a, c]$
$r_3 = 0.8[a, d]$
$r_4 = 0.9[c, d]$

First, the base records are converted to packages. For example, for the first record we create package p_1 with $b(p_1) = \{r_1\}$, $r(p_1) = r_1$. In Phase 1, say we first compare p_1 and p_2, yielding package p_5:

$b(p_5) = \{r_1, r_2\}$; $r(p_5) = 0.3[a, b, c]$

Later, p_5 matches p_3, yielding p_6:

$b(p_6) = \{r_1, r_2, r_3\}$; $r(p_6) = 0.15[a, b, c, d]$

Package p_4 does not match anything, so at the end of Phase 1 we have two packages, p_6 and p_4. In Phase 2, when we expand p_6 we obtain:

$c(p_6) = 0.9[a, b], 0.6[a, c], 0.8[a, d], 0.4[a, b, c],$
$0.4[a, b, d], 0.3[a, c, d], 0.2[a, b, c, d], 0.15[a, b, c, d]$

Note that the record $[a, b, c, d]$ has two confidences, depending on the order of the merges. The higher confidence $0.2[a, b, c, d]$ is obtained when r_1 and r_3 are merged first.

5.6.4 Packages-ND

As with Koosh, there is a variant of Packages that handles domination. To remove dominated records from the final result, we simply use Koosh-ND in

Phase 2 of the Packages algorithm. Note that it is not necessary to explicitly remove dominated packages in Phase 1. To see this, say at some point in Phase 1 we have two packages, p_1 and p_2 such that $r(p_1) \leq r(p_2)$, and hence $r(p_1) \sqsubseteq r(p_2)$. Then p_1 will match p_2 (by Property P3 and idempotence), and both packages will be merged into a single one, containing the base records of both.

5.7 Thresholds

Another opportunity to reduce the resolution workload lies within the confidences themselves. Some applications may not need to know every record that could possibly be derived from the input set. Instead, they may only care about the derived records that are above a certain confidence threshold.

Definition 5.7.1. *Given a threshold value* T *and a set of base records* R, *we define the above-threshold entity-resolved set,* $TER(R)$ *that contains all records in* $GER(R)$ *with confidences above* T. *That is,* $r \in TER(R)$ *if and only if* $r \in GER(R)$ *and* $r.\mathcal{C} \geq T$.

As we did with domination, we would like to remove below-threshold records, not after completing the resolution process (as suggested by the definition), but as soon as they appear. However, we will only be able to remove below-threshold records if they cannot be used to derive above-threshold records. Whether we can do that depends on the semantics of confidences.

As we mentioned earlier, models for the interpretation of confidences vary. Under some interpretations, two records with overlapping information might be considered as independent evidence of a fact, and the merged record will have a higher confidence than either of the two base records.

Other interpretations might see two records, each with their own uncertainty, and a match and merge process which is also uncertain, and conclude that the result of a merge must have lower confidence than either of the base records. For example, one interpretation of $r.\mathcal{C}$ could be that it is the probability that r correctly describes a real-world entity. Using the "possible

worlds" metaphor [55], if there are N equally-likely possible worlds, then an entity containing at least the attributes of r will exist in $r.\mathcal{C} \times N$ worlds. With this interpretation, if r_1 correctly describes an entity with probability 0.7, and r_2 describes an entity with probability 0.5, then $\langle r_1, r_2 \rangle$ cannot be true in more worlds than r_2, so its confidence would have to be less than or equal to 0.5.

To be more formal, some interpretations, such as the example above, will have the following property.

Threshold Property: If $r \approx s$ then $\langle r, s \rangle.\mathcal{C} \leq r.\mathcal{C}$ and $\langle r, s \rangle.\mathcal{C} \leq s.\mathcal{C}$.

Given the threshold property, we can compute TER(R) more efficiently. The next lemma tells us that if the threshold property holds, then all results can be obtained from above-threshold records. Thus, below-threshold records can be immediately dropped during the resolution process.

Lemma 5.7.2. *When the threshold property holds, every record in* TER(R) *has a well-formed derivation tree consisting exclusively of above-threshold records.*

Proof. Consider $r \in$ TER(R). Since r is also in GER(R), it has a well-formed derivation tree, call it D. Suppose that one of the records x in D is below threshold, $x.\mathcal{C} \leq T$. By the threshold property, the parent of x is below threshold. The parent of the parent of x is also below threshold, and we continue in this fashion until we show that the root itself, r is below threshold, a contradiction. Thus, r has a well-formed derivation tree with above-threshold records. □

5.7.1 Algorithms Koosh-T and Koosh-TND

As with removing dominated records, Koosh can be easily modified to drop below-threshold records. First, we add an initial scan to remove all base records that are already below threshold. Then, we simply add the following conjunct to the condition of Line 10 of the algorithm:

merged.$\mathcal{C} \geq$ T

Thus, merged records are dropped if they are below the confidence threshold.

Theorem 5.7.3. *If* TER(R) *is finite, Koosh-T terminates and computes* TER(R).

Proof. Koosh computes GER(R), and the only changes made to Koosh-T remove records that are below threshold. Because of Lemma 5.7.2, we can remove below-threshold records at any point without missing records in TER(R). Therefore, Koosh-T generates a superset of TER(R). Since below-threshold records are removed from the initial set, and because Koosh-T drops all newly formed records that are below threshold, Koosh-T cannot generate any records that are not in TER(R). Therefore, Koosh-T correctly computes TER(R). □

By performing the same modification as above on Koosh-ND, we obtain the algorithm Koosh-TND, which computes the set NER(R) ∩ TER(R) of records in GER(R) that are neither dominated nor below threshold.

5.7.2 Packages-T and Packages-TND

If the threshold property holds, Koosh-T or Koosh-TND can be used for Phase 2 of the Packages algorithm, to obtain algorithm Packages-T or Packages-TND. In that case, below-threshold and/or dominated records are dropped as each package is expanded.

Note that we cannot apply thresholds in Phase 1 of the Packages algorithm. The confidence associated with the root of a package is not an upper bound. So even if r(p).$\mathcal{C}$ is below threshold, there can still be above-threshold records in c(p). For instance, in the example of Section 5.6.3, r(p_6).$\mathcal{C}$ = 0.15, but records 0.2[a, b, c, d] and 0.95[a, c] are in c(p_6). We can, however, as a minor optimization discard below-threshold records in the base set of a package. Any such records will the thrown away at the beginning of Phase 2 (by Koosh-T), so there is no need to carry them.

5.8 Experiments

To summarize, we have discussed three main algorithms: BFA, Koosh, and Packages. For each of those basic three, there are three variants, adding in thresholds (T), non-domination (ND), or both (TND). In this section, we will compare the three algorithms against each other using both thresholds and nondomination. We will also investigate how performance is affected by varying threshold values, and, independently, by removing dominated records.

To test our algorithms, we ran them on synthetic data. Synthetic data gives us the flexibility to carefully control the distribution of confidences, the probability that two records match, as well as other important parameters. Our goal in generating the data was to emulate a realistic scenario where n records describe various aspects of m real-world entities ($n > m$). If two of our records refer to the same entity, we expect them to match with much higher probability than if they referred to different entities.

To emulate this scenario, we assume that the real-world entities can be represented as points on a number line. Records about a particular entity with value x contain an attribute A with a value "close" to x. (The value is normally distributed with mean x, see below.) Thus, the match function can simply compare the A attribute of records: if the values are close, the records match. Records are also assigned a confidence, as discussed below.

For our experiments we use an "intelligence gathering" merge function as discussed in Section 5.6, which unions attributes. Thus, as a record merges with others, it accumulates A values and increases its chances of matching other records related to the particular real-world entity.

To be more specific, our synthetic data was generated using the following parameters (and their default values):

- n, the number of records to generate (default: 1000)
- m, the number of entities to simulate (default: 100)
- margin, the separation between entities (default: 75)

- σ, the standard deviation of the normal curve around each entity. (default: 10)
- μ_c, the mean of the confidence values (default: 0.8)

To generate one record r, we proceed as follows: First, pick a uniformly distributed random integer i in the range $[0, m-1]$. This integer represents the value for the real-word entity that r will represent. For the A value of r, generate a random floating point value v from a normal distribution with standard deviation σ and a mean of $margin \cdot i$. To generate r's confidence, compute a uniformly distributed value c in the range $[\mu_c - 0.1, \mu_c + 0.1]$ (with $\mu_c \in [0.1, 0.9]$ so that c stays in $[0, 1]$). Now create a record r with $r.\mathcal{C} = c$ and $r.\mathcal{A} = \{A : v\}$. Repeat all of these steps n times to create n synthetic records.

Our merge function takes in the two records r_1 and r_2, and creates a new record r_m, where $r_m.\mathcal{C} = r_1.\mathcal{C} \times r_2.\mathcal{C}$ and $r_m.\mathcal{A} = r_1.\mathcal{A} \cup r_2.\mathcal{A}$. The match function detects a match if for the A attribute, there exists a value v_1 in $r_1.\mathcal{A}$ and a value v_2 in $r_2.\mathcal{A}$ where $|v_1 - v_2| < k$, for a parameter k chosen in advance ($k = 25$ except where otherwise noted). We defined the domination order as follows. If $r.\mathcal{C} \leq s.\mathcal{C}$ and $r.A \subseteq s.A$, then $r \leq s$.

Naturally, our first experiment compares the performance of our three algorithms, BFA-TND, Koosh-TND and Packages-TND, against each other. We varied the threshold values to get a sense of how much faster the algorithms are when a higher threshold causes more records to be discarded. Each algorithm was run at the given threshold value three times, and the resulting number of comparisons was averaged over the three runs to get our final results.

Figure 5.1 shows the results of this first experiment. The first three lines on the graph represent the performance of our three algorithms. On the horizontal axis, we vary the threshold value. The vertical axis (logarithmic) indicates the number of calls to the match function, which we use as a measure of the work performed by the algorithms. The first thing we notice is that work performed by the algorithms grows exponentially as the threshold is decreased. Thus, clearly thresholds are a very powerful tool: one can get

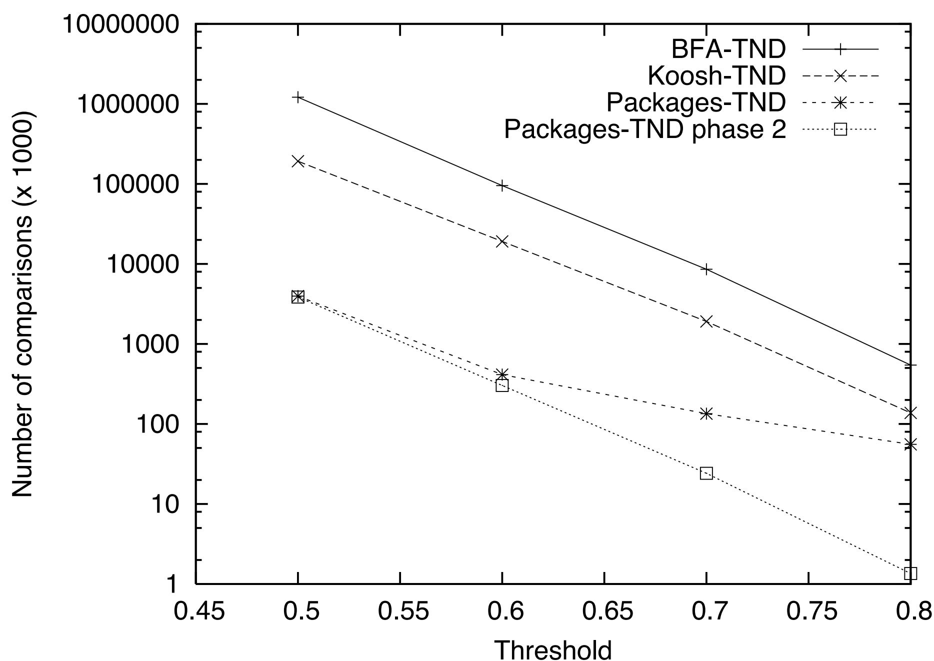

Figure 5.1: Thresholds vs. Matches

high-confidence results at a relatively modest cost, while computing the lower confidence records gets progressively more expensive! Also interestingly, the BFA-TND and Koosh-TND lines are parallel to each other. This means that they are consistently a constant factor apart. Roughly, BFA does 10 times the number of comparisons that Koosh does.

The Packages-TND algorithm is far more efficient than the other two algorithms. Of course, Packages can only be used if Properties P1, P2 and P3 hold, but when they do hold, the savings can be dramatic. We believe that these savings can be a strong incentive for the application expert to design match and merge function that satisfy the properties. Note incidentally that the Packages line is not quite parallel to the other two. The lines are not parallel because the first phase of the Packages algorithm does not consider thresholds at all, and therefore there is a constant amount of work included in each data point. When we consider only the comparisons performed in the second phase of the algorithm, we get the fourth line in the graph, which is indeed

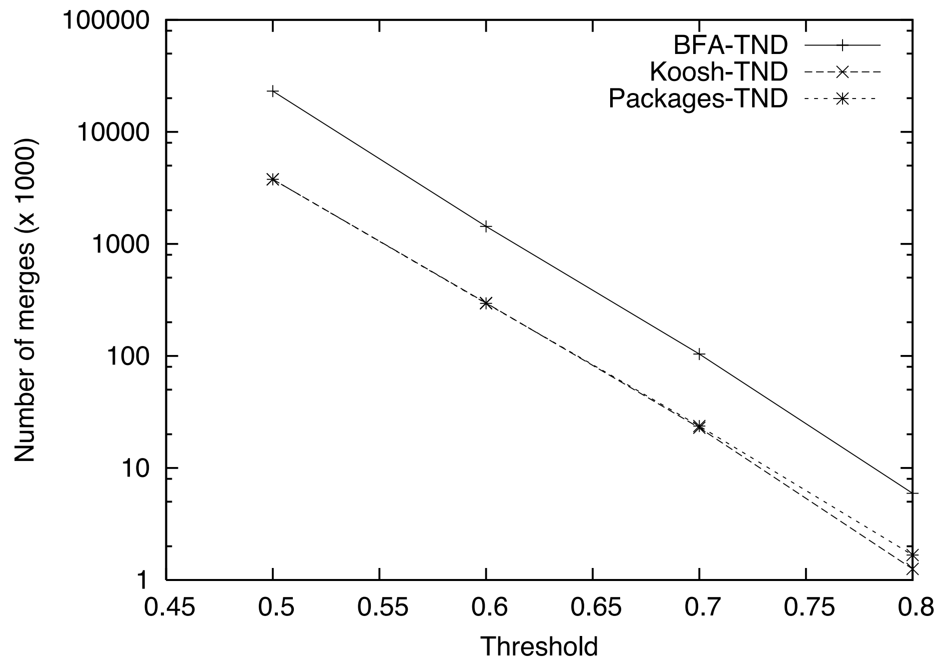

Figure 5.2: Thresholds vs. Merges

parallel to the BFA and Koosh lines. Therefore, the work done in the second phase is also a constant factor away from BFA and Koosh. In this case, the constant factor is at least 100.

We also compared our algorithms based on the number of merges performed. In Figure 5.2, the vertical axis indicates the number of merges that are performed by the algorithms. We can see that Koosh-TND and the Packages-TND are still a great improvement over BFA. BFA performs extra merges because in each iteration of its main loop, it recompares all records and merges any matches found. The extra merges result in duplicate records which are eliminated when they are added to the result set. Packages performs slightly more merges than Koosh, since the second phase of the algorithm does not use any of the merges that occurred in the first phase. If we subtract the Phase 1 merges from Packages (not shown in the figure), Koosh and Packages perform roughly the same number of merges.

In our next experiment, we compare the performance of our algorithms as

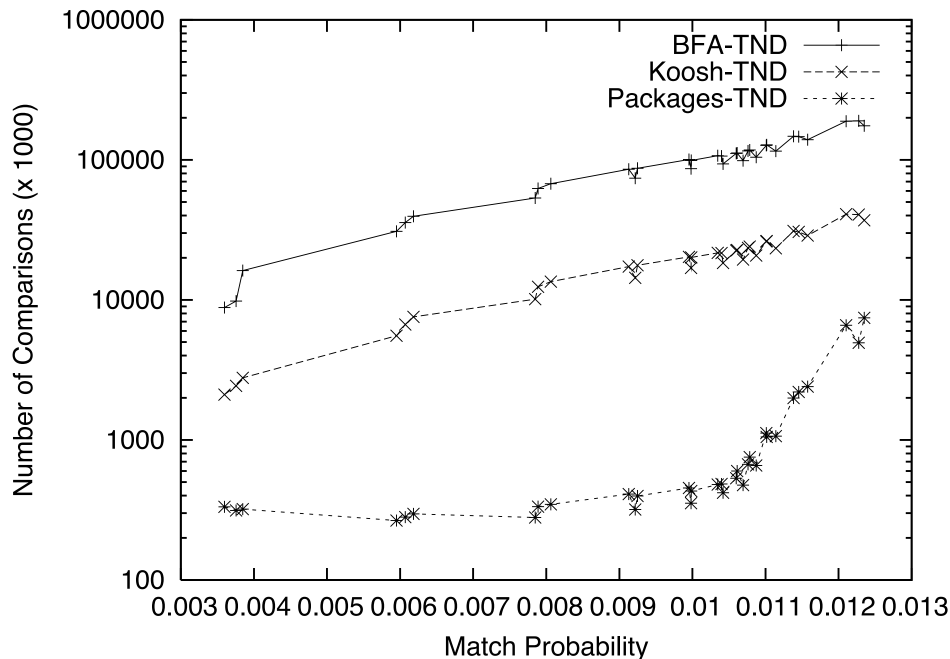

Figure 5.3: Selectivity vs. Comparisons

we vary the probability that base records match. We can control the match probability by changing parameters k or σ, but instead of reporting our results in terms of those parameter values, we use the resulting match probability as the horizontal axis to provide more intuition. In particular, to generate Figure 5.3, we vary parameter k from 5 to 55 in increments of 5 (keeping the threshold value constant at 0.6). During each run, we measure the match probability as the fraction of base record matches that are positive. (The results are similar when we compute the match probability over all matches.) For each run, we then plot the match probability versus the number of calls to the match function, for our three algorithms.

As expected, the work increases with greater match probability, since more records are produced. Furthermore, we note that the BFA and Koosh lines are roughly parallel, but the Packages line stays level until a quick rise in the amount of work performed once the match probability reaches about 0.011. The Packages optimization takes advantage of the fact that records

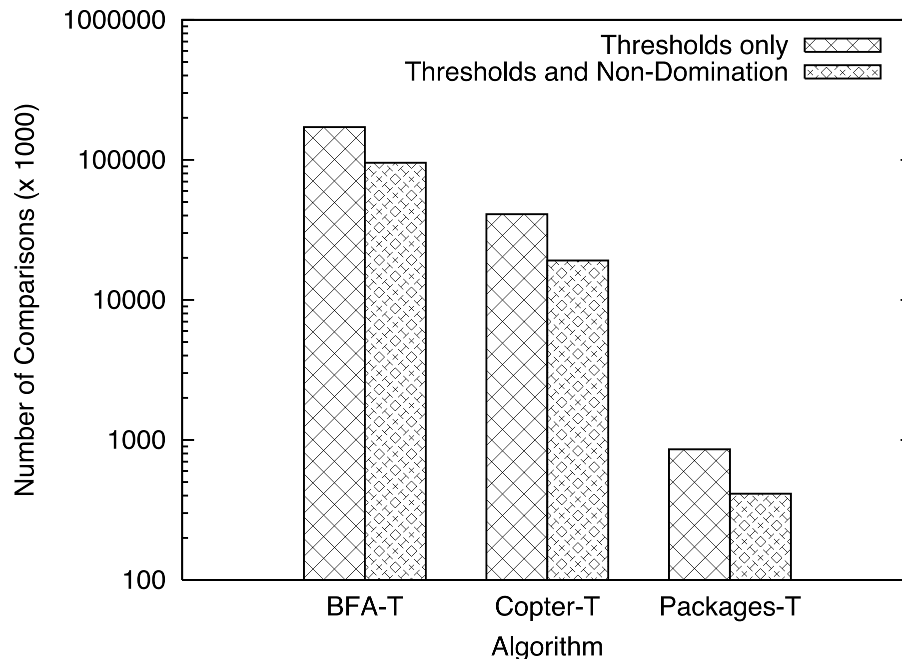

Figure 5.4: Effects of removing dominated records

can be separated into packages that do not merge with one another. As the match probability increases, these packages quickly begin to merge into larger packages, which reduces the effectiveness of the optimization. If this curve were to continue, we would expect it to approach the Koosh line, and slightly surpass it. We have this expectation because when all records fall into one package, then the second phase of Packages is precisely Koosh on the set of base records.

In practice, we would expect to operate in the range of Figure 5.3 where the match probability is low and Packages outperforms Koosh. In our scenario with high match probabilities, records that refer to different entities are being merged, which means the match function is not doing its job. One could also get high match probabilities if there were very few entities, so that packages do not partition the problem finely. But again, in practice one would expect records to cover a large number of entities.

We ran a similar test obtaining different match probabilities by varying σ

instead of k. This test revealed similar results, indicating that match probability is a good metric for predicting the performance of these algorithms.

Our results so far show the advantages of using thresholds and domination together, but do not illustrate the gains that using domination alone yields. These gains are explicitly shown in Figure 5.4 (threshold = 0.6). The results show that removing dominated records reduces the amount of work done by each algorithm by a factor of about 2.

5.9 Related Work

Most of the work in this area (see [90, 41] for recent surveys) focuses on the "matching" problem, i.e., on deciding which records do represent the same entities and which ones do not. Matching is generally done in two phases: Computing measures of how similar atomic values are (e.g., using edit-distances [78], TF-IDF [27], or adaptive techniques such as q-grams [24]), then feeding these measures into a model (with parameters), which makes matching decisions for records. Proposed models include unsupervised clustering techniques [46, 25], Bayesian networks [84], decision trees, SVMs, conditional random fields [77]. The parameters of these models are learned either from a labeled training set (possibly with the help of a user, through active learning [74]), or using unsupervised techniques such as the EM algorithm [92].

All the techniques above manipulate and produce numerical values, when comparing atomic values (e.g. TF-IDF scores), as parameters of their internal model (e.g., thresholds, regression parameters, attribute weights), or as their output. But these numbers are often specific to the techniques at hand, and do not have a clear interpretation in terms of "confidence" in the records or the values. On the other hand, representations of uncertain data exist, which soundly model confidence in terms of probabilities (e.g., [7, 38]), or beliefs [57]. However these approaches focus on computing the results and confidences of exact queries, extended with simple "fuzzy" operators for value

comparisons (e.g., see [31]), and are not capable of any advanced form of entity resolution. We propose a flexible solution for ER that accommodates any model for confidences, and proposes efficient algorithms based on their properties.

The first phase of the Packages algorithm is similar to the set-union algorithm described in [67], but our use of a merge function allows the selection of a true representative record. The Packages algorithm stores the lineage of resolved packages in order to compute confidence values later in processing. This two-phase approach bears similarity to Widom's Trio system [88], which uses lineage to support lazy evaluation of record confidences. The presence of "custom" merges is an important part of ER, and it makes confidences non-trivial to compute. The need for iterating matches and merges was identified by [14] and is also used in [34], but their record merges are simple aggregations (similar to our "information gathering" merge), and they do not consider the propagation of confidences through merges.

5.10 Conclusion

In entity resolution processing, the input data is often inaccurate, and the match and merge rules for resolution yield uncertain results. While there has been work on ER with confidences, most of the work to date has been couched in terms of a specific application or has not dealt directly with the problems of performance and scalability. In this chapter we looked at ER with confidences as a "generic database" problem, where we are given black-boxes that compare and merge records, and we focus on efficient algorithms that reduce the number of calls to these boxes.

The key to reducing work is to exploit generic properties (like the threshold property) that an application may have. If such properties hold we can use the optimizations we have studied (e.g., Koosh-T when the threshold property holds). Of the three optimizations, thresholds is the most flexible one, as it

gives us a "knob" (the threshold) that one can control: For a high threshold, we only get high-confidence records, but we get them very efficiently. As we decrease the threshold, we start adding lower-confidence results to our answer, but the computational cost increases. The other two optimizations, domination and packages, can also reduce the cost of ER very substantially but do not provide such a control knob.

Chapter 6

Evaluating Entity Resolution Results

6.1 Introduction

Usually when we evaluate entity resolution algorithms, we run them on a dataset and compare the results to a "gold standard". Section 1.4 introduced the two concepts of correctness and comprehensiveness as measures of the quality of entity resolution results. When we compare an entity resolution result to a human-generated gold standard, we are measuring the *correctness* of the result. In the absence of a human-generated gold standard, we can instead measure the *comprehensiveness* by comparing results to a result generated by an exhaustive ER algorithm—a result we still refer to as a gold standard.

Whether dealing with correctness or comprehensiveness, a key component of this type of evaluation is a method of assigning a number to express how close a given ER result is to the gold standard. Many ER measures (e.g., pairwise F_1 and cluster F_1, as detailed in Section 6.3) have been proposed for comparing the results of various ER algorithms [91, 56, 61, 1], but there is currently no agreed standard measure for evaluating ER results. Most works tend to use one ER measure over another without a clear explanation of why that ER measure is most appropriate. The pitfall of using an arbitrary measure is that different measures may disagree on which ER results are the best.

Let us consider a brief example. Using letters to represent records, consider an entity resolution problem with an input set of records $I = \{a, b, c, d, e, f, g, h, i\}$. Three possible ER results are shown in Table 6.1, along with the gold standard. We have used angle brackets to denote groups of records that have been determined to refer to the same real-world entity in a result. For example, the algorithm that generated result R_1 decided that records a, b, c, and d all refer to distinct entities, while records e, f, g, h, i, and j all refer to the same entity. Note that we are reusing angle brackets with a subtly different meaning from other chapters. In other chapters, angle brackets represented the application of the merge function. In this chapter, since we may be evaluating the results of ER algorithms that do not merge resolved records, we relax the meaning of the angle brackets as simply indicating that records within the angle brackets refer to the same entity.

Suppose we are evaluating two ER results R_1 and R_2, against the gold standard G. Using an ER measure that evaluates a result based on the number of base record pairs that match, R_1 could be a better solution because it found 15 correct pairs (i.e., all base record pairs in $\langle e, f, g, h, i, j\rangle$) while R_2 only found 8 correct pairs. On the other hand, if we use a measure that evaluates results based on correctly resolved entities in the gold standard, R_2 could be considered better than R_1 because R_2 contains two correctly resolved entities $\langle a, b\rangle$ and $\langle c, d\rangle$ while R_1 only has one correct entity $\langle e, f, g, h, i, j\rangle$. As another example, suppose that we compare R_2 and R_3. One measure could be more focused on high precision and prefer R_2 over R_3 because R_2 has only found correctly matching base records while R_3 has found some non-matching base records (e.g., a and c do not match). On the other hand, another measure might consider recall to be more important and prefer R_3 over R_2 because R_2 has not found all the matching base record pairs (unlike R_3).

Surprisingly, such conflicts between ER measures can occur frequently. It is tempting to suggest that when conflicts arise, one of the measures involved must be faulty in some way. However, since different applications may have different criteria that define the "goodness" of a result, we cannot simply

Set	ER Result
Gold Standard	$\{\langle a,b\rangle, \langle c,d\rangle, \langle e,f,g,h,i,j\rangle\}$
R_1	$\{\langle a\rangle, \langle b\rangle, \langle c\rangle, \langle d\rangle, \langle e,f,g,h,i,j\rangle\}$
R_2	$\{\langle a,b\rangle, \langle c,d\rangle, \langle e,f,g\rangle, \langle h,i,j\rangle\}$
R_3	$\{\langle a,b,c,d\rangle, \langle e,f,g,h,i,j\rangle\}$

Table 6.1: Comparing two ER results

claim one measure to be better than another.

The main contributions of this chapter are twofold. First, we provide a survey of ER measures that have been used to date and provide an analysis of how the measures differ. In studying these measures, we noticed a missing component in the space of existing measures. So the second main contribution of this chapter is a new measure for evaluating ER and an exploration of its relationships to other measures.

Our new measure is inspired by the edit distance of strings [85, 83]. Rather than the insertions, deletions and swaps of characters used in edit distance, our measure is based upon the elementary operations of merging and splitting clusters. We therefore call this measure "merge distance". A basic merge distance that simply counts the number of splits and merges may be a good choice for certain applications, but as we have mentioned, no single ER measure is better than all the others. However, if we generalize merge distance by letting the costs of merge and split operations be determined by functions, we arrive at an intuitive, configurable measure that can support the needs of a wide variety of applications. Surprisingly, at least two state-of-the-art measures are closely related to generalized merge distance: the Variation of Information (VI) [63] clustering measure is a special case of generalized merge distance while the pairwise F_1 [61] measure can be directly computed using generalized merge distance.

We further propose a linear-time algorithm (called the Slice algorithm) that efficiently computes generalized merge distance for a large class of cost functions that satisfy reasonable properties. As we argue in Section 6.5, gold

standards can be very large, so computing measures can be expensive, especially with the quadratic algorithms used for many measures. To the best of our knowledge, the Slice algorithm is the first provably scalable algorithm for ER measures. A non-trivial result is that the pairwise F_1 and VI distances can be computed using our Slice algorithm in linear time.

In summary, our contributions are as follows:

- We define our model for ER measures and conduct an extensive survey on ER measures (Sections 6.2–6.3).
- We propose generalized merge distance (GMD), a new measure that uses the elementary operations of cluster splits and merges to measure the distance from one ER result to another. We propose an efficient linear-time algorithm (called the Slice algorithm) that computes GMD for a large class of cost functions that satisfy reasonable properties (Sections 6.4–6.5).
- We experimentally demonstrate the scalability of the Slice algorithm (Section 6.7).

The contents of this chapter are the joint work of Menestrina, Whang, and Garcia-Molina. The theory, proofs, and algorithms involved in the generalized merge distance framework are predominantly the work of the author of this thesis, whereas the survey, implementation, and experimental evaluation were performed mostly by Whang.

6.2 Evaluating ER Results

To evaluate an ER algorithm, we must compare its results to a gold standard. We define the gold standard S as a partition of the input records I. So we would like to assign a number to characterize how far away a result R is from the gold standard S. Let $g(R, S)$ be the function that performs this task of assigning a number.

We would first like to discuss a few properties that g may have. First, for g to be useful, we would like to be able to compare the values $g(R_1, S)$ with $g(R_2, S)$ to determine whether R_1 or R_2 is closer to S. If $g(R, S)$ generally increases as R and S get closer to each other, then we call g a similarity measure. If g generally decreases under the same circumstances, then we call g a distance measure. We note that if $g(x, y)$ is a similarity measure, then $g_d(x, y) = -g(x, y)$ is a distance measure.

Another property of $g(x, y)$ has to do with its range. If $g(x, y) \in [0, 1]$ for all x, y, then we say g is normalized. In some applications, it may be desirable to use a normalized similarity or distance measure. If g is not normalized, but has some other bounded range, it is trivial to normalize g to the range $[0, 1]$.

To provide some examples, Hamming distance and edit distance of strings are non-normalized distance measures. The Jaccard coefficient of sets is a normalized similarity measure.

We have so far avoided using the term "metric", as a metric is a formally defined mathematical concept. We now consider whether we should expect a distance measure g to be a metric. A proper metric must satisfy the following five properties: [89]

1. $g(x, y)$ is non-negative: $g(x, y) \geq 0$,
2. $g(x, y)$ satisfies the triangle inequality: $g(x, y) + g(y, z) \geq g(x, z)$,
3. $g(x, y)$ is symmetric: $g(x, y) = g(y, x)$,
4. $g(x, x) = 0$,
5. if $g(x, y) = 0$, then $x = y$.

We also define a similarity metric as a similarity measure $s(x, y)$ where the function $g(x, y) = c - s(x, y)$ is a metric for some value of c.

Of the five properties of metrics, we consider Properties 1 and 4 as reasonable enough to assume outright of any distance measure that we consider

in this chapter. Properties 2, 3, and 5, however, are subject to debate. Property 5 requires that the distance between distinct values be non-zero. However, when using common similarity measures such as precision or recall, it is certainly possible for two distinct values to have 100% similarity (zero distance). Property 2 does not have to hold where we only compare two sets R and S at a time (i.e., we do not measure the distance of a sequence of more than two ER results).

The question of symmetry (Property 3) is more interesting. Consider the following two cases:

1. $R = \{\langle a \rangle, \langle b \rangle\}$ and $S = \{\langle a, b \rangle\}$
2. $R = \{\langle a, b \rangle\}$ and $S = \{\langle a \rangle, \langle b \rangle\}$

In the first case, our ER algorithm has missed a match. In the second, it has found a match where there should not have been one. If false negatives and false positives are considered equally bad, then the two cases have equal distance and our similarity or distance measures may be symmetric. However, in many cases, we may wish to consider measures that have different penalties for different types of errors. So in the most general case, symmetry may not be a property of a method of evaluating the results of ER.

Functions that satisfies all properties of a metric except for Properties 2, 3, or 5 are called semimetrics, quasimetrics, or pseudometrics, respectively [89]. Since the measures we consider here may not satisfy the three properties above, they may be referred to as premetrics [89]. In this chapter we will avoid the use of the term "metric" altogether and use the more general term "measure".

6.3 Existing Measures

We review state-of-the-art measures for evaluating ER results and motivate our edit distance measure. There are many measures used in the Information Retrieval (IR) and AI communities that measure the quality of clustering.

Evaluating clusters is a broader topic than evaluating ER results because ER is a special case of clustering, in which the clusters tend to be small and items in each cluster are typically quite distinct from items in other clusters [62]. Hence, the ER literature has historically only adopted a small subset of all clustering measures for IR.

6.3.1 Pairwise Comparison

The pairwise comparison approach counts the number of pairs of base records to evaluate ER results. To define pairwise measures, we define a function Pairs(P) that takes in a partition P and returns the set of distinct pairs of records that are in the same cluster in P. For example, if $P = \{\langle a, b, c\rangle, \langle d, e\rangle\}$, then $\text{Pairs}(P) = \{(a, b), (b, c), (a, c), (d, e)\}$. We can now define the similarity measures pairwise precision and pairwise recall:

$$\text{PairPrecision}(R, S) = \frac{|\text{Pairs}(R) \cap \text{Pairs}(S)|}{|\text{Pairs}(R)|}$$
$$\text{PairRecall}(R, S) = \frac{|\text{Pairs}(R) \cap \text{Pairs}(S)|}{|\text{Pairs}(S)|}$$

A number of ER papers [25, 54, 81, 35] use pairwise precision and pairwise recall to evaluate ER results while earlier works [95, 46, 3] use the rate of false positives (i.e., 1-PairPrecision(R, S)) and the rate of false negatives (i.e., 1-PairRecall(R, S)) for evaluation. A few works [35, 9] use a variant of pairwise recall while taking into account the reduced number of record comparisons due to blocking techniques. Another work [44] uses a variant of pairwise precision where precision is penalized based on the difference between $|R|$ and $|S|$.

pF_1 The pairwise F_1 measure [61, 19, 28, 74, 20, 30, 33, 62, 75, 49, 79, 56, 73] is the dominant measure in the ER literature and is defined as the harmonic

mean of pairwise precision and pairwise recall:

$$pF_1(R, S) = \frac{2 \times PairPrecision(R, S) \times PairRecall(R, S)}{PairPrecision(R, S) + PairRecall(R, S)}$$

For example, if R={⟨a, b⟩, c, d} and S={⟨a, b⟩, ⟨c, d⟩}, Pr = $\frac{1}{1}$ and Re = $\frac{1}{2}$, making the pairwise $F_1 = \frac{2\times1\times(1/2)}{1+(1/2)} = \frac{2}{3}$ = 66.67%.

6.3.2 Cluster-level Comparison

The cluster-level comparison approach sums the similarity of clusters to evaluate ER results instead of counting pairs of base records.

cF_1 The cluster F_1 measure [72, 49, 79, 56] counts clusters that exactly match and is defined as the harmonic mean of the cluster precision and cluster recall. The cluster precision is defined as $\frac{|R\cap S|}{|R|}$ while the cluster recall is defined as $\frac{|R\cap S|}{|S|}$. Notice that we are now comparing R and S at the cluster level instead of the base record level as in pF_1. Returning to our previous example where R={⟨a, b⟩, c, d} and S={⟨a, b⟩, ⟨c, d⟩}, the precision is $\frac{1}{3}$ while the recall is $\frac{1}{2}$ because exactly one cluster matches among three clusters in R and two clusters in S. The Cluster F_1 is thus $\frac{2\times(1/3)\times(1/2)}{(1/3)+(1/2)} = \frac{2}{5}$ = 40%. We denote the cluster F_1 measure as cF_1.

K The K measure [29, 56] sums the similarities of all cluster pairs and is defined as the geometric mean of the Average Cluster Purity (ACP) and the Average Author Purity (AAP). (Here, Author can be thought of as a cluster in the gold standard.) The ACP is defined as $\frac{1}{N}\sum_{r\in R}\sum_{s\in S}\frac{|r\cap s|^2}{|r|}$ where N is the number of base records. (Notice that the records r and s are considered as sets of base records.) Similarly, the AAP is defined as $\frac{1}{N}\sum_{s\in S}\sum_{r\in R}\frac{|r\cap s|^2}{|s|}$. The K measure is then $\sqrt{ACP \times AAP}$. For example, the ACP value for R and S is $\frac{(2^2/2)+(1^2/2)+(1^2/2)}{4} = \frac{3}{4}$ while the AAP value is $\frac{(2^2/2)+(1^2/1)+(1^2/1)}{4} = 1$, making the K value $\sqrt{\frac{3}{4} \times 1}$= 86.6%.

ccF_1 The closest cluster F_1 measure [11] sums the similarities of all "closest" cluster pairs and is defined as the harmonic mean of the closest cluster precision and closest cluster recall values. The closest cluster precision is defined as $\frac{\sum_{r \in R} \max_{s \in S}(J(r,s))}{|R|}$ where $J(r,s)$ is the Jaccard similarity $\frac{|r \cap s|}{|r \cup s|}$. The closest cluster precision is thus the sum of the maximum Jaccard similarity coefficients for all r's divided by $|R|$. Similarly, the closest cluster recall is defined as $\frac{\sum_{s \in S} \max_{r \in R}(J(s,r))}{|S|}$. For example, the closest cluster precision for R against S is $\frac{(2/2)+(1/2)+(1/2)}{3} = \frac{2}{3}$ while the closest cluster recall is $\frac{(2/2)+(1/2)}{2} = \frac{3}{4}$, making the closest cluster $F_1 = \frac{2 \times (2/3) \times (3/4)}{(2/3)+(3/4)} = \frac{12}{17} = 70.59\%$. We denote closest cluster F_1 as ccF_1. Reference [44] uses a variant of ccF_1 that uses a different similarity equation and gives weights to the coefficients when adding them.

6.3.3 Basic Merge Distance

Edit distance is a common measure in other domains such as string-to-string matching [85, 83] where the basic operations are inserts, deletes, updates, and swaps. In the ER domain (as in clustering), there are fundamental "edit" operations such as cluster splits and merges [64] that are frequently used to resolve records.

A measure based on splits and merges was first proposed by Al-Kamha et al. [2], which we call basic merge distance. Since basic merge distance will be the basis for our generalized merge distance measure, we will describe basic merge distance in more detail than the other measures we have covered.

The basic merge distance (BMD) is defined as the minimum number of cluster merges and splits required to modify an ER result R into another result S. (In most cases, we will have S = G, the gold standard.) In the example from Table 6.1, only one merge is required to get from R_2 to G (i.e., BMD = 1). Result R_1 is comparatively further away from G, as BMD = 2.

In addition, we require that the editing of clusters is only done based on the given clustering information in R and S. Specifically, a merge cannot create newly clustered records that are not in the same cluster in S. For example,

consider $R = \{\langle a, c\rangle, \langle b, d\rangle\}$ and $S = \{\langle a, b\rangle, \langle c, d\rangle\}$. Notice that by merging $\langle a, c\rangle$ and $\langle b, d\rangle$ into $\langle a, b, c, d\rangle$ and then splitting $\langle a, b, c, d\rangle$ into $\langle a, b\rangle$ and $\langle c, d\rangle$, we have a BMD of 2, which is better than splitting $\langle a, c\rangle$ and $\langle b, d\rangle$ into the records a, b, c, d, and then merging a, b into $\langle a, b\rangle$ and c, d into $\langle c, d\rangle$ (resulting in a BMD of 4). However, the first approach creates new clusterings in $\langle a, b, c, d\rangle$ (i.e., a clusters with d, and b clusters with c) that do not appear in the clusters of S, violating our condition. Intuitively, editing R to S requires removing the clustering information found in R only and adding the new information in S.

We now formalize the definition of BMD.

Definition 6.3.1. *A split is an operation* $c \rightarrow c_1, c_2$ *where* $c_1 \cap c_2 = \emptyset$, $c_1 \cup c_2 = c$, *and* $c_1, c_2 \neq \emptyset$. *The result of applying a split to a partition* P *is* $(P - \{c\}) \cup \{c_1, c_2\}$. *A split is a valid operation on* P *if and only if* $c \in P$.

Definition 6.3.2. *A merge is an operation* $c_1, c_2 \rightarrow c$ *where* $c = c_1 \cup c_2$. *The result of applying a merge to a partition* P *is* $(P - \{c_1, c_2\}) \cup \{c\}$. *A merge is a valid operation on* P *if and only if* $c_1, c_2 \in P$.

As a matter of notation, the result of applying an operation o (which can be either a merge or split) to a partition P can be written $P : o$. Note that the result of an operation on a partition is still a partition, so we may apply operations to a partition in sequence. The application of operations o_1 and o_2 to P in sequence can be written $P : o_1 : o_2$. However, we will use commas to separate operations instead: $P : o_1, o_2$.

Definition 6.3.3. *A path from partition* R *to partition* R′ *is a sequence of operations* $o_1, o_2, \ldots, o_n$ *where* $R' = R : o_1, o_2, \ldots, o_n$ *and* o_i *is a valid operation on* $R : o_1, o_2, \ldots, o_{i-1}$ *for all* o_i. *We say that a path is a* legal *path from* R *to* R′ *if for any operation that is a merge* $o_1 = c_1, c_2 \rightarrow c$, *then there exists a cluster* $p \in R'$ *where* $c \subseteq p$.

Definition 6.3.4. *The* BMD *from a partition* R *to a partition* S *is the number of operations in the shortest legal path from* R *to* S.

While the BMD measure also operates at the cluster level, we categorize it separately from cluster-level comparison approaches because clusters are dynamically edited using basic operations instead of being statically compared.

6.3.4 Variation of Information

Another work closely to the merge distance measure is a state-of-the-art clustering measure called Variation of Information [63] (VI) where we measure the "information" lost and gained while converting one clustering to another as follows:

$$VI(R,S) = H(R) + H(S) - 2I(R,S)$$

Functions H and I represent, respectively, the total entropy of the individual clusters and the mutual information between R and S.

$$H(R) = -\sum_{r \in R} \frac{|r|}{N} \log \frac{|r|}{N}$$

$$I(R,S) = \sum_{r \in R} \sum_{s \in S} \frac{|r \cap s|}{N} \log \frac{|r \cap s| \times N}{|r| \times |s|}$$

6.4 Generalized Merge Distance

The definition of basic merge distance (Section 6.3.3) immediately raises some questions on how we can generalize it. In some cases, we may want to penalize splits more than merges, or vice versa. Further, the "badness" of a split or merge may depend on the sizes of the clusters that are being merged or split. In this section, we define a generalized merge distance (GMD) that creates a larger space of possible measures. In Section 6.4.3 we show that this space

includes distance measures closely related to the pairwise precision and recall measures of Section 6.3.1, as well as the VI measure of Section 6.3.4.

Definition 6.4.1. *The* f_m, f_s *generalized merge distance* $GMD_{f_m,f_s}(R, S)$ *from a partition* R *to another partition* S *is the minimum cost of a legal path from* R *to* S*, where:*

- *the cost of a merge operation* $x, y \rightarrow z$ *is* $f_m(|x|, |y|)$*, and*
- *the cost of a split operation* $z \rightarrow x, y$ *is* $f_s(|x|, |y|)$*.*

Clearly, the BMD measure described in Section 6.3.3 is the same as the GMD measure when $f_m(x, y) = f_s(x, y) = 1$.

We assume some reasonable properties of the functions f_m and f_s:

1. Operations cannot have negative cost: $f_m(x, y) \geq 0$ and $f_s(x, y) \geq 0$.
2. The cost functions are symmetric: $f_m(x, y) = f_m(y, x)$ and $f_s(x, y) = f_s(y, x)$.
3. The cost functions monotonically increase with their parameters: $f_m(x, y) \leq f_m(x + j, y + k)$ and $f_s(x, y) \leq f_s(x + j, y + k)$ for non-negative j, k.

Given the above three properties, we can prove there exists a minimum-cost legal path from a partition R to a partition S where all of the split operations precede the merge operations. This result vastly reduces the search space for a minimum-cost path and thus leads to an efficient algorithm for computing GMD.

We begin by noting that merge and split operations are often commutative. That is, $R : o_1, o_2 = R : o_2, o_1$ for many pairs of operations. In fact, operations are always commutative except when the input (or one of the two inputs) of one operation is the output of another.

As an example, consider the path defined by operations o_1, o_2, o_3, o_4, o_5 in Table 6.2. Operations o_1 and o_3 can be executed in either order, as their inputs and outputs do not overlap. Operation o_4, on the other hand, takes the output of o_3 and one of the outputs of o_2 as input. Therefore, o_4 must be performed after both o_2 and o_3.

The commutativity of operations in a path can be represented with a precedence graph.

Table 6.2: Example path

Operation	Result
—	$\{\langle a,b,c\rangle, \langle d,e\rangle, \langle f\rangle, \langle g\rangle\}$
$o_1 = \langle a,b,c\rangle, \langle d,e\rangle \to \langle a,b,c,d,e\rangle$	$\{\langle a,b,c,d,e\rangle, \langle f\rangle, \langle g\rangle\}$
$o_2 = \langle a,b,c,d,e\rangle \to \langle a,e\rangle, \langle b,c,d\rangle$	$\{\langle a,e\rangle, \langle b,c,d\rangle, \langle f\rangle, \langle g\rangle\}$
$o_3 = \langle f\rangle, \langle g\rangle \to \langle f,g\rangle$	$\{\langle a,e\rangle, \langle b,c,d\rangle, \langle f,g\rangle\}$
$o_4 = \langle f,g\rangle, \langle b,c,d\rangle \to \langle b,c,d,f,g\rangle$	$\{\langle a,e\rangle, \langle b,c,d,f,g\rangle\}$
$o_5 = \langle a,e\rangle, \langle b,c,d,f,g\rangle \to \langle a,b,c,d,e,f,g\rangle$	$\{\langle a,b,c,d,e,f,g\rangle\}$

Definition 6.4.2. *The precedence graph G for a path $o_1, o_2, \ldots, o_n$ is a directed graph that specifies the order in which the operations must take place. We build the graph by creating a vertex for each operation. For each pair of operations o_i, o_j with $i < j$, if an output cluster p of o_i is an input of o_j and there is no k between i and j where o_k also has p as an input, then we add an edge from o_i to o_j.*

Figure 6.1 shows the precedence graph for the path from our example. For clarity, the edges are labeled with the cluster that creates the dependence between two operations. Dashed lines in this figure are not part of the precedence graph; they are present only to show the inputs and outputs of operations that would not be apparent from the precedence graph alone. The graph clearly illustrates that o_3 can commute with o_1 and o_2, but no other pairs of operations are commutable. To construct a path with all splits preceding all merges, we can often simply use commutativity to move the split operations to the front. However, if there is an edge from a merge to a split in the precedence graph, commutativity will not help. In that case, we must invoke a more complex transformation which we call a "merge-split swap".

We will first illustrate a merge-split swap with an example, and then formally define the transformation. In our running example, merge operation o_1 has an edge to split operation o_2 (which we would like to move "up" to the beginning of the path). We will "swap" the order of the operations by replacing o_1 with a split operation o'_1 with an edge to a merge operation o'_2 that replaces o_2. The input of o'_1 will be $\langle a,b,c\rangle$, the larger of the two inputs of

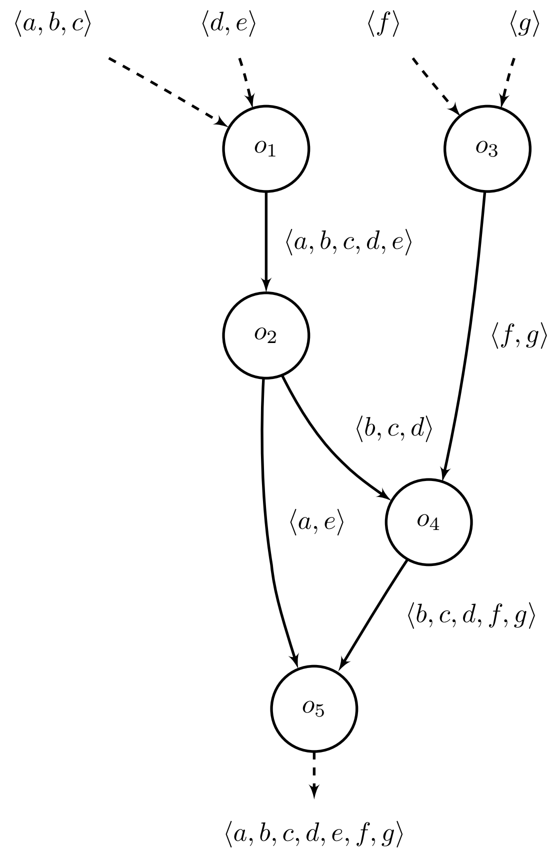

Figure 6.1: A precedence graph.

o_1. We then split that input into two clusters, one of which has the same size as the smaller of the outputs of o_2. We will arbitrarily choose $\langle a, b\rangle$ as the output of size 2, and thus $o'_1 = \langle a, b, c\rangle \rightarrow \langle a, b\rangle, \langle c\rangle$. We will define o'_2 as the merge of the "leftover" cluster $\langle c\rangle$ with the smaller of the inputs to o_1. So $o'_2 = \langle c\rangle, \langle d, e\rangle \rightarrow \langle c, d, e\rangle$.

Note that the sequences o_1, o_2 and o'_1, o'_2 do not produce the same result (the former yields $\langle a, e\rangle, \langle b, c, d\rangle$ and the latter yields $\langle a, b\rangle, \langle c, d, e\rangle$). However, the two sequences each produce 2 clusters of the same size. To compensate for the different output, we modify operations "downstream" from o'_1, o'_2 so they work with the new clusters, as shown in Figure 6.2.

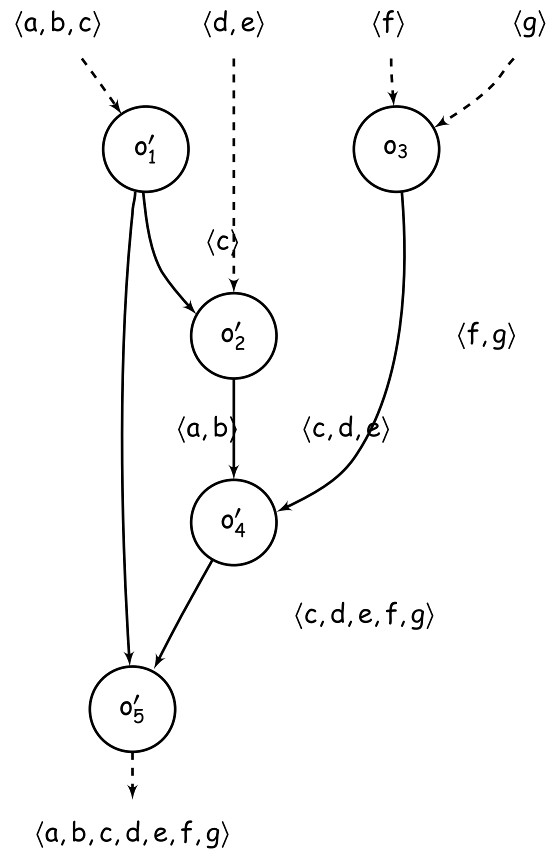

Figure 6.2: Precedence graph for the transformed path.

We note that the result of the transformed path is the same as the result of the original path: $\{\langle a, b, c, d, e, f, g\rangle\}$. But suppose the original path only had operations $o_1, \ldots, o_4$. Then the result before the transformation would be $\{\langle a, e\rangle, \langle b, c, d, f, g\rangle\}$, and the result of the transformed path would be $\{\langle a, b\rangle, \langle c, d, e, f, g\rangle\}$—a different result! As it turns out, as long as the original path was a legal path, the result after the transformation will be the same. Removing o_5 from the path changes the result in such a way that o_1 becomes an invalid merge ($\langle a, b, c, d, e\rangle$ is not a subset of any cluster in the result). Lemma 6.4.4 will prove that the merge-split swap transformation can be applied to any legal path without changing the result of the path.

Figure 6.3 provides a generalized depiction of the merge-split swap transformation. It will be useful as a reference for the following formal definition.

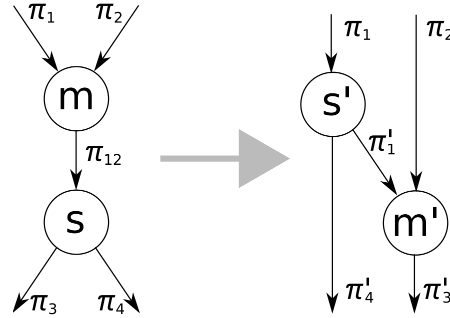

Figure 6.3: A merge with an edge to a split is rewritten into a split with an edge to a merge.

Definition 6.4.3. *The merge-split swap transformation is defined as follows. Consider a path with a merge operation* $m = \pi_1, \pi_2 \rightarrow \pi_{12}$ *with an edge to a split operation* $s = \pi_{12} \rightarrow \pi_3, \pi_4$ *in the precedence graph. Let* $|\pi_1| \geq |\pi_2|$ *and* $|\pi_3| \geq |\pi_4|$.

We construct two new operations: $s' = \pi_1 \rightarrow \pi'_1, \pi'_4$ *and* $m' = \pi'_1, \pi_2 \rightarrow \pi'_3$. *When constructing these operations, we choose a* $\pi'_4 \subseteq \pi_1$ *where* $|\pi'_4| = |\pi_4|$. *(It is possible in this scenario that* $|\pi'_1| = 0$*, which would make* s' *and* m' *illegal operations. However, in this case we can consider* s' *and* m' *to be zero cost no-ops) The transformation begins by replacing operation* m *with* s'*, and replacing* s *with* m'.

The remaining changes to the path simply fix subsequent operations to use π'_3 *and* π'_4 *as input, instead of* π_3 *and* π_4. *Defining this process formally, we construct a one-to-one mapping* M *of records. We will employ an extended notation that allows us to apply* M *to sets and operations, e.g.* $M(S) = \{M(x) : x \in S\}$ *and* $M(\pi_1, \pi_2 \rightarrow \pi_{12}) = M(\pi_1), M(\pi_2) \rightarrow M(\pi_{12})$. *For all records* $r \notin \pi_3 \cup \pi_4$*, we define* $M(r) = r$. *For the records in* π_3 *and* π_4*, we define* M *such that* $M(\pi_3) = \pi'_3$ *and* $M(\pi_4) = \pi'_4$.

The merge-split swap transformation is complete when we replace each subsequent operation o *in the path with* $M(o)$.

Lemma 6.4.4. *Given a legal path* p *from* R *to* S*, applying a merge-swap split transformation results in a legal path* p' *from* R *to* S *with cost less than or equal to the cost of* p.

Proof. In this proof, we will reuse the terms introduced in the definition of the merge-split swap: clusters $\pi_1, \pi_2, \pi_3, \pi_4, \pi'_3, \pi'_4$, operations m, s, s', m', and mapping M.

We make three claims about p':

1. Path p' is a path from R to S.
2. Path p' is a *legal* path from R to S.
3. The cost of p' is less than or equal to the cost of p.

To prove Claim 1, we first name the operations in our paths $o_1, \ldots, o_n$ and $o'_1, \ldots, o'_n$, respectively. Suppose that the split to be transformed is $o_k = s$, and therefore $o'_k = m'$. We note that $R : o'_1, \ldots, o'_k = M(R : o_1, \ldots, o_k)$. That is, after applying p' up through the kth operation, we have the same result as applying the first k operations of p and then applying M to each set in the resulting clusters. For each subsequent operation o, $o' = M(o)$, so it is clear that $R : o'_1, \ldots, o'_{k+i} = M(R : o'_1, \ldots, o'_{k+i})$ for positive i. Therefore, $R : p' = M(R : p) = M(S)$.

Now, the operation m was a merge of π_1 and π_2. Since p is a legal path, $\pi_1 \cup \pi_2$ must be a subset of some cluster π_f in S. Since $\pi_1 \cup \pi_2 = \pi_3 \cup \pi_4$, we can write $\pi_f = \pi_3 \cup \pi_4 \cup \pi_{extra}$. Applying M to π_f, we get $\pi'_3 \cup \pi'_4 \cup \pi_{extra} = \pi_1 \cup \pi_2 \cup \pi_{extra}$. So $M(\pi_f) = \pi_f$. Since M is an identity function for all records not in π_1 or π_2, the other clusters in S are also unaffected by the application of M. Therefore, $M(S) = S$, which proves Claim 1.

Further, since records r and $M(r)$ are always in the same cluster in the result, if any merge operation o is a legal merge in p, then the corresponding operation $M(o)$ in p' must also be a legal merge. The only questionable merge remaining is m', which is legal since π'_3 is a subset of $\pi_f \in S$. Therefore, p' is a legal path, which proves Claim 2.

To prove Claim 3 we note that applying M to an operation o cannot change the size of the clusters involved, and therefore, the cost of M(o) must be the same as the cost of o. So all corresponding operations in p and p′ have equal cost, other than m, s, s′ and m′. The only difference in cost can come from the cost difference between s and s′, and m and m′. The combined cost for m and s is:

$$f_m(|\pi_1|, |\pi_2|) + f_s(|\pi_3|, |\pi_4|)$$

The combined cost for m′ and s′ is:

$$f_m(|\pi'_1|, |\pi_2|) + f_s(|\pi'_1|, |\pi'_4|)$$

Recall from Definition 6.4.3 that $|\pi_1| \geq |\pi_2|$, $|\pi_3| \geq |\pi_4|$, and $|\pi'_4| = |\pi_4|$. We also have that $|\pi_1| + |\pi_2| = |\pi_3| + |\pi_4| = |\pi'_3| + |\pi'_4|$ and $|\pi_1| = |\pi'_1| + |\pi'_4|$.

From these facts, we find that $|\pi'_1| \leq |\pi_1|$, $|\pi'_1| \leq |\pi_3|$, and of course $|\pi'_4| = |\pi_4|$. Comparing the cost expressions with these three facts, we see that all arguments to the f_m and f_s functions are no greater for operations m′ and s′ than they are for m and s. By the monotonicity property of the split and merge functions, the cost for m′ and s′ must be less than or equal to the cost of m and s. Therefore the cost of p′ is less than or equal to the cost of p. This proves Claim 3 and therefore we have proven the lemma. □

Theorem 6.4.5. *For any partitions* R *and* S*, there exists a minimum-cost legal path from* R *to* S *where the precedence graph for the path has no edge from any merge operation to a split operation.*

Proof. Consider a minimum-cost legal path p from R to S. If the precedence graph of p has a merge with an edge to a split, we can apply a merge-split swap transformation in order to obtain a path that (according to Lemma 6.4.4) must also be a legal and minimum-cost path from R to S. In fact, we can repeat the transformation until there no longer exists an edge from a merge operation to a split operation in the precedence graph of the path. □

Theorem 6.4.6. *For any partitions* R *and* S*, there exists a minimum-cost legal path from* R *to* S *where all split operations precede all merge operations.*

Proof. This result follows directly from Theorem 6.4.5 and the rules of commutativity. Since there is no edge from a merge operation to a split operation in the precedence graph, the split operations may all be commuted to the beginning of the path. □

We will use the term "splits-first path" to refer to a path with all split operations preceding all merge operations. We note that any splits-first path from R to S is also a legal path, as if there were an operation that merged two clusters that are not merged in S, then subsequent operations cannot be splits, and thus the result of the path could not be S.

We also define the "split point" of a splits-first path.

Definition 6.4.7. *In a splits-first path from* R *to* S*, we can apply all of the split operations to* R *to get a partition we call the* split point*. We denote the split point of a path* p *applied to a partition* R *as* R : splits(p)*.*

6.4.1 Operation Order Independence

Definition 6.4.8. *A* fragment *of partitions* R *and* S *is any set of the form* $s \cap r$ *where* $r \in R$*,* $s \in S$*, and* $r \cap s \neq \varnothing$*. We denote the set of all fragments of* R *and* S *as* $R \sqcap S$*. Since every record is in exactly one fragment,* $R \sqcap S$ *is a partition as well.*

As an example, suppose $R = \{\langle a, c, e\rangle, \langle b, d, f\rangle\}$ and $S = \{\langle a, b, c\rangle, \langle d, e, f\rangle\}$. In that case, $R \sqcap S = \{\langle a, c\rangle, \langle e\rangle, \langle b\rangle, \langle d, f\rangle\}$.

If a splits-first path fails to break R down to the fragments of R and S by the end of the split phase, then it cannot possibly result in S after the merge phase is complete. Therefore, the fragments of R and S can be seen as the minimum required splitting performed by a splits-first path from R to S.

Lemma 6.4.9. *Given a splits-first path from* R *to* S*, any cluster in* R : splits(p) *is a subset of some* $f \in R \sqcap S$*.*

Proof. Consider any cluster $\pi \in R : \text{splits}(p)$. Since $\pi \in R : \text{splits}(p)$, it is the result of a series of split operations on R. Therefore, $\pi \subseteq r$ for some $r \in R$. The remaining operations in the path are all merges, so $\pi \subseteq s$ for some cluster $s \in S$. Now we have that $\pi \subseteq r \cap s$, and $r \cap s \in R \sqcap S$. So any cluster in $R : \text{splits}(p)$ must be a subset of some $f \in R \sqcap S$. □

We now define a term for a path that performs this minimum required splitting:

Definition 6.4.10. *A* bare necessities path *from* R *to* S *is any splits-first path* p *from* R *to* S *where* $R : \text{splits}(p) = R \sqcap S$.

For many split and merge functions, there always exists a bare necessities path from R to S that is also a minimum-cost path. In fact, if the functions satisfy a property we call "operation order independence", then any bare necessities path from R to S is in fact a minimum-cost path.

Definition 6.4.11. *We say that a function* F *is* operation order independent *if it satisfies* $F(x, y) + F(x + y, z) = F(x, z) + F(x + z, y)$ *for all* x, y, z.

We call this property operation order independence because it implies that the order in which certain operations are performed is unimportant. Suppose that we wish to merge three clusters π_x, π_y, and π_z (with sizes x, y, and z, respectively) all together into a single cluster. If we merge π_x and π_y together first, and then merge the resulting cluster with π_z, observe that the resulting cost would be given by the left-hand side of the equation in Definition 6.4.11. The cost of merging π_x and π_z together first, and then merging π_y with the result is given by the right-hand side of the equation, and therefore with operation order independence, these two paths would have the same cost.

Now that we have defined operation order independence, we note two simple classes of functions that satisfy this property: $F(x, y) = k$ and $F(x, y) = kxy$. (One can easily verify the property holds for these classes by plugging them into the equation in Definition 6.4.11.) The first class includes the BMD measure of Section 6.3.3 and the second class of measures can be used to directly compute (see Section 6.4.3) the pairwise precision and recall measures

of Section 6.3.1. We also note that functions of the form $F(x, y) = k_1 + k_2xy$ also satisfy this property, and these may provide an interesting "blend" of the two classes above. These are not the only functions that satisfy this property, and it turns out that the class of functions that satisfy the property (studied in [48]) is, in fact, quite vast.

The class of operation order independent functions has been studied in [48]. That paper proves the general form of an operation order independent function to be:

$$F(x, y) = B(x, y) + f(x + y) - f(x) - f(y)$$

In this formulation, $f(x)$ is any arbitrary function. On the other hand, $B(x, y)$ is a function that must satisfy many properties, including skew symmetry: $B(x, y) + B(y, x) = 0$. Since we assume the property of symmetry on our cost functions, we require that $B(x, y) = 0$. Therefore, the most general form for operation order independent functions is $f(x + y) - f(x) - f(y)$. We can verify that if $f(x) = -k$ then $F(x) = k$ and if $f(x) = \frac{k}{2}x^2$ then $F(x, y) = kxy$. This result shows that there is actually a large class of functions that satisfy operation order independence.

Lemma 6.4.12. *Given clusters $\pi_1, \pi_2, \ldots, \pi_n$, if f_m is operation order independent, then all sequences of merges that result in $\bigcup \pi_i$ have equal cost.*

Proof. Consider the class of pairs P with the left element being a cluster and the right element being a numerical cost associated with that cluster. We map the clusters π_i to the corresponding pair $(\pi_i, 0)$, indicating that we can obtain any π_i with zero cost. We can now define an operator $\oplus$ on P that performs the merge of the clusters in two pairs, and computes the total cost necessary to obtain that cluster:

$$(p_1, x) \oplus (p_2, y) = (p_1 \cup p_2, x + y + f_m(|p_1|, |p_2|)$$

Due to the properties f_m has (symmetry and operation order independence), it is easy to show that $\oplus$ is both associative and commutative.

Any sequence of merges of the base clusters that results in $\bigcup \pi_i$ maps to an application of the $\oplus$ operator on all of the π_i clusters. Since $\oplus$ is commutative and associative, the order of merge operations does not affect the cost to generate the result. So all sequences of merges with that result have equal cost. □

Lemma 6.4.13. *If f_s is operation order independent, then all sequences of splits starting from a cluster π that result in $\pi_1, \pi_2, \ldots, \pi_n$ have equal cost.*

Proof. We note that any sequence of splits leading to the π_i clusters can be reversed to obtain a sequence of merges from the π_i clusters to π. If we let $f_m = f_s$, then the cost of the merge sequence is equal to the cost of the split sequence. By applying Lemma 6.4.12, we get that all such sequences have equal cost. □

Theorem 6.4.14. *If both f_m and f_s are operation order independent, then any bare necessities path from R to S is a minimum-cost legal path from R to S.*

Proof. Take any minimum-cost splits-first path p_{min} from R to S. (By Theorem 6.4.6 such a path always exists.) Now take any bare necessities path p from R to S. By Lemma 6.4.9, every cluster in $R : splits(p_{min})$ is a subset of some cluster in $R : splits(p)$ (which is $R \sqcap S$ by definition of a bare necessities path). Given this subset relationship, it is clear that we can append split operations to the splits phase of p such that we arrive at $R : splits(p_{min})$. Further, we can append merge operations to the end of the splits phase to merge these clusters back down to $R \sqcap S$. With this process, we can extend p to a new path p' from R to S where $R : splits(p') = R : splits(p_{min})$.

The new path p' contains all of the operations of p plus some extra operations in the middle. Therefore, the cost of p' is greater than or equal to the cost of p. However, by Lemma 6.4.12 and Lemma 6.4.13, the cost of p' must be equal to the cost of p_{min}, since $R : splits(p') = R : splits(p_{min})$. Path p_{min} is a minimum-cost legal path from R to S, and p must have lower or equal cost. So the bare necessities path p is also a minimum-cost legal path from R to S. □

We now demonstrate that a bare necessities path may be easily constructed for any given partitions R and S. Let p_0 be the null path that performs no operations, so $R = R : p_0$. We define the splits in this path inductively: $p_{i+1} = p_i, \pi \rightarrow \pi_1, \pi_2$ where $\pi \in R : p_0$, and non-empty π_1 and π_2 are, respectively, $\pi \cap s$ and $\pi - s$ for some $s \in S$. We extend the path until all clusters in $R : p_i$ are subsets of clusters in S.

We then extend the path further with only merge operations: $\pi_1, \pi_2 \rightarrow \pi_1 \cup \pi_2$ where $\pi_1, \pi_2 \subseteq s$ for some $s \in S$. This completes a bare necessities path from R to S.

This construction suggests an algorithm for computing GMD when the merge and split functions are operation order independent functions, since it suffices to construct a bare necessities path and compute its cost. We will describe such an algorithm (called the Slice algorithm) in Section 6.5.

6.4.2 Merge Precision and Recall

Another idea inspired by Theorem 6.4.6 is that we can consider the costs of splitting and merging separately. When a cluster in the result must be split in a path to the gold standard, it is usually due to a false positive in the result.[1] When two clusters are merged, it is usually due to a false negative. Therefore, the total cost of the split operations is much like an inverse measure of precision, and the total cost of the merge operations is much like an inverse measure of recall.

We can further use Theorem 6.4.6 to normalize these measures. The cost of all the split operations can never be more than the cost of splitting all clusters in R down into individual base records. Likewise, the cost of all the merge operations can never be more than the cost of merging individual base records up into the clusters in S. Let $\perp$ represent a partition in which each base

[1] In some configurations, a minimum-cost path may split clusters that are found together in the destination. Consider $R = \{\langle a, b, c\rangle, d\}$, $S = \{\langle a, b, c, d\rangle\}$, $f_s(x, y) = 0$, and $f_m = x^3 + y^3$. A minimum-cost path will split $\langle a, b, c\rangle$ to avoid the large cost of a merge with a cluster of size 3.

records is alone in its own partition. We can then normalize merge precision and recall using the factors $GMD_{f_m,f_s}(R, \perp)$ and $GMD_{f_m,f_s}(\perp, S)$, respectively.

Let $C_m(R, S)$ and $C_s(R, S)$ refer to the total cost of merges and splits (respectively) in a minimum-cost path from R to S. We can then define merge precision and recall as follows:

Definition 6.4.15. *The* f_m, f_s *merge precision from* R *to* S *is defined according to the following formula:*

$$MP_{f_m,f_s}(R, S) = 1 - \frac{C_s(R, S)}{GMD_{f_m,f_s}(R, \perp)}$$

The f_m, f_s *merge recall from* R *to* S *is defined according to this similar formula:*

$$MR_{f_m,f_s}(R, S) = 1 - \frac{C_m(R, S)}{GMD_{f_m,f_s}(\perp, S)}$$

Note that we subtract the normalized distance from 1 to turn these measures into similarity measures, rather than distance measures.

With a definition of precision and recall from the merge perspective, it is possible to use the standard methods (e.g., F_1) to combine the two into a single number.

6.4.3 Relationship to Other Measures

Several other measures are closely related to GMD. First, the BMD measure in [2] is exactly our GMD measure when $f_m(x, y) = f_s(x, y) = 1$. Second, the VI measure (Section 6.3.4) is a special case of GMD where f_m and f_s are chosen as follows:

Theorem 6.4.16. $VI(R, S) = GMD(R, S)$ *when* $f_m(x, y) = f_s(x, y) = h(x + y) - h(x) - h(y)$*, with* $h(x) = \frac{x}{N} \log \frac{x}{N}$.

Proof. First, we note that f_m and f_s here are order operation independent functions, as they have the form shown in Section 6.4.1 to be the most general

form of an order independent function. Theorem 6.4.14 therefore tells us that the GMD from R to S is the cost of any bare necessities path from R to S.

Construct a bare necessities path $p = o_1, o_2, \ldots, o_m$ with $R : p = S$. We will use the shorthand notation $R^{(i)}$ to refer to $R : o_1, \ldots, o_i$, with $R^{(0)} = R$ and $R^{(m)} = S$. We show by induction that $GMD_{f_m,f_s}(R, R^{(k)}) = VI(R, R^{(k)})$ for all $0 \le k \le m$. We note that $GMD_{f_m,f_s}(R, R^{(0)}) = VI(R, R^{(0)}) = 0$ which provides with a base case for our induction.

Now, assuming the inductive hypothesis, $GMD_{f_m,f_s}(R, R^{(i-1)}) = VI(R, R^{(i-1)})$, we will show that $GMD_{f_m,f_s}(R, R^{(i)}) = VI(R, R^{(i)})$. Let us evaluate the change in VI when we perform operation o_i.

$$\begin{aligned}
\Delta VI &= VI(R, R^{(i)}) - VI(R, R^{(i-1)}) \\
&= H(R) + H(R^{(i)}) - 2I(R, R^{(i)}) \\
&\quad - H(R) - H(R^{(i-1)}) + 2I(R, R^{(i-1)}) \\
&= H(R^{(i)}) - H(R^{(i-1)}) - 2(I(R, R^{(i)}) - I(R, R^{(i-1)})) \\
&= \Delta H - 2\Delta I
\end{aligned}$$

In the above equation, we have introduced $\Delta H = H(R^{(i)}) - H(R^{(i-1)})$ and $\Delta I = I(R, R^{(i)}) - I(R, R^{(i-1)})$. The value of ΔH is the change in entropy created by performing the operation, and ΔI is the change in information shared with R.

We must handle the case of a split separately from the case of a merge. So for the time being, suppose that o_i is a split: $o_i = \pi \to \pi_1, \pi_2$. First, let us consider ΔH:

$$\begin{aligned}
\Delta H &= H(R^{(i)}) - H(R^{(i-1)}) \\
&= -\sum_{r \in R^{(i)}} h(|r|) + \sum_{r \in R^{(i-1)}} h(|r|) \\
&= -h(|\pi_1|) - h(|\pi_2|) + h(|\pi|)
\end{aligned}$$

$$= h(|\pi|) - h(|\pi_1|) - h(|\pi_2|)$$

Now, we consider ΔI:

$$\begin{aligned}
\Delta I &= I(R, R^{(i)}) - I(R, R^{(i-1)}) \\
&= \sum_{r \in R} \sum_{s \in R^{(i)}} \frac{|r \cap s|}{N} \log \frac{|r \cap s| \times N}{|r| \times |s|} \\
&\quad - \sum_{r \in R} \sum_{s \in R^{(i-1)}} \frac{|r \cap s|}{N} \log \frac{|r \cap s| \times N}{|r| \times |s|}
\end{aligned}$$

The only difference between these two summations is in the terms involving π for the first sum, and π_1, π_2 in the second sum. Since o_i is a split in a splits-first path, it is preceded only by splits, and therefore π, π_1, π_2 are all subsets of some single cluster $r^* \in R$. Therefore, for any $r \in R$ where $r \neq r^*$, $r \cap \pi = \emptyset$, and the same is true of π_1 and π_2. This fact brings ΔI down to the following:

$$\begin{aligned}
\Delta I &= \frac{|r^* \cap \pi|}{N} \log \frac{|r^* \cap \pi| \times N}{|r^*| \times |\pi|} \\
&\quad - \frac{|r^* \cap \pi_1|}{N} \log \frac{|r^* \cap \pi_1| \times N}{|r^*| \times |\pi_1|} \\
&\quad - \frac{|r^* \cap \pi_2|}{N} \log \frac{|r^* \cap \pi_2| \times N}{|r^*| \times |\pi_2|} \\
&= \frac{|\pi|}{N} \log \frac{|\pi| \times N}{|r^*| \times |\pi|} - \frac{|\pi_1|}{N} \log \frac{|\pi_1| \times N}{|r^*| \times |\pi_1|} \\
&\quad - \frac{|\pi_2|}{N} \log \frac{|\pi_2| \times N}{|r^*| \times |\pi_2|} \\
&= \frac{|\pi|}{N} \log \frac{N}{|r^*|} - \frac{|\pi_1|}{N} \log \frac{N}{|r^*|} - \frac{|\pi_2|}{N} \log \frac{N}{|r^*|} \\
&= \frac{|\pi| - |\pi_1| - |\pi_2|}{N} \log \frac{N}{|r^*|} \\
&= \frac{0}{N} \log \frac{N}{|r^*|}
\end{aligned}$$

$$= 0$$

Now, we continue to compute ΔVI.

$$\begin{aligned}\Delta VI &= \Delta H - 2\Delta I \\ &= h(|\pi|) - h(|\pi_1|) - h(|\pi_2|) - 2 \times 0 \\ &= h(|\pi_1| + |\pi_2|) - h(|\pi_1|) - h(|\pi_2|) \\ &= f_s(|\pi_1|, |\pi_2|)\end{aligned}$$

So the cost of the operation o_i is exactly ΔVI, which allows us to prove the inductive step in the case that o_i is a split:

$$\begin{aligned}VI(R, R^{(i)}) &= VI(R, R^{(i-1)}) + \Delta VI \\ &= GMD_{f_m,f_s}(R, R^{(i-1)}) + f_s(|\pi_1|, |\pi_2|) \\ &= GMD_{f_m,f_s}(R, R^{(i)})\end{aligned}$$

Now, we need to repeat this procedure assuming that o_i is a merge: $o_i = \pi_1, \pi_2 \rightarrow \pi$. Starting with computing ΔH:

$$\begin{aligned}\Delta H &= H(R^{(i)}) - H(R^{(i-1)}) \\ &= -\sum_{r \in R^{(i)}} h(|r|) + \sum_{r \in R^{(i-1)}} h(|r|) \\ &= h(|\pi_1|) + h(|\pi_2|) - h(|\pi|)\end{aligned}$$

Note that ΔH in the merge case is the opposite of ΔH in the split case.

Now, we proceed to compute ΔI. The merge case is trickier, though, as π_1 and π_2 may have components of many clusters of R. Since the path is a bare necessities path, we can let $\pi_1 = \bigcup_{f \in F} f$ for some set of fragments $F \subset R \sqcap S$. We will use the notation f_j to refer to an individual element of F. Similarly, let $\pi_2 = \bigcup_{g \in G} g$, with $G \subset R \sqcap S$, and g_j referring to an individual element of G.

Since $F, G \subset R \sqcap S$, each element of these sets is a subset of some cluster

in R. That is, for each f_j, there is a corresponding $c_j \in R$ where $f_j \subseteq c_j$. Likewise, for each g_j, there is a corresponding $d_j \in R$ where $g_j \subseteq d_j$. We will continue to use c_j and d_j to refer to the clusters in R that f_j and g_j came from, respectively. We note that $\pi_1 \cap \pi_2 = \varnothing$, so for all j, k, $c_j \neq d_k$.

We begin with the ΔI equation from earlier, and simplify it by leaving only the terms that use the clusters involved in the merge operation:

$$
\begin{aligned}
\Delta I &= \sum_{r \in R} \sum_{s \in R^{(i)}} \frac{|r \cap s|}{N} \log \frac{|r \cap s| \times N}{|r| \times |s|} \\
&\quad - \sum_{r \in R} \sum_{s \in R^{(i-1)}} \frac{|r \cap s|}{N} \log \frac{|r \cap s| \times N}{|r| \times |s|} \\
&= \sum_{j} \frac{|c_j \cap \pi|}{N} \log \frac{|c_j \cap \pi| \times N}{|c_j| \times |\pi|} \\
&\quad + \sum_{j} \frac{|d_j \cap \pi|}{N} \log \frac{|d_j \cap \pi| \times N}{|d_j| \times |\pi|} \\
&\quad - \sum_{j} \frac{|c_j \cap \pi_1|}{N} \log \frac{|c_j \cap \pi_1| \times N}{|c_j| \times |\pi_1|} \\
&\quad - \sum_{j} \frac{|d_j \cap \pi_2|}{N} \log \frac{|d_j \cap \pi_2| \times N}{|d_j| \times |\pi_2|} \\
&= \sum_{j} \left[\frac{|f_j|}{N} \log \frac{|f_j| \times N}{|c_j| \times |\pi|} + \frac{|g_j|}{N} \log \frac{|g_j| \times N}{|d_j| \times |\pi|} \right] \\
&\quad - \sum_{j} \left[\frac{|f_j|}{N} \log \frac{|f_j| \times N}{|c_j| \times |\pi_1|} + \frac{|g_j|}{N} \log \frac{|g_j| \times N}{|d_j| \times |\pi_2|} \right]
\end{aligned}
$$

$$
\begin{aligned}
&= \sum_j \frac{|f_j|}{N}\left(\log\frac{|f_j|\times N}{|c_j|\times|\pi|} - \log\frac{|f_j|\times N}{|c_j|\times|\pi_1|}\right) \\
&\quad + \sum_j \frac{|g_j|}{N}\left(\log\frac{|g_j|\times N}{|d_j|\times|\pi|} - \log\frac{|g_j|\times N}{|d_j|\times|\pi_2|}\right) \\
&= \sum_j \left[\frac{|f_j|}{N}\log\frac{|\pi_1|}{|\pi|} + \frac{|g_j|}{N}\log\frac{|\pi_2|}{|\pi|}\right] \\
&= \frac{|\pi_1|}{N}\log\frac{|\pi_1|}{|\pi|} + \frac{|\pi_2|}{N}\log\frac{|\pi_2|}{|\pi|} \\
&= \frac{|\pi_1|}{N}\log|\pi_1| + \frac{|\pi_2|}{N}\log|\pi_2| - \frac{|\pi_1|+|\pi_2|}{N}\log|\pi| \\
&= \frac{|\pi_1|}{N}\log\frac{|\pi_1|}{N} + \frac{|\pi_2|}{N}\log\frac{|\pi_2|}{N} - \frac{|\pi|}{N}\log\frac{|\pi|}{N} \\
&= h(|\pi_1|) + h(|\pi_2|) - h(|\pi|)
\end{aligned}
$$

Again, we continue to compute ΔVI.

$$
\begin{aligned}
\Delta VI &= \Delta H - 2\Delta I \\
&= h(|\pi_1|) + h(|\pi_2|) - h(|\pi|) \\
&\quad - 2(h(|\pi_1|) + h(|\pi_2|) - h(|\pi|)) \\
&= h(|\pi|) - h(|\pi_1|) - h(|\pi_2|) \\
&= h(|\pi_1| + |\pi_2|) - h(|\pi_1|) - h(|\pi_2|) \\
&= f_m(|\pi_1|, |\pi_2|)
\end{aligned}
$$

So the cost of the operation o_i is exactly ΔVI, which allows us to prove the inductive step in the case that o_i is a merge:

$$
\begin{aligned}
VI(R, R^{(i)}) &= VI(R, R^{(i-1)}) + \Delta VI \\
&= GMD_{f_m,f_s}(R, R^{(i-1)}) + f_m(|\pi_1|, |\pi_2|)
\end{aligned}
$$

$$= GMD_{f_m,f_s}(R, R^{(i)})$$

We have proven the inductive step in the case that o_i is either a merge or a split, and therefore we have proven the inductive step completely. Therefore, by induction, $GMD_{f_m,f_s}(R, R^{(k)}) = VI(R, R^{(k)})$ for all $0 \leq k \leq m$, and therefore $GMD_{f_m,f_s}(R, S) = VI(R, S)$. □

Third, the pF_1 distance can be computed directly using GMD. The theorem below shows how to compute pairwise precision and pairwise recall using GMD. The pF_1 distance is then the harmonic mean of the two values. We use the symbol $\perp$ to refer to a partition with each record alone in its own cluster.

Theorem 6.4.17. $PairPrecision(R, S) = 1 - \frac{GMD(R,S)}{GMD(R,\perp)}$ *if* $f_m(x, y) = 0$ *and* $f_s(x, y) = xy$. $PairRecall(R, S) = 1 - \frac{GMD(R,S)}{GMD(\perp,S)}$ *if* $f_m(x, y) = xy$ *and* $f_s(x, y) = 0$.

Proof. We will prove this theorem for $f_m(x, y) = 0$ and $f_s(x, y) = xy$. The other case is symmetric.

A split operation $\pi \to \pi_1, \pi_2$ has a cost of $|\pi_1| \times |\pi_2|$. When we apply this operation to a partition, the result will be missing all pairs consisting of a record in π_1 and a record in π_2. There are $|\pi_1| \times |\pi_2|$ such pairs, so it turns out the cost of the split is the same as the reduction in the number of pairs.

Since f_m and f_s are operation order independent functions, Theorem 6.4.14 tells us that any bare necessities path from R to S has minimum cost. The splits in a bare necessities path will remove all pairs in Pairs(R) – Pairs(S) and no other pairs. The merges all have zero cost, so $GMD_{f_m,f_s}(R, S) = |Pairs(R) - Pairs(S)|$.

Now we follow through with the derivation:

$$\begin{aligned} 1 - \frac{GMD_{f_m,f_s}(R, S)}{GMD_{f_m,f_s}(R, \perp)} &= 1 - \frac{|Pairs(R) - Pairs(S)|}{|Pairs(R)|} \\ &= 1 - \frac{|Pairs(R)| - |Pairs(R \cap S)|}{|Pairs(R)|} \\ &= \frac{|Pairs(R \cap S)|}{|Pairs(R)|} \end{aligned}$$

$$= \text{PairPrecision}(R, S) \qquad \square$$

The various relationships are possible because of the configurability of GMD. Since the f_m and f_s functions used in this section are all operation order independent, we can use the linear time Slice algorithm described in the next section to compute all the measures above. This is exciting, especially because the straightforward implementation of pF_1 and VI would be quadratic in the worst case.

6.5 Computing Measures

Computing measures efficiently is important because the number of entities to resolve can be huge. Recall the discussion of correctness and comprehensiveness of Section 1.4. When measuring correctness, the gold-standard is human-generated, and therefore will rarely exceed thousands of records. The gold standard for measuring comprehensiveness, on the other hand, is automatically generated and could contain a much larger number of records. While large exhaustive ER results may be very expensive to generate, they need only be generated once, whereas the computation of the distance measure will be performed multiple times for a diverse set of blocking algorithms and parameters. The distance computation can therefore take a great deal of time, and a more efficient algorithm provides practitioners more time to tune their algorithms (e.g., experiment with different matching thresholds) over a wide range of options.

Many measures take (or appear to take) quadratic time for computation, which could be prohibitive. For example, a straightforward implementation of the pF_1 measure requires a quadratic number of base record pairs to be compared against the actual matching pairs. Similarly, the K measure sums the similarities of all pairs of clusters need to be computed, requiring quadratic time computation. The ccF_1 measure finds the the closest clusters for all clusters and requires a quadratic number of cluster comparisons because finding

each closest cluster requires a linear scan of the other ER result in the worst case.

Surprisingly, the topic of efficiency of measure computation is not discussed in any ER paper. Fortunately in this chapter, we propose an efficient algorithm that computes GMD in linear time for a large class of configurations. We also show that the pF_1 and VI measures can be computed in linear time using our algorithm. It is an open question if there are linear algorithms for the ccF_1 and K measures.

The algorithm we propose is called the Slice algorithm, which computes GMD when f_m and f_s are operation order independent functions. A complete description of the Slice algorithm is given in Section 6.6.2. The gist of the algorithm is to independently compute the cost of generating each cluster in S by splitting off the necessary components from clusters in R and then merging them together. For example, suppose $R = \{\langle a, c, e\rangle, \langle b, d, f\rangle\}$ and $S = \{\langle a, b, c\rangle, \langle d, e, f\rangle\}$. Cluster $\langle a, b, c\rangle$ must be generated by merging two fragments (one from each cluster in R): $\langle a, c\rangle$ and $\langle b\rangle$. The cost to split these fragments from their clusters in R is $f_s(2, 1) + f_s(1, 2)$, and the cost to merge them is $f_m(1, 2)$. When computing the cost to generate $\langle d, e, f\rangle$, we remember how many records have already been split from the clusters in R to properly compute the cost of splitting R further. In the example, no splits are necessary to get the fragments $\langle e\rangle$ and $\langle d, f\rangle$ because $\langle a, c\rangle$ and $\langle b\rangle$ have already been split off. So the cost to generate $\langle d, e, f\rangle$ is just $f_m(1, 2)$. We can then add the costs of generating these two clusters to obtain the total GMD.

We can compute the fragments needed to generate a cluster $\pi_S \in S$ by considering each record in π_S and looking up its location in R. We can then group the records by their R location to obtain the fragments. This step takes time $O(|\pi_S|)$ and thus the entire algorithm runs in time $O(N)$.

6.6 Computing Merge Distance

In this section we provide the details of algorithms for computing merge distance.

6.6.1 General Algorithm

A simple method for computing merge distance is the direct application of Dijkstra's algorithm. If we treat partitions as nodes in a graph and merge and split operations as the edges between nodes, then Dijkstra's algorithm will find a minimum cost path from R to S.

For this application, however, Dijkstra's algorithm would be highly inefficient. There are a few modifications to the algorithm that can help improve the performance. For one, Theorem 6.4.6 allows us to prune all paths that have a split after a merge has been performed. Further, at any state, we may consider if it is possible to reach the destination S through merges alone. If not, then more splits are required, so we need not consider any merges at this point. (See Lemma 6.4.9 for an explanation of this idea.)

Finally, we may make use of the monotonicity property of the merge and split cost functions to construct a lower-bound on the cost from any state to the destination S. Given this lower bound, we can apply the $A*$ algorithm instead of Dijkstra's algorithm, which may help narrow the search.

Unfortunately, we do not expect any of these optimizations to improve the exponential worst-case time for computing merge distance. We instead focus on a specific class of merge and split functions for which we have found an efficient algorithm.

6.6.2 Slice Algorithm

In this section, we describe a linear time algorithm called the Slice algorithm for computing generalized merge distance. The algorithm computes the cost

of an arbitrarily selected bare necessities path, and therefore produces the correct answer only when the split and merge cost functions are order operation independent.

The algorithm takes two partitions R and S, as well as the functions f_m and f_s as input. The output of the algorithm will be the f_m, f_s merge distance from R to S (as long as f_m and f_s are operation order independent). The gist of this algorithm is to find the cost to build each cluster $S_i \in S$ by breaking off pieces from clusters in R and then merging them together. We can find the cost to build each S_i independently and then compute the sum for the total cost to move from R to S.

We now explain the details of the algorithm, and execute it over an example input. For the purposes of the example, let $R = \{\langle a, c, e \rangle, \langle b, d, f \rangle\}$ and $S = \{\langle a, b, c \rangle, \langle d, e, f \rangle\}$. We'll refer to the individual clusters with one-based indexes: R_1, R_2 and S_1, S_2.

The algorithm begins with a loop over all clusters in R. Lines 4-8 set up the loop, which builds up a mapping M from each record to the cluster in R it is a member of. For brevity, we will not show the entire contents of M, but as examples, M[a] = 1 and M[b] = 2. The loop also computes an array Rsizes that stores the size of each cluster in R. The Rsizes array will be updated over the course of the algorithm as we split pieces off of each cluster in R. In our example, the Rsizes array will have the value 3 for both entries.

The algorithm then continues compute the cost of building each cluster $S_i \in S$. The first step is determining which clusters in R contain the records in S_i. Lines 14-21 build a structure pMap that, for each cluster R_j in R, keeps a count of the records in S_i that are in R_j. If $R_j \cap S_i = \varnothing$, then there will be no entry in pMap for S_j. Therefore there is at most one entry in pMap for each record in S_i. In our example, the pMap generated for S_1 will have pMap[1] = 2 and pMap[2] = 1, since the two records a, c are in R_1, whereas the one remaining record b comes from R_2. When pMap is generated for S_2, it will have pMap[1] = 1 and pMap[2] = 2.

To build S_i from the clusters in R, we must first split off the parts of the

```
Input:    R, S: the result and gold standard.
          (R_i and S_i refer to the ith clusters of R and S respectively.)
          f_m, f_s: the cost functions for merge and split operations
Output:   the f_m, f_s merge distance from R to S
 1: MergeDistance(R, S)
 2: // build a map M from record to cluster number
 3: // and store sizes of each cluster in R
 4: for all R_i ∈ R do
 5:   for all r ∈ R_i do
 6:     M[r] ← i
 7:   end for
 8:   Rsizes[i] ← |R_i|
 9: end for
10: // begin computing cost
11: cost ← 0
12: for all S_i ∈ S do
13:   // determine which clusters in R contain the records in S_i
14:   pMap ← {}
15:   for all r ∈ S_i do
16:     // if we haven't seen this R cluster before, add it to the map
17:     if M[r] ∉ keys(pMap) then
18:       pMap[M[r]] ← 0
19:     end if
20:     // increment the count for this partition
21:     pMap[M[r]] ← pMap[M[r]] + 1
22:   end for
```

Algorithm 6.1: Slice algorithm, part 1

clusters in R that have the records in S_i. We can perform this splitting with a single split operation for each cluster that intersects S_i. Once those k pieces are split off, then we can merge them all together with $k-1$ merge operations. Lines 24-37 compute the cost for this series of operations, consulting pMap to find out how many records must be split off of each cluster in R. In our example, the cost to construct S_1 would be computed as follows. First, pMap[1] is 2, so we would have to split two records off of R_1. Since R_1 currently has size 3 (according to Rsizes), the cost for this split would be $f_s(2, 3-2)$. In the next iteration, we would consider pMap[2] = 1, and split 1 record away from

```
23:    // compute cost to generate S_i
24:    SiCost ← 0
25:    totalRecs ← 0
26:    for all (i, count) ∈ pMap do
27:      // add the cost to split R_i
28:      if Rsizes[i] > count then
29:        SiCost ← SiCost + f_s(count, Rsizes[i] − count)
30:      end if
31:      Rsizes[i] ← Rsizes[i] − count
32:      if totalRecs ≠ 0 then
33:        // cost to merge into S_i
34:        SiCost ← SiCost + f_m(count, totalRecs)
35:      end if
36:      totalRecs ← totalRecs + count
37:    end for
38:    cost ← cost + SiCost
39:  end for
40:  return cost
```

Algorithm 6.2: Slice algorithm, part 2

R_2. This split would cost $f_s(1, 3-1)$. Now that there are two "fragments", we compute the cost to merge them together: $f_m(2, 1)$. This would end the loop and the cost for constructing S_1 would be $2 * f_s(2, 1) + f_m(2, 1)$.

We note that on line 31, we update the Rsizes array to reflect the fact that records have been split off of the clusters in R. After computing the cost to construct S_1, Rsizes will have been updated to the sizes of the clusters in R without records in S_1. Specifically, Rsizes[1] = 3−2 = 1 and Rsizes[2] = 3−1 = 2.

The final details of the algorithm are to simply sum up the costs to construct all the S_i clusters, which is the merge distance from R to S.

6.7 Experiments

As discussed in Section 6.5, ER datasets can be huge, and the computation times for measures can be very significant. In this section we compare the computation times for the BMD, pF_1,cF_1,K,ccF_1, and VI measures. (We omit

the other configured GMD measures because their runtimes are similar to that of BMD.) For BMD, we used the Slice algorithm. For pF_1, we used two implementations: one uses the Slice algorithm while the other is a straightforward implementation that iterates through all base record pairs of the ER result and the gold standard. Similarly for VI, we used an implementation using Slice (i.e., GMD_V) and a straightforward implementation that iterates through all pairs of clusters between the ER result and the gold standard. We implemented cF_1, ccF_1, and K in a straightforward way (as described in Section 6.5) because there are no better published algorithms. As a result, cF_1 was implemented with a linear time algorithm while ccF_1 and K were implemented with quadratic time algorithms. All the algorithms were implemented in Java, and our experiments were run in memory on a 2.4GHz Intel(R) Core 2 processor with 4GB of RAM.

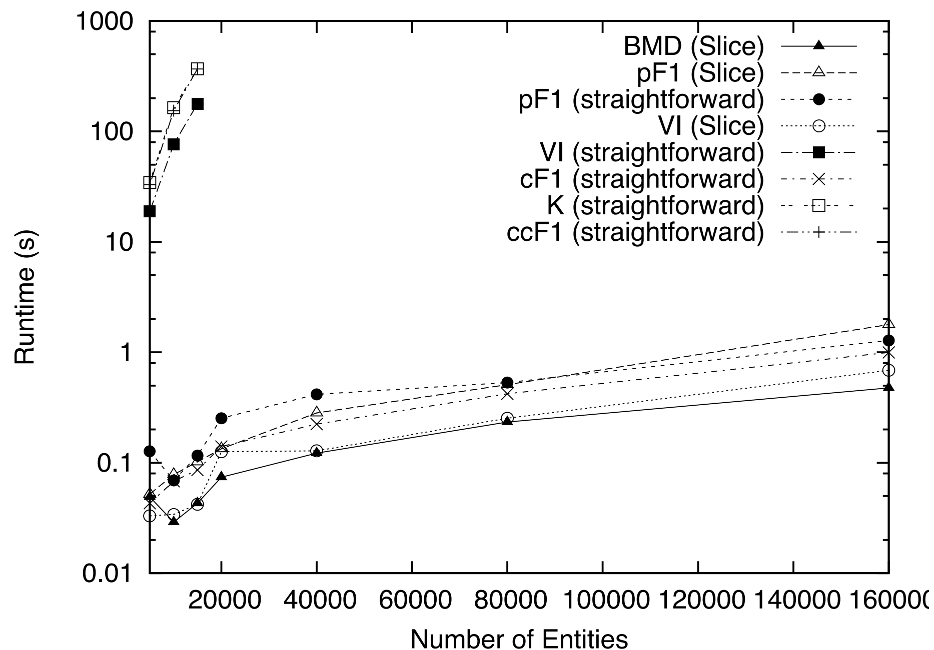

Figure 6.4: Scalability

Figure 6.4 shows the runtime plots for the measures. We experimented on 10K to 160K entities with the Zipfian exponent e = 1.5, and each ER result

R had $\frac{|R|}{10}$ misplaced entities. Any implementation using the Slice algorithm is scalable to large ER results, with a runtime increasing linearly by the number of entities. Although the straightforward implementation of pF_1 is worst-case quadratic, in this experiment it shows linear average behavior (because clusters are small — average size is 3.5 — making the number of base records to iterate over small). The straightforward implementation of VI is expensive even for a small number of entities, highlighting the runtime improvements when using Slice. The cF_1 algorithm is efficient and shows a linear increase in runtime. Finally, the runtimes of the K and ccF_1 algorithms grow quadratically against the number of entities and show the worst runtimes.

6.8 Conclusion

We have proposed an edit distance measure for ER (called "generalized merge distance" or GMD) that computes the shortest edit distance from an ER result to a gold standard using merges and splits as the basic operations on clusters. A powerful feature is that the merge and split costs can be configured based on record sizes. We proposed an efficient algorithm (called Slice), which computes GMD in linear time for a large class of merge and split cost functions. Interestingly, the state-of-the-art VI clustering measure is a special case of GMD, and the dominantly used pF_1 measure for ER can be directly computed using GMD. As a result, both VI and pF1 can be computed efficiently using our Slice algorithm.

We also conclude that GMD can be configured on two important parameters: sensitivity to error type and sensitivity to cluster size. As a result, one could more precisely define a measure that correctly evaluates a given application. Finally, we have demonstrated that the Slice algorithm is scalable and can be used to evaluate very large datasets. Thus, we believe that the GMD measure fills a hole in the space of available ER measures, and that it clarifies the relationship between the available ER measures.

There are interesting open issues for the GMD measure. We have already shown that the pF_1 and VI measures are closely related to GMD. We believe that edit distance measures for ER and clustering have yet to be fully explored and suspect that GMD could be a fundamental way of generating ER and in general clustering measures.

Chapter 7

Conclusion

7.1 Summary

Entity resolution is an expensive problem and we must therefore resort to specialized techniques in order to reduce the processing time. Distributing the process as in Chapter 2 is an excellent way to increase performance, as it may be done without suffering a loss in quality of the ER result. We described how to improve performance further by using the multiple blocking techniques of Chapter 3, optionally in conjunction with the LSH function families from Chapter 4. Confidences add complexity to entity resolution processing, so Chapter 5 described how to efficiently handle confidences. Chapter 6 presented various methods of evaluating the quality of the results of an ER algorithm.

Having reviewed the overall contributions of this thesis, we now proceed to summarize the individual contributions of each chapter.

In Chapter 2 we studied how to efficiently distribute the ER workload across multiple processors. We presented D-Swoosh, a distributed ER framework that allowed us to plug in and test many distribution strategies. We found that the full replication scheme had the best runtimes for lower numbers of processors, while the grid scheme scaled the best over all. We also demonstrated that schemes that blindly compare all records often outperform

simple techniques that take advantage of domain knowledge.

Chapter 3 took the other path towards improving ER performance: blocking. Specifically, the chapter presented the Duplo algorithm, a highly scalable algorithm for performing multiple blocking on datasets too large to fit in memory at once. We discovered that the performance of the algorithm is highly dependent on the order in which blocks are processed. We therefore compared four different ordering strategies and found that the best strategy was to select the next block to process according to how many records in the block were already found to be merged.

In Chapter 4 we presented multiple extensions to minhash for use with different data types. We defined "map minhash" for map data types and proved that it satisfies the LSH property for a similarity measure based on Jaccard similarity suitable for use on maps. We further defined an LSH hash family applicable to composed data types. Finally, we introduced a hash family that satisfies the LSH property for weighted Jaccard similarity on sets with weighted values. We compared this final hash family against Manasse's scheme, the only other known LSH hash family for weighted Jaccard similarity. Although the Manasse scheme supports more general weighted sets, our experiments demonstrated that for the more specific case of sets with weighted values, our scheme allows one to generate hashes 5 times faster. All of these hashing schemes can work in conjunction with the techniques of Chapter 3 and thus add power to those techniques.

Chapter 5 considered the application of the pairwise entity resolution model to the special case of data with confidences. We defined the result of entity resolution with confidences and presented a naive, brute force algorithm to compute the result. We then described Koosh, an optimal algorithm for computing ER with confidences in the absence of extra known properties of the match and merge functions. Despite Koosh's optimality, we found that ER with confidences is extremely expensive, as the result can be very large. We therefore introduced the concepts of thresholds and domination to deliver a useful subset of the full ER result. Finally, we described the Packages algorithm,

which—given the existence of a more relaxed match function with necessary properties—computes the ER result with a much lower number of comparisons. Our conclusions are that thresholds and domination are both crucial tools in reducing the workload of ER with confidences. Koosh is an effective algorithm, but the Packages algorithm should be used in cases when it applies. Moreover, the result of the first phase of the Packages algorithm may be a useful result that is cheaper to compute than the full ER result.

Finally, in Chapter 6 we analyzed the process of evaluating the results of entity resolution algorithms by comparing them to a gold standard result. We surveyed the many existing measures and proposed Generalized Merge Distance, a new, flexible measure that can be configured to suit many applications. Providing more evidence for its flexibility, we proved that three existing measures (pairwise precision/recall, merge distance, and VI distance) are special cases of Generalized Merge Distance. Finally, we identified a family of cost functions that allow Generalized Merge Distance to be computed in linear time with our Slice algorithm. The Slice algorithm can be used to efficiently compute Generalized Merge Distance in many of its configurations, including those that lead to the pairwise measures and the VI distance. This algorithm is a great improvement over the naive approach to computing these measures.

7.2 Future Work

While this thesis has made several contributions in the field of entity resolution, the problem is in no way "solved". This work in fact opens up many avenues for future work.

First, Chapter 2 did find that the Grid scheme was nearly optimal in terms of network communication, and was overall the most scalable scheme of the ones we tried. However, the Grid scheme requires a triangular number of processors to function properly. There is likely a scheme that retains the near-optimality of the Grid scheme while scaling smoothly with any number of

processors.

Chapter 2 also showed that "blind" schemes often outperform naive methods of using domain knowledge. One could combine a blind scheme with domain knowledge by adding extra predicates to the resp function. This scheme would reduce the workload by invoking fewer comparison functions, however the network communication cost would be the same as a blind scheme. Further research may identify a hybrid scheme that takes advantage of domain knowledge to reduce network communication, yet also has the smooth scalability of the blind schemes.

We have mentioned that entity resolution can be sped up using either distribution as in Chapter 2 or blocking as in Chapter 3. However, we have not given a detailed look at how to do both at the same time. Future work in this area may prove to be fruitful.

Finally, most of the work in this thesis has been done with the idea that all records refer to the same type of entity. However, many data sets may have records that refer to different types of entities. For example, one may have records that refer to papers along with other records that refer to authors. Once we have multiple types in our input data, it becomes imperative to use any relationship information between the records to either improve the accuracy or the performance of the entity resolution process. Thus, multiple domain entity resolution provides a vast area of exploration for future work.

Bibliography

[1] Latif Al-Hakim. *Information Quality Management: Theory and Applications*. Idea Group Inc, 2007.

[2] Reema Al-Kamha and David W. Embley. Grouping search-engine returned citations for person-name queries. In *WIDM '04: Proceedings of the 6th annual ACM international workshop on Web information and data management*, pages 96–103, New York, NY, USA, 2004. ACM.

[3] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *Proc. of VLDB*, pages 586–597, 2002.

[4] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[5] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 918–929. VLDB Endowment, 2006.

[6] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. In *FOCS '02: Proceedings of the 43rd Symposium on Foundations of Computer Science*, page 238, Washington, DC, USA, 2002. IEEE Computer Society.

[7] D. Barbará, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, 1992.

[8] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 651–660, New York, NY, USA, 2005. ACM.

[9] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *Proc. of ACM SIGKDD'03 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[10] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Identification*, 2003.

[11] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 2009.

[12] Omar Benjelloun, Hector Garcia-Molina, Heng Gong, Hideki Kawai, Tait E. Larson, David Menestrina, and Sutthipong Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 37, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Deepavali Bhagwat, Kave Eshghi, and Pankaj Mehra. Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 105–112, New York, NY, USA, 2007. ACM.

[14] I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, 2004.

[15] Indrajit Bhattacharya and Lise Getoor. Collective entity resolution in relational data. *ACM Trans. Knowl. Discov. Data*, 1(1):5, 2007.

[16] Indrajit Bhattacharya, Lise Getoor, and Louis Licamele. Query-time entity resolution. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 529–534, New York, NY, USA, 2006. ACM.

[17] M. Bilenko, S. Basu, and M. Sahami. Adaptive Product Normalization: Using Online Learning for Record Linkage in Comparison Shopping . In *Proc. of IEEE Int. Conf. on Data Mining*, Houston, Texas, 2005.

[18] M. Bilenko, B. Kamath, and R. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM*, 2006.

[19] M. Bilenko and R. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD*, 2003.

[20] M. Bilenko, R. J. Mooney, W. W. Cohen, P. Ravikumar, and S. E. Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.

[21] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *J. Comput. Syst. Sci.*, 60(3):630–659, 2000.

[22] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[23] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02: Proceedings of the thiry-fourth annual ACM*

symposium on Theory of computing, pages 380–388, New York, NY, USA, 2002. ACM.

[24] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. of ACM SIGMOD*, pages 313–324, 2003.

[25] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. of ICDE*, Tokyo, Japan, 2005.

[26] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 5, Washington, DC, USA, 2006. IEEE Computer Society.

[27] William Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems*, 18:288–321, 2000.

[28] William W. Cohen and Jacob Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 475–480, New York, NY, USA, 2002. ACM.

[29] Ricardo G. Cota, Marcos André Gonçalves, and Alberto H. F. Laender. A heuristic-based hierarchical clustering method for author name disambiguation in digital libraries. In Altigran Soares da Silva, editor, *SBBD*, pages 20–34. SBC, 2007.

[30] Aron Culotta and Andrew McCallum. Joint deduplication of multiple record types in relational data. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 257–258, New York, NY, USA, 2005. ACM.

[31] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875, 2004.

[32] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[33] X. Dong, A. Y. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *Proc. of ACM SIGMOD*, 2005.

[34] Xin Dong, Alon Halevy, and Jayant Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 85–96, New York, NY, USA, 2005. ACM.

[35] M. Elfeky, V. Verykios, and A. Elmagarmid. TAILOR: A record linkage toolbox. In *ICDE*, 2002.

[36] 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.

[37] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[38] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[39] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[40] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson Prentice Hall, 2nd edition, 2009.

[41] L. Gu, R. Baxter, D. Vickers, and C. Rainsford. Record linkage: Current practice and future directions. Technical Report 03/83, CSIRO Mathematical and Information Sciences, 2003.

[42] Lifang Gu and Rohan A. Baxter. Adaptive filtering for efficient record linkage. In Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn, editors, *SDM*. SIAM, 2004.

[43] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: an efficient clustering algorithm for large databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 73–84, New York, NY, USA, 1998. ACM.

[44] 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.

[45] M. Hernandez and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[46] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proc. of ACM SIGMOD*, pages 127–138, 1995.

[47] Thomas N. Herzog, Fritz J. Scheuren, and William E. Winkler. *Data Quality and Record Linkage Techniques*. Springer, July 2007.

[48] M. Hosszú. On the functional equation f(x+y,z) + f(x,y) = f(x,y+z) + f(y,z). *Periodica Mathematica Hungarica*, 1(3):213–216, 09 1971.

[49] Jian Huang, Seyda Ertekin, and C. Lee Giles. Efficient name disambiguation for large-scale databases. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *PKDD*, volume 4213 of *Lecture Notes in Computer Science*, pages 536–544. Springer, 2006.

[50] T. Ibaraki, H. Nagamochi, and T. Kameda. Optimal coteries for rings and related networks. *Distrib. Comput.*, 8(4):191–201, 1995.

[51] Piotr Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001.

[52] Piotr Indyk. A small approximately min-wise independent family of hash functions. *J. Algorithms*, 38(1):84–90, 2001.

[53] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[54] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets. In *Proc. of Intl. Conf. on Database Systems for Advanced Applications*, pages 137–, 2003.

[55] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[56] Alberto H.F. Laender, Marcos André Gonçalves, Ricardo G. Cota, Anderson A. Ferreira, Rodrygo L.T. Santos, and Allan J.C. Silva. Keeping a digital library clean: new solutions to old problems. In *DocEng '08: Proceeding of the eighth ACM symposium on Document engineering*, pages 257–262, New York, NY, USA, 2008. ACM.

[57] Suk Kyoon Lee. An extended relational database model for uncertain and imprecise information. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, pages 211–220. Morgan Kaufmann, 1992.

[58] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[59] Mark Manasse, Frank McSherry, and Kunal Talwar. Consistent weighted sampling. (Unpublished technical report) **http://research.microsoft.com/en-us/people/manasse/**.

[60] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. *Introduction to Information Retrieval.* Cambridge University Press, Cambridge, England, 2008.

[61] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, July 2008.

[62] A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of KDD*, pages 169–178, Boston, MA, 2000.

[63] Marina Meila. Comparing clusterings by the variation of information. In Bernhard Schölkopf and Manfred K. Warmuth, editors, *COLT*, volume 2777 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2003.

[64] Marina Meilă. Comparing clusterings: an axiomatic view. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 577–584, New York, NY, USA, 2005. ACM.

[65] David Menestrina, Omar Benjelloun, and Hector Garcia-Molina. Generic entity resolution with data confidences. In *CleanDB*, 2006.

[66] M. Michelson and C. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, 2006.

[67] A. E. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 23–29, 1997.

[68] H. B. Newcombe. *Handbook of record linkage: methods for health and statistical studies, administration, and business.* Oxford University Press, Inc., New York, NY, USA, 1988.

[69] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.

[70] 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.

[71] C. H. Papadimitriou and M. Sideri. Optimal coteries. In *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 75–80, New York, NY, USA, 1991. ACM Press.

[72] Hanna Pasula, Bhaskara Marthi, Brian Milch, Stuart J. Russell, and Ilya Shpitser. Identity uncertainty and citation matching. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *NIPS*, pages 1401–1408. MIT Press, 2002.

[73] Daniel Ramage, Paul Heymann, Christopher D. Manning, and Hector Garcia-Molina. Clustering the tagged web. In *WSDM '09: Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 54–63, New York, NY, USA, 2009. ACM.

[74] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of ACM SIGKDD*, Edmonton, Alberta, 2002.

[75] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *KDD*, 2002.

[76] Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

[77] P. Singla and P. Domingos. Object identification with attribute-mediated dependences. In *Proc. of PKDD*, pages 297 – 308, 2005.

[78] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[79] Yang Song, Jian Huang, Isaac G. Councill, Jia Li, and C. Lee Giles. Efficient topic-based unsupervised name disambiguation. In *JCDL '07: Proceedings*

of the 7th ACM/IEEE-CS joint conference on Digital libraries*, pages 342–351, New York, NY, USA, 2007. ACM.

[80] Terry Tang. Set 'em up, baker—frosting shots, neat, all around. *The Arizona Republic*, July 30 2008. Available at: **http://www.azcentral.com/arizonarepublic/food/articles/2008/07/30/20080730frosting0730.html**.

[81] S. Tejada, C. A. Knoblock, and S. Minton. Learning domain-independent string transformation weights for high accuracy object identification. In *KDD*, 2002.

[82] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[83] Esko Ukkonen. On approximate string matching. In Marek Karpinski, editor, *FCT*, volume 158 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 1983.

[84] V. S. Verykios, G. V. Moustakides, and M. G. Elfeky. A bayesian decision model for cost optimal record matching. *The VLDB Journal*, 12(1):28–40, 2003.

[85] Robert A. Wagner. On the complexity of the extended string-to-string correction problem. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 218–223, New York, NY, USA, 1975. ACM.

[86] Eric W. Weisstein. "Beta Distribution". From MathWorld–A Wolfram Web Resource. **http://mathworld.wolfram.com/BetaDistribution.html**.

[87] Steven Euijong Whang, David Menestrina, Georgia Koutrika, Martin Theobald, and Hector Garcia-Molina. Entity resolution with iterative

blocking. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 219–232, New York, NY, USA, 2009. ACM.

[88] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[89] Wikipedia. Metric (mathematics) — Wikipedia, the free encyclopedia, 2009. [Online; accessed 11-June-2009].

[90] W. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 1999.

[91] W. Winkler. Overview of record linkage and current research directions. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 2006.

[92] W. E. Winkler. Using the EM algorithm for weight computation in the fellegi-sunter model of record linkage. *American Statistical Association, Proceedings of the Section on Survey Research Methods*, pages 667–671, 1988.

[93] W. E. Winkler. Approximate string comparator search strategies for very large administrative lists. Technical report, US Bureau of the Census, 2005.

[94] W.E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research Methods*, pages 354–359. American Statistical Association, 1990.

[95] William E. Winkler and Yves Thibaudeau. An application of the fellegi-sunter model of record linkage to the 1990 U.S. decennial census. In *U.S. Decennial Census. Technical report, US Bureau of the Census*, 1987.

[96] W.E. Yancey. Bigmatch: A program for extracting probable matches from a large file for record linkage. Technical report, US Bureau of the Census, 2002. document available at http://www.census.gov/srd/www/byyear.html.