SCALABLE HUMAN-CENTRIC ENTITY MATCHING

by

Sanjib Kumar Das

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 12/14/2016

The dissertation is approved by the following members of the Final Oral Committee:

AnHai Doan, Professor, Computer Sciences

Jeffrey F. Naughton, Professor, Computer Sciences

Paraschos Koutris, Assistant Professor, Computer Sciences

Jude W. Shavlik, Professor, Computer Sciences and Biostatistics & Medical Informatics

C. David Page Jr., Professor, Biostatistics & Medical Informatics and Computer Sciences

To Baba, Maa and Tukai

ACKNOWLEDGMENTS

As I reflect back on my wonderful journey as a Ph.D. student, I have many people to thank. First and foremost, I would like to thank my advisor, Prof. AnHai Doan, for his continued support, guidance, and patience. He has been an advisor in *all* senses of the word, advising me on my research, communication, career choices, dating, marriage, managing finances, managing time and life at home, and so on. I have learned from him the importance of telling a good story. He has always urged me to think hard until "the head literally hurts" and to put "an extra 1%" of work each day. I have matured a lot from his honest feedback. His perseverance and work ethics will continue to inspire me for years. AnHai, I cannot thank you enough for being an inspiring "boss".

Next, I would like to thank my thesis committee members: Professors Paris Koutris, Jeff Naughton, David Page, and Jude Shavlik for their invaluable inputs. I would also like to thank my mentors and collaborators at WalmartLabs: Omkar Deshpande, Ganesh Krishnan, Digvijay Lamba, and Shishir Prasad for helping me evaluate my research "in the wild".

I would like to thank my direct collaborators at UW-Madison: Chaitanya Gokhale, Fatemah Panahi, and Paul Suganthan. Thank you Chaitanya for starting the *Corleone* project which gave an initial direction to my dissertation. I would always remember you as the "Don Vito" of our Corleone family. Thank you Fatemah for contributing to *Rmony*. Thank you Paul for contributing to *Falcon*, particularly for working on it single-handedly when I was travelling for my job interviews. Also, a big thanks to Adel, Bruhathi, Han, Haojun, Harshad, Jeff B., Pradap, Yash, and to the entire database group for their critical feedback on my research and for their friendly support.

Next, I would like to thank my friends for the night-long poker/mafia sessions and the adventurous road trips: Akshar, Borna, Deepti, Ishani, Jai, Nilay, Nisha, Sandeep, Sathya, Saurabh, Subhadip, and Zainab. Thanks to Anurag, Bill, Sanish, Tapas, and Vijay for being extremely "tolerant" roommates. A special thanks to Aren, Gina, Teja, and all the members of the UWTT club. All you folks helped me maintain my sanity when the going sometimes got insane. Thanks for being my "family" away from home.

I thank my family back at home: my father (Baba) and mother (Maa) for giving me an excellent education (and making a lot of sacrifices all along), my elder brother (Dada) and sister (Didi) for pampering me always, my in-laws (Maa, Pratik, Shreya, Piu, Masi and Meso) for encouraging me throughout. Thank you for patiently waiting for that one phone call every Sunday and never complaining about my lack of time for you.

Finally, I would like to thank my "boss" at home, my wife Priyanka (Tukai). The bold decision of marrying her in the middle of the Ph.D. program paid off as she brought order to my otherwise "haphazard" life and helped me accelerate my Ph.D. Thank you for your love, care, and support. You are inarguably the best thing that happened to me during this eventful journey, period.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Entity matching (EM) finds data records referring to the same real-world entity. This dissertation studies the problem of scalable human-centric EM. By "human-centric" we mean EM problems that involve a crowd of workers (in crowdsourcing) or analysts. By "scalable" we mean problems that involve a large number of EM tasks or a large number of tuple pairs to match.

Today solving an EM task often involves a lot of developer effort. As such it is difficult for an organization that wants to solve a large number of EM tasks, because there are simply not enough developers. Recent approaches apply crowdsourcing to EM in order to reduce developer effort. While promising, these approaches are limited in that they still require a developer to be in the loop. To address this, we propose Corleone, a hands-off crowdsourced solution to EM. Corleone crowdsources the *entire* EM workflow, requiring no developers. As a result, Corleone can scale up EM at enterprises and crowdsourcing startups, and open up crowdsourcing for the masses.

Many EM settings require non-technical analysts to write EM rules. To address this problem, we build upon techniques developed in Corleone to develop Rmony, a system that supports analysts in writing EM rules. We evaluate Rmony with real-world users and data to show that it is effective. We report on lessons learned that eventually lead to the inception of an open-source EM management system in the Python ecosystem.

For important emerging EM settings, such as EM as a service on the cloud, the hands-off crowdsourced approach of Corleone is ideally suited. Corleone however is severely limited in that it does not scale to large tables. To address this limitation, in the last direction of this dissertation, we propose Falcon, a solution that scales up Corleone, using RDBMS-style query execution

and optimization over a Hadoop cluster. Extensive experiments show that Falcon can scale up to tables of millions of tuples, thus providing a practical solution for hands-off crowdsourced EM.

Chapter 1

Introduction

Entity matching (EM) is the problem of finding data records that refer to the same real-world entity, such as (John Doe, Univ. of Wisc. Madison, 608-772-9642) and (J. Doe, UW-Madison, 772 9642). EM is a critical problem in many real-world applications, including comparison shopping [2, 5], citation tracking [50], knowledge base construction [1], matching health records [51, 41], matching census data [52, 108], and many more.

As a concrete example of a real-world use case of EM, consider product matching in e-commerce companies (e.g., Amazon, eBay, Walmart). These companies have to regularly match products in their catalog with those of competitor sites so that they can offer the best price for a product. Another related example is comparison shopping on Web sites such as Google Shopping [2], Nextag [5], PriceGrabber [7], and SlickDeals [9], where customers can compare prices of a product across multiple online retailers. The underlying challenge in building these sites is identifying product descriptions from multiple retailers that refer to the same real-world product, which is an EM problem.

As yet another real-world use case of EM, consider matching patient records in health care industry. Patient data is often spread across hospitals, clinics, insurance companies, and pharmacies. Data from all these sources must be matched in order to provide a single comprehensive view of patient data to doctors and health care researchers. This can significantly improve patient diagnosis and reduce the overall cost of health care. Matched patient data also allows researchers to identify key disease patterns, e.g., matching patient addresses to location data helps identify hot-spots for diseases and correlations between diseases [51, 41].

In the current era, where large volumes of data is being integrated from multiple sources, the problem of EM is becoming even more critical. For example, consider the recent introduction of marketplaces by e-commerce companies such as Amazon and Walmart. In a marketplace, the e-commerce company now sells items from smaller vendors (e.g., Tech for Less Inc., Unbeatable-Sale, Circuit City) alongside products from its own catalog on the shopping Web site. Different vendors may sell the same real-world product in the marketplace, and these matching products need to be identified to avoid duplication.

This problem arises in many other domains as well. Today, data is being collected and analyzed by users in virtually every occupation and field. Examples of these users include domain scientists, small business workers, end users, and other "data enthusiasts" [58]. Many such users need to perform EM. For example, a journalist may need to match two long lists of political donors for an election campaign to find donors that contributed to two different campaigns.

These real world use cases show that EM is an important problem spanning different domains (e.g., e-commerce, health care, census data) and different practitioners (e.g., enterprises, small businesses, end users). As a result, this problem has received significant attention (see [27, 28, 37, 44] for surveys, overviews and books on EM). Despite this attention, EM is far from being solved. Many research challenges remain and new directions have emerged to be explored.

In this dissertation we focus on the EM problem, in particular on the scalable human-centric EM direction. By "human-centric" we mean EM problems that involve a crowd of workers (in crowdsourcing) or an analyst (as we motivate below). By "scalable" we mean EM settings that involve a very large number of EM tasks or very large number of tuple pairs to match. Specifically, we consider the following three concrete research directions:

- **Hands-Off Crowdsourcing for Entity Matching:** Today solving an EM task often involves a developer (who knows how to program or use machine learning techniques or both). However, there are many EM settings that involve a very large number of EM tasks to be solved (e.g., in the hundreds). Such settings would require a large number of developers, which is

often not feasible in practice. We address this problem by completely crowdsourcing the entire EM task, thereby not requiring a developer. This is joint work with Chaitanya Gokhale, and Section 1.1 details my contributions.

- **Developing a Rule-Based Entity Matching Management System for Analysts:** Many EM settings ask non-technical analysts to write EM rules. Currently there is virtually no support for such rule writing. To address this problem, we build on techniques developed in the above direction ("hands-off crowdsourcing for EM") to develop a system that supports analysts in writing EM rules. We evaluate this system and report observations and lessons learned.

- **Scaling Up Crowdsourced Entity Matching to Large Tables:** Many EM settings involve matching very large tables (e.g., each consisting of millions of tuples). In the third and final direction of the dissertation, we examine how to scale up hands-off crowdsourcing to such settings using RDBMS-style query execution and optimization over a Hadoop cluster.

Next we elaborate on the above three directions and give an overview of the solutions.

1.1 Hands-Off Crowdsourcing for Entity Matching

Over the past few decades many different approaches have been proposed for EM such as hand-crafted rules [30, 50, 61], supervised learning [20, 43], clustering [79], probabilistic models [47, 91], and collective matching [19, 39, 90].

However, these solutions require significant developer effort (e.g., writing rules, selecting features, training models) to produce high matching quality. In order to decrease developer effort, crowdsourcing is being increasingly applied to entity matching [103, 104, 35, 106, 107]. In these solutions, certain parts of the entity matching problem are solved using a crowd of workers. For example, some works [103, 104, 35] use the crowd to verify the matches predicted by their EM solutions.

While these works look promising, they crowdsource only parts of the EM problem and thus still require substantial developer effort. For example, works that use the crowd only to verify

predicted matches still need the developer to perform the matching (e.g., by writing rules or training models). As a result, these solutions do not scale to the growing EM need at enterprises and crowdsourcing startups. This is because these firms need to solve tens to hundreds of EM tasks on a regular basis; with a developer in the loop this becomes very difficult. Furthermore, the current solutions cannot help ordinary users such as domain scientists, journalists, and other "data enthusiasts" [58], who can neither code like developers nor have enough budget to employ service from crowdsourcing startups.

To address these problems, we propose Corleone, a hands-off solution for crowdsourced entity matching [54]. Here, we crowdsource the entire EM problem, thus requiring no developers. As such, Corleone can scale up EM at enterprises and crowdsourcing startups, and also enable ordinary users to perform EM.

We evaluate Corleone over three real-world datasets and show that Corleone achieves comparable or better accuracy than traditional solutions at a reasonable crowdsourcing cost. For example, Corleone completes the task of matching 2.6K citations from DBLP and 64K citations from Google Scholar at a crowdsourcing cost of \$70 with an F_1 score of 92.1%, which is comparable to the 88-92% F_1 in the previously published works [71, 70, 17].

This work was carried out in collaboration with Chaitanya Gokhale. My main contributions here are the following:

- collaborating with Chaitanya Gokhale to work out the Corleone architecture and to design the experiments;
- developing a MapReduce-based solution to apply blocking rules to the input tables;
- developing a MapReduce-based solution to generate feature vectors for a given set of tuple pairs; and
- designing and performing experiments on one data set, namely Citations.

1.2 Developing a Rule-Based EM Management System for Analysts

In a recent study on Big Data [75], McKinsey Global Institute (MGI) estimated that by 2018, in the US alone up to 1.5 million new managers and data analysts will be needed to analyze the data being generated across various sectors. Data analysts are individuals who have been trained with the business needs but generally do not have programming experience or background in Computer Science. They are less expensive to hire than developers and hence can be hired in larger numbers.

EM is one such data management problem which may be performed by data analysts. For example, product matching in e-commerce companies, such as Walmart, is often performed by data analysts [88]. However, currently there are no interactive tools available for these "novice" data analysts to perform matching quickly and accurately.

Discussions with analysts at WalmartLabs suggest that for such a tool to be useful, it should have the following three features. First, the system should be end-to-end, i.e., it must be a "management" rather than just a "matching" system. Second, it must provide support for EM rules. The analysts often prefer writing rules to do matching, such as "two books match if they agree on the ISBNs and the number of pages". This is because rules are relatively easy to understand, tune and debug. Further, domain knowledge can be easily encoded in the form of rules. Finally, such a tool must have good visualizations and debugging capabilities.

It turns out that some of the techniques we develop in Corleone could potentially be used here. For example, in Corleone we extract rules from a trained Random Forest. This technique could be used in the new system to learn rules. Another example is the use of active learning in Corleone where the crowd iteratively labels examples to train a matcher. Again, this technique could be used in the system to learn rules except that instead of the crowd, the analyst could label example pairs.

Thus, in this research direction, we build on Corleone to develop a rule-based EM management system for analysts. Specifically:

- First, we develop Rmony[1], a system that helps an analyst manually write EM rules to match two input tables, using a GUI. We experimentally demonstrate that Rmony helps analysts write rules with high accuracy.

- Manually writing rules can be time consuming. Thus in the next step, we extend Rmony to automatically suggest EM rules to the analyst, using the technology underlying Corleone. Specifically, Corleone performs active learning with the crowd, where the crowd iteratively labels examples to train a learning-based matcher. EM rules are then extracted from this matcher. Rmony uses the same technology except that instead of the crowd, an analyst labels the example pairs. We experimentally demonstrate that Rmony can suggest EM rules that have high accuracy.

The research questions that we seek to answer in this direction are the following:

- Can we develop a system using which an analyst can manually write EM rules effectively? What will be the quality of such rules? Will the analyst be able to write EM rules for diverse domains?

- Can we develop a system to automatically suggest EM rules to analysts? What will be the quality of such rules? Can high-quality rules be suggested for diverse domains?

- Can such a system be effectively used in real-world settings?

- What lessons can we learn?

To explore these questions, we have developed and experimented with several system variations, both with graduate students at UW-Madison and with analysts at WalmartLabs. In the course of this, we have drawn a set of observations and lessons that we detail in the dissertation. Some of these lessons have led to the inception of Magellan, an open-source EM management system in the Python ecosystem.

[1] Rmony is a pun on "harmony" which means "agreement/match" and the 'R' in Rmony is for "rules".

1.3 Scaling Up Crowdsourced Entity Matching to Large Tables

Corleone demonstrates that hands-off crowdsourcing for EM is promising for many settings, e.g., EM as a service on the cloud. However, Corleone does not consider scaling to large tables. To address this problem, in this direction we introduce Falcon (fast large-table Corleone), a solution that scales up Corleone to tables of millions of tuples.

We begin by identifying three reasons for Corleone's being slow. First, it often performs too many crowdsourcing iterations without a noticeable accuracy improvement, resulting in large crowd time and cost. Second, many of its machine activities take too long. In particular, in the blocking step Corleone simply applies the blocking rules to *all* tuple pairs in the Cartesian product of the two input tables A and B. This is clearly unacceptable for large tables. Finally, when Corleone performs crowdsourcing, the machines sit idly, a waste of resources. If we can "mask the machine time" by scheduling as many machine activities as possible during crowdsourcing, we may be able to significantly reduce the total runtime.

We then describe how Falcon addresses the above problems. It is difficult to address all three simultaneously. So Falcon provides a relatively simple solution to cap the crowdsourcing time and cost to an acceptable level (for now), then focuses on minimizing and masking machine time. Realizing these goals raises three important challenges:

- First, we do not want to scale up a monolithic standalone EM workflow. Rather we want a solution that is modular and extensible so that we can focus on scaling up pieces of it, and can easily extend it later to more complex EM workflows. To address this, we introduce an RDBMS-style execution and optimization framework, in which an EM task is translated into a plan composed of operators, then optimized and executed. Compared to traditional RDBMSs, this framework is distinguished in that its operators can use crowdsourcing.

- The second challenge is to provide efficient implementations for the operators. We describe a set of implementations in Hadoop that significantly advances the state of the art. In particular, we focus on the blocking step as this step consumes most of the machine time. Current Hadoop-based solutions to execute blocking rules either do not scale or have considered

only simple rule formats. We develop a solution that can efficiently process complex rules over large tables. Our solution uses indexes to avoid enumerating the Cartesian product, but faces the problem of what to do when indexes do not fit in memory. We show how the solution can nimbly adapt to these situations by redistributing the indexes and the associated workloads across the mappers and reducers.

- Finally, we consider the challenge of optimizing EM plans. We show that combining machine operations with crowdsourcing introduces novel optimization opportunities, such as using crowd time to mask machine time. We develop masking techniques that use the crowd time to build indexes and to speculatively execute machine operations. We also show to replace an operator with an approximate one which has almost the same accuracy yet introduces significant additional masking opportunities.

1.4 Contributions and Outline of This Dissertation

To summarize, in this dissertation I make the following contributions:

- First, I describe Corleone, a hands-off crowdsourcing system for EM. Corleone is novel in that it crowdsources the entire EM pipeline. As a result, it can scale up EM for enterprises and startups, and can open up crowdsourcing for the masses, e.g., EM as a service on the cloud. Since this is a joint work with Chaitanya Gokhale (and has been described in detail in his PhD dissertation), I explicitly outline my contributions in this dissertation.

- Second, I introduce and describe Rmony, a rule-based EM management system for the analysts. I present extensive evaluation of Rmony with students at UW-Madison and analysts at Walmart to show the efficacy of the system. I describe a real-world case study to show how Rmony was instrumental in improving the accuracy of an EM system in production. Finally, I list the important lessons learned during the development of Rmony which eventually led to the inception of Magellan, an open-source EM management system in the Python ecosystem.

- Third, I show that for emerging topics, e.g., EM as a cloud service, Corleone is ideally suited but must be scaled to large tables to make such services a reality. I introduce and describe Falcon, a solution that scales up crowdsourced EM using RDBMS-style query execution and optimization over a Hadoop cluster. The Hadoop-based solution in Falcon to execute complex rules over the Cartesian product significantly advances the state of the art. I develop three novel optimization techniques to mask machine time by scheduling certain machine activities during crowdsourcing activities. Finally, I present extensive experiments with real-world data sets (using real and synthetic crowds) to show that Falcon can efficiently perform hands-off crowdsourced EM over tables of 1.0M - 2.5M tuples at the cost of $54 - $65.5.

The rest of this dissertation is organized as follows. Chapters 2, 3, and 4 describe Corleone, Rmony, and Falcon, respectively. Chapter 5 concludes this dissertation.

Parts of this dissertation have been published in database conferences. In particular, Corleone (Chapter 2) is described in a SIGMOD-2014 paper [54], and Falcon (Chapter 4) has been accepted to SIGMOD-2017. Magellan is described in two VLDB-2016 papers [69, 68].

Chapter 2

Hands-Off Crowdsourcing for Entity Matching

2.1 Introduction

Entity matching (EM) finds data records that refer to the same real-world entity, such as (David Smith, JHU) and (D. Smith, John Hopkins). This problem has received significant attention (e.g., [13, 33, 70, 37]). In particular, in the past few years crowdsourcing has been increasingly applied to EM. In crowdsourcing, certain parts of a problem are "farmed out" to a crowd of workers to solve. As such, crowdsourcing is well suited for EM, and indeed several crowdsourced EM solutions have been proposed (e.g., [103, 104, 35, 107, 106]).

These pioneering solutions demonstrate the promise of crowdsourced EM, but suffer from a major limitation: they crowdsource only parts of the EM workflow, requiring a developer who knows how to code and match to execute the remaining parts. For example, several recent solutions require a developer to write heuristic rules to reduce the number of candidate pairs to be matched, then train and apply a matcher to the remaining pairs to predict matches (see Section 2.2). They use the crowd only at the end, to verify the predicted matches. The developer must know how to code (e.g., to write heuristic rules in Perl) and match entities (e.g., to select learning models and features).

As described, current solutions do not scale to the growing EM need at enterprises and crowdsourcing startups. Many enterprises (e.g., eBay, Microsoft, Amazon, Walmart) routinely need to solve tens to hundreds of EM tasks, and this need is growing rapidly. It is not possible to crowdsource all these tasks if crowdsourcing each requires the involvement of a developer (even when sharing developers across tasks). To address this problem, enterprises often ask crowdsourcing

startups (e.g., CrowdFlower) to solve the tasks on their behalf. But again, if each task requires a developer, then it is difficult for a startup, with a limited staff, to handle hundreds of EM tasks coming in from multiple enterprises. This is a bottleneck that we have experienced firsthand in our crowdsourcing efforts at two e-commerce enterprises and two crowdsourcing startups, and this was a major motivation behind this work.

Furthermore, current solutions cannot help ordinary users (i.e., the "masses") leverage crowdsourcing to match entities. For example, suppose a journalist wants to match two long lists of political donors, and can pay up to a modest amount, say $500, to the crowd on Amazon's Mechanical Turk (AMT). He or she typically does not know how to code, thus cannot act as a developer and use current solutions. He or she cannot ask a crowdsourcing startup to help either. The startup would need to engage a developer, and $500 is not enough to offset the developer's cost. The same problem would arise for domain scientists, small business workers, end users, and other "data enthusiasts" [58].

To address these problems, in this chapter we introduce the notion of *hands-off crowdsourcing (HOC)*. HOC crowdsources the *entire* workflow of a task, thus requiring no developers. HOC can be a next logical direction for EM and crowdsourcing research, moving from no-, to partial-, to complete crowdsourcing for EM. By requiring no developers, HOC can scale up EM at enterprises and crowdsourcing startups.

HOC can also open up crowdsourcing for the masses. Returning to our example, the journalist wanting to match two lists of donors can just upload the lists to a HOC Web site, and specify how much he or she is willing to pay. The Web site will use the crowd to execute a HOC-based EM workflow, then return the matches. Developing crowdsourcing solutions for the masses (rather than for enterprises) has received rather little attention, despite its potential to magnify many times the impact of crowdsourcing. HOC can significantly advance this direction.

In this chapter we describe Corleone, a HOC solution for EM (named after Don Corleone, the fictional Godfather figure who managed the mob in a hands-off fashion). Corleone uses the crowd (no developers) in all four major steps of the EM process:

- Virtually any large-scale EM problem requires blocking, a step that uses heuristic rules to reduce the number of tuple pairs to be matched (e.g., "if the prices of two products differ by at least $20, then they do not match"). Current solutions require a developer to write such rules. We show how to use the crowd instead. As far as we know, ours is the first solution that uses the crowd, thus removing developers from this important step.

- We develop a solution that uses crowdsourcing to train a learning-based matcher. We show how to use active learning [98] to minimize crowdsourcing costs.

- Users often want to estimate the matching accuracy, e.g., as precision and recall. Surprisingly, very little work has addressed this problem, and we show that this work breaks down when the data is highly skewed by having very few matches (a common situation). We show how to use the crowd to estimate accuracy in a principled fashion. As far as we know, this is the first in-depth solution to this important problem.

- In practice developers often do EM iteratively, with each iteration focusing on the tuple pairs that earlier iterations have failed to match correctly. So far this has been done in an ad-hoc fashion. We show how to address this problem in a rigorous way, using crowdsourcing.

We present extensive experiments over three real-world data sets, showing that Corleone achieves comparable or significantly better accuracy (by as much as 19.8% F_1) than traditional solutions and published results, at a reasonable crowdsourcing cost.

2.2 Background and Related Work

Entity matching has received extensive attention (see Chapter 7 in [37]). A common setting finds all tuple pairs ($a \in A$, $b \in B$) from two relational tables A and B that refer to the same real-world entity. In this work we will consider this setting (leaving other EM settings as ongoing work).

Recently, crowdsourced EM has received increasing attention in academia (e.g., [103, 104, 35, 107, 106]) and industry (e.g., CrowdFlower, CrowdComputing, and SamaSource). These works

either use the crowd to verify predicted matches [103, 104, 35], or find the best questions to ask the crowd [106], or find the best UI to pose such questions [107]. These works still crowdsource only parts of the EM workflow, requiring a developer to execute the remaining parts. In contrast, Corleone tries to crowdsource the entire EM workflow, thus requiring no developers.

Specifically, virtually any large-scale EM workflow starts with blocking, a step that uses heuristic rules to reduce the number of pairs to be matched. This is because the Cartesian product $A \times B$ is often very large, e.g., 10 billion tuple pairs if $|A| = |B| = 100,000$. Matching so many pairs is very expensive or highly impractical. Hence many blocking solutions have been proposed (e.g., [33, 37]). These solutions however do not employ crowdsourcing, and still require a developer (e.g., to write and apply rules, create training data, and build indexes). In contrast, Corleone completely crowdsources this step.

After blocking, the next step builds and applies a matcher (e.g., using hand-crafted rules or learning) to match the surviving pairs (Chapter 7 in [37]). Here the works closest to ours are those that use active learning [94, 13, 17, 80]. These works however either do not use crowdsourcing (requiring a developer to label training data) (e.g., [94, 13, 17]), or use crowdsourcing [80] but do not consider how to effectively handle noisy crowd input and to terminate the active learning process. In contrast, Corleone considers both of these problems, and uses only crowdsourcing, with no developer in the loop.

The next step, estimating the matching accuracy (e.g., as precision and recall), is vital in real-world EM (e.g., so that the user can decide whether to continue the EM process), but surprisingly has received very little attention in EM research. Here the most relevant work is [63, 97]. [63] uses a continuously refined stratified sampling strategy to estimate the accuracy of a classifier. However, it can not be used to estimate recall which is often necessary for EM. [97] considers the problem of constructing the optimal labeled set for evaluating a given classifier given the size of the sample. In contrast, we consider the different problem of constructing a minimal labeled set, given a maximum allowable error bound.

Subsequent steps in the EM process involve "zooming in" on difficult-to-match pairs, revising the matcher, then matching again. While very common in industrial EM, these steps have received

little or no attention in EM research. Corleone shows how they can be executed rigorously, using only the crowd.

Finally, crowdsourcing in general has received significant recent attention [38]. In the database community, the works [78, 85, 48] build crowdsourced RDBMSs. Many other works crowdsource joins [77], find the maximal value [55], collect data [100], match schemas [113], and perform data mining [11] and analytics [74].

2.3 Proposed Solution

We now discuss hands-off crowdsourcing and our proposed Corleone solution.

Hands-Off Crowdsourcing (HOC): Given a problem P supplied by a user U, we say a crowdsourced solution to P is hands-off if it uses no developers, only a crowd of ordinary workers (such as those on AMT). It can ask user U to do a little initial setup work, but this should require no special skills (e.g., coding) and should be doable by any ordinary workers. For example, Corleone only requires a user U to supply

1. two tables A and B to be matched,
2. a short textual instruction to the crowd on what it means for two tuples to match (e.g., "these records describe products sold in a department store, they should match if they represent the same product"), and
3. four examples, two positive and two negative (i.e., pairs that match and do not match, respectively), to illustrate the instruction. EM tasks posted on AMT commonly come with such instructions and examples.

Corleone then uses the crowd to match A and B (sending them information in (2) and (3) to explain what user U means by a match), then returns the matches. As such, Corleone is a hands-off solution. The following real-world example illustrates Corleone and contrasts it with current EM solutions.

Example 2.3.1. *Consider a retailer that must match tens of millions of products between the online division and the brick-and-mortar division (these divisions often obtain products from different sets of suppliers). The products fall into 500+ categories: toys, electronics, homes, etc. To obtain high matching accuracy, the retailer must consider matching products in each category separately, thus effectively having 500 EM problems, one per category.*

Today, solving each of these EM problems (with or without crowdsourcing) requires extensive developer's involvement, e.g., to write blocking rules, to create training data for a learning-based matcher, to estimate the matching accuracy, and to revise the matcher, among others. Thus current solutions are not hands-off. One may argue that once created and trained, a solution to an EM problem, say for toys, is hands-off in that it can be automatically applied to match future toy products, without using a developer. But this ignores the initial non-negligible developer effort put into creating and training the solution (thus violating our definition). Furthermore, this solution cannot be transferred to other categories (e.g., electronics). As a result, extensive developer effort is still required for all 500+ categories, a highly impractical approach.

In contrast, using Corleone*, per category the user only has to provide Items 1-3, as described above (i.e., the two tables to be matched; the matching instruction which is the same across categories; and the four illustrating examples which virtually any crowdsourcing solution would have to provide for the crowd).* Corleone *then uses the crowd to execute all steps of the EM workflow. As such, it is hands-off in that it does not use any developer when solving an EM problem, thus potentially scaling to all 500+ categories.*

We believe HOC is a general notion that can apply to many problem types, such as entity matching, schema matching, information extraction, etc. In this work we will focus on entity matching. Realizing HOC poses serious challenges, in large part because it has been quite hard to figure out how to make the crowd do certain things. For example, how can the crowd write blocking rules (e.g., "if prices differ by at least $20, then two products do not match")? We need rules in machine-readable format (so that we can apply them). However, most ordinary crowd workers cannot write such rules, and if they write in English, we cannot reliably convert them into machine-readable ones. Finally, if we ask them to select among a set of rules, we often can

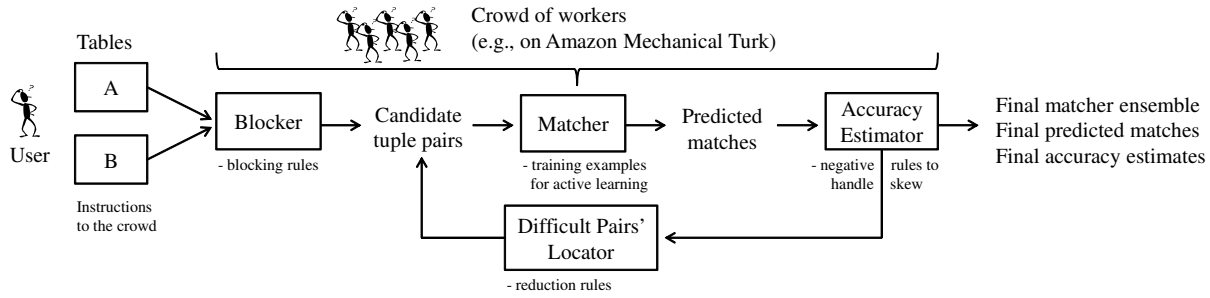


Figure 2.1: The Corleone architecture.

only work with relatively simple rules and it is hard to construct sophisticated ones. Corleone addresses such challenges, and provides a HOC solution for entity matching.

The Corleone Solution: Figure 2.1 shows the Corleone architecture, which consists of four main modules: Blocker, Matcher, Accuracy Estimator, and Difficult Pairs' Locator. The Blocker generates and applies blocking rules to $A \times B$ to remove obviously non-matched pairs. The Matcher uses active learning to train a random forest [22], then applies it to the surviving pairs to predict matches. The Accuracy Estimator computes the accuracy of the Matcher. The Difficult Pairs' Locator finds pairs that the current Matcher has matched incorrectly. The Matcher then learns a better random forest to match these pairs, and so on, until the estimated matching accuracy no longer improves.

As described, Corleone is distinguished in three important ways. (1) All four modules do not use any developer, but heavily use crowdsourcing. (2) In a sense, the modules use crowdsourcing not just to label the data, as existing work has done, but also to create complex rules (blocking rules for the Blocker, negative rules for the Estimator, and reduction rules for the Locator, see Sections 2.4-2.7). And (3) Corleone can be run in many different ways. The default is to run multiple iterations until the estimated accuracy no longer improves. But the user may also decide to just run until a budget (e.g., $300) has been exhausted, or to run just one iteration, or just the Blocker and Matcher, etc.

Next we describe Corleone in detail. Sections 2.4-2.7 describe the Blocker, Matcher, Estimator, and Locator, respectively. Section 2.8 describes the experiments. Section 2.9 discusses my contributions.

2.4 Blocking to Reduce a Set of Pairs

We now describe the Blocker, which generates and applies blocking rules. As discussed earlier, this is critical for large-scale EM. Prior work requires a developer to execute this step. Our goal however is to completely crowdsource it. To do so, we must address the challenge of using the crowd to generate machine-readable blocking rules.

To solve this challenge, Blocker takes a relatively small sample S from $A \times B$; applies crowdsourced active learning, in which the crowd labels a small set of informative pairs in S, to learn a random forest matcher; extracts potential blocking rules from the matcher; uses the crowd again to evaluate the quality of these rules; then retain only the best ones. We now describe these steps in detail.

2.4.1 Generating Candidate Blocking Rules

1. Decide Whether to Do Blocking: Let A and B be the two tables to be matched. Intuitively, we want to do blocking only if $A \times B$ is too large to be processed efficiently by subsequent steps. Currently we deem this is the case if $A \times B$ exceeds a threshold t_B, set to be the largest number such that if after blocking we have t_B tuple pairs, then we can fit the feature vectors of all these pairs in memory (we discuss feature vectors below), thus minimizing I/O costs for subsequent steps. The goal of blocking is then to generate and apply blocking rules to remove as many obviously non-matched pairs from $A \times B$ as possible.

2. Take a Small Sample S from $A \times B$: We want to learn a random forest F, then extract candidate blocking rules from it. Learning F directly over $A \times B$ however is impractical because this set is too large. Hence we will sample a far smaller set S from $A \times B$, then learn F over S. Naïvely, we can randomly sample tuples from A and B, then take their Cartesian product to be

S. Random tuples from A and B however are unlikely to match. So we may get no or very few positive pairs in S, rendering learning ineffective.

To address this problem, we sample as follows. Let A be the smaller table. We randomly sample $t_B/|A|$ tuples from B, then take S to be the Cartesian product between this set of tuples and A. Note that we also add the four examples (two positive, two negative) supplied by the user to S. This way, S has roughly t_B pairs, thus having the largest possible size that still fits in memory, to ensure efficient learning. Furthermore, if B has a reasonable number of tuples that have matches in A, and if these tuples are distributed uniformly in B, then the above strategy ensures that S has a reasonable number of positive pairs. We show empirically later that this simple sampling strategy is effective; exploring better sampling strategies is future work.

3. Apply Crowdsourced Active Learning to S: In the next step, we convert each tuple pair in S into a feature vector, using features taken from a pre-supplied feature library. Example features include edit distance, Jaccard measure, Jaro-Winkler, TF/IDF, Monge-Elkan, etc. (See Chapter 4.2 in [37]). Then we apply crowdsourced active learning to S to learn a random forest F. Briefly, we use the two positive and two negative examples supplied by the user to build an initial forest F, use F to find informative examples in S, ask the crowd to label them, then use the labeled examples to improve F, and so on. A random forest is a set of decision trees [22]. We use decision trees because blocking rules can be naturally extracted from them, as we will see, and we use active learning to minimize the number of examples that the crowd must label. We defer describing this learning process in detail to Section 2.5.

4. Extract Candidate Blocking Rules from F: The active learning process outputs a random forest F, which is a set of decision trees, as mentioned earlier. Figures 2.2.a-b show a toy forest with just two trees (in our experiments each forest has 10 trees, and the trees have $8 - 655$ leaves). Here, the first tree states that two books match only if the ISBNs match and the numbers of pages match. Observe that the leftmost branch of this tree forms a decision rule, shown as the first rule in Figure 2.2.c. This rule states that if the ISBNs do not match, then the two books do not match. It is therefore a negative rule, and can clearly serve as a blocking rule because it identifies book pairs

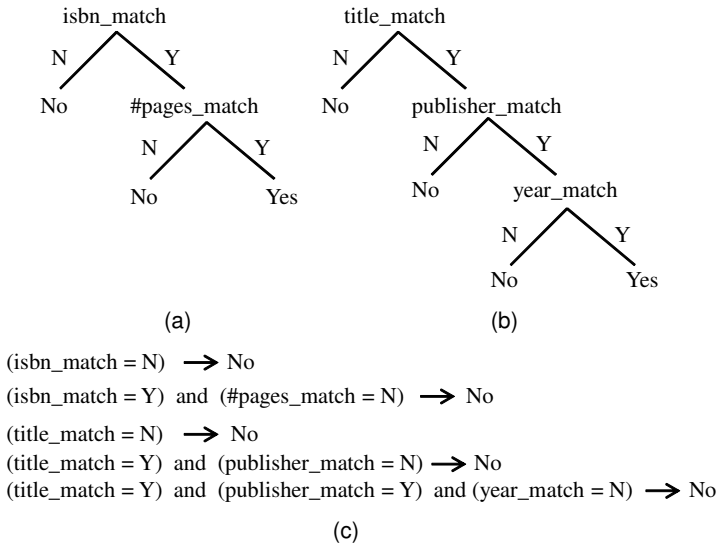


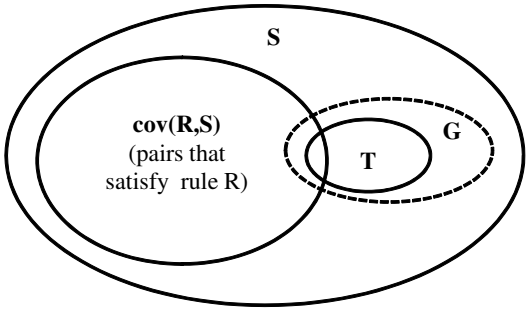
Figure 2.2: (a)-(b) A toy random forest consisting of two decision trees, and (c) negative rules extracted from the forest.

that do not match. In general, given a forest F, we can extract all tree branches that lead from a root to a "no" leaf to form negative rules. Figure 2.2.c show all five negative rules extracted from the forest in Figures 2.2.a-b. We return all negative rules as the set of candidate blocking rules.

2.4.2 Evaluating Rules Using the Crowd

1. Select k Blocking Rules: The extracted blocking rules can vary widely in precision. So we must evaluate and discard the imprecise ones. Ideally, we want to evaluate all rules, using the crowd. This however can be very expensive money-wise (we have to pay the crowd), given the large number of rules (e.g., up to 8943 in our experiments). So we pick only k rules to be evaluated by the crowd (current $k = 20$).

Specifically, for each rule R, we compute the coverage of R over sample S, $cov(R, S)$, to be the set of examples in S for which R predicts "no". We define the precision of R over S, $prec(R, S)$, to be the number of examples in $cov(R, S)$ that are indeed negative divided by $|cov(R, S)|$ (see Figure 2.3). Of course, we cannot compute $prec(R, S)$ because we do not know the true labels of examples in $cov(R, S)$. However, we can compute an upper bound on $prec(R, S)$. Let T



$$\text{prec}(R,S) = \frac{|\,\text{cov}(R,S) - G\,|}{|\,\text{cov}(R,S)\,|}$$

S: sample from A X B
G: actual matching pairs in S
T : pairs in S labeled as matches by the crowd during active learning

Figure 2.3: Coverage and precision of rule R over S.

be the set of examples in S that (a) were selected during the active learning process in Step 3, Section 2.4.1, and (b) have been labeled by the crowd as positive. Then clearly $prec(R, S) \leq |cov(R, S) - T|/|cov(R, S)|$. We then select the rules in decreasing order of the upper bound on $prec(R, S)$, breaking tie using $cov(R, S)$, until we have selected k rules, or have run out of rules. Intuitively, we prefer rules with higher precision and coverage, all else being equal.

2. Evaluate the Selected Rules Using the Crowd: Let V be the set of selected rules. We now use the crowd to estimate the precision of rules in V, then keep only highly precise rules. Specifically, for each rule $R \in V$, we execute the following loop:

1. We randomly select b examples in $cov(R, S)$, use the crowd to label each example as matched / not matched, then add the labeled examples to a set X (initially set to empty).

2. Let $|cov(R, S)| = m$, $|X| = n$, and n_- be the number of examples in X that are labeled negative (i.e., not matched) by the crowd. Then we can estimate the precision of rule R over S as $P = n_-/n$, with an error margin $\epsilon = Z_{1-\delta/2}\sqrt{\left(\frac{P(1-P)}{n}\right)\left(\frac{m-n}{m-1}\right)}$ [105]. This means that the true precision of R over S is in the range $[P - \epsilon, P + \epsilon]$ with a δ confidence (currently set to 0.95).

3. If $P \geq P_{min}$ and $\epsilon \leq \epsilon_{max}$ (which are pre-specified thresholds), then we stop and add R to the set of precise rules. If (a) $(P + \epsilon) < P_{min}$, or (b) $\epsilon \leq \epsilon_{max}$ and $P < P_{min}$, then we stop and drop R (note that in case (b) with continued evaluation P may still exceed P_{min}, but we judge the continued evaluation to be costly, and hence drop R). Otherwise return to Step 1.

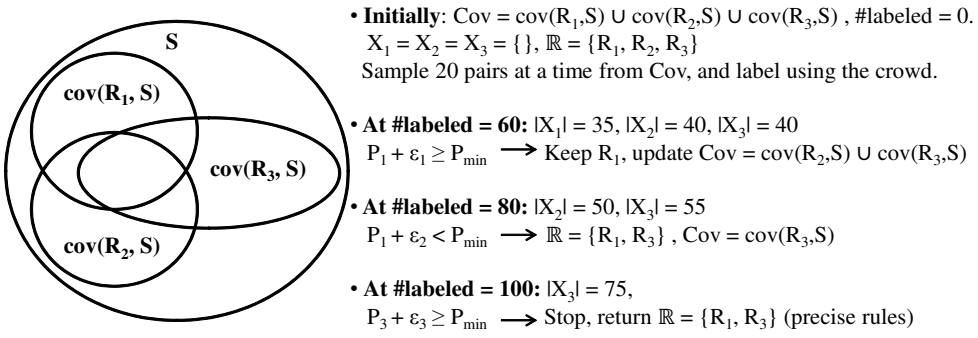


Figure 2.4: Example illustrating joint evaluation of rules.

Currently we set $b = 20, P_{min} = 0.95, \epsilon_{max} = 0.05$.

The above procedure evaluates each rule in V in isolation. We can do better by evaluating all rules in V jointly, to reuse examples across rules. Specifically, let $R_1, \ldots, R_q$ be the rules in V. Then we start by randomly selecting b examples from the *union* of the coverages of $R_1, \ldots, R_q$, use the crowd to label them, then add them to $X_1, \ldots, X_q$, the set of labeled examples that we maintain for the $R_1, \ldots, R_q$, respectively. (For example, if a selected example is in the coverage of only R_1 and R_2, then we add it to X_1 and X_2.) Next, we use $X_1, \ldots, X_q$ to estimate the precision of the rules, as detailed in Step 2, and then to keep or drop rules, as detailed in Step 3. If we keep or drop a rule, we remove it from the union, and sample only from the union of the remaining rules. Figure 2.4 shows an example illustrating joint evaluation of three rules.

2.4.3 Applying Blocking Rules

Let Y be the set of rules in V that have survived crowd-based evaluation. We now consider which subset of rules $\mathcal{R}$ in Y should be applied as blocking rules to $A \times B$.

This is highly non-trivial. Let $Z(\mathcal{R})$ be the set of pairs obtained after applying the subset of rules $\mathcal{R}$ to $A \times B$. If $|Z(\mathcal{R})|$ falls below threshold t_B (recall that our goal is to try to reduce $A \times B$ to t_B pairs, if possible), then among all subsets of rules that satisfy this condition, we will want to select the one whose set $Z(\mathcal{R})$ is the *largest*. This is because we want to reduce the number of pairs to be matched to t_B, but do not want to go too much below that, because then we run the risk of eliminating many true positive pairs. On the other hand, if no subset of rules from Y can reduce

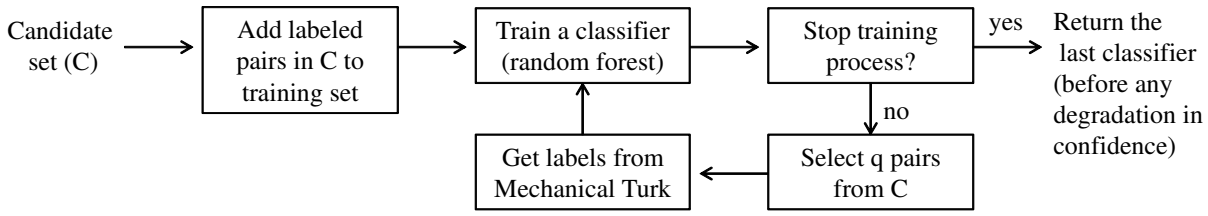


Figure 2.5: Crowdsourced active learning in Corleone.

$A \times B$ to below t_B, then we will want to select the subset that does the most reduction, because we want to minimize the number of pairs to be matched.

We cannot execute all subsets of Y on $A \times B$, in order to select the optimal subset. So we use a greedy solution. First, we rank all rules in Y based on the precision $prec(R, S)$, coverage $cov(R, S)$, and the tuple cost. The tuple cost is the cost of applying rule R to a tuple, primarily the cost of computing the features mentioned in R. We can compute this because we know the cost of computing each feature in Step 3, Section 2.4.1. Next, we select the first rule, apply it to reduce S to S', re-estimate the precision, coverage, and tuple cost of all remaining rules on S', re-rank them, select the second rule, and so on. We repeat until the set of selected rules when applied to S has reduced it to a set of size no more than $|S| * (t_B/|A \times B|)$, or we have selected all rules. We then apply the set of selected rules to $A \times B$ (using a Hadoop cluster), to obtain a smaller set of tuple pairs to be matched. This set is passed to the Matcher, which we describe next.

2.5 Training and Applying a Matcher

Let C be the set of tuple pairs output by the Blocker. We now describe Matcher M, which applies crowdsourcing to learn to match tuple pairs in C. We want to maximize the matching accuracy, while minimizing the crowdsourcing cost. To do this, we use active learning. Figure 2.5 shows the overall workflow for learning the matcher. Specifically, we train an initial matcher M, use it to select a small set of informative examples from C, ask the crowd to label the examples, use them to improve M, and so on. A key challenge is deciding when to stop training M. Excessive training wastes money, and yet surprisingly can actually *decrease*, rather than increase the matcher's accuracy. We now describe matcher M and our solution to the above challenge.

2.5.1 Training the Initial Matcher

We begin by converting all examples (i.e., tuple pairs) in C into feature vectors, for learning purposes. This is done at the end of the blocking step: any surviving example is immediately converted into a feature vector, using all features that are appropriate (e.g., no TF/IDF features for numeric attributes) and available in our feature library. In what follows we use the terms "example", "pair", and "feature vector" interchangeably, when there is no ambiguity.

Next, we use all labeled examples available at that point (supplied by the user or labeled by the crowd) to train an initial classifier that when given an example (x, y) will predict if x matches y. Currently we use an ensemble-of-decision-trees approach called *random forest* [22]. In this approach, we train k decision trees independently, each on a random portion (typically set at 60%) of the original training data. When training a tree, at each tree node we randomly select m features from the full set of features $f_1, \ldots, f_n$, then use the best feature among the m selected to split the remaining training examples. The values k and m are currently set to be the default 10 and $log(n) + 1$, respectively. Once trained, applying a random forest classifier means applying the k decision trees, then taking the majority vote.

2.5.2 Consuming the Next Batch of Examples

Once matcher M has trained a classifier, M evaluates the classifier to decide whether further training is necessary (see Section 2.5.3). Suppose M has decided yes, then it must select new examples for labeling.

In the simplest case, M can select just a single example (as current active learning approaches often do). A crowd however often refuses to label just one example, judging it to be too much overhead for little money. Consequently, M selects q examples (currently set to 20) for the crowd to label. Intuitively, M wants these examples to be "most informative". A common way to measure the "informativeness" of an example e is to measure the disagreement of the component classifiers using entropy [98]:

$$entropy(e) = -[P_+(e) \cdot ln(P_+(e)) + P_-(e) \cdot ln(P_-(e))], \tag{2.1}$$

where $P_+(e)$ and $P_-(e)$ are the fractions of the decision trees in the random forest that label example e positive and negative, respectively. The higher the entropy, the stronger the disagreement, and the more informative the example is.

Thus, M selects the p examples (currently set to 100) with the highest entropy from set C (excluding those that have been selected in the previous iterations). Next, M selects q examples from these p examples, using weighted sampling, with the entropy values being the weights. This sampling step is necessary because M wants the q selected examples to be not just informative, but also diverse. M sends the q selected examples to the crowd to label, adds the labeled examples to the current training data, then re-trains the classifier.

2.5.3 Deciding When to Stop

Recall that matcher M trains in iteration, in each of which it pays the crowd to label q training examples. We must decide then when to stop the training. Interestingly, more iterations of training not only cost more, as expected, but can actually *decrease* rather than increase M's accuracy. This happens because after M has reached peak accuracy, more training, even with perfectly labeled examples, does not supply any more informative examples, and can mislead M instead. This problem became especially acute in crowdsourcing, where crowd-supplied labels can often be incorrect, thereby misleading the matcher even more.

To address this problem, we develop a solution that tells M when to stop training. Our solution defines the "confidence" of M as the degree to which the component decision trees agree with one another when labeling. We then monitor M and stop it when its confidence has peaked, indicating that there are no or few informative examples left to learn from.

Specifically, let $conf(e) = 1 - entropy(e)$, where $entropy(e)$ is computed as in Equation 4.1, be the *confidence* of M over an example e. The smaller the entropy, the more decision trees of M agree with one another when labeling e, and so the more confident M is that it has correctly labeled e.

Before starting the active learning process, we set aside a small portion of C (currently set to be 3%), to be used as a monitoring set V. We monitor the confidence of M over V, defined as

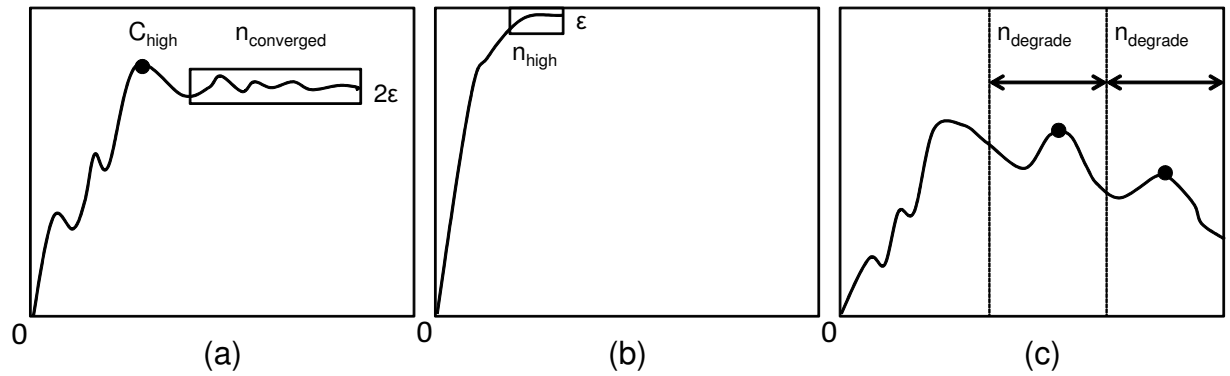


Figure 2.6: Typical confidence patterns that we can exploit for stopping.

$conf(V) = \sum_{e \in V} conf(e)/|V|$. We expect that initially $conf(V)$ is low, reflecting the fact that M has not been trained sufficiently, so the decision trees still disagree a lot when labeling examples. As M is trained with more and more informative examples (see Section 2.5.2), the trees become more and more "robust", and disagree less and less. So $conf(V)$ will rise, i.e., M is becoming more and more confident in its labeling. Eventually there are no or few informative examples left to learn from, so the disagreement of the trees levels off. This means $conf(V)$ will also level off. At this point we stop the training of matcher M.

We now describe the precise stopping conditions, which, as it turned out, was quite tricky to establish. Ideally, once confidence $conf(V)$ has leveled off, it should stay level. In practice, additional training examples may lead the matcher astray, thus reducing or increasing $conf(V)$. This is exacerbated in crowdsourcing, where the crowd-supplied labels may be wrong, leading the matcher even more astray, thus causing drastic "peaks" and "valleys" in the confidence line. This makes it difficult to sift through the "noise" to discern when the confidence appears to have peaked. We solve this problem as follows.

First, we run a smoothing window of size w over the confidence values recorded so far (one value per iteration), using average as the smoothing function. That is, we replace each value x with the average of the w values: $(w-1)/2$ values on the left of x, $(w-1)/2$ values on the right, and x itself. (Currently $w = 5$.) We then stop if we observe any of the following three patterns over the smoothed confidence values:

- **Converged confidence:** In this pattern the confidence values have stabilized and stayed within a 2ϵ interval (i.e., for all values v, $|v - v^*| \leq \epsilon$ for some v^*) over $n_{converged}$ iterations. We use $\epsilon = 0.01$ and $n_{converged} = 20$ in our experiments (these parameters and those described below are set using simulated crowds). Figure 2.6.a illustrates this case. When this happens, the confidence is likely to have converged, and unlikely to still go up or down. So we stop the training.

- **Near-absolute confidence:** This pattern is a special case of the first pattern. In this pattern, the confidence is at least $1 - \epsilon$, for n_{high} consecutive iterations (see Figure 2.6.b). We currently use $n_{high} = 3$. When this pattern happens, confidence has reached a very high, near-absolute value, and has no more room to improve. So we can stop, not having to wait for the whole 20 iterations as in the case of the first pattern.

- **Degrading confidence:** This pattern captures the scenarios where the confidence has reached the peak, then degraded. In this pattern we consider two consecutive windows of size $n_{degrade}$, and find that the maximal value in the first window (i.e., the earlier one in time) is higher than that of the second window by more than ϵ (see Figure 2.6.b). We currently use $n_{degrade} = 15$. We have experimented with several variations of this pattern. For example, we considered comparing the average values of the two windows, or comparing the first value, average value, and the last value of a (relatively long) window. We found however that the above pattern appears to be the best at accurately detecting degrading confidence after the peak.

Afterward, M selects the last classifier before degrading to match the tuple pairs in the input set C.

2.6 Estimating Matching Accuracy

After applying matcher M, Corleone estimates M's accuracy. If this exceeds the best accuracy obtained so far, Corleone continues with another round of matching (see Section 2.7). Otherwise, it stops, returning the matches together with the estimated accuracy. This estimated accuracy is

Datasets	Table A	Table B	# of Matches
Restaurants	533	331	112
Citations	2616	64263	5347
Products	2554	22074	1154

Table 2.1: Data sets for our experiments.

especially useful to the user, as it helps decide how good the crowdsourced matches are and how best to use them. [54] describes how Corleone estimates the matching accuracy in detail.

2.7 Iterating to Improve

In practice, entity matching is not a one-shot operation. Developers often estimate the matching result, then revise and match again. A common way to revise is to find tuple pairs that have proven difficult to match, then modify the current matcher, or build a new matcher specifically for these pairs. For example, when matching e-commerce products, a developer may find that the current matcher does reasonably well across all categories, except in Clothes, and so may build a new matcher specifically for Clothes products.

Corleone operates in a similar fashion. It estimates the matching accuracy (as discussed earlier), then stops if the accuracy does not improve (compared to the previous iteration). Otherwise, it revises and matches again. Specifically, it attempts to locate difficult-to-match pairs, then build a new matcher specifically for those. The challenge is how to locate difficult-to-match pairs. Our key idea is to identify precise positive and negative rules from the learned random forest, then remove all pairs covered by these rules (they are, in a sense, easy to match, because there already exist rules that cover them). We treat the remaining examples as difficult to match, because the current forest does not contain any precise rule that covers them. [54] describes this idea in detail.

2.8 Empirical Evaluation

We now empirically evaluate Corleone. Table 2.1 describes three real-world data sets for our experiments. Restaurants matches restaurant descriptions. Citations matches citations between DBLP and Google Scholar [71]. These two data sets have been used extensively in prior EM

Datasets	Corleone					Baseline 1			Baseline 2			Published Works
	P	R	F_1	Cost	# Pairs	P	R	F_1	P	R	F_1	F_1
Restaurants	97.0	96.1	**96.5**	$9.2	274	10.0	6.1	**7.6**	99.2	93.8	**96.4**	**92-97** [103, 70]
Citations	89.9	94.3	**92.1**	$69.5	2082	90.4	84.3	**87.1**	93.0	91.1	**92.0**	**88-92** [71, 70, 17]
Products	91.5	87.4	**89.3**	$256.8	3205	92.9	26.6	**40.5**	95.0	54.8	**69.5**	Not available

Table 2.2: Comparing the performance of Corleone against that of traditional solutions and published works.

work (Section 2.8.1 compares published results on them with that of Corleone, when appropriate). Products, a new data set created by us, matches electronic products between Amazon and Walmart. Overall, our goal is to select a diverse set of data sets, with varying matching difficulties.

We used Mechanical Turk and ran Corleone on each data set three times, each in a different week. The results reported below are averaged over the three runs. In each run we used common turker qualifications to avoid spammers, such as allowing only turkers with at least 100 approved HITs and 95% approval rate. We paid 1 cent per question for Restaurants & Citations, and 2 cents for Products (it can take longer to answer Product questions due to more attributes involved).

2.8.1 Overall Performance

Accuracy and Cost: We begin by examining the overall performance of Corleone. The first five columns of Table 2.2 (under "Corleone") show this performance, broken down into P, R, F_1, the total cost, and the total number of tuple pairs labeled by the crowd. The results show that Corleone achieves high matching accuracy, 89.3-96.5% F_1, across the three data sets, at a reasonable total cost of $9.2-256.8. The number of pairs being labeled, 274-3205, is low compared to the total number of pairs. For example, after blocking, Products has more than 173,000 pairs, and yet only 3205 pairs need to be labeled, thereby demonstrating the effectiveness of Corleone in minimizing the labeling cost.

Comparison to Traditional Solutions: In the next step, we compare Corleone to two traditional solutions: Baseline 1 and Baseline 2. Baseline 1 uses a developer to perform blocking, then trains a random forest using the same number of labeled pairs as the average number of labeled pairs used by Corleone. Baseline 2 is similar to Baseline 1, but uses 20% of the candidate set (obtained

after blocking) for training. For example, for Products, Baseline 1 uses 3205 pairs for training (same as Corleone), while Baseline 2 uses 20% * 180,382 = 36,076 pairs, more than 11 times what Corleone uses. Baseline 2 is therefore a very strong baseline matcher.

The next six columns of Table 2.2 show the accuracy (P, R, and F_1) of Baseline 1 and Baseline 2. The results show that Corleone significantly outperforms Baseline 1 (89.3-96.5% F_1 vs. 7.6-87.1% F_1), thereby demonstrating the importance of active learning, as used in Corleone. Corleone is comparable to Baseline 2 for Restaurants and Citations (92.1-96.5% vs. 92.0-96.4%), but significantly outperforms Baseline 2 for Products (89.3% vs. 69.5%). This is despite the fact that Baseline 2 uses 11 times more training examples.

Comparison to Published Results: The last column of Table 2.2 shows F_1 results reported by prior EM work for Restaurants and Citations. On Restaurants, [70] reports 92-97% F_1 for several works that they compare. Furthermore, CrowdER [103], a recent crowdsourced EM work, reports 92% F_1 at a cost of \$8.4. In contrast, Corleone achieves 96.5% F_1 at a cost of \$9.2 (including the cost of estimating accuracy). On Citations, [71, 70, 17] report 88-92% F_1, compared to 92.1% F_1 for Corleone. It is important to emphasize that due to different experimental settings, the above results are not directly comparable. However, they do suggest that Corleone has reasonable accuracy and cost, while being hands-off.

Summary: The overall result suggests that Corleone achieves comparable or in certain cases significantly better accuracy than traditional solutions and published results, at a reasonable crowdsourcing cost. The important advantage of Corleone is that it is totally hands-off, requiring no developer in the loop, and it provides accuracy estimates of the matching result.

2.8.2 Performance of the Components

We now "zoom in" to examine Corleone in more details.

Blocking: Table 2.3 shows the results for crowdsourced automatic blocking executed on the three data sets. From left to right, the columns show the size of the Cartesian product (of tables A and B), the size of the candidate set (i.e., the set after applying the blocking rules), recall (i.e., the

Datasets	Cartesian Product	Candidate Set	Recall (%)	Cost	# Pairs
Restaurants	176.4K	176.4K	100	$0	0
Citations	168.1M	38.2K	99	$7.2	214
Products	56.4M	173.4K	92	$22	333

Table 2.3: Blocking results for Corleone.

Datasets	Iteration 1				Estimation 1				Reduction 1		Iteration 2				Estimation 2			
	# Pairs	P	R	F_1	# Pairs	P	R	F_1	# Pairs	Reduced Set	# Pairs	P	R	F_1	# Pairs	P	R	F_1
Restaurants	140	97	96.1	96.5	134	95.6	96.3	96	0	157								
Citations	973	89.4	94.2	91.7	366	92.4	93.8	93.1	213	4934	475	89.9	94.3	92.1	0	95.2	95.7	95.5
Products	1060	89.7	82.8	86	1677	90.9	86.1	88.3	94	4212	597	91.5	87.4	89.3	0	96	93.5	94.7

Table 2.4: Corleone's performance per iteration on the data sets.

percentage of positive examples in the Cartesian product that are retained in the candidate set), total cost, and total number of pairs being labeled by the crowd. Note that Restaurants is relatively small and hence does not trigger blocking.

The results show that automatic crowdsourced blocking is quite effective, reducing the total number of pairs to be matched to just 0.02-0.3% of the Cartesian product, for Citations and Products. This is achieved at a low cost of $7.2-22, or just 214-333 examples having to be labeled. In all the runs, Corleone applied 1-3 blocking rules. These rules have 99.9-99.99% precision. Finally, Corleone also achieves high recall of 92-99% on Products and Citations. For comparison purposes, we asked a developer well versed in EM to write blocking rules. The developer achieved 100% recall on Citations, reducing the Cartesian product to 202.5K pairs (far higher than our result of 38.2K pair). Blocking on Products turned out to be quite difficult, and the developer achieved a recall of 90%, compared to our result of 92%. Overall, the results suggest that Corleone can find highly precise blocking rules at a low cost, to dramatically reduce the Cartesian products, while achieving high recall.

Performance of the Iterations: Table 2.4 shows Corleone's performance per iteration on each data set. To explain, consider for example the result for Restaurants (the first row of the table). In Iteration 1 Corleone trains and applies a matcher. This step uses the crowd to label 140 examples, and achieves a true F_1 of 96.5%. Next, in Estimation 1, Corleone estimates the matching accuracy in Iteration 1. This step uses 134 examples, and produces an estimated F_1 of 96% (very close to

the true F_1 of 96.5%). Next, in Reduction 1, Corleone identifies the difficult pairs and comes up with 157 such pairs. It uses no new examples, being able to re-use existing examples. At this point, since the set of difficult pairs is too small (below 200), Corleone stops, returning the matching results of Iteration 1.

The result shows that Corleone needs 1-2 iterations on the three data sets. The estimated F_1 is quite accurate, always within 0.5-5.4% of true F_1. Note that sometimes the estimation error can be larger than our desired maximal margin of 5% (e.g., Estimation 2 for Products). This is due to the noisy labels from the crowd. Despite the crowd noise, however, the effect on estimation error is relatively insignificant. Note that the iterative process can indeed lead to improvement in F_1, e.g., by 3.3% for Products from the first to the second iteration (see more below). Note further that the cost of reduction is just a modest fraction (3-10%) of the overall cost.

2.9 My Contributions

This work was carried out in collaboration with Chaitanya Gokhale. My main contributions here are the following:

- collaborating with Chaitanya to work out the Corleone architecture and to design the experiments (see Sections 2.3 and 2.8);
- developing a MapReduce solution to apply blocking rules to the input tables (see Section 2.9.1 below for details);
- developing a MapReduce solution to generate a feature vector for every tuple pair in a given set of tuple pairs (see Section 2.9.2 below for details); and
- performing experiments on the Citations data set (see Section 2.8).

2.9.1 Applying Blocking Rules to the Input Tables

In Section 2.4.3, we apply a set of blocking rules to the two input tables to obtain a candidate set of tuple pairs. (This is the step behind going from (a) to (b) in Figure 2.7.) I implemented

two MapReduce-based solutions, MapSide and ReduceSplit, to apply the blocking rules to the two input tables on a Hadoop cluster. The two solutions were proposed in [65]. Next, I describe the two solutions at a high-level, complete details can be found in [65].

id	isbn	title	publisher	year	#pages
1	981	Cosmos	Random House	1980	550
2	937	Twilight	Little, Brown	2005	320
…	…	…	…	…	…

A

id	isbn	title	publisher	year	#pages
1	937	Twilight	Yen Press	2010	120
2	981	Cosmos	Random	1980	550
…	…	…	…	…	…

B

(a) The two tables A and B

aid	bid	aisbn	bisbn	atitle	btitle	…
1	2	981	981	Cosmos	Cosmos	…
2	1	937	937	Twilight	Twilight	…
…	…	…	…	…	…	…

(b) Candidate set C

aid	bid	isbn_ match	title_ match	publisher _match	year_ match	pages_ match
1	2	1	1	0	1	1
2	1	1	1	0	0	0
…	…	…	…	…	…	

(c) Feature vectors for pairs in C

Figure 2.7: (a) The two tables A and B, (b) candidate set C after blocking, and (c) feature vectors for pairs in C after feature vector generation.

1. MapSide: This solution is an implementation of broadcast join, where the smaller table (say A) is broadcast to all map tasks. All map tasks load A in memory at the initialization time. The larger table, B, resides on the distributed file system as several partitions. Each map task reads the partition of B assigned to it, one tuple at a time. For each tuple b of B (read from disk) and each tuple a in A (read from memory), we form a tuple pair (a, b). We apply the blocking rules to the pair (a, b). The pairs that do not get eliminated by the blocking rules are output as candidate pairs. There are no reduce tasks in this solution.

2. ReduceSplit: This solution is employed when table A is not small enough to fit in memory of a mapper node. It applies the blocking rules on the pairs in the reduce phase. In the map phase, the tuples of A and B (both residing on the distributed file system) are distributed in such a way that: (1) all possible pairs (a, b) reach the reducers, and (2) all reducers get (almost) the same load. To achieve this, the solution uses special composite keys for the map and reduce tasks and a specific

partitioning scheme for the shuffle phase. The reducer on receiving a pair, applies the blocking rules on it. The pair is output as a candidate pair if it does not get eliminated by the blocking rules.

MapSide is usually faster than ReduceSplit, because there is no Reduce phase in MapSide. If the smaller table can fit in memory of a mapper node, then MapSide is our preferred choice. (In fact, for all the data sets in our experiments, we used the MapSide implementation). If the smaller table does not fit in memory of a mapper node, then we choose ReduceSplit.

2.9.2 Generating Features for a Set of Tuple Pairs

We convert a set of tuple pairs into a set of feature vectors for learning purposes (See Section 2.5.1). This is done at two places: (1) at the end of the sampling step (Section 2.4.1) when we convert the sample of tuple pairs into a set of feature vectors, and (2) at the end of the blocking step when any tuple pair that survived the blocking rules is immediately converted into a feature vector (e.g., going from (b) to (c) in Figure 2.7). We describe the solution below.

Given a list of features $F = \{f_1, ..., f_m\}$ and a set of tuple pairs S, we want to generate a feature vector for each tuple pair (a, b) in S. This step is trivially parallelizable. We implement a MapReduce solution with a single Map-only job. Each Mapper reads a pair (a, b) from S (residing on HDFS); computes a feature value $f_i(a, b)$ for each feature $f_i \in F$; and outputs a key-value pair where key is the composition of ids of a and b: $<a.id, b.id>$, and value is the feature vector: $<f_1(a, b), ..., f_m(a, b)>$. There is no reduce phase.

2.10 Conclusion

We have proposed hands-off crowdsourcing (HOC), which crowdsources the entire EM workflow, without using a developer. We showed how HOC can scale to EM needs at enterprises and startups, and open up crowdsourcing (by the masses) for the masses. We believe HOC can represent a next logical direction for crowdsourcing research. We have also presented Corleone, a HOC solution for EM, and showed that it achieves comparable or better accuracy than traditional solutions and published results, at a reasonable crowdsourcing cost.

Chapter 3

Developing a Rule-Based EM Management System for Analysts

3.1 Introduction

In a recent study on Big Data [75], McKinsey Global Institute (MGI) estimated that by 2018, in the US alone up to 1.5 million new managers and data analysts will be needed to analyze the data being generated across various sectors. Data analysts are individuals who have been trained in the business needs but generally do not have programming experience or background in Computer Science.

EM is one such data management problem which may be performed by data analysts. For example, product matching in e-commerce companies, such as Walmart, is often performed by data analysts [88]. However, currently there are no interactive tools available for these "non-technical" data analysts to perform matching quickly and accurately.

Discussions with analysts at WalmartLabs suggest that for such a tool to be useful, it should have the following three features. First, the system should be end-to-end, i.e., it must be a "management" rather than just a "matching" system. Second, it must provide support for EM rules. The analysts often prefer writing rules to do matching, such as "two books match if they agree on the ISBNs and the number of pages". This is because rules are relatively easy to understand, tune and debug. Further, domain knowledge can be easily encoded in the form of rules. Finally, such a tool must have good visualizations and debugging capabilities.

It turns out that some of the techniques that we develop in Corleone could potentially be used here. For example, in Corleone we extract rules from a trained random forest. This technique could be used in the new system to learn rules. Another example is the use of active learning in

Corleone where the crowd iteratively labels examples to train a matcher. Again, this technique could be used in the system to learn rules except that instead of the crowd, the analyst could label example pairs.

Thus, in this research direction, we build on Corleone to develop a rule-based EM management system for analysts. Specifically, we explore the following topics:

- First, we develop Rmony[1], a system that helps an analyst manually write EM rules to match two input tables, using a GUI.
- Manually writing rules can be time consuming. Thus in the next step, we extend Rmony to automatically suggest EM rules to the analyst, using the technology underlying Corleone. Specifically, the analyst can perform active learning to train a learning-based matcher. EM rules are then extracted from this matcher.
- Finally, we evaluate Rmony in real-world settings, e.g., product matching at WalmartLabs.

The research questions that we seek to answer in this direction are the following:

- Can we develop a system using which an analyst can manually write EM rules effectively? What will be the quality of such rules? Will the analyst be able to write EM rules for diverse domains?
- Can we develop a system to automatically suggest EM rules to analysts? What will be the quality of such rules? Can high-quality rules be suggested for diverse domains?
- Can such a system be effectively used in real-world settings? What lessons can we learn?

To explore these questions, we have developed and experimented with several system variations, both with students at UW-Madison and analysts at WalmartLabs. During the course, we have drawn a set of observations and lessons that we detail in this chapter.

The rest of this chapter is organized as follows. Section 3.2 describes the first version of our system, Rmony 0.1, where an analyst can manually write EM rules. Section 3.3 describes our

[1] Rmony is a pun on "harmony" which means "agreement/match" and the 'R' in Rmony is for "rules".

extended version Rmony 1.1, which can automatically suggest EM rules to analysts. Section 3.4 presents the empirical evaluation of the two versions of Rmony. Section 3.5 describes a case study to demonstrate that Rmony can be effectively used in real-world settings. Section 3.6 discusses the important lessons learned. Section 3.7 describes the related work and Section 3.8 concludes.

3.2 Rmony 0.1: A System to Manually Write EM Rules

Using Rmony 0.1, an analyst can match two tables by writing EM rules manually. Typically, she will start by uploading two tables of records A and B. Then she will try to understand the data by browsing the tables, viewing statistics, searching for keywords, querying on attributes, etc. She can optionally perform basic cleaning of the tables by editing values, deleting tuples/attributes, etc. Next, she will perform blocking on the two tables to obtain a candidate set of tuple pairs C.

Next, she will manually create a matcher, which is a set of EM rules that she wants to apply on C to perform matching. Next, she will apply the matcher on C to obtain predicted matches M. To know the quality of the matcher, she will evaluate it using labeled data. Finally, she can debug the predicted matches M to analyze the different errors (e.g., false positives, false negatives). If she is satisfied with the quality of the results, she will stop here. If not, she will go back and refine the matcher, or go back further to understand the data better, and continue iterating. We now discuss these steps in detail.

3.2.1 Uploading Tables and Understanding Data

Typically, an analyst starts by uploading the two tables (into Rmony) that she wants to match. Rmony supports uploading of tables in CSV and JSON file formats. Next the analyst may want to understand the data before proceeding with matching. To this end, Rmony supports the following actions:

- Browse the records of a table through a paginated view (e.g., Figure 3.1 shows the records of a “books” table),

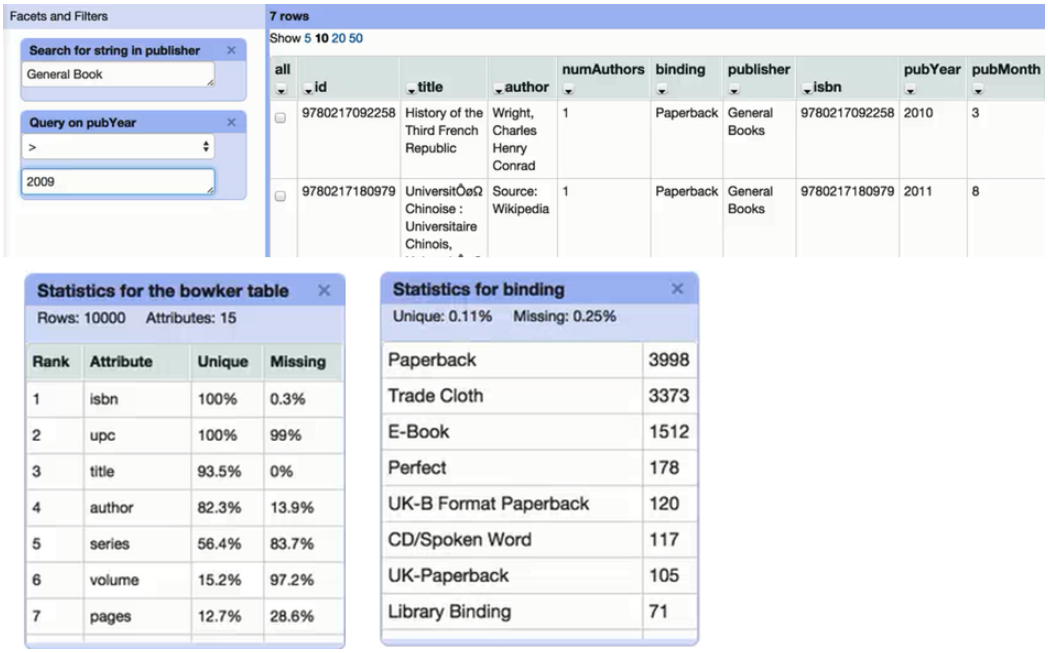


Figure 3.1: Understanding data by browsing tables, viewing statistics, querying tables, etc.

- View table-wide statistics (e.g., how many attributes, percentage of missing and unique values for each attribute as shown in Figure 3.1),
- View attribute-specific statistics (e.g., Figure 3.1 shows the frequency of values in the "binding" attribute),
- Sort the table on an attribute,
- Search for keywords in an attribute (e.g., Figure 3.1 shows searching for "General Books" in the "Publisher" attribute) and write basic selection queries (e.g., show books that were published after 2009 as in Figure 3.1), and
- Take a uniform random sample of a table.

The analyst can also perform basic data cleaning actions, e.g., edit attribute values in a table, delete tuples/attributes from a table, etc.

3.2.2 Performing Blocking to Reduce the Candidate Set of Tuple Pairs

It is often impractical to perform matching on the Cartesian product $A \times B$ of two tables A and B because $A \times B$ can be very large. For example, two tables of 100K tuples each will result in a Cartesian product of 10B tuple pairs. Typically, an analyst will perform blocking on two tables to remove the obviously non-matched tuple pairs, thereby producing a candidate set of tuple pairs much smaller than the Cartesian product. Rmony supports the following two blocking methods:

1. Attribute equivalence based blocking, where a tuple pair is kept in the candidate set if the two tuples have the exact same value for an attribute (e.g., two persons match if they live in the same state), and

2. Rule based blocking, where a tuple pair is kept in the candidate set if it satisfies a rule (e.g., two books match only if they differ in price by at most $10).

3.2.3 Manually Creating Features, Rules, and Matchers

Next the analyst will manually create features, rules, and matchers that she can use to match the two tables. For example, suppose the analyst knows (from her domain knowledge) that two books match if they have the exact same ISBNs, or their titles are highly similar and their prices differ by at most $10. To encode this knowledge into a matcher, in Rmony, she needs to execute the following three steps (in order):

1. First, she will create 3 features:

   - f1(a, b): exactMatch(a.isbn, b.isbn),
   - f2(a, b): Jaccard.word(a.title, b.title), and
   - f3(a, b): absDiff(a.price, b.price).

Each feature f(a, b) maps a tuple pair (a, b) into a numeric score. These features use standard functions (e.g., exactMatch, Jaccard.word) from Rmony's library, which currently comprises 24 standard similarity and distance functions (e.g., Jaccard, edit distance, Monge-Elkan, TF-IDF, Smith-Waterman-Gotoh, Jaro-Winkler, etc.). In the above example, Jaccard.word(a.title, b.title) stands for the Jaccard measure applied to the two sets of tokens obtained after word-based tokenization of title attribute values of the tuples a and b respectively.

2. Next, she will create 2 rules:

   - r1(a, b): f1(a, b) == 1, and
   - r2(a, b): f2(a, b) $>$ 0.9 and f3(a, b) $\leq$ 10.

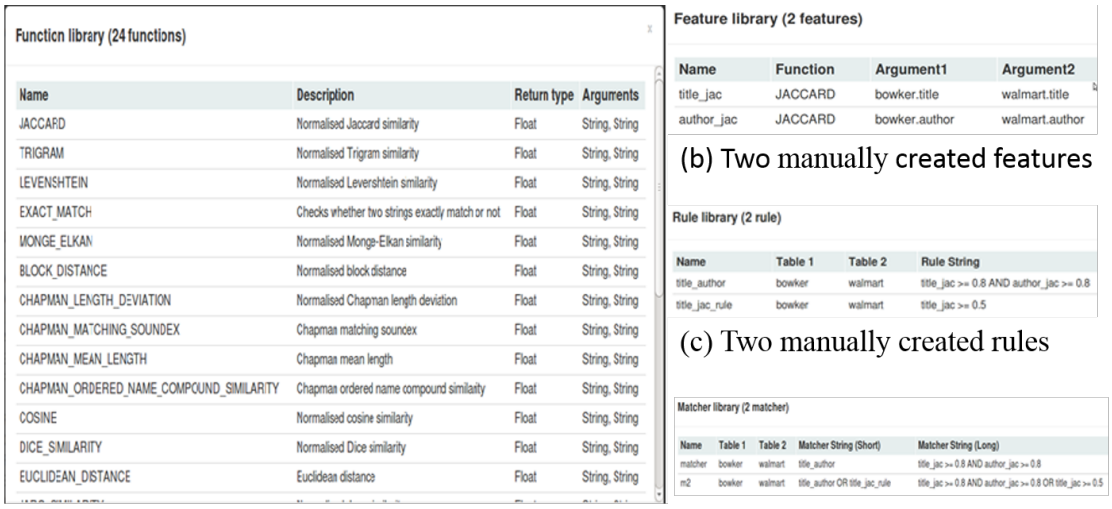
   In Rmony a rule is a conjunction of predicates. Each predicate is of the form $[f(a, b)\ op\ v]$, where $f(a, b)$ is a feature, op is an operator (e.g., $=$, $<$, $\leq$) and v is a value. A rule r(a, b) maps a tuple pair (a, b) to true/false; true indicating that the tuple pair satisfies the rule, false otherwise.

3. Finally, she will create a matcher:

   - m1(a, b): r1(a, b) or r2(a, b).

   In Rmony a matcher m(a, b) is a disjunction of rules. It evaluates to true if one of the rules evaluates to true, indicating that the tuples a and b match.

Figure 3.2 shows screen shots showing (a) the library of 24 functions currently supported in Rmony, (b) features manually created by an analyst using the library of functions, (c) rules manually created by an analyst, and (d) matchers manually created by an analyst.



Function library (24 functions)

Name	Description	Return type	Arguments
JACCARD	Normalised Jaccard similarity	Float	String, String
TRIGRAM	Normalised Trigram similarity	Float	String, String
LEVENSHTEIN	Normalised Levenshtein similarity	Float	String, String
EXACT_MATCH	Checks whether two strings exactly match or not	Float	String, String
MONGE_ELKAN	Normalised Monge-Elkan similarity	Float	String, String
BLOCK_DISTANCE	Normalised block distance	Float	String, String
CHAPMAN_LENGTH_DEVIATION	Normalised Chapman length deviation	Float	String, String
CHAPMAN_MATCHING_SOUNDEX	Chapman matching soundex	Float	String, String
CHAPMAN_MEAN_LENGTH	Chapman mean length	Float	String, String
CHAPMAN_ORDERED_NAME_COMPOUND_SIMILARITY	Chapman ordered name compound similarity	Float	String, String
COSINE	Normalised cosine similarity	Float	String, String
DICE_SIMILARITY	Normalised Dice similarity	Float	String, String
EUCLIDEAN_DISTANCE	Euclidean distance	Float	String, String

(a) Library of 24 standard string similarity functions

Feature library (2 features)

Name	Function	Argument1	Argument2
title_jac	JACCARD	bowker.title	walmart.title
author_jac	JACCARD	bowker.author	walmart.author

(b) Two manually created features

Rule library (2 rule)

Name	Table 1	Table 2	Rule String
title_author	bowker	walmart	title_jac >= 0.8 AND author_jac >= 0.8
title_jac_rule	bowker	walmart	title_jac >= 0.5

(c) Two manually created rules

Matcher library (2 matcher)

Name	Table 1	Table 2	Matcher String (Short)	Matcher String (Long)
matcher	bowker	walmart	title_author	title_jac >= 0.8 AND author_jac >= 0.8
m2	bowker	walmart	title_author OR title_jac_rule	title_jac >= 0.8 AND author_jac >= 0.8 OR title_jac >= 0.5

(d) Two manually created matchers

Figure 3.2: Screen shots from Rmony showing (a) the library of supported functions, (b) manually created features, (c) manually created rules, and (d) manually created matchers.

3.2.4 Performing Matching on the Candidate Set Using a Matcher

Next the analyst will apply the matcher (that she has manually created) to the candidate set of tuple pairs to obtain a prediction ("match" or "no-match") for each tuple pair in the candidate set. Matching in Rmony can be performed in two modes:

1. Fast, where Rmony uses short-circuiting techniques (e.g., skipping evaluation of rules in a matcher and evaluation of predicates in a rule wherever appropriate). Specifically, since the matcher is a disjunction of rules, if one rule evaluates to true for a tuple pair, then the evaluation of the other rules is skipped. Similarly, for a rule which is a conjunction of predicates, if one predicate evaluates to false for a tuple pair, then the evaluation of the other predicates is skipped.

2. Debug, where Rmony evaluates every predicate in a rule, and every rule in the matcher and also collects a lot of matching details (e.g., which rules evaluated to true). These details help the analyst debug the results later as we shall see shortly.

3.2.5 Evaluating a Matcher with Labeled Data

Next the analyst can evaluate how good a matcher is (in terms of precision and recall) using a labeled set of tuple pairs. Each tuple pair in the labeled set is either labeled as "match" or "no-match". The matcher is first applied on the labeled set to get a predicted label for each tuple pair. These predicted labels are then compared against the actual labels to compute precision and recall for the matcher.

3.2.6 Debugging the Results

EM is never a one-shot process. An analyst usually writes an initial set of rules, creates a matcher, applies the matcher on the candidate set of tuple pairs, and evaluates the quality. If the quality does not meet the requirements, then she needs to iteratively improve the matching pipeline.

This is a very time-consuming process. In fact, based on our discussions with the analysts at WalmartLabs, we anticipate that the analyst will spend most of her time trying to iteratively improve the quality of the matching pipeline. Therefore, it is critical to save the analyst time.

Correspondingly, Rmony provides debugging support to help the analyst quickly and iteratively improve the quality of a matcher. Briefly, the analyst can perform the following debugging actions in Rmony:

- Identify which rules in a matcher and which predicates in a rule evaluated to true on a tuple pair and which did not,
- Quickly browse through the set of precision errors (i.e., false positives) and recall errors (i.e., false negatives), and
- Fine-tune the thresholds of the existing rules in the matcher and re-evaluate the quality. Internally, Rmony does not perform the matching process from scratch but quickly recomputes the effect of the change using the matching details that it had previously computed.

Figure 3.3 shows a screen shot of the debugging interface in Rmony. To the left, Rmony shows the evaluation summary of a matcher which includes precision, recall, and F_1 of the matcher on a

Figure 3.3: Debugging the results.

labeled candidate set of tuple pairs. Rmony also shows the number of precision and recall errors (i.e., false positives and false negatives respectively) which are clickable links to help the analyst zoom in and debug.

The matching summary section shows the number of candidate pairs and how many are predicted as "match" by the matcher. It also shows the total number of rules in the matcher. In the rule summary section, for each rule Rmony shows how many candidate pairs were matched by this rule.

In the matching details box (to the right), for each tuple pair Rmony shows which rule is satisfied and which is not. For each rule, Rmony digs deeper to show which predicate in the rule is satisfied and which is not. This helps an analyst identify exactly why a tuple pair matched or did not match.

3.3 Rmony 1.1: Automatically Suggesting EM Rules

Manually creating features and rules can be very challenging for an analyst for two reasons. First, it is very time consuming. Second, the analyst may not have the technical know how (e.g., which functions to use for which attributes when creating features). So we extended Rmony from 0.1 to 1.1 to support the following:

- Automatic generation of features,
- Automatic suggestion of rules using training data, and

Attribute Type and Characteristic	Similarity Measures	Intuition
Single word string	Jaro, Jaro-Winkler, Levenshtein, Jaccard.3gram, Exact Match	May be first names, last names, zip codes, etc.
Multi-word short (#words <= 5) string	Monge-Elkan, Jaccard.word, Jaccard.3gram, Needleman-Wunsch, Smith-Waterman, Smith-Waterman-Gotoh, Cosine	May be product titles, full names, etc.
Multi-word mid-sized (6 <= #words <= 10) string	Monge-Elkan, Jaccard.word, Cosine	May be street addresses, short product descriptions, etc.
Multi-word long (#words >= 11) string	Jaccard.word, TF/IDF, Soft TF/IDF, Cosine	May be long product descriptions, product reviews, etc.
Numeric	Absolute Difference, Relative Difference, Levenshtein, Exact Match	May be age, size, weight, height, price, etc.
Categorical	Exact Match, Levenshtein	The assumption is that the normalization has been done first (e.g., Paperback = Softcover = Trade Paper) using a dictionary
Set-valued	Jaccard, Dice, Overlap, TF/IDF, Soft TF/IDF	May be book authors, ingredients in a food item, etc.

Figure 3.4: Guiding rules for automatically generating features.

- Automatic suggestion of rules using active learning.

We describe these extensions next.

3.3.1 Automatic Generation of Features

Manually creating features can be very challenging for an analyst. First, she may not have the technical know-how of which functions to use for which attributes. Second, she can create only a handful of features manually, which may not be sufficient to write (or learn) good rules.

Rmony 1.1 can automatically generate features. Given two tables, Rmony examines the attribute types (e.g., string, numeric), and characteristics (e.g., short string, long string) to automatically generate features. The guiding rules used by Rmony for automatic feature generation are tabulated in Figure 3.4. We next describe generating features automatically in detail.

Conceptually, a feature is a function that maps a tuple pair (a, b) into a numeric score. However, in Rmony we currently only consider features of the form $f(a, b) = sim(a.x, b.y)$, where sim is a similarity function (e.g., Jaccard, Levenshtein), $a.x$ is the value of attribute x of tuple a (from table A), and $b.y$ is the value of attribute y of tuple b (from table B). For example, if Rmony has inferred

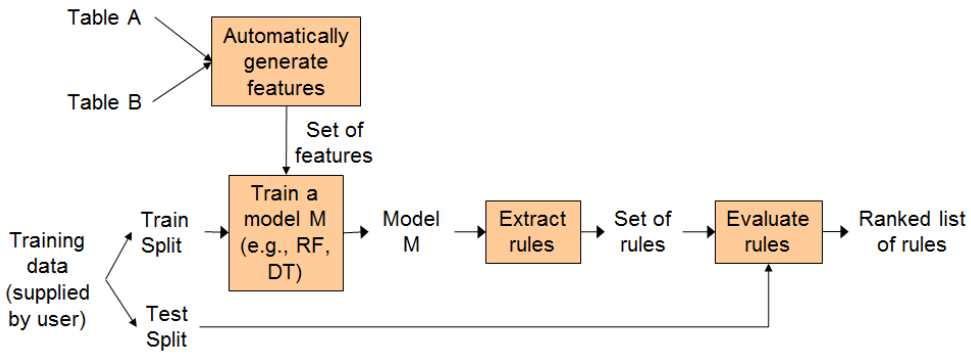


Figure 3.5: An overview of learning rules using training data.

that attributes $A.title$ and $B.title$ are multi-word mid-sized string attributes, then it will automatically generate features such as $Monge{-}Elkan(A.title, B.title)$, $Jaccard.word(A.title, B.title)$, and $cosine(A.title, B.title)$ (see row 3 in Figure 3.4).

To decide on the set of features $F = \{f_1, ..., f_m\}$ to generate automatically, Rmony first asks the analyst to manually specify the attribute correspondences between the two tables A and B by pairing an attribute in table A with an attribute in table B (e.g., $A.title$ with $B.title$, $A.address$ with $B.location$). Next, Rmony scans through the tables to determine the characteristics (e.g., single-word string, multi-word long string, etc.) of each attribute in the two tables.

Next, for each attribute correspondence (x, y), Rmony includes in F a set of features, each of the form $sim(a.x, b.y)$ where sim is a similarity function chosen based on the rules in Figure 3.4. If x and y have different attribute characteristics (but same attribute types), Rmony chooses the characteristic that is at a lower row in Figure 3.4. For example, suppose x is a multi-word short string attribute (row 2 in Figure 3.4) and y is a multi-word mid-sized string attribute (row 3 in Figure 3.4), then Rmony chooses the similarity functions corresponding to multi-word mid-sized string attribute (row 3 in Figure 3.4) for feature generation. Attribute correspondences between attributes of different types (e.g., string with numeric) are ignored (and alerted to the analyst).

3.3.2 Automatic Suggestion of Rules Using Training Data

Rmony 1.1 can automatically suggest rules to analysts using training data. The analyst must supply the training data, which is a set of tuple pairs, each labeled "match" or "no-match". Figure 3.5 shows an overview of learning rules using training data.

First the analyst uploads the two tables A and B that she wants to match. Next she either manually creates features or automatically generates features, or does a mix of both. Automatic feature generation (shown in Figure 3.5) is often the preferred choice. Next she supplies training data. Rmony splits this training data into train and test splits (the split-ratio is configurable, default being 67%-33%).

Next Rmony encodes the tuple pairs in the train split into a set of feature vectors (using the features specified by the analyst). Rmony then trains a model (either a decision tree or a random forest, default being random forest) using the train split (encoded as feature vectors), extracts matching rules from the model, evaluates the rules (in terms of precision and recall) using the test split and then outputs a ranked list of top rules. The analyst then selects a subset from the ranked list of rules. Note that Rmony uses the ideas in Corleone, specifically, the idea of training a random forest and extracting rules from it.

3.3.3 Automatic Suggestion of Rules Using Active Learning

Often it is difficult to get sufficient training data for learning rules. For such situations, Rmony 1.1 supports learning rules using active learning. Figure 3.6 gives an overview of learning rules using active learning with the analyst.

The analyst supplies a seed training data S (which can be as small as four tuple pairs, two labeled "match", and two labeled "no-match"). Again she either manually creates features or automatically generates features, or does a mix of both. Rmony then converts each tuple pair in S into a feature vector, using the specified features. Next Rmony trains an initial model (e.g., a random forest F as shown in Figure 3.6), uses the trained model to find informative examples in S, asks the analyst to label them, then uses the labeled examples to improve the trained model, and so on.

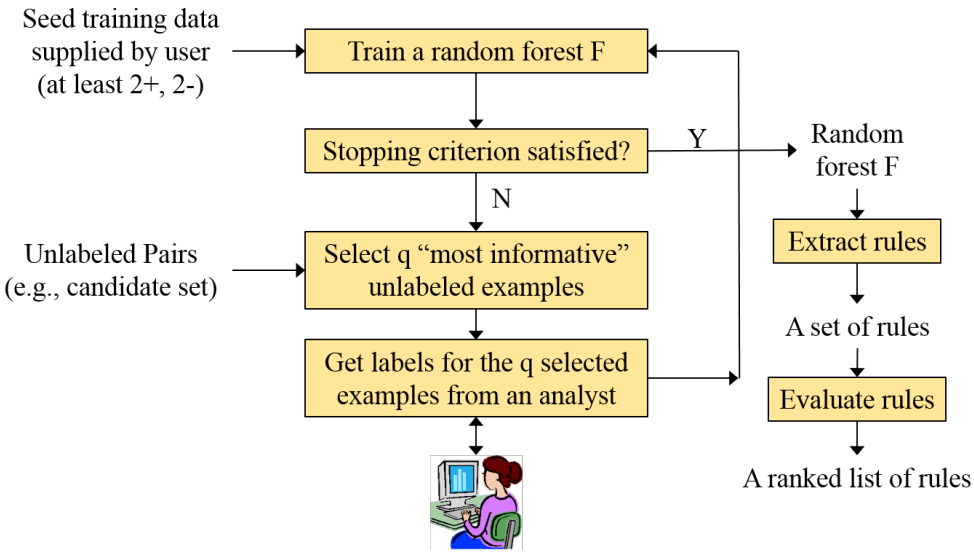


Figure 3.6: An overview of learning rules using active learning with the analyst.

Note that Rmony builds upon the similar ideas in Corleone except that instead of the crowd, here the analyst labels the tuple pairs. Another difference is that Rmony does not yet support the automatic stopping criteria in Corleone (see Chapter 2) but instead lets the analyst decide when to stop.

In order to decide when to stop, Rmony lets the analyst see and evaluate the rules extracted from the trained model after every iteration. For evaluating the rules, the analyst can either use the labeled data obtained in the course of active learning iterations, or a separately supplied test data. The former is a highly optimistic evaluation but sometimes is the only option when obtaining a separate test data is difficult.

3.4 Empirical Evaluation

We now empirically evaluate Rmony using real-world data and users. First, we evaluate Rmony 0.1 which helps an analyst match two tables by writing EM rules manually. Specifically, we ask UW-Madison students in a graduate class to apply Rmony 0.1 to match real-world data from diverse domains collected from the Web. The goal is to demonstrate that Rmony 0.1 can be used to match real-world data with high accuracy by writing EM rules manually. Second,

Team	Domain	Table Sizes (# of Tuples)	# of Rules	Precision	Recall
1	Restaurants	25K x 3.2K	4	1	0.98
2	Car Ads	25K x 5K	5	0.97	0.88
3	Electronics	21.5K x 3.6K	4	0.98	0.94
4	Movies	17K x 9K	5	1	0.91
5	Video Games	6.7K x 3.7K	2	1	0.89
6	Movies	5.5K x 4.3K	5	0.97	0.98
7	Books	5K x 6.5K	3	0.9	0.96
8	Breakfast Products	4K x 3.6K	4	1	0.79
9	Books	3K x 3.5K	2	0.97	0.99
10	Books	3.4K x 3.2K	2	1	0.82
11	Books	3.6K x 3.5K	5	0.96	0.93

Table 3.1: Evaluation of manual rule writing in Rmony on Web data.

we evaluate Rmony 1.1 to see if it can automatically suggest high-quality rules using (a) training data, and (b) active learning. We now elaborate on these evaluations.

3.4.1 Evaluation of Rmony 0.1

Using Rmony 0.1, a user can match two tables by writing EM rules manually. We released Rmony 0.1 to graduate students at UW-Madison in a Fall 2014 database class. These students were grouped into 11 teams, each comprising 2-3 members.

We asked each team to find two data-rich Web sites, extract and convert data from them into two relational tables, then apply Rmony 0.1 to match tuples across the tables. Table 3.1 summarizes the evaluation. Each row in the table shows a team, domain of data, sizes of the two tables to match, number of rules used in the final matcher, and quality of the matcher in terms of precision and recall. Note that two teams may cover the same domain, e.g., "Books", but extract from different sites.

Overall, there are 7 domains. The average precision and recall of EM rules written by students is 98% and 91% respectively. This suggests that students were able to write high quality EM rules using Rmony 0.1 to match real-world data of diverse domains.

Encouraging Feedback: We received a lot of encouraging feedback from the students. We quote some of them below:

- *"The system has the complete pipeline for matching and it is easy for us to find our way around."*
- *"The system is designed well in that projects are easily portable and also uses standard data formats."*
- *"It has good interface for browsing and viewing intermediate results."*
- *"Easy UI and very well descriptive debugging and refining rules interface, helped us a lot in improving precision and/or recall."*
- *"The tool helped us to identify data cleaning or attribute extraction problems."*
- *"The system provides useful statistics."*

Suggestions for Improvement: We also received many useful suggestions for improvement (some of which have been addressed in Rmony 1.1). We quote some of them below:

- *"The system may have a bit of learning curve for non-technical users. Specifically, it is not trivial to know which function is suitable for what attribute."* (Note that Rmony 1.1 supports automatic feature generation.)
- *"Add support for machine learning."* (Note that Rmony 1.1 automatically suggests rules to analysts using training data or active learning.)
- *"Make the system more robust to user errors and provide useful hints for them to resolve the issue."* (Rmony 1.1 handles exceptions better and is more robust to errors.)
- *"Enable simple data cleaning (e.g., applying transforms) inside the system."* (This has not been addressed yet.)

3.4.2 Evaluation of Rmony 1.1

Rmony 1.1 can automatically suggest rules to analysts using two methods: (a) training data, and (b) active learning. We want to validate the following two claims for each method:

Domain	Table A (#tuples, #attributes)	Table B (#tuples, #attributes)	Gold	#Features (auto-gen)	#Rules	Precision (%)	Recall (%)	F1 (%)
Books	Amazon (5111, 11)	Barnes & Noble (6429, 11)	600	52	102	[0-100] 62 rules have 100%	[0-72]	[0-82]
Video Games	Moby Games (6738, 9)	The Games DB (3742, 11)	1000	33	36	[0-100] 20 rules have 100%	[0-82]	[0-90]
Car Ads	Craigslist (4953, 23)	Kelly Blue Books (25412, 11)	591	42	30	[0-100] 12 rules have 100%	[0-86]	[0-89]
Breakfast Products	Amazon (3669, 8)	Walmart (4171, 9)	337	13	46	[0-100] 16 rules have 100%	[0-58]	[0-67]

Figure 3.7: Rmony automatically suggests high-quality rules using training data.

1. Rmony can automatically suggest high-quality rules when matching data of diverse domains, and

2. The suggested rules are comparable in quality to those manually written by the analysts.

To validate the claims we pick four data sets of varied domains, namely "Books" (Team 7), "Video Games", "Car Ads", and "Breakfast Products" from Table 3.1. For each of these four data sets, Rmony automatically suggests rules using training data and active learning as described below.

(a) Automatically Suggest Rules Using Training Data: For each of the four chosen data sets, the students have manually labeled a set of a few hundred tuple pairs. We use those sets of labeled pairs as our training data. As in Figure 3.5 we split the training data into 67% train and 33% test splits. We train a random forest of 10 trees on the train split encoded as feature vectors using features automatically generated by Rmony. We extract the positive rules from the random forest, evaluate these rules using the test split and retain the top rules in terms of precision and recall.

Figure 3.7 validates our first claim that Rmony can automatically suggest high-quality EM rules for data sets of varied domains. For example, for the "Books" data set, Rmony learns 102 rules. The students have manually labeled a set of 600 tuple pairs (termed as "Gold" in Figure 3.7) which we use as the training data to learn the rules. These rules have precision in the range [0-100%], recall in the range [0-72%], and F_1 in the range [0-82%]. 62 out of the 102 rules have a precision of 100%.

Domain	Manual Matcher {P, R, F1} (%)	Manual Matcher #Rules	{P, R, F1} % of rules in the manual matcher	#Learned Rules	Precision (%)	Recall (%)	F1 (%)	{P, R, F1} (%) of 3 good rules
Books	90, 96, 93	3	{100, 33, 50}, {93, 95, 94}, {92, 69, 79}	144	[17-100] 106 rules have 100%	[1-74]	[1-85]	{100, 33, 49}, {99, 75, 85}, {97, 65, 78}
Video Games	100, 89, 94	2	{100, 79, 88}, {100, 58, 73}	54	[50-100] 32 rules have 100%	[1-83]	[3-90]	{100, 71, 83}, {100, 62, 76}, {98, 83, 90}
Car Ads	97, 88 , 92	5	{97, 73, 82}, {96, 66, 78} {96, 61, 75}, {96, 59, 73}, {96, 54, 69}	67	[25-100] 40 rules have 100%	[2-78]	[4-86]	{100, 68, 81}, {97, 78, 86}, {97, 73, 81}
Breakfast Products*	100, 79, 88	1	100, 79, 88	95	[20-100] 52 rules have 100%	[3-58]	[5-70]	{100, 21, 35}, {95, 47, 63}, {88, 58, 70}

* This dataset is unclean; students obtained good results by folding data cleaning into their custom features.

Figure 3.8: The automatically suggested rules using training data by Rmony are comparable to the manually written rules.

Figure 3.8 validates our second claim that the automatically suggested rules using training data are comparable in quality with the rules manually written by the students. For example, for the "Books" data set, the students have manually created a matcher of 3 rules (column 4). These rules look comparable in quality with the top-3 F_1 rules suggested by Rmony (last column).

(b) Automatically Suggest Rules Using Active Learning: We choose the same four data sets as before. For each data set, from the labeled data that the students have manually created, we choose four tuple pairs, two labeled "match" and two labeled "no-match", to seed the active learning. Again we use the automatically generated features and train a random forest of 10 trees. We run 10 rounds of active learning, where an analyst labels 10 pairs in each round (i.e., a total of 100 labeled pairs). We extract the positive rules from the trained random forest and evaluate the rules using the entire labeled data manually created by the students.

Figure 3.9 validates our first claim that Rmony is able to suggest high-quality EM rules using active learning for data sets of varied domains. For example, for the "Books" data set, Rmony learns 53 rules. These rules have precision in the range [4-100%], recall in the range [1-91%], and F_1 in the range [1-93%]. 3 out of the 53 rules have a precision of 100%.

Domain	Table A (#tuples)	Table B (#tuples)	Candidate Set (#tuples)	Gold	#Features (auto-gen)	#Rules	Precision (%)	Recall (%)	F1 (%)
Books	Amazon (5111)	Barnes&Noble (6429)	3,327	600	52	53	[4-100] 3 rules have 100%	[1-91]	[1-93]
Video Games	MobyGames (6738)	TheGamesDB (3742)	33,466	1000	33	15	[4-100] 4 rules have 100%	[1-97]	[2-82]
Car Ads	Craigslist (4953)	Kelly Blue Books (25412)	863,207	591	42	13	[3-100] 3 rules have 100%	[1-83]	[2-88]
Breakfast Products	Amazon (3669)	Walmart (4171)	20,146	337	13	15	[2-100] 3 rules have 100%	[1-67]	[1-67]

Figure 3.9: Rmony automatically suggests high-quality rules using active learning.

Domain	Manual Matcher {P, R, F1} (%)	Manual Matcher #Rules	{P, R, F1} (%) of rules in the manual matcher	#Learned Rules	Precision (%)	Rec. (%)	F1 (%)	{P, R, F1} (%) of 3 good rules
Books	90, 96, 93	3	{100, 33, 50}, {93, 95, 94}, {92, 69, 79}	53	[4-100] 4 rules have 100%	[1-91]	[1-93]	{100, 6, 10}, {97, 77, 86} {96, 91, 93}
Video Games	100, 89, 94	2	{100, 79, 88}, {100, 58, 73}	15	[4-100] 4 rules have 100%	[1-97]	[2-82]	{100, 62, 76}, {97, 57, 72}, {95, 72, 82}
Car Ads	97, 88 , 92	5	{97, 71, 82}, {96, 66, 78}, {96, 61, 75}, {96, 59, 73}, {96, 54, 69}	13	[3-100] 3 rules have 100%	[1-83]	[2-88]	{100, 50, 67} {94, 83, 88} {92, 82, 87}
Breakfast Products*	100, 79, 88	1	100, 79, 88	15	[2-100] 3 rules have 100%	[1-67]	[1-67]	{100, 26, 42}, {87, 34, 49}, {67, 67, 67}

* This dataset is unclean; students obtained good results by folding data cleaning into their custom features.

Figure 3.10: The automatically suggested rules using active learning by Rmony are comparable to the manually written rules.

Figure 3.10 validates our second claim that the automatically suggested rules are comparable in quality with the rules manually written by the students. For example, for the "Books" data set, the students have manually created a matcher of 3 rules (column 4). These rules look comparable in quality with the top-3 F_1 rules suggested by Rmony (last column).

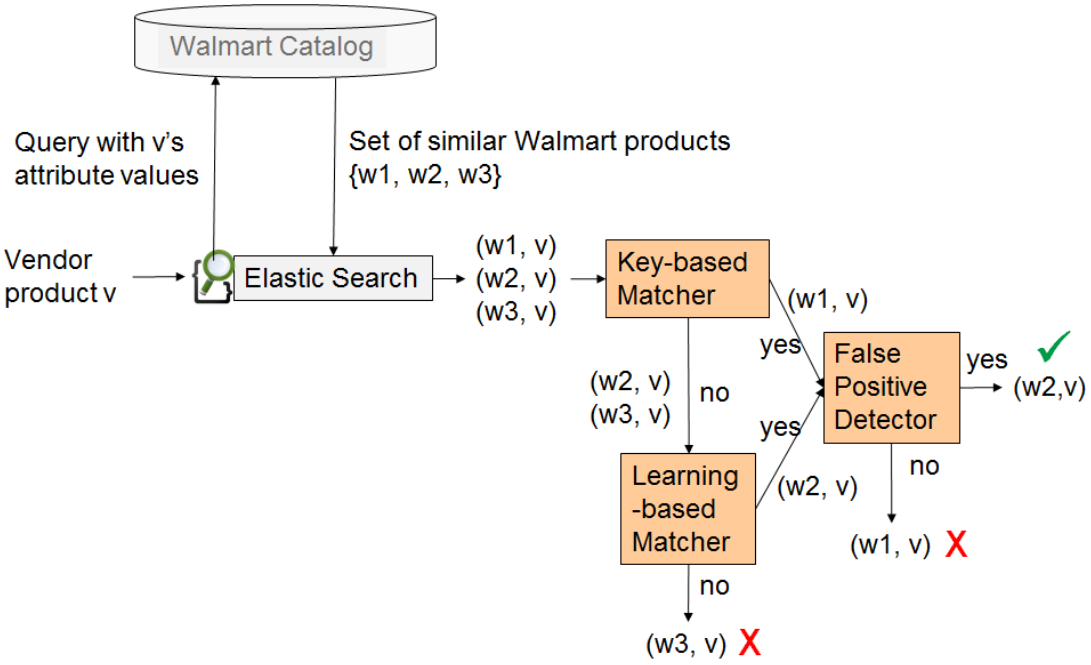


Figure 3.11: A highly simplified high-level architecture of product matching system at Walmart.

3.5 Product Matching at Walmart: A Case Study

Walmart has recently added a "marketplace" feature to its e-commerce business. In a marketplace, Walmart sells items from smaller vendors (e.g., Tech for Less Inc., UnbeatableSale, Circuit City) alongside products from its own catalog on the shopping Web site. Different vendors may sell the same real-world product in the marketplace, so it is critical to avoid duplication of the matching products.

3.5.1 Existing Solution at Walmart

Walmart has deployed multiple EM systems for product matching and other purposes. Figure 3.11 presents a highly simplified high-level architecture of one of their EM solutions.

An incoming vendor product v is queried against Walmart's catalog of products using an Elastic Search interface. This interface returns a set of similar Walmart products (say $w1$, $w2$, and $w3$).

The similar Walmart products are then paired with the vendor product to form the candidate set of product pairs (e.g., ($w1$, v), ($w2$, v), and ($w3$, v)). Think of this step as blocking.

The candidate set of tuple pairs are then passed through a key-based matcher (e.g., a matcher that looks for equivalence in key attributes such as UPC, ISBN, GTIN, etc.). If an exact equivalence is found for the key attribute, then the product pair is most likely a "match", and hence it is passed to a "false positive detector" module which we will discuss shortly. If an exact equivalence is not found, then the product pair is passed to a more sophisticated learning-based matcher. If the learning-based matcher decides that a pair is "no-match", then it is immediately labeled as as "no-match" and is not carried forward in the EM pipeline. On the other hand, if the learning-based matcher decides that a pair is a "match", then it is passed to the "false positive detector" module.

False positives are highly undesirable in product matching. The final module in the EM pipeline, "false positive detector", tries to detect and eliminate false positives. All pairs that have reached this module are highly likely to be matching pairs (as these were passed by some previous module in the pipeline). This module consists of many rules that encode domain knowledge, business requirements, etc. and is designed to "kill off" the "false positives" (pairs that are not actually "matching" but have reached this module). Finally the pairs that survive this module are labeled "match".

3.5.2 Applying Rmony to Walmart Data

Goal: In the summer of 2015, I interned at WalmartLabs. The goal was to match Walmart products with third-party vendor products using Rmony 1.1 and thereby verify if Rmony could be used to perform rule-based EM in real-world settings. To scope the problem, I considered matching products only in the "Electronics" domain.

Collecting Labeled Data: Walmart's analysts have prepared a large set of labeled product pairs for training purposes. I sampled a set of 10K product pairs, with 6K pairs labeled "match" and 4K labeled "non-match", as my training data. I sampled another set of 10K product pairs, with 6K pairs labeled "match" and 4K labeled "no-match", as my test data.

Error Category	Error Description	# of Occurrences	Proposed Solution
P1	Incorrect label in Test data	5/59	Correct the labels in the Test data
P2	Manufacturing Part Number (MPN) highly similar, but differing only slightly	22/59	Create only exact match feature (and no fuzzy match features) for MPN
P3	Relevant attribute value (e.g., color) has not been extracted from text (e.g., product name)	27/59	Custom features to do "quick and dirty" extraction
P4	Errors not belonging to any specific category	5/59	None

Figure 3.12: Analysis of precision errors.

Attribute	Walmart Product	Vendor Product
Product Name	Blue Basic Mouse Pad (Pack of 3)	Blue Basic Mouse Pad (Pack of 3)
Product Short Description	Blue Basic Mouse Pad (Pack of 3)	
Product Long Description	Allsop 28228 Basic Mouse Pad (Blue)	Allsop 28228 Basic Mouse Pad (Blue)
Product Segment	Electronics	Electronics
Product Type	Mouse Pads	Mouse Pads
Manufacturer Part Number	921483	

Figure 3.13: An example of P1 error category.

Training an Initial Matcher: Next I used Rmony to automatically generate 68 features for this data. Then I trained a random forest of 10 trees on 80% of training data (encoded as feature vectors using the automatically generated features). Then I extracted the positive rules from the trained random forest and evaluated them on the remaining 20% of the training data. Finally, I constructed a matcher comprising the top-10 F_1 rules. This matcher obtained a precision of 98.61% and a recall of 69.58% on the test data.

Debugging the Errors: Next I debugged the data, features, and matcher to improve the matching accuracy (i.e, precision and recall). Specifically, I took the following three steps. First, I manually analyzed a sample of precision and recall errors (i.e, false positives and negatives respectively) to understand the errors. Second, I categorized these errors. Third, I took actions to fix the common categories of errors. I describe my debugging efforts below.

The initial matcher M1 made 59 precision errors and 1824 recall errors on the test data. Product matching is an EM setting that requires a very high precision (often close to 100%). Any precision error is highly undesirable. So I decided to first analyze the precision errors and then analyze the recall errors.

Attribute	Walmart Product	Vendor Product
Product Name	rooCASE Samsung Galaxy Tab Pro 12.2 / Note Pro 12.2 Origami Slim Shell Case	rooCASE Samsung Galaxy Tab Pro 12.2 / Note Pro 12.2 Origami Slim Shell Case
Product Short Description	rooCASE Samsung Galaxy Tab Pro 12.2 / Note Pro 12.2 Origami Slim Shell Case The rooCASE Slim Shell Case for …	
Product Long Description	The rooCASE Slim Shell Case for Samsung Galaxy Note Pro / Tab Pro 12.2" boasts a slim silhouette, and it slides easily in and out of …	The rooCASE Slim Shell Case for Samsung Galaxy Note Pro / Tab Pro 12.2" boasts a slim silhouette, and it slides easily in and out of …
Product Segment	Electronics	Electronics
Product Type	Electronics Carrying Cases	Electronics Carrying Cases
Brand	rooCASE	
Color	Blue	
Manufacturer Part Number	RC-GALX12.2-PRO-OG-SS-BL	RC-GALX12.2-PRO-OG-SS-BK

Figure 3.14: An example of P2 error category.

I manually analyzed all the 59 precision errors and categorized them into 4 categories P1-P4 as shown in Figure 3.12. P1 errors, constituting about 8.5% of all the precision errors, were those where the pairs were incorrectly labeled as "no-match". Figure 3.13 shows an example of this category. P2 errors, constituting about 37.3% of the precision errors, were those where the "Manufacturer Part Number" attribute values in the products were highly similar but not exactly the same. Figure 3.14 shows one example of this category. P3 errors, constituting about 45.8% of the precision errors, were those where a relevant attribute value had not been extracted from text. Figure 3.15 shows an example of this category where a relevant attribute, "Color", was missing from the product details but the color values were present in the text of "Product Name" attribute and have not been extracted. Finally, there were P4 errors, constituting about 8.5% of the precision errors, for which I was not able to assign any particular reason. Figure 3.16 shows an example of this category.

I started by resolving P1 errors. I manually corrected the actual labels (from "no-match" to "match") for the 5 erroneous tuple pairs in test data. Ideally one should correct all the incorrect labels in the train and test data but detecting the incorrect labels automatically is difficult.

Next I tried to resolve P2 errors. A high similarity in the values of Manufacturer Part Number (MPN) attribute is not enough for a tuple pair to be a "match" (e.g., as in Figure 3.14). In fact, slight differences in values of MPN capture subtle differences between two products. For example,

Attribute	Walmart Product	Vendor Product
Product Name	rooCASE Samsung Galaxy Tab Pro 12.2 / Note Pro 12.2 Origami Slim Shell Case	rooCASE Samsung Galaxy Tab Pro 12.2 / Note Pro 12.2 Origami Slim Shell Case
Product Short Description	rooCASE Samsung Galaxy Tab Pro 12.2 / Note Pro 12.2 Origami Slim Shell Case The rooCASE Slim Shell Case for …	
Product Long Description	The rooCASE Slim Shell Case for Samsung Galaxy Note Pro / Tab Pro 12.2" boasts a slim silhouette, and it slides easily in and out of …	The rooCASE Slim Shell Case for Samsung Galaxy Note Pro / Tab Pro 12.2" boasts a slim silhouette, and it slides easily in and out of …
Product Segment	Electronics	Electronics
Product Type	Electronics Carrying Cases	Electronics Carrying Cases
Brand	rooCASE	
Color	Blue	
Manufacturer Part Number	RC-GALX12.2-PRO-OG-SS-BL	RC-GALX12.2-PRO-OG-SS-BK

Figure 3.15: An example of P3 error category.

in Figure 3.14, "BL" and "BK" in the MPN values of the two products actually stand for the colors "Blue" and "Black" respectively, thereby indicating that the two products (which otherwise have almost identical attribute values) differ only in color. To address this category of errors, I created only the "ExactMatch" feature on MPN and discarded all the "fuzzy" similarity features (e.g., Levenshtein) on MPN. This is because all the fuzzy features will fail to capture the slight differences in the MPN values. I retrained a matcher with the new set of features. This matcher M2 obtained 99.27% precision and 90.05% recall. Precision and recall errors go down significantly to 40 and 597 respectively. It is worth noting that the change improved recall significantly (even though it was targeted towards precision). This may be due to the fact that with "fuzzy" similarity features, the matcher was trying to learn complex patterns on MPN. But when trained with a relevant "exact" match feature on MPN, a more accurate matcher was learned, resulting in improvements in both precision and recall.

Next I tried to resolve P3 errors (e.g., Figure 3.15). A long-term solution for these errors is to extract the relevant attribute values (e.g., color) from text (e.g., product name). For example, for the example shown in Figure 3.15, color ("Black" for Walmart product and "White" for vendor product) must be extracted. Ideally, this full-fledged information extraction should precede EM to obtain better matching accuracy. As a short-term solution, I manually created a custom feature that can do "quick and dirty extraction" in some cases. Specifically, I created a custom feature

Attribute	Walmart Product	Vendor Product
Product Name	Airbac Journey	AIRBAC Mens Blue Skater Lightweight School Sports Travel Laptop Backpack Bag SKR
Product Short Description	Airbac Journey Has one large pocket with smaller interior pockets within for small extras plus an iPad sleeve. Has one large ...	
Product Long Description	Has one large pocket with smaller interior pockets within for small extras plus an …	Ideal solution to a heavy backpack Scientifically proven technology to alleviate pressure and …
Product Segment	Travel, Luggage & Accessories	Electronics
Product Type	Backpacks	Laptop Bags & Cases
Brand	Airbac	Airbac
Color	Blue	
Actual Color	Blue	
Assembled Product Length	8.0	
Assembled Product Width	12.0	
Assembled Product Height	19.0	
UPC	857527002181	0085752700281
Manufacturer Part Number	JNY-BL	

Figure 3.16: An example of P4 error category.

PN_Colors_Jaccard(walmart.pn, vendor.pn), which extracts a set of color words, $S1$ from the Walmart product's "Product Name" attribute and a set of color words, $S2$, from the vendor product's "Product Name" attribute. It then computes a Jaccard score on the two sets $S1$ and $S2$. For the example pair in Figure 3.15, $S1 = \{\text{“}Black\text{”}\}$ and $S2 = \{\text{“}White\text{”}\}$ and the feature score will be 0. To extract the color words from "Product Name", I used a dictionary-based approach where I constructed a dictionary of all the color values from all the Walmart and vendor products. Next, I retrained a matcher with this additional custom feature. This matcher M3 obtained 99.26% precision and 93.87% recall. The number of precision errors went up marginally to 42, but the recall errors went down significantly from 597 to 368. Again this is a case where the change was directed towards the improvement of precision but we observed an improvement in recall. This was due to the fact that this change resolved some of the false positives that I had manually observed but introduced a few other false positives (e.g., "black printers with blue cartridges" were matched with "blue printers with black cartridges") thereby keeping precision nearly the same. Recall improved because pairs that were previously not matched but had color attribute values hidden in the "Product Name" attribute values were now matched due to this additional feature.

Error Category	Error Description	# of Occurrences	Proposed Solution
R1	Relevant attribute value (e.g., MPN) is present in text (e.g., product name)	18/50	Create custom features to do "quick and dirty extraction"
R2	UPC formats are different in Walmart and vendor products (e.g., 12-digit v/s 13-digit)	12/50	Normalize UPC values (i.e., change UPC values to the same format)
R3	MPN values must be cleaned (e.g., hyphens and leading zeroes must be removed)	27/59	Clean the MPN values before matching
R4	Most of the attribute values are missing from the product details	8/50	None
R5	Errors not belonging to any specific category	4/50	None

Figure 3.17: Analysis of recall errors.

Next I wanted to analyze the recall errors. Since manually analyzing 368 recall errors would be time consuming, I took a random sample of 50 recall errors. I analyzed them manually and categorized them into 5 categories R1-R5 as shown in Figure 3.17. R1 errors, constituting 36% of the sample recall errors, were those pairs where a relevant attribute value had not been extracted from text. Figure 3.18 shows an example of this category. R2 errors, constituting 24% of the sample recall errors, were due to differing formats in the UPCs of the Walmart and vendor products. The Walmart products had 12-digit UPCs whereas the vendor products had 13-digit UPCs. R3 errors, constituting 16% of the sample recall errors, were those pairs where the MPN values would match but after cleaning (e.g., removing hyphens and leading zeroes). R4 errors, constituting 8% of the sample recall errors, were pairs where a lot of relevant attribute values were missing from the product details. Figure 3.19 shows an example. For these pairs, enough information was not present in the product details to guide the matcher. Finally there were R5 errors, constituting 16% of the sample recall errors, for which I was not able to assign any specific reason. Figure 3.20 shows an example of this category.

Next I tried to resolve R1 errors. In Figure 3.18, observe that the MPN value of the vendor product is missing but is present in the text of "Product Name" attribute value. Had it been extracted it would have matched exactly with the MPN value of the Walmart product. So I manually created two custom features to do "quick and dirty extraction" before matching. The first feature, Walmart_MPN_in_vendor_PN, probes Walmart product's MPN attribute value in vendor

Attribute	Walmart Product	Vendor Product
Product Name	Intel Storage Controller Battery	Raid Smart Battery AXXRSBBU9
Product Short Description	1500mAh Lithium ion (Li-ion) 3.7V DC	
Product Long Description	Intel Storage Controller Battery: • 1500 mAh • Lithium ion (Li-ion) • 3.7V DC	Raid Smart Battery AXXRSBBU9
Product Segment	Electronics	Electronics
Product Type	General Purpose Batteries	General Purpose Battery Chargers
Brand	Intel	
Manufacturer	Intel Corp.	
Assembled Product Length	10.9	
Assembled Product Width	9.6	
Assembled Product Height	3.1	
UPC	735858220286	0073585822028
Manufacturer Part Number	AXXRSBBU9	

Figure 3.18: An example of R1 error category.

product's "Product Name" attribute value and returns 1 if found and 0 if not found. For example, this feature will return a score of 1 for the example pair shown in Figure 3.18. The second feature, Walmart_MPN_in_vendor_PLD, probes Walmart product's MPN attirbute value in vendor product's "Product Long Description" attribute value, and returns 1 if found and 0 if not found. For example, this feature will return a score of 1 for the example pair shown in Figure 3.18. With these additional custom features, I retrained a matcher. This matcher M4 obtained 99.14% precision and 94.47% recall. The number of recall errors decrease from 368 to 332 but the precision errors increase 42 to 49.

Next I tried to resolve R2 errors. I observed that many Walmart products had 12-digit UPC values whereas the vendor products had 13-digit UPC values. For example, a Walmart product had an UPC value of "085854222624", a matching vendor product had a UPC value of "0008585422262" but our matcher was unable to predict this pair as a "match". Had normalization to 11-digits been done, both the products would have the identical UPC value of "08585422262". So to address the R2 errors, I normalized the UPC values (to 11 digits) for all the products in the train and test data

Attribute	Walmart Product	Vendor Product
Product Name	Oki 43487734 Toner OKI43487734	Oki Data Americas Inc Toner Cartridge 6000 Page Yield Magenta
Product Short Description	Prints an amazing array of shades and solids, creating life-like images. Creates fine lines ...	
Product Long Description	Prints an amazing array of shades and solids, creating life-like images. Creates fine lines ...	Toner cartridge is designed for use with Oki C8800. Yields 6000 pages …
Product Segment	Electronics	Electronics
Product Type	Printer Cartridges	Printer Cartridges
Brand	OKI	OKI Data
Manufacturer	Oki	
Color	Multicolor	
Actual Color	Multicolor	
Assembled Product Length	16.0	
Assembled Product Width	4.8	
Assembled Product Height	2.6	
UPC		0005185135441
Manufacturer Part Number	43487734	

Figure 3.19: An example of R4 error category.

sets. I retrained a matcher with the same features but on the normalized data. This matcher M5 obtained 99.65% precision and 95.87% recall. The number of precision errors went down from 49 to 20 and the number of recall errors went down from 332 to 303, thereby proving that the normalization of UPC values helped improve the overall accuracy (i.e., both precision and recall) of the matcher.

Next I tried to resolve R3 errors. I observed that in many of the recall errors, the MPN values of the Walmart and vendor products were almost matching except for punctuations (e.g., hyphens) and leading zeroes. For example, one recall error had “ENST-213Pink” and “ENST-213-Pink” as MPN values in the Walmart and vendor products respectively. Another recall error had “08873” and “8873” as MPN values in the Walmart and vendor products respectively. Note that since we were using “exact” match as the only feature on MPN, these subtle differences would not be captured by this feature. So to address R3 errors, I wrote a custom feature that would first clean MPN values of punctuations and leading zeroes, then perform an “exact” match. I retrained a

Attribute	Walmart Product	Vendor Product
Product Name	CHAMP Bluetooth Survival Solar Multi-Function Skybox with Emergency AM/FM NOAA Weather Radio (RCEP600WR)	CHAMP Bluetooth Survival Solar Multi-Function Skybox with Emergency AM/FM NOAA Weather Radio (RCEP600WR)
Product Short Description	BLTH SURVIVAL SKYBOX W WR	
Product Long Description	BLTH SURVIVAL SKYBOX W WR	BLTH SURVIVAL SKYBOX W WR
Product Segment	Electronics	Electronics
Product Type	CB Radios & Scanners	Portable Radios
Brand	CHAMP	
Color	Black	
Actual Color	Black	
UPC		0004447611732

Figure 3.20: An example of R5 error category.

matcher using this new feature. This matcher M6 obtained a precision of 99.55% and a recall of 95.87%. Precision errors increased from 20 to 26 and recall errors went down form 303 to 248.

On analyzing the 6 additional false positives introduced by M6, I realized that my naïve cleaning of MPN values can potentially introduce many false positives as the position of punctuations (e.g., hyphens) matters. For example, “A65-06” and “A6-506” are MPN values of two very different products. So I discarded the new feature and reverted to matcher M5.

As a last debugging step, I wrote a manual rule to improve the recall. From my manual inspection of a few hundreds of pairs, I had observed that if both UPC and MPN values are identical for a product pair, then that pair is a “match”. I added a manual rule (encoding this domain knowledge) to matcher M5 to obtain a new matcher M7. M7 had the same precision of 99.65% as M5 (thereby showing that my rule did not introduce any false positives) but an improved recall of 95.30%. The recall errors went down from 303 to 282.

M7 was my final matcher. Figure 3.21 summarizes my debugging efforts, i.e., how I iteratively improved the data (e.g., by fixing incorrect labels), features, and matcher.

Final Results: My final matcher M7 obtained a precision of 99.65% and a recall of 95.30% on test data. Since I performed debugging by looking at the test data, I collected another set of 28K labeled pairs (as my final test data) and applied M7 on it. M7 obtained a precision of 98.45% and a recall of 94.03% thereby suggesting that M7 achieved high matching accuracy.

Matcher	Description	Precision (%)	Recall (%)	False Positives	False Negatives
M1	Initial matcher	98.61	69.58	59	1824
M1	Addressed P1: corrected labels in Test data	98.72	69.61	54	1824
M2	Addressed P2: created only exact match feature for MPN	99.27	90.05	40	597
M3	Addressed P3: added a custom feature PN_Colors_JAC	99.26	93.87	42	368
M4	Addressed R1: added features to do "quick and dirty extraction" from text	99.14	94.47	49	332
M5	Addressed R2: normalized UPC	99.65	94.95	20	303
M6	Addressed R3: cleaned MPN	99.55	95.87	26	248
M7	Added a rule from domain knowledge to matcher M5	99.65	95.30	20	282

Figure 3.21: Iteratively improving the data, features, and matcher.

Revising Walmart's Existing Solution: The precision/recall of my final matcher on the newly created test data of 28K pairs was significantly higher than Walmart's solution on the same test data. (For confidentiality purposes I cannot disclose the exact numbers.) As a result, some of my findings (e.g., cleaning the data, fixing the labels, and adding custom features) were incorporated into their existing solution. This significantly improved the system in production: increasing recall by 34 percentage points while reducing precision only slightly by 0.75 percentage points.

3.6 Lessons Learned and the Inception of Magellan

During the course of the development and evaluation of Rmony we learned the following main lessons:

1. Proxy debugging: The case study of product matching at Walmart using Rmony suggested a new debugging method. To debug a complex system, we can debug a simpler one and then transfer the findings where appropriate. For example, to debug Walmart's complex black-box learning-based solution, I first debugged my rule-based matcher in Rmony and then transferred the findings. This "proxy debugging" is promising (as demonstrated in Section 3.5) as it can uncover problems in data, labels, and features. Of course, it cannot help with errors specific to the complex system. But once "low hanging" errors have been debugged, we can better focus on errors specific to the complex system.

2. A declarative, operator-driven end-to-end EM system may not be a good idea: Most current works on EM systems (e.g., Ajax [49]) have tried to build a declarative framework. These works first identify a small set of core operators. All analyst actions are then compiled into a workflow of these operators. The workflow is then optimized and executed.

We tried this in Rmony but found it hard to come up with a small set of core operators that can capture all user actions. For example, there are so many debugging actions that the analysts may want to do, it is not clear how to represent all of them using a a small set of core operators. Also, some operators are human-centric such as exploring a data set. It is not clear how to represent them in a workflow. As a result we found it very hard to extend the conceptual model of Rmony to incorporate debugging and exploration activities.

3. Need a different approach to building EM management systems: Our work suggested a different approach to building EM management systems. We should first distinguish between the development and production stages. For the development stage, the goal is to come up with a good workflow (e.g., accurate) that can then be executed in the production stage. Finding a good workflow is akin to data mining or data analysis. There is no clear small set of core operators. Instead people use a rich range of commands in a data analysis stack, such as the Python data analysis stack.

So we propose that we develop the development stage of an EM system on top of the Python data analysis stack, by developing as many commands as necessary. The production stage can probably benefit from compiling and optimizing the workflow, as done in prior work.

4. Need a methodology for the analysts: Our work suggested that we need a methodology for analysts to use an EM system. There are many questions that remain unanswered. What actions should an analyst perform? In what order? How should she debug? Our current debugging solution is iterative: analyze precision and recall errors, then modify data, labels, and matcher accordingly. Is this a good approach? We need to explore these questions systematically.

Magellan: Some of these lessons have led to the inception of Magellan [69], an open-source EM management system situated within the PyData ecosystem. Magellan adopts a different approach to building EM management systems. This approach is novel in several important aspects.

First, Magellan provides a methodology to users in the form of how-to guides which guide users through the EM workflow, step by step. Second, it provides automated tools to address the "pain points" of the steps, and these tools seek to cover the entire EM workflow.

Finally, the tools are built on top of the Python data analysis and Big Data stacks. Magellan proposes that users solve an EM scenario in two stages. In the development stage users find an accurate EM workflow using data samples. In the production stage users execute this workflow on the entirety of data. The development stage is akin to data analysis. So Magellan builds tools for this stage on top of Python data analysis stack, which includes packages such as pandas, scikit-learn, matplotlib, etc. The production stage on the other hand focuses on scalability. So Magellan develops tools for this stage on top of the Python Big Data stack, which includes packages such as PyDoop, PySpark, mrjob, etc.

3.7 Related Work

Many EM systems are available both commercially and non-commercially (see Chapter 10 in [27] for a detailed overview and [70] for a comparison).

Commercial systems include industry leaders (e.g., IBM InfoSphere [3], Informatica Data Quality [4], SAS Data Quality [8] and Oracle Enterprise Data Quality [6]) and recent players (e.g., Tamr [10]). One limitation of most of these systems is that they support EM only as a small part of a bigger data cleaning/integration pipeline. As a result, the support for EM is often limited.

Non-commercial systems include D-Dupe [21], Dedoop [67], Dedupe, Dude [40], Febrl [26], NADEEF, OYSTER, pydedupe, RecordLinkage, SERF [18], Silk, TAILOR [43], etc. (See [27] for more.)

Rmony is distinguished in three aspects. First, it has extensive support for rules. Second, it is designed for the "non-technical" user (e.g., data analyst). Third, it seeks to cover critical but

often-ignored steps (e.g., debugging) of the EM pipeline. So next we compare EM systems with Rmony with respect to the above three aspects:

- Support for rules: Most commercial systems and many non-commercial systems (e.g., DuDe, NADEEF, OYSTER, pydedupe, Silk) support rules. Most of these systems support manual rule creation. However very few systems (e.g., Tamr) support learning rules using training data, and almost none (except NADEEF) support learning rules using active learning.

- Support for "non-technical" users: Most commercial systems and some non-commercial systems (e.g., Dedoop, Febrl, NADEEF, TAILOR) have rich standalone or Web-based GUI with excellent visualization capabilities. They are still very difficult to use for a "non-technical" user because they do not automate the technically hard steps. For example, none of the systems supports automatic feature generation. Commercial systems however sell consulting services (sometimes called "data stewarding") to help "non-technical" users.

- Support for debugging: There is very limited or no support for debugging in these systems. Support for debugging (if present) is limited to showing which EM rules fire on a given tuple pair. Rmony on the other hand has a much richer set of debugging capabilities.

Many declarative data cleaning systems (e.g., Ajax [49], HIL [60], Dedupalog [14], Potter's Wheel [89], BigDansing [64], and Wisteria [56]) have also been proposed. These systems differ from Rmony in at least three aspects. First, they define operators for EM at a very coarse granularity. For example, feature vector generation is not modeled as an operator in these systems. As another example, Ajax uses just a single operator "Match", and Wisteria uses two operators "SimilarityJoin" (for blocking) and "Filtering" (for matching) to model the EM process. Second, the data cleaning workflows in these systems must be manually specified by the user using their domain-specific languages; there is little provision to learn the workflows. Third, these systems are designed to optimize and execute already known workflows (e.g., in production stage), but not for discovering workflows (e.g., in development stage). Rmony on the other hand is seeks to help the user discover an accurate EM workflow in the development stage.

Prior work has also addressed debugging learning-based models. They help users build, inspect and visualize specific models (e.g., decision trees [12], Naïve Bayes [16], SVM [23], ensemble models [99]). But, unlike Rmony, they do not allow users to examine errors and inspect raw data. Another related work [31] queries a complex model (e.g., a trained neural network) to obtain input-output pairs which serve as training examples to build a decision tree. This decision tree is an approximate but more interpretable representation of the complex model. We believe that this work is complementary to ours. In the future, when we extend Rmony to train learning-based matchers (e.g., SVM, neural networks), we can potentially use this work to approximate the matcher by a decision tree, and then extract rules from the tree (using the current techniques).

3.8 Conclusion

In this chapter, we have presented Rmony, a rule-based entity matching management system for the analysts. Using Rmony an analyst can either manually write rules or Rmony can automatically suggest rules using training data or active learning. We present evaluation of Rmony with students at UW-Madison and analysts at Walmart to show that Rmony is effective. Finally, we discuss the important lessons learned during the course of development of Rmony. Some of these lessons have led to the inception of Magellan, a novel EM management system in the Python ecosystem.

Rmony has opened up many interesting future research directions, some of which are being pursued in Magellan, but a lot more needs to be done. These include extending the system with more capabilities (e.g., crowdsourcing, cloud-based service, etc.), handling more EM scenarios (e.g., linking a table into a knowledge base), handling more EM workflows (e.g., information extraction and data cleaning before performing EM), and providing more methodologies (e.g., how-to guides) for various critical steps (e.g., blocking, sampling, debugging) to users. Finally, we hope that Rmony (and now Magellan) will motivate the research community to devote far more efforts toward system building in the future.

Chapter 4

Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching

4.1 Introduction

Corleone demonstrates that hands-off crowdsourcing for EM is promising for many EM settings, e.g., EM as a service on the cloud. But it suffers from a major limitation: it does not scale to large tables, as the following example illustrates.

Example 4.1.1. *We are currently seeking to provide EM services to hundreds of domain scientists. Such users often do not know how to, or are reluctant to, deploy EM systems locally (such systems often require a Hadoop cluster, as we will see). So we want to provide such EM services on the cloud, supported in the backend by a cluster of machines.*

During any week, we may have tens of submitted EM tasks running. Many of these tasks require blocking, but the users do not know how to write blocking rules (which often involve string similarity functions, e.g., edit distance, Jaccard, TF/IDF), and we simply cannot afford to ask our two busy developers to assist the users in all of these tasks.

Thus, we planned to deploy the hands-off solution of Corleone. *A user can just submit the two tables to be matched on a Web page and specify the crowdsourcing budget. We will run* Corleone *internally, which uses the crowd to match. (In fact, if users do not want to engage the crowd, they can label the tuple pairs themselves. Most users we have talked to, however, prefer if possible to just pay a few hundreds crowdsourcing dollars to obtain the result in 1-2 days.)*

As described, Corleone *seems perfect for our situation. Unfortunately, it executes a single-machine in-memory EM workflow, and does not scale at all to tables of moderate and large sizes. Our users often need to match tables of 50-200K tuples each (and some have tables of millions*

of tuples), e.g., an applied economist studying non-profit organizations in the US must match two lists of hundreds of thousands of organizations. For such tables, Corleone *took weeks, a simply unacceptable time (and use of machine resources).*

The above example shows that Corleone is highly promising for certain EM situations, e.g., EM as a service on the cloud, but that it is critical to scale Corleone up to large tables, to make such cloud-based services a reality.

To address this problem, in this chapter we introduce Falcon (fast large-table Corleone), a solution that scales up Corleone to tables of millions of tuples.

We begin by identifying three reasons for Corleone's being slow. First, it often performs too many crowdsourcing iterations without a noticeable accuracy improvement, resulting in large crowd time and cost. Second, many of its machine activities take too long. In particular, in the blocking step Corleone simply applies the blocking rules to *all* tuple pairs in the Cartesian product of the two input tables A and B. This is clearly unacceptable for large tables. Finally, when Corleone performs crowdsourcing, the machines sit idly, a waste of resources. If we can "mask the machine time" by scheduling as many machine activities as possible during crowdsourcing, we may be able to significantly reduce the total runtime.

We then describe how Falcon addresses the above problems. It is difficult to address all three simultaneously. So Falcon provides a relatively simple solution to cap the crowdsourcing time and cost to an acceptable level (for now), then focuses on minimizing and masking machine time.

Challenges: Realizing the above goals raised three challenges. First, we do not want to scale up a monolithic stand-alone EM workflow. Rather, we want a solution that is modular and extensible so that we can focus on scaling up pieces of it, and can easily extend it later to more complex EM workflows. To address this, we introduce an RDBMS-style execution and optimization framework, in which an EM task is translated into a plan composed of operators, then optimized and executed. Compared to traditional RDBMSs, this framework is distinguished in that its operators can use crowdsourcing.

The second challenge is to provide efficient implementations for the operators. We describe a set of implementations in Hadoop that significantly advances the state of the art. We focus on the

blocking step as this step consumes most of the machine time. Current Hadoop-based solutions to execute blocking rules either do not scale or have considered only simple rule formats. We develop a solution that can efficiently process complex rules over large tables. Our solution uses indexes to avoid enumerating the Cartesian product, but faces the problem of what to do when the indexes do not fit in memory. We show how the solution can nimbly adapt to these situations by redistributing the indexes and associated workloads across the mappers and reducers.

Finally, we consider the challenge of optimizing EM plans. We show that combining machine operations with crowdsourcing introduces novel optimization opportunities, such as using crowd time to mask machine time. We develop masking techniques that use the crowd time to build indexes and to speculatively execute machine operations. We also show how to replace an operator with an approximate one which has almost the same accuracy yet introduces significant additional masking opportunities. To summarize, our main contributions are:

- We show that for important emerging topics such as EM as a cloud service, Corleone is ideally suited, but must be scaled up to make such services a reality.
- We show that an RDBMS-style execution and optimization framework is a good way to address scaling for crowdsourced EM, and we develop the first end-to-end solution to scale up hands-off crowdsourced EM.
- We define a set of operators and plans for crowdsourced EM that uses machine learning.
- We develop a Hadoop-based solution to execute complex rules over the Cartesian product of two tables (without materializing the Cartesian product), a problem that arises in many settings (not just in EM). The solution significantly advances the state of the art.
- We develop three novel optimization techniques to mask machine time by scheduling certain machine activities during crowdsourcing activities.

Finally, extensive experiments with real-world data sets (using real and synthetic crowds) show that Falcon can efficiently perform hands-off crowdsourced EM over tables of 1.0M - 2.5M tuples at the cost of $54 - $65.5.

4.2 Related Work

Parallel Execution of DAGs of Operators: Several pioneering works have developed platforms for the specification, optimization, and parallel execution of directed acyclic graphs (DAGs) of operators (e.g., [64, 92, 62, 53]).

While highly scalable for many applications, these platforms are not applicable to our context for two reasons. First, it is difficult to encode our workflows, which are specific to *learning-based EM*, in their DAG languages. For example, some platforms consider only key-based blockers, i.e., grouping tuples with the same key into blocks [64]. Falcon however learns a more general kind of blockers called rule-based blockers, which cannot be easily encoded using the current operators of these platforms. Similarly, crowd-based active learning (to learn blockers/matchers) is common in Falcon, but difficult to encode in the current platforms.

Second, even if we can encode our workflows (using UDFs, say), the platforms cannot execute them scalably because they do not yet have scalable solutions for rule-based blocking. In most cases, rule-based blocking will be treated as a "blackbox" UDF to be applied to all tuple pairs in the Cartesian product of the input tables, an impractical solution.

RDBMS-Style Solutions for Data Cleaning: Several such solutions have been developed, e.g., Ajax, BigDansing, and Wisteria [49, 64, 56]. Compared to these works, Falcon is novel in four aspects. First, Falcon focuses on learning-based EM (which uses active learning to learn blockers/matchers). It provides eight "atomic" operators that we believe are appropriate for (a) modeling such EM processes, (b) facilitating efficient operator implementation, and (c) providing opportunities for inter-operator optimization. In contrast, current works either do not consider learning-based EM [64], or define operators at granularity levels that are too coarse for the above purposes [49, 56]. For example, feature vector generation, a very common step in learning-based EM, is not modeled as an atomic operation. As another (extreme) example, Ajax uses just a single operator called Match to model the entire EM process.

Second, current works consider only certain types of blocking, such as key-based ones [64]. However, such blocking types are not accurate for many real-world data sets, due to dirty/missing

data (see Section 4.3.2). As a result, Falcon considers a far more general type of blocking called rule-based blocking and develops efficient MapReduce solutions.

Third, current works do not provide *comprehensive end-to-end solutions* for parallel crowdsourced EM. Ajax considers neither parallel processing nor crowdsourcing. BigDansing develops a highly effective parallel platform but does not consider crowdsourcing. Wisteria crowdsources only the matching step and provides parallel processing for a limited set of blockers and matchers (e.g., only for string similarity join-style blockers). In contrast, Falcon can handle more general types of blockers and matchers. It crowdsources and provides parallel processing (where necessary) for *all* steps of the EM process. It also provides effective novel optimizations, e.g., masking machine time using crowd time.

Finally, both Ajax and BigDansing require users to manually specify blockers/matchers. In contrast, Falcon automatically learns them. Wisteria also considers learning, but it supports only learning the matchers.

Blocking: Key-based blocking (KBB) partitions tuples into blocks based on associated keys (the subsequent matching step then considers only tuples within each block). As such, KBB is highly scalable and is employed in many recent works [64, 42, 111, 33, 29]. Our experience however suggests that it is not always accurate on real-world data, in that it can "kill off" too many true matches (see Section 4.3.2). As a result, we elect to use rule-based blocking (RBB), as used in Corleone. RBB subsumes KBB, i.e., each KBB method can be expressed as an RBB rule. RBB proves highly accurate in our experiments (Section 4.11), but is challenging to scale. As far as we can tell, Falcon provides the first MapReduce solution to scale such rules (each being a Boolean expression of predicates).

Recent work has also examined scaling up sorted neighborhood blocking [66] and meta-blocking [42, 111], which combines multiple blocking methods in a scalable fashion. Such methods are complementary to our work here, and can potentially be used in future versions of Falcon.

Similarity Joins: Falcon is also related to scaling up similarity joins (SJs) [101, 110, 93, 109, 102] and theta joins [82]. To avoid examining all tuple pairs in the Cartesian product, work on SJs

uses inverted indexes [95], prefix filtering [24], partition-based filtering [36], and other pruning techniques [110] (see [112] for a recent survey). Some have considered special similarity functions such as Euclidean distance [96] and edit distance [109, 102]. Most works however consider join conditions of just a single predicate [101, 110] or a conjunction of predicates [73], and develop specialized solutions for these. In contrast, Falcon develops general solutions to handle far more powerful join conditions in our blocking rules, which are Boolean expressions of predicates.

Active Learning and Optimizing: Like Falcon, [80] also proposes using active learning to reduce the number of tuple pairs to be labeled by the crowd. However, it applies learning to the Cartesian product, and thus does not scale to large tables. The idea of combining and optimizing crowd- and relational operators is also discussed in [86]. But as far as we know, Falcon is the first work to do so for crowdsourced EM. Further, some works on optimizing crowd operators have focused on minimizing cost [78, 87], minimizing crowd latency [57], or studying the trade-offs between the two [45]. These works are complementary to ours, which focuses on minimizing the machine time. As far as we know, no other work has proposed the "masking machine time" optimizations in Section 4.10.2.

Crowdsourced RDBMSs: Finally, works have proposed crowdsourced RDBMSs [48, 78, 85, 86] and have addressed crowdsourcing enumeration, select, max, count, and top-k queries, among others (e.g., [100, 84, 55, 76, 34, 83, 11]). Crowdsourced joins (CSJs) which at the heart solve the EM problem, have been addressed in [48, 77, 78, 45, 104]. Initial CSJ works [48, 77] however crowdsource all tuple pairs in the Cartesian product of the two tables and hence do not scale. Recent CSJ works [45, 78] ask users to write filters to reduce the number of tuple pairs to be crowdsourced. Such hand-crafted filters can be difficult to write and using them severely limits the applicability of crowdsourced RDBMSs. Falcon can automatically learn such filters (i.e., blocking rules) using crowdsourcing, and thus can potentially be used to perform CSJ over large tables.

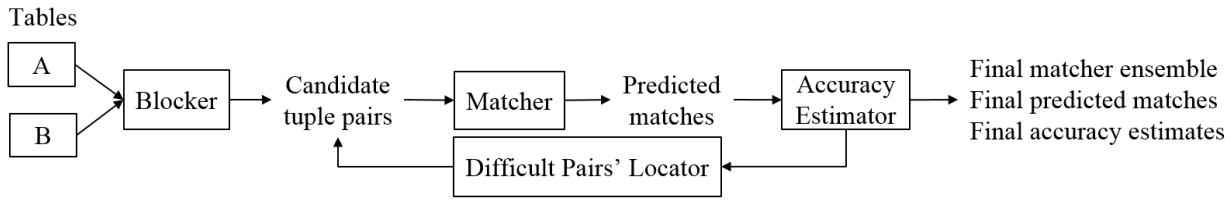


Figure 4.1: The EM workflow of Corleone.

4.3 Problem Definition

Chapter 2 describes Corleone in detail. In this section, we briefly describe the EM workflows considered by Corleone and Falcon, analyze Corleone's limitations, and define the problem considered by Falcon.

4.3.1 The EM Workflows of Corleone

Many types of EM tasks exist, e.g., matching across two tables, within a single table, matching into a knowledge base, etc. Corleone (and Falcon) consider one such kind of tasks: matching across two tables. Specifically, given two tables A and B, Corleone applies the EM workflow in Figure 4.1 to find all tuple pairs $(a \in A, b \in B)$ that match. This workflow consists of four main modules: Blocker, Matcher, Accuracy Estimator, and Difficult Pairs' Locator.

The Blocker generates and applies blocking rules to $A \times B$ to remove obviously non-matched pairs (Figure 4.2.b shows two such rules). Since $A \times B$ is often very large, considering all tuple pairs in it is impractical. So blocking is used to drastically reduce the number of pairs that subsequent modules must consider. The Matcher uses active learning to train a random forest classifier [22], then applies it to the surviving pairs to predict matches. The Accuracy Estimator computes the accuracy of the Matcher. The Difficult Pairs' Locator finds pairs that most likely the current Matcher has matched incorrectly. The Matcher then learns a better random forest to match these pairs, and so on, until the estimated matching accuracy no longer improves.

Corleone is distinguished in that the above four modules use no developers, only crowdsourcing. For example, to perform blocking, most current works would require a developer to examine Tables A and B to come up with heuristic blocking rules (e.g., "If prices differ by at least \$20,

then two products do not match"), code the rules (e.g., in Python), then execute them over A and B. In contrast, the Blocker in Corleone uses crowdsourcing to learn such blocking rules (in a machine-readable format), then automatically executes those rules. Similarly, the remaining three modules also heavily use crowdsourcing but no developers.

Corleone can also be run in many different ways, giving rise to many different EM workflows. The default is to run multiple iterations until the estimated accuracy no longer improves. But the user may also decide to just run until a budget (e.g., $300) has been exhausted, or to run just one iteration, or just the Blocker and Matcher, or just the Matcher if the two tables are relatively small, making blocking unnecessary, etc.

4.3.2 The EM Workflows Considered by Falcon

As a first step, Falcon will consider EM workflows that consist of just the Blocker followed by the Matcher, or just the Matcher. (Virtually all current works consider similar EM workflows.) As we will see, these workflows already raise difficult scaling challenges. Considering more complex EM workflows is future work.

We now briefly describe the Blocker and the Matcher, focusing only on the aspects necessary to understand Falcon (see Chapter 2 for more details).

The Blocker: The key idea underlying this module is to use crowdsourced active learning to learn a random forest based matcher (i.e., binary classifier) M [22], then extract certain paths of M as blocking rules.

Specifically, learning on $A \times B$ is impractical because it is often too large. So this module first takes a small sample of tuple pairs S from $A \times B$ (without materializing the entire $A \times B$), then uses S to learn matcher M.

To learn, the module first trains an initial random forest matcher M, uses M to select a set of controversial tuple pairs from sample S, then asks the crowd to label these pairs as matched / no-matched. In the second iteration, the module uses these labeled pairs to re-train M, uses M to select a new set of tuple pairs from S, and so on, until a stopping criterion has been reached.

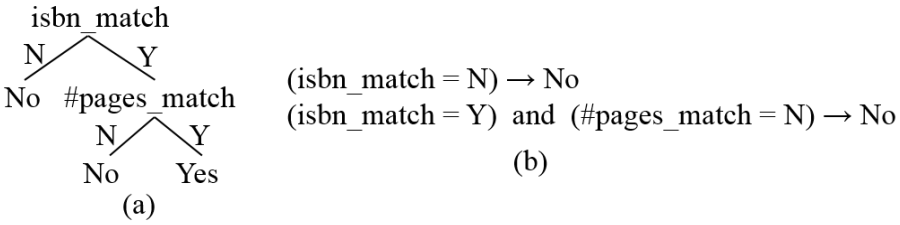


Figure 4.2: (a) A decision tree learned by Corleone and (b) blocking rules extracted from the tree.

At this point the module returns a final matcher M, which is a random forest classifier consisting of a set of decision trees. Each tree when applied to a tuple pair will predict if it matches, e.g., the tree in Figure 4.2.a predicts that two book tuples match only if their ISBNs match and the number of pages match. Given a tuple pair p, matcher M applies all of its decision trees to p, then combines their predictions to obtain a final prediction for p.

Next, the module extracts all tree branches that lead from the root of a decision tree to a "No" leaf as candidate blocking rules. Figure 4.2.b shows two such rules extracted from the tree in Figure 4.2.a. The first rule states that if two books do not agree on ISBNs, then they do not match.

Next, for each extracted blocking rule r, the module computes its precision. The basic idea is to take a sample T from S, use the crowd to label pairs in T as matched / no-matched, then use these labeled pairs to estimate the precision of rule r. To minimize crowdsourcing cost and time, T is constructed (and expanded) incrementally in multiple iterations, only as many iterations as necessary to estimate the precision of r with a high confidence (see Chapter 2).

Finally, the Blocker applies a subset of high-precision blocking rules to $A \times B$ to remove obviously non-matched pairs. The output is a set of candidate tuple pairs C to be passed to the Matcher.

The Matcher: This module applies crowdsourced active learning on C to learn a new matcher N, in the same way that the Blocker learns matcher M on sample S. The module then applies N to match the pairs in C.

Reasons for Not Using Key-Based Blocking: Recall that we plan to learn blocking rules such as those in Figure 4.2.b. As we will see in Section 4.7, it is a major challenge to execute such rules

over two tables A and B efficiently, without enumerating the entire Cartesian product as Corleone does.

Given this, one may ask why consider rule-based blocking (RBB) at all. In particular, many recent works have used key-based blocking (KBB, see the related work section), where tuples are grouped into blocks based on associated keys, and only tuples in each block are considered in the subsequent matching step. As such, KBB is highly scalable.

It turns out that KBB does not work well for many data sets, due to dirty data, variations in data values, and missing values. For example, on the Products, Songs, and Citations data sets in Section 4.11, our extensive effort at KBB produces recalls of 72.6%, 98.6%, and 38.8% (recall measures the fraction of true matches that survive the blocking step; ideally we want 100% recall). In contrast, rule-based blocking produces recalls of 98.09%, 99.99%, and 99.67%.

Thus, we decide to use rule-based blocking. This does not mean we execute blocking rules on the *materialized* Cartesian product, like Corleone. Instead, we analyze the rules, build indexes over the tables, then use them to quickly identify only a small fraction of tuple pairs to which the rules should be applied (see Section 4.7). In particular, it can be shown that when a blocking rule performs key-based blocking (e.g., the first rule in Figure 4.2.b, "(isbn_match = N) $\rightarrow$ No", considers only tuples that share the same ISBN), our solution in Section 4.7 reduces to current key-based blocking solutions on MapReduce.

4.3.3 Limitations of Corleone

Corleone is highly promising because it uses only crowdsourcing to achieve high EM accuracy at a relatively low cost. However it suffers from a major limitation: it does not scale to large tables. The largest table pair considered by Corleone is 2.6K tuples $\times$ 64K tuples (see Chapter 2).

Several real-world applications that we have been working with, however, must match tables ranging from 100K to several millions of tuples. On two tables of 100K tuples each, Corleone had to be stopped after more than a week, with no result. Clearly, we must drastically scale up Corleone to make it practical.

To scale, our analysis reveals that we must address three problems. First, we must *minimize the crowd time* of Corleone. As described earlier, the Blocker and Matcher crowdsource in iterations (until reaching a stopping criterion). Each iteration requires the crowd to label a certain number of tuple pairs (e.g., 20). The number of iterations can be quite large (e.g., close to 100 for the Blocker in certain cases), thus incurring a large crowd time (and cost).

Second, we must *minimize the machine time* of Corleone. The single biggest "consumer" of machine time turned out to be the step of executing the blocking rules. For this step Corleone applies the rules to each tuple pair in $A \times B$. This clearly does not scale, e.g., two tables of 100K tuples each already produce 10 billion tuple pairs, too large to be exhaustively enumerated. Given the single-machine in-memory nature of Corleone, certain other steps also consume considerable time, e.g., the set C of tuple pairs output by the Blocker is often quite large (often in the tens of millions), making active learning on C very slow.

Finally, Corleone performs crowdsourcing and machine activities sequentially. For example, in each iteration of active learning in the Blocker, the machine is idle while Corleone waits for the crowd to finish labeling a set of tuple pairs. Thus, we should consider *masking the machine time*, by scheduling as many machine activities as possible during crowdsourcing. As we will see in Section 4.10.2, this raises very interesting optimization opportunities, and can significantly reduce the total execution time.

4.3.4 Goals of Falcon

It is difficult to address all of the above performance factors simultaneously. So as a first step, in Falcon we will develop a relatively simple solution to keep the crowd time (and cost) at an acceptable level (for now), then focus on minimizing and masking machine time.

Keeping Crowd Time Manageable: The total crowd time t_c is the sum of t_{ab}, crowd time for active learning of the Blocker, t_{er}, crowd time for evaluating the blocking rules of the Blocker, and t_{am}, crowd time for active learning of the Matcher.

We observe that active learning in the Blocker and Matcher can take up to 100 iterations. Yet after 30 iterations or so the accuracy of the learned matcher stays the same or increases only

minimally. As a result, in Falcon we stop active learning when the stopping criterion is met or when the number of iterations has reached a pre-specified threshold k (currently set to 30). This caps the crowd times t_{ab} and t_{am}. As for t_{er}, we can show that:

Proposition 1. *The procedure of evaluating blocking rules (described in Chapter 2) is guaranteed to execute at most 20 iterations per rule.*

Proof. We prove that evaluating the blocking rules is guaranteed to execute at most 20 iterations per rule. Section 2.4.2 describes in detail the algorithm for evaluating a set of rules V by the crowd. Briefly, for each rule $R \in V$, the following 3-step loop is executed:

1. Randomly select b examples in $cov(R, S)$, get these examples labeled by crowd, then add the labeled examples to set X (initially empty).
2. Let $|cov(R, S)| = m$, $|X| = n$, and n_- be the number of examples in X that are labeled "not matched". Estimate the precision of rule R over S as $P = n_-/n$, with an error margin $\epsilon = Z_{(1-\delta)/2}\sqrt{\left(\frac{P(1-P)}{n}\right)\left(\frac{m-n}{m-1}\right)}$.
3. If $P \geq P_{min}$ and $\epsilon \leq \epsilon_{max}$ then stop and retain R. If (a) $(P + \epsilon) < P_{min}$, or (b) $\epsilon \leq \epsilon_{max}$ and $P < P_{min}$, then stop and drop R. Otherwise return to Step 1.

Note that for each rule, this algorithm proceeds in iterations and stops either by retaining or dropping the rule. Also observe (from Step 3) that if $\epsilon \leq \epsilon_{max}$, then the algorithm either retains the rule (if $P \geq P_{min}$) or drops the rule (if $P < P_{min}$). In either case, it terminates the iterations for the rule.

So we can compute the minimum n that will guarantee $\epsilon \leq \epsilon_{max}$ as follows:

$$\begin{aligned}
\epsilon &= Z_{(1-\delta)/2}\sqrt{\frac{P(1-P)}{n} \cdot \frac{m-n}{m-1}} && \text{(from step 2 of algorithm)} \\
&\approx Z_{(1-\delta)/2}\sqrt{\frac{P(1-P)}{n}} && \text{(m >> n, } \frac{m-n}{m-1} \approx 1) \\
&\leq Z_{(1-\delta)/2}\sqrt{\frac{1}{4n}} && (0 \leq P \leq 1, P(1-P) \leq \frac{1}{4})
\end{aligned}$$

$$\epsilon \leq \epsilon_{max} \implies Z_{(1-\delta)/2}\sqrt{\frac{1}{4n}} \leq \epsilon_{max} \implies n \geq \frac{Z^2_{(1-\delta)/2}}{4\epsilon^2_{max}}$$

Corleone (and hence Falcon) uses the following parameter setting: $\epsilon_{max} = 0.05$, and $\delta = 0.95$. With this parameter setting, we get $n \geq 384$.

This means that when at least 384 examples are labeled by the crowd, it is guaranteed that $\epsilon \leq \epsilon_{max}$ and hence the algorithm retains or drops the rule. In the current setting, to get 384 examples labeled, Falcon will run for 20 iterations (since it gets $b = 20$ examples labeled by the crowd per iteration).

Hence, the procedure for evaluating blocking rules in Falcon is guaranteed to execute at most 20 iterations per rule. □

As a result, we can estimate an upper bound on the total crowd time (regardless of the table sizes):

Proposition 2. *For active learning in the Blocker and Matcher, let k be the upper bound on the number of iterations, q_1 be the number of pairs to be labeled in each iteration, and t_a be the average time it takes the crowd to label a pair (e.g., the time it takes to obtain three answers from the crowd, then take majority voting). For rule evaluation in the Blocker, let n be the number of rules to be evaluated, and q_2 be the number of pairs to be labeled in each iteration. Then the total crowd time t_c is upper bounded by $t_a(2kq_1 + 20nq_2)$.*

In practice, when crowdsourcing tables of several million tuples each, we found t_c in the range 9h 59m - 15h 48m on Mechanical Turk. While still high, this time is already acceptable in many settings, e.g., many users are satisfied with letting the system run overnight. Thus, we turn our attention to reducing machine time, which poses a far more serious problem as it can easily consume weeks.

Minimizing and Masking Machine Time: Let t_m be the total machine time (i.e., the sum of the times of all machine activities). The total time of Corleone is $(t_c + t_m)$. We seek to minimize this time by (a) minimizing t_m, and (b) masking, i.e., scheduling as many machine activities as possible during crowd activities. This will result in a (hopefully far smaller) total time $(t_c + t_u)$, where $t_u < t_m$ is the total time of machine activities that cannot be masked.

We will seek to preserve the EM accuracy of Corleone, which are shown to be already quite high in a variety of experiments (see Chapter 2). Yet we will also explore optimization techniques that may reduce this accuracy slightly, if they can significantly reduce $(t_c + t_u)$.

Reasons for Focusing on Machine Time: As hinted above, we focus on machine time for several reasons. First, for now machine time *is* the main bottleneck. It often takes weeks on moderate data sets, rendering Corleone unusable. On the other hand, crowd time (say on Mechanical Turk) is already in the range of being acceptable for many applications. So our first priority is to reduce machine time to an acceptable range (say hours), to be able to build practical systems.

Second, Section 4.11 shows that we have achieved this goal, reducing machine time from weeks to 52m - 2h 32m on several data sets. Since crowd time on Mechanical Turk was 11h 25m - 13h 33m, it may appear that the next goal should be to minimize crowd time because it makes up a large portion of total time. This however is not quite correct. As we discuss in Section 4.11.1, crowd time can vary widely depending on the platform. In fact, we describe an application on drug matching that uses in-house crowds, where crowd time was only 1h 37m, but machine time was 2h 10m, constituting a large portion (57%) of the total run time. For such applications further optimizing machine time is important.

Finally, once we have made major progress on reducing machine time, we fully intend to focus on crowd time, potentially using the techniques in [57].

4.4 The Falcon Solution

4.4.1 Adopting an RDBMS Approach

Recall that Falcon considers EM workflows consisting of the Blocker followed by the Matcher, or just the Matcher if the tables are small. A straightforward solution is to just optimize these two stand-alone monolithic EM workflows.

This solution however is unsatisfying. First, soon we may want to add more operators (e.g., the Accuracy Estimator), resulting in more kinds of EM workflows. Second, we focus for now on machine time, but soon we may consider other objectives, e.g., minimizing crowd time/cost,

maximizing accuracy, etc. In fact, users often have differing preferences for trade-offs among accuracy, cost, and time. It would be difficult to extend an "opaque" solution focusing on stand-alone monolithic EM workflows to such scenarios. Finally, the Blocker and Matcher actually share common operations, e.g., crowdsourced active learning. An opaque solution makes it hard to factor out and optimize such commonalities.

For these reasons, we propose that Falcon adopt an RDBMS approach. Specifically, (1) we will identify a set of basic operators that underlie the Blocker and Matcher (as well as constitute a big part of other modules, e.g., Accuracy Estimator). We will compose these operators to form EM workflows. (2) We will develop efficient implementations of these operators, using Hadoop. And (3) we will develop both intra- and inter-operator optimization techniques for the resulting EM workflow, focusing on rule-based optimization for now (and considering cost-based optimization in the future).

We now define a set of operators and show how to compose them to form EM workflows, henceforth called *EM plans*. Sections 4.5-4.10 describe efficient implementations of operators, then plan generation, execution, and optimization.

4.4.2 Operators

We have defined the following eight operators that we believe are sufficient to compose a wide variety of EM plans.

sample_pairs: takes two tables A, B and a number n, and outputs a set S of n tuple pairs $(a, b) \in A \times B$. This operator is important because we want to learn blocking rules on the sample S instead of $A \times B$, as learning on $A \times B$ is impractical for large A and B.

gen_fvs: takes a set S of tuple pairs and a set F of m features, then converts each pair $(a, b) \in S$ into a feature vector $\langle f_1(a, b), \ldots, f_m(a, b)\rangle$, where each feature $f_i \in F$ is a function that maps (a, b) into a numeric score. For example, a feature may compute the edit distance between the values of the attributes $title$ of a and b. This operation is important because we want to learn

blocking rules (during the blocking stage) and a matcher (during the matching stage), and we need feature vectors to do the learning. (Section 4.10.1 discusses how Falcon generates features.)

al_matcher: Suppose we have taken a sample S from $A \times B$ and have converted S into a set S' of feature vectors. This operator performs crowdsourced active learning on S' to learn a matcher M. Specifically, it trains an initial matcher M, uses M to select a set of controversial pairs from S', asks the crowd to label these pairs, uses them to improve M, and so on, until reaching a stopping criterion.

get_blocking_rules: extracts a set of blocking rules from a matcher M (typically output by operator $al\_matcher$). This operator assumes that M is such that we can extract rules from it. To be concrete, Falcon will assume that M is a random forest, from which we can extract a set of blocking rules $\{R_1, \ldots, R_n\}$ such as those shown in Figure 4.2.b. Each rule R_i is of the form

$$p_1^i(a,b) \wedge \ldots \wedge p_{m_i}^i(a,b) \rightarrow \ drop\ (a,b), \tag{4.1}$$

where each predicate $p_j^i(a,b)$ is of the form $[f_j^i(a.x, b.y)\ op_j^i\ v_j^i]$. Here f_j^i is a function that computes a score between the values of attribute x of tuple $a \in A$ and attribute y of tuple $b \in B$ (e.g., string similarity functions such as edit distance, Jaccard). Thus predicate p_j^i compares this score via operation op_j^i (e.g., $=$, $<$, $\leq$) with a value v_j^i.

eval_rules: takes a set of blocking rules, computes their precision and coverage, then retains only those with high precision and coverage. Precisions are computed using crowdsourcing. This operator is important because some blocking rules may be imprecise, i.e., eliminating too many matching tuples when applied to $A \times B$.

select_opt_seq: Let $\mathcal{R}$ be the set of n blocking rules output by $eval\_rules$. Then there are $\sum_{k=0}^{n} \binom{n}{k} * k!$ possible rule sequences, each containing a subset of rules in $\mathcal{R}$. Executing a rule sequence $\bar{R}$ on a tuple pair means executing each rule in $\bar{R}$ in that order, until a rule "fires" or all rules have been executed. It turns out that the rule sequences of $\mathcal{R}$ can vary drastically in terms of precision, selectivity, and run time. Thus this operator returns a rule sequence $\bar{R}*$ from $\mathcal{R}$ that when applied to $A \times B$ would minimize run time while maximizing precision and selectivity (i.e.,

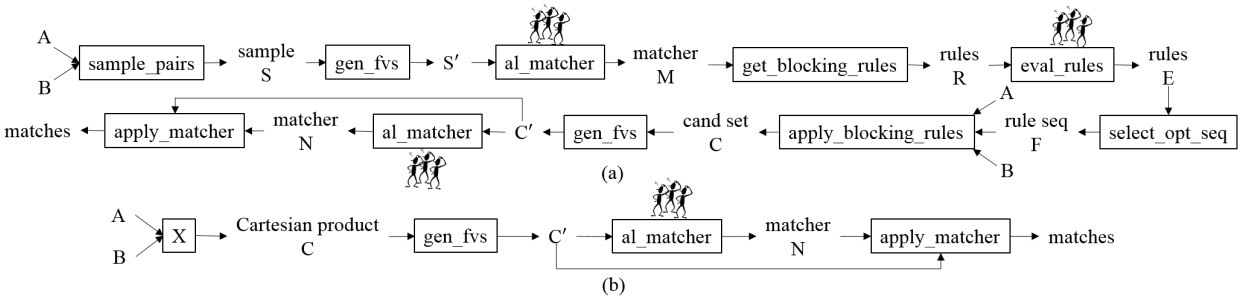


Figure 4.3: The two plan templates used in Falcon.

it would produce a set of tuple pairs C that is as small as possible and yet contains as many true matching pairs as possible).

apply_blocking_rules: applies a sequence of blocking rules $\bar{R}$ to two tables A and B, producing a set of tuple pairs $C \subseteq A \times B$ to be matched in the matching stage. Applying $\bar{R}$ in a naïve way to all pairs in $A \times B$ is clearly impractical. So this operator uses indexes to apply $\bar{R}$ only to certain tuple pairs, on a Hadoop cluster (see Section 4.7.3).

apply_matcher: applies a matcher to a set of tuple pairs C, where each pair is encoded as a feature vector, to predict "matched"/"not matched" for each pair in C.

4.4.3 Composing Operators to Form Plans

The above eight operators (together with relational operators such as selection, join, and projection) can be combined in many different ways to form EM plans. As a first step, Falcon will consider the two common plan templates in Figure 4.3, which correspond to the EM workflows that use both the Blocker and Matcher, and just the Matcher, respectively.

The first plan template (Figure 4.3.a, where operators with crowd symbol use crowdsourcing) performs both blocking and matching. Specifically, we apply $sample\_pairs$ to Tables A and B to obtain a sample S, then convert S into a set of feature vectors S'. Next, we do crowdsourced active learning on S' to obtain a matcher M. Next we extract blocking rules R from M, then use crowdsourcing to evaluate and retain only the best rules E. Next, we select the best rule sequence F from E, then apply F on Tables A and B to obtain a set of tuple pairs C. Finally, we convert C

into a set of feature vectors C', do crowdsourced active learning on C' to learn a matcher N, then apply N to match pairs in C'.

The second plan template (Figure 4.3.b) performs only matching. It computes the Cartesian product C of A and B, converts C into a set of feature vectors C', does crowdsourced active learning on C' to learn a matcher N, then applies N to match pairs in C'.

Falcon selects the first plan template if it deems Tables A and B sufficiently large, necessitating blocking, otherwise it selects the second plan template (see Section 4.10.1).

In the next few sections we describe how to implement the operators efficiently. First we focus on the four operators $sample\_pairs$, $select\_opt\_seq$, $apply\_blocking\_rules$, and gen_fvs as implementing them is most challenging. Then we discuss the rest of the operators.

4.5 Sampling Input Tables

The $sample\_pairs$ operator samples a set S from $A \times B$, so that subsequent operators can learn on S. To sample, first we must decide on a size. We want S (when encoded as a set of feature vectors) to fit in memory, and large enough so that we can learn effective blocking rules, yet not too large so that learning would be slow. Our experiments show $|S|$ in the range 500K-1M to be reasonable (see Section 4.11.4).

Let n be the size of S. We now consider sampling n pairs from $A \times B$. Naïvely, we can *randomly* sample tuples from A and B, then take their Cartesian product to be S. Random tuples from A and B however are unlikely to match, so S may contain very few positive (i.e., matching) pairs, rendering learning ineffective.

To address this, Corleone randomly samples $n/|A|$ tuples from B (the larger table), then takes the Cartesian product of these with A to be S. If B has a reasonable number of tuples that have matches in A and if these tuples are distributed uniformly in B, then this strategy ensures that S contains a reasonable number of matches.

It turns out that this solution does not work for large A and B. First, if A is larger than n (as in some of our experiments), the solution is not applicable. Second, even if A is smaller than n, it may not be much smaller, in which case we sample very few tuples from B. For example, if

$|A| = 500K$ and $n = 1M$, then we sample only 2 tuples from B. This can produce very few matches in S, especially if we unluckily pick two tuples from B that have no matches in A.

To address this, we develop the following solution. First, we create an inverted index on A, the smaller table. Specifically, we convert each tuple a in A into a document $d(a)$ that contains only the (white space delimited) tokens in the string attributes of a (we use a procedure that analyzes the tables to recognize string attributes). Then we build an inverted index I with entries $\langle w : id_1, \ldots, id_k \rangle$, indicating that token w appear in the documents of the tuples $id_1, \ldots, id_k$ in A.

Next, we randomly select n/y tuples from B (where y is a tunable parameter, currently set to 100). For each selected tuple $b \in B$, we select y tuples from A (to be paired with b) as follows. First, we convert b into a document $d(b)$ that contains only the tokens in the string attributes of b (similar to the way tuples of A have been converted). Next, we use the inverted index I to find the set X of tuples a in A whose documents $d(a)$ share at least a token with $d(b)$. We sort the tuples in X in decreasing order of the number of tokens shared with b, then select the top $y_1 = min(y/2, |X|)$ tuples from X. Next, we randomly select $(y - y_1)$ tuples from the remaining tuples of A. We then pair these y selected tuples with b. The sample S contains all such pairs, for all n/y selected tuples b in B.

Intuitively, we have constructed S such that for each tuple b selected from B, we have tried to pair it with (1) roughly $y/2$ tuples from A that are likely to match (judged by the number of shared tokens), and (2) roughly $y/2$ random tuples from A. Thus we try to get a reasonable number of matches into S yet keep it as representative of $A \times B$ as possible.

Section 4.11.4 shows that this simple and fast sampling strategy is highly effective, in that the blocking rules learned on the samples achieve high recall of 98.09%-99.99% on our data sets (recall measures the fraction of true matches that survive blocking). We implement this strategy using two MapReduce jobs, to create the inverted index and to generate the pairs of S, respectively.

4.6 Selecting Optimal Rule Sequence

Given a set of blocking rules $\mathcal{R}$, this operator finds the optimal rule sequence with respect to precision, selectivity, and run time, which are defined as follows:

Definition 1. *Let $\bar{X}$ be a rule sequence and Y be a set of tuple pairs. Precision $prec(\bar{X}, Y)$ is the fraction of negative predictions $\bar{X}$ made on Y that are indeed negative. Selectivity $sel(\bar{X}, Y)$ is the number of pairs in Y that survive $\bar{X}$ (which drops all pairs predicted negative) divided by $|Y|$. Finally, run time $time(\bar{X}, Y)$ is the average run time of $\bar{X}$ on a tuple pair in Y.* □

We then define an overall score for $\bar{X}$ on Y as follows:

$$score(\bar{X}, Y) = \alpha * prec(\bar{X}, Y) - \beta * sel(\bar{X}, Y) - \gamma * time(\bar{X}, Y),$$

where α, β, γ can be tuned by the application, e.g., those that must minimize the matches lost to blocking can choose a high weight for α, memory-constrained applications that can process only a small candidate set can choose a high β, whereas real-time applications that must perform blocking fast can choose a high γ.

Let $p(\mathcal{R})$ be the set of all rule sequences of $\mathcal{R}$. Our problem is to find the rule sequence $\bar{R} \in p(\mathcal{R})$ that maximizes $score(\bar{R}, A \times B)$. Since we cannot realistically enumerate $A \times B$, we develop an approximate solution by finding the rule sequence $\bar{R} \in p(\mathcal{R})$ that maximizes $score(\bar{R}, S)$, where S is the sample produced by $sample\_pairs$ from $A \times B$.

To solve this problem, we will enumerate all possible subsets of $\mathcal{R}$, find the optimal sequence for each subset, then compare them to find the globally optimal sequence. Since $eval\_rules$ outputs only a small set of rules, e.g., up to 20 in the current Falcon, the number of subsets of $\mathcal{R}$ is relatively small (e.g., 2^{20}), and direct enumeration is fast.

So the main problem left is to find the optimal sequence for a given subset of rules. Suppose this subset has k rules, then there are $k!$ sequences. It is not difficult to see that these sequences have the same precision and selectivity, but can differ drastically in run time (depending on how time intensive and how selective the rules in the sequence are, and how these rules are ordered in the sequence). Thus, our goal is to find the sequence with minimal run time.

This problem is NP-hard as shown in [15]. Specifically, [15] shows how the problem of ordering pipelined filters for stream processing (when the stream and filter characteristics have stabilized) reduces to the min-sum set cover problem, which is known to be NP-hard [81, 46]. Although [15] shows the problem of ordering a set of pipelined filters (for stream processing) to be NP-hard,

the problem at the core is the same as ours. To elaborate, [15] considers the problem of optimally ordering a set of n filters $F_1, F_2, ..., F_n$ in conjunction, where each filter F_i takes a stream tuple t as input and returns either $true$ or $false$. If F_i returns $false$ for tuple t, then F_i is said to drop t. A tuple is emitted in the final result if and only if all n filters return $true$. The goal is to optimally order the n filters so that the expected time to process an incoming tuple t is minimized. This problem is equivalent to our problem of selecting an optimal sequence of rules because we can view each rule as a filter that passes or drops a tuple pair.

The work [81] also proves that any polynomial time algorithm to solve the min-sum set cover problem can at best provide a constant-factor approximation algorithm guarantee. To this end [15] proposes a 4-approximation greedy algorithm to optimally order pipelined filters for data streams. We adapt this greedy algorithm as a solution to our problem. Specifically, given a set $T = \{R_1, R_2, ..., R_m\}$ of rules, we first choose rule R_i that maximizes $[1 - sel(R_i, S)]/time(R_i, S)$, then choose rule R_j $(j \neq i)$ that maximizes $[1 - sel([R_i, R_j], S)/sel(R_i, S)]/time(R_j, S)$, then choose rule R_k $(k \neq j, i)$ that maximizes $[1 - sel([R_i, R_j, R_k], S)/sel([R_i, R_j], S)]/time(R_k, S)$, and so on. The chosen rules form the optimal sequence $[R_i, R_j, R_k, \ldots]$.

What makes the problem *hard* is the fact that the rules in the sequence can be correlated, i.e., non-independent. For example, many rules may share the same feature (e.g., Jaccard(a.title, b.title)) and hence are correlated. When the rules are independent, the optimal ordering of rules in a sequence can be computed in polynomial time. Prior work [25, 59, 72] on related problems (e.g., pipelined filters, ordering selection and join conditions in SQL queries) makes the independence assumption and uses the *rank*-based ordering technique. The solution there is to order the rules in decreasing order of *rank*, where for a rule R_i, $rank(R_i) = [1 - sel(R_i)]/time(R_i)$. Note that our greedy algorithm described above also simplifies to the *rank*-based solution when rules are independent. This is because if two rules R_i and R_j are independent, we can write $sel([R_i, R_j]) = sel(R_i)sel(R_j)$. However, the assumption that rules are independent is not very realistic as it is pretty common for rules to share the same or related features. Nevertheless, the approximate solution (given by the above algorithm) seems to work well in practice (see Section 4.11.2).

After we have obtained an optimal sequence for each subset of rules, we want to find the globally optimal sequence. Recall that the globally optimal sequence is the one that has maximal *score*. However, in order to compute the *score* for a sequence, we must compute its precision, selectivity, and run time. To compute these, we first need to compute the coverage and selectivity of a rule, as described next.

Coverage of a Rule: For each rule $R_i \in \mathcal{R}$, we have already computed (when extracting the top rules from the random forest) the coverage of rule R_i over sample S, $cov(R_i, S)$, which is the set of tuple pairs in S that R_i would "drop". For each rule R_i, Falcon maintains $cov(R_i, S)$ in the form of a bitmap B_i of size $|S|$. Each bit b_j in B_i indicates whether for the j^{th} tuple pair in S, the rule R_i would "drop" it (bit 1), or "keep" it (bit 0). Maintaining these bitmaps helps Falcon compute the coverages of rule sequences efficiently (as we shall see shortly).

Selectivity of a Rule: Having computed the coverage of a rule R_i, Falcon computes the selectivity of the rule as $sel(R_i, S) = 1 - (|cov(R_i, S)|/|S|)$.

Coverage of a Sequence: Coverage of a rule sequence $\bar{R}$ on S, $cov(\bar{R}, S)$, is the set of pairs in S that $\bar{R}$ would "drop". $\bar{R}$ would "drop" a tuple pair if *any one* rule $R_i \in \bar{R}$ would "drop" it. So it is not hard to see that this can easily be computed by OR-ing the bitmaps B_i of each rule R_i in $\bar{R}$.

Selectivity of a Sequence: Having computed the coverage of a sequence, Falcon computes the selectivity of a sequence as $sel(\bar{R}, S) = 1 - (|cov(\bar{R}, S)|/|S|)$.

Run Time of a Sequence: Falcon estimates the run time of a sequence $\bar{R} = [R_1, .., R_m]$ from the run times and selectivities of its sub-sequences as $time(\bar{R}, S) = time(R_1, S) + sel(R_1, S) * time(R_2, S) + sel([R_1, R_2], S) * time(R_3, S) + ... + sel([R_1, ..., R_{m-1}], S) * cost(R_m, S)$. Falcon estimates the run times of sequences in a bottom-up fashion proceeding from shorter sequences to longer ones.

Precision of a Sequence: To compute the actual precision, $prec(\bar{R}, S)$, we need the true labels of all pairs in $cov(\bar{R}, S)$. However, we do not have these labels. So we estimate the precision of a rule sequence from the estimated precisions of its constituent rules. Note that we have obtained

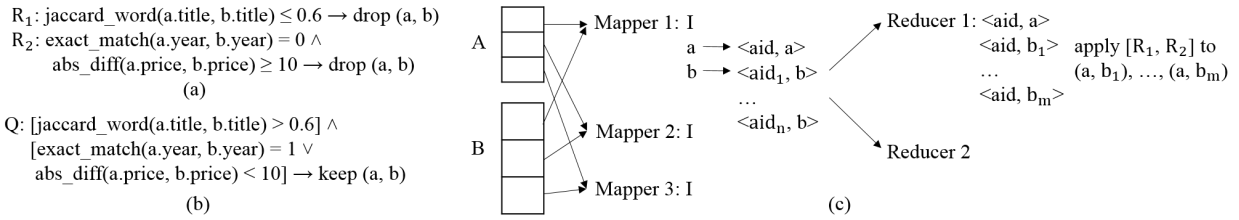


Figure 4.4: (a) A rule sequence, (b) the same rule sequence converted into a single "positive" rule, and (c) an illustration of how $apply\_all$ works.

estimates of the precision of the constituent rules from crowd-based evaluation (in $eval\_rules$ operator). Using these we can compute a lower bound on the estimate of precision, $prec(\bar{R}, S)$, of a rule sequence $\bar{R} = [R_1, ..., R_m]$ as:

$$prec(\bar{R}, S) \geq 1 - \frac{\sum_{i=1}^{m} |cov(R_i, S)|(1 - prec(R_i, S))}{|cov([R_1, ..., R_m], S)|}.$$

We use the above lower bound on $prec(\bar{R}, S)$ as the substitute for $prec(\bar{R}, S)$.

4.7 Applying the Blocking Rules

In this section we will focus on $apply\_blocking\_rules$ operator which consumes by far the most of machine time, and is also the most difficult to implement.

Recall that $apply\_blocking\_rules$ takes two tables A and B, and a sequence of rules $\bar{R} = [R_1, \ldots, R_n]$, where each rule R_i is of the form shown in Formula 4.1, then outputs all tuple pairs $(a, b) \in A \times B$ that satisfy at least one rule in $\bar{R}$.

Example 4.7.1. *Consider the sequence of two rules $[R_1, R_2]$ in Figure 4.4.a. Rule R_1 states that two books do not match if their titles are not sufficiently similar (using a Jaccard similarity function over the two titles tokenized as two sets of words). Rule R_2 states that two books do not match if they disagree on years and their prices differ by at least \$10 (here $exact\_match(a.year, b.year)$ returns 1 if the years match and 0 otherwise, and $abs\_diff(a.price, b.price)$ returns the absolute difference in prices).*

In what follows we describe the limitations of the current solutions for this operator, the key ideas underlying our solution, then the implementation of these ideas.

4.7.1 Limitations of Current Solutions

Two MapReduce solutions to apply rules to tuple pairs in $A \times B$ have been proposed: $MapSide$ and $ReduceSplit$ [65].

$MapSide$ assumes the smaller table fits in the memory of the mappers, in which case it can execute a straightforward map-only job to enumerate the pairs and apply the rules. If neither table fits in memory, then $ReduceSplit$ uses the mappers to enumerate the pairs, then spreads them evenly among the Reducers, which apply the rules.

As far as we can tell, these are state-of-the-art solutions that can be applied to our setting. (The works [101, 110, 73] are related, but consider specialized types of rules and develop specialized solutions for these. Hence they do not apply to our setting that uses a far more general type of rules.)

Both $MapSide$ and $ReduceSplit$ are severely limited in that they still enumerate the entire $A \times B$, which is often very large (e.g., 10 billion pairs for two tables of 100K tuples each).

4.7.2 Key Ideas Underlying Our Solution

Both $MapSide$ and $ReduceSplit$ assume the rules are "blackboxes", necessitating the enumeration of $A \times B$. This is not true in Falcon, where the rules use the features automatically generated by Falcon (see Section 4.10.1), and these features in turn often use well-known similarity functions, e.g., edit distance, Jaccard, exact match, etc. (see Example 4.7.1). Thus, we can exploit certain properties of these functions to build index-based filters, then use them to avoid enumerating $A \times B$.

Example 4.7.2. *Suppose we want to find all tuple pairs in $A \times B$ that satisfy the predicate $jaccard\_word(a.title, b.title) > 0.6$. It is well known that for a pair of string (x, y), $jaccard(x, y) \geq t$ implies $|y|/t \geq |x| \geq |y| \cdot t$ [112]. This property can be exploited to build a length filter for the above predicate. Specifically, we build a B-tree index I_l over the lengths of attribute $a.title$*

(counted in words). Given a tuple $b \in B$ the filter uses I_l to find all tuples a in A where the length of $a.title$ falls in the range $[|b.title| \cdot 0.6, |b.title|/0.6]$, then returns only these (a, b) pairs. We can then evaluate $jaccard\_word(a.title, b.title) > 0.6$ only on these pairs.

Realizing this idea in MapReduce however raises the challenge that the indexes may not fit into memory. So we propose four solutions that balance between the amount of available memory and the amount of work done at the mappers and reducers, then develop rules for when to select which solutions.

4.7.3 The End-to-End Solution

We now build on the above ideas to describe the end-to-end solution for $apply\_blocking\_rules$.

1. Convert the Rule Sequence into a CNF Rule: We begin by rewriting the rule sequence $\bar{R} = [R_1, \ldots, R_n]$ into a form that is amenable to distributed processing in subsequent steps. Specifically, we first rewrite $\bar{R}$ as a single "negative" rule P in disjunctive normal form (DNF):

$$[p_1^1(a,b) \wedge \ldots \wedge p_{m_1}^1(a,b)] \vee \ \ldots \vee [p_1^n(a,b) \wedge \ldots \wedge p_{m_n}^n(a,b)]$$
$$\rightarrow \ drop\ (a,b).$$

Then we convert this negative rule into a "positive" rule Q in conjunctive normal form (CNF):

$$[q_1^1(a,b) \vee \ldots \vee q_{m_1}^1(a,b)] \wedge \ \ldots \wedge [q_1^n(a,b) \vee \ldots \vee q_{m_n}^n(a,b)]$$
$$\rightarrow \ keep\ (a,b)\ as\ they\ may\ match,$$

where each predicate q_j^i is the complement of the corresponding predicate p_j^i in the "negative" rule P.

Example 4.7.3. *The rule sequence $[R_1, R_2]$ in Figure 4.4.a is converted into the "positive" rule Q in CNF in Figure 4.4.b.*

2. Analyze CNF Rule to Infer Index-Based Filters: Next, we analyze the CNF rule to infer index-based filters. Work on string matching has studied several such filters for similarity functions

(e.g., [95, 24]). Falcon builds on this work. It currently uses eight similarity functions (e.g., edit distance, Jaccard, overlap, cosine, exact match, etc.), and five filters. The filters are discussed in detail in Section 4.7.4.

Example 4.7.4. *Consider again rule Q in Figure 4.4.b.* Falcon *assigns three filters to predicate $jaccard\_word(a.title, b.title) > 0.6$: length filter, prefix filter, and position filter [112].* Falcon *assigns an equivalence filter to $exact\_match(a.year, b.year) = 1$. Given a tuple $b \in B$, this filter uses a hash index to find all tuples in A that have the same year as $b.year$. Finally,* Falcon *assigns a range filter to $abs\_diff(a.price, b.price) < 10$. Given a tuple $b \in B$, this filter uses a B-tree index to find all tuples in A whose prices fall into the range $(b.price - 10, b.price + 10)$.*

Once we have inferred all filters for rule Q, we execute several MapReduce (MR) jobs to build the indexes for these filters (more details in Section 4.7.5).

3. Apply the Filters to the Rule Sequence: Let $\mathcal{F}$ and $\mathcal{I}$ be the set of filters and indexes that have been constructed for rule Q, respectively. We now consider how to use MapReduce to apply $\mathcal{F}$ to $A \times B$ (without materializing $A \times B$) to find a set of tuple pairs that may match, then apply Q to these pairs. A reasonable solution is to copy the set of indexes $\mathcal{I}$ to each of the mappers, use $\mathcal{I}$ to quickly locate candidate pairs (a, b), send them to the reducers, then apply Q to these pairs.

A challenge however is that $\mathcal{I}$ (which can be as large as 3G in our experiments) may not fit into the memory of each mapper. So we propose four solutions that balance between the amount of memory available for the indexes at the mappers and the amount of work done at the reducers. Section 4.10.1 discusses how to select among these four solutions.

(a) apply-all: This solution loads the entire set of indexes $\mathcal{I}$ into the memory of each mapper, which uses $\mathcal{I}$ to locate pairs (a, b) that may match. The reducers then apply rule Q to these pairs (see the pseudo code in Algorithm 4.1).

Example 4.7.5. *Consider three mappers into whose memory we already load indexes $\mathcal{I}$ (Figure 4.4.c). We first partition table A three ways sending each partition to a mapper. We do the same for table B. Now consider Mapper 1. For each arriving tuple $a \in A$, it emits a key-value pair*

Algorithm 4.1 apply-all

Input: Tables *A* and *B*, Rule sequence $\mathcal{R}$, L: set of length indexes, O: set of token orderings, P: set of inverted indexes (on prefix tokens), H: set of hash indexes, and T: set of tree indexes
Output: Candidate tuple pairs C

map-setup: /* before running map function */
Load L, O, P, H, and T into memory
$Q \Leftarrow$ Translate $\mathcal{R}$ into a positive rule in CNF

map(K: null, V: record from a split of either A or B):
if $V \in B$ **then**
    /* $Q = q_1 \wedge q_2 ...$ where each q_i is $p_{i1} \vee p_{i2} ...$*/
    $C_Q \Leftarrow \bigcap_{q \in Q} (\bigcup_{p \in q}$ FindProbableCandidates(V, p))
    for each $a.id \in C_Q$, emit $(a.id, V)$
else /* $V \in A$ */
    emit($V.id, V$)
end if

reduce(K': $a.id$ where $a \in A$, $LIST\_V'$: contains $a \in A$ and a set of B tuples, C_B):
for each $b \in C_B$ **do**
    if (a, b) does not satisfy rule sequence $\mathcal{R}$, emit (a, b)
end for

Procedure FindProbableCandidates(b, p)
Input: $b \in B$, p: predicate of the form $sim(a.col1, b.col2)$ *op* v
Output: $C_p = \{a.id \mid a \in A, (a, b)$ passes all filters$\}$
if sim = ExactMatch **then**
    $H_p \Leftarrow$ Get hash index for p from H
    $C_p \Leftarrow$ Probe H_p with $b.col2$
else if $sim \in \{$AbsDiff, RelDiff$\}$ **then**
    $T_p \Leftarrow$ Get tree index for p from T
    $C_p \Leftarrow$ Probe T_p with range $[b.col2 - v, b.col2 + v]$
else /* $sim \in \{$Jaccard, Dice, Overlap, Cosine, Levenshtein$\}$ */
    $\{P_p, L_p, O_p\} \Leftarrow$ Get inverted index, length index and token ordering for p from P, L and O
    $l \Leftarrow$ Compute prefix length of $b.col2$ using v
    $b_l \Leftarrow$ Get prefix tokens of $b.col2$ using l and O_p
    $C_p \Leftarrow$ Probe P_p with b_l, apply position and length filters using P_p and L_p
end if
return C_p

$\langle aid, a \rangle$, where aid is the ID of a. For each arriving tuple $b \in B$, Mapper 1 applies the filters by using $\mathcal{I}$ to find a set of IDs of tuples in A that may match with b. Let these IDs be $aid_1, \ldots, aid_n$.

Then Mapper 1 emits key-value pairs $\langle aid_1, b\rangle, \ldots, \langle aid_n, b\rangle$ (we discuss below optimizations to avoid emitting multiple copies of the same tuple). The other mappers proceed similarly.

Each emitted key-value pair is sent to one of the two reducers. For example, for a particular key aid, Reducer 1 receives all key-value pairs with that key: $\langle aid, a\rangle, \langle aid, b_1\rangle, \ldots, \langle aid, b_m\rangle$ (see Figure 4.4.c). Then this reducer can apply rule Q to the pairs $(a, b_1), \ldots, (a, b_m)$.

(b) apply-greedy: loads only the indexes of the most selective conjunct of rule Q into the mappers' memory. The mappers apply only the filters of this conjunct. The reducers then apply Q to all surviving pairs. The selectivity of each conjunct in Q can be computed from the selectivity of the corresponding rule in $\bar{R}$. Section 4.11 shows how to estimate rule selectivities when we evaluate the rules on sample S.

(c) apply-conjunct: uses multiple mappers, each loading into memory only the indexes of one conjunct (of rule Q). There are at most as many mappers as the number of conjuncts (no mapper for those conjuncts whose indexes do not fit into the mappers' memory). The reducers first perform intersection on the pairs surviving various mappers, then apply Q to the pairs in the intersection.

(d) apply-predicate: is similar to $apply\_conjunct$, except that here each mapper loads the indexes of one predicate (of rule Q), and the reducers need to process the pairs surviving the mappers in a more complicated fashion (than just taking intersection as in $apply\_conjunct$).

Optimizations: We have extensively optimized the above solutions as follows:

1. **Load Balancing at Map Phase:** In the default mode some mappers process only tuples from A and some process only tuples from B. This incurs highly unbalanced loads. This is because the map tasks that process A tuples complete very fast (because the processing is trivial; see lines 13-14 of Algorithm 4.1) whereas the map tasks that process B tuples run longer (because the processing is more involved; see lines 9-12 of Algorithm 4.1). This leads to under-utilization of the resources. To address this, we have optimized so that each mapper processes both A's and B's tuples in a way that evens out the loads. Specifically, we create one combined file with interleaved A and B tuples (depending on the ratio of sizes of A and B). This combined file is then distributed on HDFS. When running the jobs, splits of this

combined file are sent to mapper nodes. As a result each mapper now has to process both A and B tuples thereby leading to better utilization of resources.

2. **Reducing Intermediate Output Size:** In the default mode, each mapper outputs the entire tuple as value. For A tuples, this is still fine because each A tuple is output only once. Each B tuple however can be output multiple times (depending upon the number of candidate A matches coming from indexes). This can sometimes result in huge intermediate output leading to increased I/O and network overhead. To prevent this, we minimize the intermediate output size, e.g., by passing only the IDs of the B tuples to the reducers, instead of passing the entire B tuples. However to do that, we have to load an index (on ID) of B tuples in memory of every Reducer node. This is needed because we need the entire B tuple when applying the rule sequence in the reducer. Due to this overhead we apply this optimization only if index for B fits in memory of a reducer node, and the selectivity of the rule sequence is above a pre-specified threshold (indicating that the intermediate output can be huge). Note that this optimization is in addition to using Hadoop's built-in feature of compressing the intermediate output.

3. **Optimizations on Rule Sequence:** We also optimize the application of the rule sequences to tuple pairs. For example, we cache and reuse computations such as $Jaccard\_word(a.title, b.title)$ (as the same rule or two different rules may refer to this), and we simplify predicate expressions such as $f < 0.5$ AND $f < 0.2$ into $f < 0.2$. We do this simplification only for predicates that have $<$, $\leq$, $>$, and $\geq$ operators. The implementation is straightforward. We parse a rule to first collect all unique features referenced in the rule. For each unique feature f we examine all its predicates of the form $f < v$ or $f \leq v$ to determine the minimal v (say v_1) and the operator ($<$ or $\leq$) associated with v_1 (say op_1). Similarly, we examine all the predicates of the form $f > v$ or $f \geq v$ to determine the maximal v (say v_2) and the operator ($>$ or $\geq$) associated with v_2 (say op_2). Finally, we replace all the predicates involving f and operators $<$, $\leq$, $>$, and $\geq$ in the rule with $f\ op_1\ v_1$ AND $f\ op_2\ v_2$.

4.7.4 Using Filters to Apply Blocking Rules

We now describe in detail the filters and indexes used in Falcon. We associate one or more filters with each predicate q_j^i in Q. A filter is a necessary (but not sufficient) condition for a tuple pair (a, b) to satisfy the predicate $q_j^i(a, b)$. In other words, if the filter does not pass (a, b) then it is guaranteed that $q_j^i(a, b)$ is not satisfied. But if the filter passes (a, b), then $q_j^i(a, b)$ must be evaluated to see if it is satisfied.

For example, if the predicate is $[Jaccard(a.x, b.y) \geq 0.6]$, then a "share-token" filter is f_1 = "a.x and b.y must share at least one token", and a "length" filter is $f_2 = length(a.x)/0.6 \geq length(b.y) \geq 0.6 * length(a.x)$.

We build on prior work [37] to come up with the various filters that can be constructed and *indexes* that can be created to quickly find tuple pairs that satisfy the filters. Below are the five filters (and the corresponding indexes) that we consider in our implementations.

1. **Equivalence Filter:** requires that "a.x" and "b.y" are equivalent for the predicate $f(a.x, b.y)$ op v to be satisfied on pair (a, b). It is implemented using a hash index on "a.x", and is used for predicates that use $exact\_match$ similarity function.

2. **Range Filter:** requires that "b.y" lie within a range of "a.x" for the predicate to be satisfied. It is implemented using a B-tree index over "a.x", and is used for predicates involving abs_diff and rel_diff.

3. **Length Filter:** requires that a constraint on the lengths of "a.x" and "b.y" be satisfied for the predicate to be satisfied. It is implemented using a length index on $length(a.x)$ (probed using $length(b.y)$), and is used for predicates involving $Jaccard$, $overlap$, $Dice$, $cosine$ and $Levenshtein$.

4. **Prefix Filter:** requires that there must be at least one shared token in the *prefixes* of "a.x" and "b.y" for the predicate to be satisfied. Note that the tokens of "a.x" and "b.y" are first re-ordered based on a global token ordering and then prefixes of the re-ordered tokens are considered. This filter is implemented using an inverted index over the prefixes of re-ordered

tokens of "a.x", and used for predicates involving $Jaccard$, $overlap$, $Dice$, $cosine$ and $Levenshtein$.

5. **Position Filter:** requires that at least a certain number of tokens be shared between the *prefixes* of "a.x" and "b.y". It is implemented using an inverted index on the prefixes of "a.x" (the same index constructed for prefix filters) and a length index (constructed for length filters). It is used for predicates involving $Jaccard$, $overlap$, $Dice$, $cosine$ and $Levenshtein$.

Since filters have been extensively used in string matching and set similarity joins, we point the reader to [37] for more details. Next we describe how we construct indexes in Falcon to implement the various filters.

4.7.5 Building Indexes for Filters in MapReduce

Once we have inferred all filters for rule Q, we run several MapReduce (MR) jobs to build the indexes for these filters. Specifically, we run three MR jobs sequentially to build all the relevant indexes for rule Q.

Before running the first MR job, Falcon first analyzes Q to determine all the unique attribute-tokenization pairs (x, T) used in Q. For example, if Q uses two features: Dice_3gram(a.title, b.title) and Jaccard_word(a.title, b.title), then there are two unique attribute-tokenization pairs: (title, word) and (title, 3gram).

For each attribute-tokenization pair (x, T):

1. The first MR job computes the frequencies of all tokens obtained by tokenizing (using T) the values of attribute x of all A tuples.

2. The second MR job sorts all the tokens obtained for that (x, T) pair in increasing order of frequencies to obtain a global token ordering for that (x, T) pair, which will be used by the next MR job to construct inverted indexes for *prefix* and *position* filters.

Attribute Type and Characteristic	Similarity Function	Intuition
Single-word string	Exact Match, Jaccard_3gram, Overlap_3gram, Dice_3gram, Levenshtein, Jaro*, Jaro-Winkler*	May be first names, last names, zip codes, etc.
Multi-word short string (#words ≤ 5)	Jaccard_3gram, Overlap_3gram, Dice_3gram, Jaccard_word, Overlap_word, Dice_word, Cosine_word, Monge-Elkan*, Needleman-Wunsch*, Smith-Waterman*, Smith-Waterman-Gotoh*	May be product brand names, full names of people, etc.
Multi-word medium string (6 ≤ #words ≤ 10)	Jaccard_word, Overlap_word, Dice_word, Cosine_word, Monge-Elkan*	May be street addresses, short product descriptions, etc.
Multi-word long string (#words ≥ 11)	Jaccard_word, Overlap_word, Dice_word, Cosine_word, TF/IDF*, Soft TF/IDF*	May be long product descriptions, product reviews, etc.
Numeric	Exact Match, Absolute Difference, Relative Difference, Levenshtein	May be age, size, weight, height, price, etc.

* Not used for blocking.

Figure 4.5: Rules for feature generation.

3. The third MR job tokenizes (using T) values of attribute x of each A tuple; reorders the tokens (using the global token ordering output by the second MR job); computes prefix length for that tuple; and indexes the prefix of the reordered tokens.

In addition to constructing inverted indexes of prefix tokens, the third MR job also simultaneously constructs the length indexes (needed for length filter), hash indexes (for equivalence filter) and B-tree indexes (for range filters). Note that each MR job scans the table A only once.

4.8 Generating Feature Vectors

Given a set S of tuple pairs and a set F of m features, this operator converts each pair $(a, b) \in S$ into a feature vector $\langle f_1(a, b), \ldots, f_m(a, b)\rangle$. Given F, converting a set of pairs into a set of feature vectors is straightforward (as we shall see shortly). The key challenge, however, is to automatically generate F. In prior EM work, selection of relevant features is usually performed by a developer. But since crowdsourced EM in Falcon must be executed in a hands-off fashion (without requiring a developer), automatically generating features F becomes a necessity. Falcon addresses this challenge by generating F based on the types (e.g., string, numeric) and characteristics (e.g., short string, long string) of the attributes of the two tables. The heuristic rules guiding the generation process are shown in Figure 4.5. We next describe generating features in detail.

Conceptually, a feature is a function that maps a tuple pair (a, b) to a numeric score. However, in Falcon we currently only consider features of the form $f(a, b) = sim(a.x, b.y)$, where sim is

a similarity function (e.g., Jaccard, edit distance), $a.x$ is the value of attribute x of tuple a (from table A), and $b.y$ is the value of attribute y of tuple b (from table B). For example, if we have inferred that attributes $A.name$ and $B.name$ are of type string, we can generate features such as $jaccard(3gram(A.name), 3gram(B.name))$, and
$edit\_dist(A.name, B.name)$, etc. To decide on the set of relevant features $F = \{f_1, ..., f_m\}$, Falcon first creates attribute correspondences between the two tables A and B by pairing string with string, numeric with numeric, etc.

Next, we scan through the tables to determine the characteristics (e.g., single-word string, multi-word long string, etc.) of every attribute. Next, for each attribute correspondence (x, y), we include in F a set of features, each of the form $sim(a.x, b.y)$ where sim is a similarity function chosen based on the rules in Figure 4.5. If x and y have different attribute characteristics, we choose the characteristic that is at a lower row in Figure 4.5.

Once we have selected $F = \{f_1, ..., f_m\}$, we want to generate the feature vector for every tuple pair (a, b) in S. This step is trivially parallelizable. We implement it as a single map-only job on a Hadoop cluster. In this job, each mapper reads in a pair (a, b) from S (residing on HDFS); computes a feature value $f_i(a, b)$ for each feature $f_i \in F$; and outputs a key-value pair where a composition of tuple IDs ($\langle a.id, b.id \rangle$) is the key, and the feature vector ($\langle f_1(a, b), ..., f_m(a, b) \rangle$) is the value.

4.9 Implementing Other Operators

We now briefly describe implementing the remaining four operators: $al\_matcher$, $get\_blocking\_rules$, $eval\_rules$, and $apply\_matcher$.

Operator al_matcher: This operator performs crowdsourced active learning on a set of pairs V. It trains an initial matcher M, uses M to select a small set of controversial pairs from V (currently set to 20), asks the crowd to label these pairs, uses them to improve M, and so on. This operator was described in Chapter 2 and is straightforward to implement. We made only two small changes to the implementation in Corleone. First, we execute pair selection on Hadoop, as it can be time

consuming (Section 4.10.2 shows further optimizations of this step). Second, Corleone performs active learning until a convergence criterion T is met. Falcon however stops active learning when either T is met *or* the number of iterations reaches a pre-specified threshold (currently set to 30). Our experiments show that this limit has negligible effects on the accuracy yet can significantly reduce the crowd time and cost.

Operator get_blocking_rules: This operator extracts candidate blocking rules from a random forest. It is trivial to implement on a single machine.

Operator eval_rules: This operator uses crowdsourcing to evaluate and retain only the most precise rules. Chapter 2 describes it and is straightforward to implement. Similar to $al\_matcher$, it also operates in iterations. Thus we also impose a threshold on the maximal number of iterations, capping the crowd time and cost. Note that we use the same crowdsourcing strategies as Corleone uses for $al\_matcher$ and $eval\_rules$ (see Chapter 2).

Operator apply_matcher: This operator applies a trained classifier to each tuple pair (encoded as a feature vector) in a set C to predict match/no-match. It is highly parallelizable and is implemented as a map-only job on Hadoop.

4.10 Plan Generation, Execution, and Optimization

4.10.1 Plan Generation and Execution

Given two tables A and B (to be matched), we generate a plan p as follows. First, we analyze A and B to automatically generate a set of features F (see Section 4.8). Later, operators such as gen_fvs (Section 4.8) will need these features (e.g., to convert tuple pairs into feature vectors).

Next, we estimate the size of $A \times B$, where each pair is encoded as a feature vector (using the features in F). If this size does not fit in the memory of machine nodes, then blocking is likely to be necessary, so we generate the plan in Figure 4.3.a. Otherwise we generate the plan in Figure 4.3.b. (Since we are currently using a rule-based optimization approach, this is just a heuristic rule encoding the intuition that in such cases the plan in Figure 4.3.b can do everything solely in

memory, and hence will be faster. In the future we will consider a cost-based approach that selects the plan with the estimated lower runtime.)

Next, we replace each logical operator in p except operator $apply\_blocking\_rules$ with a physical operator. Currently each such logical operator has just a single physical operator, so these replacements are straightforward. We cannot yet replace $apply\_blocking\_rules$ because this logical operator has six physical operators: four provided by us (e.g., $apply\_all$, $apply\_greedy$, etc.) and two from prior work: $MapSide$ and $ReduceSplit$ (Section 4.7). Selecting the appropriate physical operator requires knowing the index sizes and the rule sequence $\bar{R}$, which are unknown at this point.

So in the next step we execute all operators in p from the start up to (and including) the operator right before $apply\_blocking\_rules$. This produces the rule sequence $\bar{R}$. Next, we convert it into a single positive rule Q, then infer filters and build indexes for Q, as described in Section 4.7.

Once index building is done, we select a physical operator for $apply\_blocking\_rules$, i.e., select among the six methods $apply\_all$, $apply\_greedy$, etc. as follows.

First, let c be the most selective conjunct in rule Q. Let $sel(c)$ and $sel(Q)$ be the selectivities of c and Q, respectively (see Section 4.11 on computing such selectivities). Clearly $sel(c) \geq sel(Q)$. If $sel(Q)/sel(c)$ exceeds a threshold (currently set to 0.8), then intuitively c is almost as selective as the entire rule Q. In this case, we will select $apply\_greedy$.

Otherwise we proceed in this order (a) if the indexes for all conjuncts fit in memory (of a mapper) then select $apply\_all$; (b) if the indexes of at least one conjunct fit in memory then select $apply\_conjunct$; (c) if the indexes of each predicate fit in memory then select $apply\_predicate$; (d) if the smaller table fits in memory then select $MapSide$, else select $ReduceSplit$.

After selecting a physical operator for $apply\_blocking\_rules$, we execute it, then execute the rest of plan p. Of course, if p does not involve blocking, then we do not have to deal with the above issues, and plan execution is straightforward.

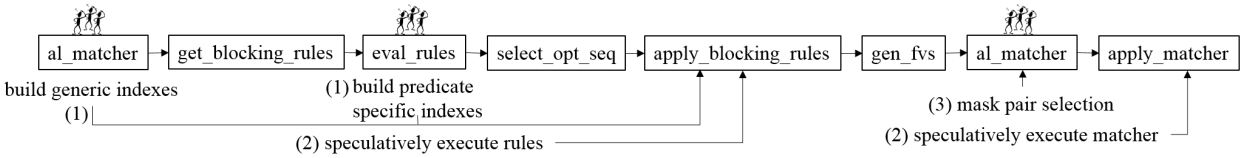


Figure 4.6: Three types of optimization solutions that use crowd time to mask machine time.

4.10.2 Plan Optimization

We now consider how to optimize plan p. The Falcon framework raises many interesting optimization opportunities regarding time, accuracy, and cost. As a first step, Falcon will focus on a kind of optimization called "using crowd time to mask machine time".

To explain, observe that plan p currently executes machine and crowd activities *sequentially*, with no overlap. For example, $eval\_rules$ uses the crowd to evaluate blocking rules. Only after this has been done would $select\_opt\_seq$ and $apply\_blocking\_rules$ start, which execute machine activities on a Hadoop cluster. Thus this cluster is idle during $eval\_rules$. This clearly raises an opportunity: while $eval\_rules$ is performing crowdsourcing, if we can do some useful machine activities on the idle cluster, we may be able to reduce the total run time. To mask machine time, we have developed three solutions, marked with (1), (2), and (3) respectively in Figure 4.6.

- Solution (1) uses the crowd time in $al\_matcher$ and $eval\_rules$ to build indexes for $apply\_blocking\_rules$.
- Solution (2) speculatively executes rules and matchers for $apply\_blocking\_rules$ and $apply\_matcher$.
- The above solutions are inter-operator optimizations. Solution (3) in contrast is an intra-operator optimization for $al\_matcher$. It interleaves "selecting pairs for labeling" with "crowdsourcing to label the pairs". As such, it learns an approximate matcher but drastically cuts down on pair selection time.

We now describe these solutions.

1. Building Indexes for apply_blocking_rules: Recall that $apply\_blocking\_rules$ must build indexes for filters. There are two earlier operators in the plan pipeline, $al\_matcher$ and $eval\_rules$, where crowdsourcing is done and the Hadoop cluster is idle. So we will move as much index building activities to these two operators as possible.

In particular, while $al\_matcher$ crowdsources, we still do not know the rules that $apply\_blocking\_rules$ will ultimately apply. So we use the Hadoop cluster to build only generic indexes that do not depend on knowing these rules, e.g., hash and range indexes for numeric and categorical attributes, and global token orderings for string attributes. This ordering will be required if later we decide to build indexes for prefix and position filters [112].

After $al\_matcher$ has finished crowdsourcing, it outputs a matcher M. $get\_blocking\_rules$ then extracts blocking rules from M. Next, $eval\_rules$ ranks then evaluates the top 20 rules using crowdsourcing. So while $eval\_rules$ crowdsources, we already know that the rules $apply\_blocking\_rules$ ultimately uses will come from this set of 20 rules. So we use the Hadoop cluster to build indexes for all predicates in all 20 rules (or for as many predicates as we can). Clearly, some of these indexes may not be used in $apply\_blocking\_rules$. But if some are used, then we have saved time.

2. Speculative Execution of Future Operations: Recall that $eval\_rules$ uses crowdsourcing to evaluate 20 rules and retain only the best ones. Then $select\_opt\_seq$ examines these rules to output an optimal rule sequence $\bar{R}$, which $apply\_blocking\_rules$ will execute.

While $eval\_rules$ crowdsources the evaluation of the 20 rules, we use the idle Hadoop cluster to speculatively execute these 20 rules (in practice we use the cluster to build indexes first, then to speculatively execute the rules). If later it turns out $\bar{R}$ contains at least one rule that has been executed, then we can reuse the result, saving significant time.

Specifically, we execute the 20 rules individually, in the order that $eval\_rules$ crowdsources (i.e., executing the most promising rules first). When $eval\_rules$ finishes, $select\_opt\_seq$ takes over and outputs an optimal rule sequence $\bar{R}$, say $[R_2, R_1, R_3]$. At this point we start executing $apply\_blocking\_rules$ as usual, but modify it to use the speculative execution results as follows. Suppose the output of one or more rules in $\bar{R}$ has been generated. Then we pick the smallest

output then apply the remaining rules to it in a map-only job. For example, suppose that the outputs $O(R_1), O(R_3)$ of rules R_1, R_3 have been generated, and that $O(R_3)$ is the smallest output. Then we apply the sequence $[R_2, R_1]$ to $O(R_3)$.

Now suppose none of the outputs of the rules in $\bar{R}$ has been generated, but we are still in the middle of running a MapReduce (MR) job to execute a rule in $\bar{R}$. Then reusing becomes quite complex, as we want to keep the MR job running, but tell it that the rule sequence $\bar{R}$ has been selected, so that it can figure out how to execute $\bar{R}$ while reusing whatever partial results it has obtained so far.

Specifically, if the MR job is still in the map stage, then a reasonable strategy is to let the mappers complete, then tell the reducers to use $\bar{R}$ to evaluate the tuple pairs. This strategy resembles $apply\_greedy$. Thus, we use it if operator $apply\_blocking\_rules$ has selected $apply\_greedy$ as the rule execution strategy. Otherwise, $apply\_greedy$ has not been selected, suggesting that similar strategies may also not work well. In this case we kill the MR job and start $apply\_blocking\_rules$ as usual.

Now if the MR job is in the reduce stage, then it has already produced some part X of the output of a rule, say R_1. We then communicate the rule sequence $\bar{R}$, say $[R_2, R_1, R_3]$, to the reducers, so that for new incoming tuple pairs, the reducers can apply $\bar{R}$ and collect the output into a set of files Y. We then run a map-only job to apply $[R_2, R_3]$ to X to obtain a set of files Z. The sets Y and Z contain the desired tuple pairs (i.e., the correct output of $apply\_blocking\_rules$).

Finally, if none of the outputs of rules in $\bar{R}$ has been generated, and none of these rules is currently being executed, then we simply start $apply\_blocking\_rules$ as usual. See Algorithm 4.2 for the pseudo-code of this optimization.

In addition to speculatively executing blocking rules (as described above), we speculatively execute matchers (in the matching phase). Recall that al_matcher trains a new matcher in each iteration of crowdsourced active learning. When it decides to stop, it outputs the "best" matcher so far, which is then applied to the candidate set of tuple pairs by the apply_matcher operator. While al_matcher crowdsources, the Hadoop cluster is idle and can potentially be used to apply a matcher to the candidate set. So in this optimization, we speculatively execute the apply_matcher

Algorithm 4.2 Speculative Rule Execution

Input: Tables A and B, a sorted list of blocking rules $[R_1, R_2, ..., R_k]$
Output: Candidate tuple pairs C
for $1 \leq i \leq k$ **do**
    $job_i \Leftarrow$ Start execution of rule R_i on tables A and B
    Wait till eval_rules is complete or job_i is complete
    if eval_rules is complete **then**
        $\mathcal{R} \Leftarrow$ Get the optimal sequence
        if any of the rules $R_1, ..., R_{i-1}$ is in $\mathcal{R}$ **then**
            Kill job_i
            $m \Leftarrow argmin_{j \in [1,i-1]} |C_j|$
            $C \Leftarrow$ Apply $\mathcal{R} - R_m$ to C_m in a map-only job
        else if $R_i \in \mathcal{R}$ **then**
            if job_i is in map phase **then**
                if apply-greedy is chosen for $\mathcal{R}$ and R_i is most selective in $\mathcal{R}$ **then**
                    Pass $\mathcal{R}$ to job_i and wait for job_i to complete
                    $C \Leftarrow$ Fetch output of job_i
                else
                    Kill job_i
                    $C \Leftarrow$ Apply $\mathcal{R}$ to A and B
                end if
            else /* job_i is in reduce phase*/
                Pass $\mathcal{R}$ to the reducer of job_i. job_i will now output tuple pairs to a different file. Let job_i complete.
                $C_1 \Leftarrow$ Fetch output of job_i before passing $\mathcal{R}$
                $C_2 \Leftarrow$ Fetch output of job_i after passing $\mathcal{R}$
                $C_1^{'} \Leftarrow$ Apply $\mathcal{R} - R_i$ to C_1 in a map-only job
                $C \Leftarrow C_1^{'} \cup C_2$
            end if
        else /* None of the rules $R_1, ...R_i$ is in $\mathcal{R}$ */
            Kill job_i
            $C \Leftarrow$ Apply $\mathcal{R}$ to A and B
        end if
        return C
    end if
end for
/* We have speculatively executed all rules */
Wait till eval_rules is complete
$m \Leftarrow argmin_{j \in [1,k]} |C_j|$
$C \Leftarrow$ Apply $\mathcal{R} - R_m$ to C_m in a map-only job
return C

operator with the "best" matcher so far (while al_matcher is crowdsourcing). If the speculatively executed matcher happens to be the final matcher output by al_matcher then we would have saved the apply_matcher run time. If not, we simply execute apply_matcher as usual.

3. Masking Pair Selection in al_matcher: Recall that after $apply\_blocking\_rules$ has applied a rule sequence $\bar{R}$ to Tables A and B to obtain a set of candidate tuple pairs C, we convert C into a set of feature vectors C', then use $al\_matcher$ to "active learn" a matcher on C'.

Specifically, $al\_matcher$ iterates. In each iteration it (a) applies the matcher learned so far to C' and uses this result to select 20 "most controversial" pairs from C', (b) uses crowdsourcing to label these pairs, then (c) adds the labeled pairs to the training data and retrains the matcher.

It turns out that when C' is large (e.g., more than 50M pairs), Step (a) can take a long time, e.g., 2 minutes per iteration in our experiments; if $al\_matcher$ takes 30 iterations, this incurs 60 minutes, a significant amount of time. Consequently, we examine how to minimize the run time of Step (a). One idea is to do Step (a) during the time allotted to crowdsourcing of Step (b). The problem, however, is that Step (b) depends on Step (a): without knowing the 20 selected pairs, we do not know what to label in Step (b).

To address this seemingly insurmountable problem, we propose the following solution. In the first iteration, we select not 20, but 40 tuple pairs. Then we send 20 pairs to the crowd to be labeled, as usual, keeping the remaining 20 pairs for the next batch. When we get back the 20 pairs labeled by the crowd, we immediately send the remaining 20 pairs for labeling. During the labeling time we use the 20 pairs already labeled to retrain the matcher and select the next batch of 20 pairs, and so on.

Thus the above solution masks the pair selection time using the pair labeling time. It approximates the original physical implementation of $al\_matcher$ since it may not learn the same matcher (because it selects 40 pairs in the first iteration, instead of 20). Our experiments however show that this loss is negligible, e.g., both matcher versions achieve 99.61% F_1 accuracy on the Songs data set, yet the optimized version drastically reduces pair selection time, from 58m 32s to 2m 5s (see Section 4.11).

Dataset	Table A	Table B	# of Correct Matches
Products	2,554	22,074	1,154
Songs	1,000,000	1,000,000	1,292,023
Citations	1,823,978	2,512,927	558,787

Table 4.1: Data sets for our experiments.

Products: A(url,brand,modelno,groupname,title,price,descr,image_url, shipweight)
B(url,brand,modelno,cat1,cat2,pcategory,title,price,features,image_url,shipweight)
Songs(title,release,artist_name,duration,artist_familiarity,artist_hotness,year)
Citations: A(title,authors,journal,month,year,pub_type)
B(title,authors,journal,month,year,pub_type)

Figure 4.7: The schemas of the data sets.

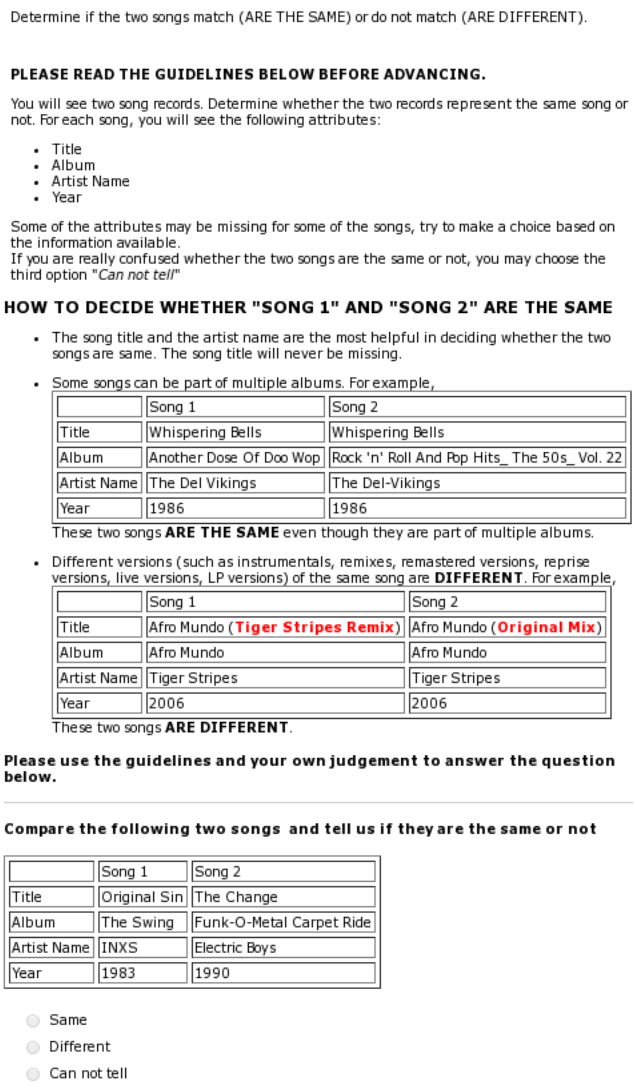
We use the above optimization for $al\_matcher$ in the matching stage, when it is applied to a large set of pairs (at least 50M in the current Falcon). We do not use it for $al\_matcher$ in the blocking stage as this operator is applied to a relatively small sample of 1M tuple pairs, incurring little pair selection time.

4.11 Empirical Evaluation

We now empirically evaluate Falcon. We consider three real-world data sets in Table 4.1. Products describes electronic products and was used in Corleone. Songs describes songs within a single table and was obtained from the freely available Million Song Dataset[1]. Citations describes citations in Citeseer and DBLP (see Figure 4.7 for the schemas). Songs and Citations have 1-2.5M tuples in each table, and are far larger than those used in crowdsourced EM experiments so far. We have made all three data sets publicly available at [32].

Falcon applied the procedure described in Section 4.8 to generate features for these data sets. Overall, it generated 50/83 features for Products, 20/47 features for Songs, and 22/30 features for Citations. "50/83" for example means that 50 and 83 features were generated for the blocking and matching steps, respectively. Note that only features involving relatively fast string similarity measures were generated for the blocking step (see Figure 4.5 for the list of string similarity measures).

[1] labrosa.ee.columbia.edu/millionsong



Determine if the two songs match (ARE THE SAME) or do not match (ARE DIFFERENT).

PLEASE READ THE GUIDELINES BELOW BEFORE ADVANCING.

You will see two song records. Determine whether the two records represent the same song or not. For each song, you will see the following attributes:

- Title
- Album
- Artist Name
- Year

Some of the attributes may be missing for some of the songs, try to make a choice based on the information available.
If you are really confused whether the two songs are the same or not, you may choose the third option "*Can not tell*"

HOW TO DECIDE WHETHER "SONG 1" AND "SONG 2" ARE THE SAME

- The song title and the artist name are the most helpful in deciding whether the two songs are same. The song title will never be missing.
- Some songs can be part of multiple albums. For example,

	Song 1	Song 2
Title	Whispering Bells	Whispering Bells
Album	Another Dose Of Doo Wop	Rock 'n' Roll And Pop Hits_ The 50s_ Vol. 22
Artist Name	The Del Vikings	The Del-Vikings
Year	1986	1986

These two songs **ARE THE SAME** even though they are part of multiple albums.

- Different versions (such as instrumentals, remixes, remastered versions, reprise versions, live versions, LP versions) of the same song are **DIFFERENT**. For example,

	Song 1	Song 2
Title	Afro Mundo (**Tiger Stripes Remix**)	Afro Mundo (**Original Mix**)
Album	Afro Mundo	Afro Mundo
Artist Name	Tiger Stripes	Tiger Stripes
Year	2006	2006

These two songs **ARE DIFFERENT**.

Please use the guidelines and your own judgement to answer the question below.

Compare the following two songs and tell us if they are the same or not

	Song 1	Song 2
Title	Original Sin	The Change
Album	The Swing	Funk-O-Metal Carpet Ride
Artist Name	INXS	Electric Boys
Year	1983	1990

Same
Different
Can not tell

Figure 4.8: Screenshot of a task to the crowd.

We used Mechanical Turk and ran Falcon on each data set three times, paying 2 cents per answer. In each run we used common turker qualifications to avoid spammers, such as allowing only turkers with at least 100 approved HITs and 95% approval rate. Figure 4.8 shows a screenshot of a task presented to the crowd on the Songs data set. The first half of the screen shows instructions to the crowd and the bottom half asks the crowd if two given song tuples match. A task contains 10 such tuple pairs. On Mechanical Turk workers prefer tasks that contain multiple pairs since it

Dataset	Accuracy (%)			Cost (# Questions)	Run Time			Candidate Set Size
	P	R	F_1		Machine Time	Crowd Time	Total Time	
Products	90.9	74.5	81.9	$57.6 (960)	52m	13h 7m	13h 25m	536K - 11.4M
Songs	96.0	99.3	97.6	$54.0 (900)	2h 7m	11h 25m	11h 58m	1.6M - 51.4M
Citations	92.0	98.5	95.2	$65.5 (1087)	2h 32m	13h 33m	14h 37m	654K - 1.06M

Table 4.2: Overall performance of Falcon on the data sets. Each row is averaged over three runs.

reduces the overhead. Thus, we present 10 pairs per task, following the exact same crowdsourcing procedure of Corleone.

We ran Hadoop on a 10-node cluster, where each node has an 8-core Intel Xeon E5-2450 2.1GHz processor and 8GB of RAM.

In addition to the above three data sets, we have recently successfully deployed Falcon to solve a real-world drug matching problem at a major medical research center. We will briefly report on that experience as well.

4.11.1 Overall Performance

We begin by examining the overall performance of Falcon. The first few columns of Table 4.2 show that Falcon achieves high accuracy, 81.9% F_1 on Products and 95.2-97.6% F_1 on Songs and Citations. Products is a difficult data set used in Corleone, and the accuracy 81.9% here is comparable to the accuracy of Corleone (86% F_1 after the first iteration, see Table 2.4). Note that each row of Table 4.2 is averaged over three runs. (Table 4.3 shows all nine runs. The results show that while the candidate set size can vary across runs, affecting the machine and crowd time, the cost and the F_1 accuracy stay relatively stable.)

The next column, labeled "Cost", shows that this accuracy is achieved at a reasonable cost of $54 - 65.5 (the numbers in parentheses show the number of questions to the crowd).

The next two columns show the total machine time and crowd time, respectively. Crowd time on Mechanical Turk is somewhat high (11h 25m - 13h 33m), underscoring the need for future work to focus on how to minimize crowd time. Machine time is comparatively lower, but is still substantial (52m - 2h 32m).

Dataset	Runs	Accuracy (%)			Cost (# Questions)	Run Time			Candidate Set Size
		P	R	F_1		Machine Time	Crowd Time	Total Time	
Products	Run 1	92.6	74.9	82.8	$61.2 (1020)	31m 52s	12h 45m 22s	13h 1m 23s	536K
Products	Run 2	88.4	75.1	81.2	$58.8 (980)	56m 9s	13h 57s	13h 18m 41s	5.3M
Products	Run 3	91.8	73.4	81.6	$52.8 (880)	1h 6m 32s	13h 35m 57s	13h 56m 3s	11.4M
Songs	Run 1	90.9	99.7	95.1	$56.4 (940)	3h 54m 4s	11h 59m 39s	12h 38m 55s	51.4M
Songs	Run 2	98.2	99.6	98.9	$55.2 (920)	1h 23m 5s	11h 44m 36s	12h 18m	15.9M
Songs	Run 3	98.9	98.7	98.8	$50.4 (840)	1h 4m 1s	10h 30m 4s	10h 57m 8s	1.6M
Citations	Run 1	92.4	99.6	95.9	$52.8 (880)	1h 49m 18s	9h 59m 8s	10h 38m 26s	654K
Citations	Run 2	93.4	96.8	95.1	$66.8 (1100)	3h 6m 12s	15h 48m	16h 27m 46s	835K
Citations	Run 3	90.2	99.2	94.5	$76.8 (1280)	2h 40m 54s	14h 51m 47s	16h 44m 31s	1.06M

Table 4.3: All runs of Falcon on the data sets.

The next column, labeled "Total Time", shows the total run time of 11h 58m - 14h 37m. This time is often less than the sum of machine time and crowd time, e.g., the Songs data set incurs a "machine time" of 2h 7m and a "crowd time" of 11h 25m; yet it incurs a "total run time" of only 11h 58m. This is because plan optimization was effective, masking parts of the machine time by executing them during the crowd time (see more below).

The last column shows the number of tuple pairs surviving blocking: 536K - 51.4M. This number varies a lot, both within and across data sets. Yet despite such drastic swings, we have observed that Falcon stays relatively stable in terms of accuracy and cost (see Table 4.3).

Drug Matching: Recently we have successfully deployed Falcon to match drug descriptions across two tables for a major medical research center. The two tables are derivatives of the First Databank's National Drug Data File (NDDF). The tables have 453K and 451K tuples. They have identical schema comprising eight attributes: (drug_id, drug, drug_name, generic_name, therapeutic_ahfs_id, therapeutic_standard_desc, ahfs_specific_category_desc, ndc_code).

Two drug products are said to match when they contain the same active ingredients or when one product contains *all* of the active ingredients of the other. Active ingredients of a drug product are usually present in the "generic_name" attribute value. For example, consider two drug products a and b with generic names "LAMIVUDINE" and "LAMIVUDINE/ZIDOVUDINE", respectively. a contains only one active ingredient "LAMIVUDINE". b contains two active ingredients: "LAMIVUDINE" and "ZIDOVUDINE". Thus, b contains all the active ingredients of a.

Attribute	Drug Product 1	Drug Product 2
drug_id	530459	521837
drug	00378516893 LAMIVUDINE	55700009604 LAMIVUDINE-ZIDOVUDINE
drug_name	LAMIVUDINE	LAMIVUDINE-ZIDOVUDINE
generic_name	LAMIVUDINE	LAMIVUDINE/ZIDOVUDINE
therapeutic_ahfs_id	8180820	8180820
therapeutic_standard_desc	Antivirals	Antivirals
ahfs_specific_category_desc	Hiv Nucleoside & Nucleotide Rt Inhibitor	Hiv Nucleoside & Nucleotide Rt Inhibitor
ndc_code	378516893	55700009604

Figure 4.9: An example of a matching pair of drug products.

Hence by the above match definition drug products a and b match. Figure 4.9 shows this example matching pair.

As another example, consider two drug products "HYDROCODONE/ACETAMINOPHEN" and "OXYCODONE/ACETAMINOPHEN". Here the first product contains "HYDROCODONE" and "ACETAMINOPHEN", whereas the second product contains "OXYCODONE" and "ACETAMINOPHEN". Thus, the two products do not contain the same active ingredients (since "HYDROCODONE" and "OXYCODONE" are not the same) and neither product contains all the active ingredients of the other. Hence by the above match definition the products do not match. Figure 4.10 shows this example non-matching pair.

Further, for two active ingredients to match, in general the first words must be the same but the second words may be different (e.g., "diclofenac sodium" and "diclofenac potassium" match). However, two active ingredients that have similar names may be very different and hence do not match (e.g., "fluoxetine" and "paroxetine" both share "oxetine" but do not match).

For privacy reasons we could not use Mechanical Turk. So an in-house scientist labeled the data, effectively forming a crowd of 1 person. The scientist labeled 830 tuple pairs, incurring a crowd time of 1h 37m. Machine time was 2h 10m, constituting a significant portion (57%) of the total run time. Our optimizations reduced this machine time by 49%, to 1h 6m, resulting in a total Falcon time of 2h 42m. The end result is 4.3M matches, with 99.18% precision and 95.29% recall on a set-aside sample.

Attribute	Drug Product 1	Drug Product 2
drug_id	4144	4441
drug	00044072503 VICODIN HP	00045052660 TYLOX
drug_name	VICODIN HP	TYLOX
generic_name	HYDROCODONE/ACETAMINOPHEN	OXYCODONE/ACETAMINOPHEN
therapeutic_ahfs_id	28080800	28080800
therapeutic_standard_desc	Narcotic Analgesics	Narcotic Analgesics
ahfs_specific_category_desc	Opiate Agonists	Opiate Agonists
ndc_code	44072503	45052660

Figure 4.10: An example of a non-matching pair of drug products.

Discussion: The results suggest that Falcon can crowdsource the matching of very large tables (of 1M-2.5M tuples each) with high accuracy, low cost, and reasonable run time. In particular, the run times 11h 58m - 14h 37m suggest that Falcon can match large tables overnight, a time frame already acceptable for many real-world applications. But there is clearly room for improvement, especially for crowdsourcing time on Mechanical Turk (11h 25m - 13h 33m).

It is also important to note that crowd time can vary widely, depending on the platform. For instance, many companies have in-house dedicated crowd workers (often as contractors) or use platforms such as Samasource and WorkFusion that can provide dedicated crowds. Many applications with sensitive data (e.g., drug matching) will use a "crowd" of one or a few in-house experts. In such cases, the crowd time can be significantly less than that on Mechanical Turk. As a result, machine time can form a significant portion of the total run time, thus requiring optimization.

4.11.2 Performance of the Components

We now "zoom in" to examine the major components of Falcon. Recall that we run Falcon three times on each data set. Table 4.4 shows the time of the first run on each data set, broken down by operator.

Table 4.4 shows that five "machine" operators: $sample\_pairs$, gen_fvs, get_block_rules, sel_opt_seq, and $apply\_matcher$, finish in seconds or minutes, suggesting that they have been successfully optimized. The remaining three operators: the two "crowd" operators, $al\_matcher$

Dataset	sample_ pairs	gen_ fvs	al_matcher	get_block_ rules	eval_rules	sel_opt_ seq	apply_block_ rules	gen_ fvs	al_matcher	apply_ matcher
Products	1m 15s	34s	8h 14m 37s	2m 9s	46m 46s	130ms	0 (1m 53s)	49s	3h 54m 40s	33s
Songs	1m 29s	33s	5h 21m 29s	30s	1h 48m 19s	52ms	0 (5m 7s)	13m 5s	5h 12m 9s (6h 40m 34s)	1m 21s
Citations	2m 23s	36s	2h 23m 12s	45s	1h 10m	144ms	7m (1h 13m 20s)	55s	6h 53m	35s

Table 4.4: Falcon's runtimes per operator on the data sets. Each row refers to the first run of each data set.

and $eval\_rules$, and the "machine" operator $apply\_block\_rules$ are the most time-consuming. In what follows we will now zoom in on the major operators described in detail in this chapter.

Operator sample_pairs: Recall that we run Falcon three times on each data set. Table 4.4 shows the time of the first run on each data set, broken down by operator. Column "sample_pairs" of this table shows that sampling is very fast, taking just 1m 15s - 2m 23s. The candidate sets in the last column of Table 4.2 contain tuple pairs surviving blocking. These sets are just 0.01-0.95% of the size of $A \times B$, and retain 98.09-99.99% of matching pairs. These results suggest that our sampling solution is fast and effective, in that it helps Falcon learn very good blocking rules.

Operators al_matcher & eval_rules: The first "al_matcher" column of Table 4.4 shows that the time we learn a matcher via active learning in the blocking step is quite significant, 2h 23m - 8h 14m, due mainly to crowdsourcing. Similarly, column "eval_rules" shows a high rule evaluation time of 46m - 1h 48m, also due to crowdsourcing. This raises an opportunity for masking machine time, which we successfully exploit. For example, column "apply_block_rules" show in parentheses the unoptimized time of $apply\_blocking\_rules$: 1m 53s - 1h 13m 20s, which in certain cases is quite significant. Masking optimization however successfully reduced these times to just 0 - 7m (the numbers outside parentheses).

The second "al_matcher" column of Table 4.4 shows that the time we learn a matcher in the matching step is also quite significant, due partly to crowdsourcing and partly to pair selection (see Section 4.10.2). Pair selection however was successfully optimized. For example, for Songs the unoptimized "al_matcher" time is 6h 40m 34s (the number in parentheses). Pair selection optimization reduced this to 5h 12m 9s (almost all of which is crowdsourcing time).

Dataset	Optimal Rule Sequence			All Rules			Top-1 Rule			Top-3 Rules		
	Run Time	|C|	Recall (%)	Run Time	|C|	Recall (%)	Run Time	|C|	Recall (%)	Run Time	|C|	Recall (%)
Products	1m 53s	536,370	98.09	2m	429,458	94.37	1m 53s	536,370	98.09	1m 56s	495,744	97.40
Songs	5m 7s	51,456,003	99.99	27m 54s	43,184,642	99.52	25m 7s	52,064,416	99.99	27m 57s	50,917,386	99.99
Citations	1h 12m	654,603	99.67	1h 14m	633,015	98.97	1h 9m	1,587,707	99.99	1h 11m	1,585,691	99.99

Figure 4.11: Comparison of optimal rule sequence with other rule sequences.

Operator sel_opt_seq: Column "sel_opt_seq" of Table 4.4 shows that selecting the optimal rule sequence is very fast, taking milliseconds. To measure the impact of the optimal sequence $\bar{R}$, we examine what happens if instead of executing $\bar{R}$, we execute all the rules (deemed precise by $eval\_rules$), top-1 rule, or top-3 rules (in the order produced by $eval\_rules$). Figure 4.11 shows a detailed comparison of the optimal rule sequence with other rule sequences for the first runs on each data set.

To summarize, for all three data sets, executing $\bar{R}$ produces (a) the highest recall (e.g., on Products and Songs), or recall within 0.3% of the highest recall (e.g., on Citations), (b) the lowest run time or run time within 4% of the lowest run time, and (c) the second smallest candidate set or candidate set of size within 25% of that of the smallest candidate set. These results suggest that different rule sequences can vary significantly in their effects, and that we successfully select a good rule sequence.

Operator apply_blocking_rules: The numbers in parentheses in column "apply_block_rules" of Table 4.4 show that this operator takes 1m 53s - 1h 13m 20s on three data sets, suggesting that our Hadoop-based solution was able to scale up to large tables. Masking optimization successfully reduced this time further, to just 0 - 7m, as shown in the same column (outside the parentheses).

For this operator, recall that we provided four solutions, $apply\_all$ (AA), $apply\_greedy$ (AG), $apply\_conjunct$ (AC), and $apply\_predicate$ (AP), as well as rules on when to select which solution. In addition, we also supplied two Hadoop-based solutions from prior work: $MapSide$ and $ReduceSplit$ [65]. We now examine the performance of these six solutions. Recall that we ran Falcon three times on each data set, resulting in nine runs. In all runs except two Falcon correctly selected the best solution (i.e., the one with lowest run time). For example, on a run of Songs,

Dataset	U	O	Reduction	$O - O_1$	$O - O_2$	$O - O_3$
Products	18m	16m	11%	17m	17m	16m
Songs	2h 12m	39m	70%	40m	43m	2h 7m
Citations	1h 46m	40m	62%	41m	1h 45m	40m

Table 4.5: Effect of optimizations on machine time.

the times for AA, AG, AC, and AP are 10m 19s, 1h 3m, 1h 40m, and 1h 45m, respectively, and Falcon correctly picked AA to run. ($MapSide$ and $ReduceSplit$ did not complete on this data set.)

In all nine runs, the best solution was either AA (4 times), AG (3 times), or $MapSide$ (2 times). Solutions $MapSide$ and $ReduceSplit$ only worked on Products, the smallest data set. For Songs and Citations they had to be killed as they took forever trying to enumerate $A \times B$.

For these nine runs, each mapper has 2G of memory, sufficiently large for AA and AG to work. When we reduced the amount of memory to 1G and 500M, AA, AG, and AC did not work on Songs and Citations because there was not enough memory to load the required indexes, but AP worked well (AC did not appear to dominate in any experiment).

Overall, the results here suggest that (a) the solutions can vary drastically in their run times, (b) Falcon often selected the best solution, which is AA, AG, or AP depending on the amount of available memory, and (c) prior solutions do not scale as they enumerate $A \times B$.

4.11.3 Effectiveness of Optimization

Recall that our goal is to minimize the machine time beyond the crowdsourcing time (i.e., the machine time that cannot be masked). Column "U" of Table 4.5 shows this unoptimized time, 18m - 2h 12m, for the first run of each data set (recall that we run Falcon three times with real crowd for each data set; the results are similar for other runs, and we only show the first runs for space reasons). Column "O" shows the optimized time 16m - 40m, a significant reduction, ranging from 11% to 70% (see Column "Reduction"). This result suggests that the current optimization techniques of Falcon are highly effective.

The next three columns show the run time when we turned off each type of optimization: index building (O_1), speculative execution (O_2), and masking pair selection (O_3). The result shows that

all three optimization types are useful, and that the effects of some are quite significant (e.g., O_2 on Citations and O_3 on Songs).

4.11.4 Sensitivity Analysis

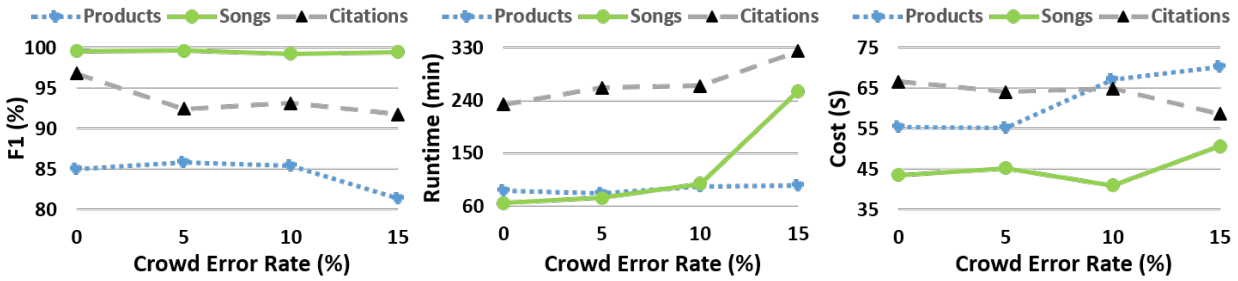


Figure 4.12: Effect of crowd error rate on F_1, runtime, and cost.

We now examine the main factors affecting Falcon's performance.

Error Rate of the Crowd: First we examine how varying crowd error rates affect Falcon. To do this, we use the random worker model in Corleone to simulate a crowd of random workers with a fixed error rate (i.e., the probability of incorrectly labeling a pair) [54]. Figure 4.12 shows F_1, run time, and cost vs. the error rate (the results are averaged over three runs). We can see that as error rate increases from 0 to 15%, F_1 decreases and run time increases, but either minimally or gracefully. Interestingly there is no clear trend on cost. This is because in some cases (e.g., when the error rate is high), active learning converged early, thereby saving crowdsourcing costs. In any case, recall that there is a cap on crowdsourcing cost (Section 4.3.4) and the costs in Figure 4.12 remain well below that cap.

Size of the Tables: So far we have shown that Falcon achieved good performance on tables of size 1-2.5M tuples. We now examine how this performance changes as we vary the table size. Figure 4.13 shows F_1, run time, cost as we run Falcon on 25%, 50%, 75%, and 100% of Songs and Citations (using simulated crowd with 5% error rate and 1.5m latency per a 10-question HIT; each data point is averaged over 3 runs). The results show that as table size increases, (a) F_1 remains

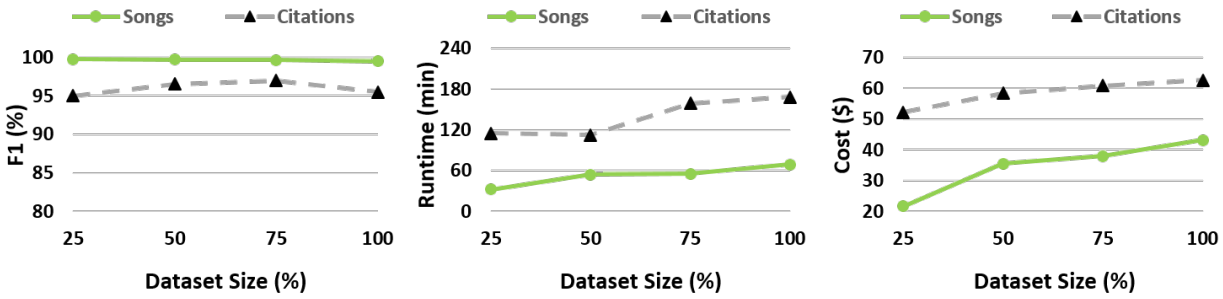


Figure 4.13: Performance of Falcon across varying sizes of Songs and Citations data.

stable or fluctuates in a small range. (b) run time increases sublinearly, and (c) cost increases sublinearly (recall that cost will not exceed the cap).

Additional Experiments: As we varied the Hadoop cluster size from 5 to 20 nodes, we found that the machine time of Falcon (i.e., total time subtracting crowd time) decreases, as expected. But this decrease is largest from 5-node to 10-node. Subsequent decrease is not as significant. For example, the times of a run of Songs on a 5-, 10-, 15-, and 20-node cluster are 31m, 11m, 7m, and 6m, respectively.

We are also interested in knowing how sample size affects Falcon. As we vary the sample size from 500K to 2M tuples, we found that it has negligible effects on F_1, and increases total run time and cost very slightly. Based on this, we believe a sample size of 1M (that we have used) or even 500K is a good default size.

Regarding memory size, its largest effect would be on $apply\_blocking\_rules$, and we have discussed this earlier in Section 4.11.2. Finally, we have experimented with varying the maximal number of iterations for active learning. As this number goes from 30 to 100, we found that (a) all active learning in our experiments terminated before 100, (b) the run time (including crowdsourcing time) increased significantly, (c) yet F_1 accuracy fluctuates in a very small range. This suggests that capping the number of iterations at some value, say 30 as we have done, is a reasonable solution to avoid high run time and cost yet achieve good accuracy.

4.12 Conclusion

In this chapter we have shown that for important emerging topics such as EM as a service on the cloud, the hands-off crowdsourcing approach of Corleone is ideally suited, but must be scaled up to make such services a reality.

We have described Falcon, a solution that adopts an RDBMS approach to scale up Corleone. Extensive experiments show that Falcon can efficiently match tables of millions of tuples. We are currently in the process of deploying Falcon as an EM service on the cloud for data scientists.

Falcon also provides a framework for many interesting future research directions. These include minimizing crowd latency / monetary cost, examining more optimization techniques (including cost-based optimization), extending Falcon with more operators (e.g., the Accuracy Estimator of Corleone), and applying Falcon to other problem settings, e.g., crowdsourced joins in crowdsourced RDBMSs.

Chapter 5

Conclusion

Entity matching (EM) is the problem of finding data records that refer to the same real-world entity. It is a critical problem in many real-world applications spanning different domains (e.g., e-commerce, healthcare) and practitioners (e.g., enterprises, data enthusiasts). Despite this problem receiving significant attention in the past, it is still far from being solved.

There are many EM settings that involve a crowd of workers (in crowdsourcing) or "non-technical" users (e.g., analysts). There are also many EM settings that involve a large number of EM tasks to be solved (e.g., in the hundreds) or a large number of tuple pairs to match. This dissertation addresses these EM settings. Below are our three key contributions in this dissertation:

1. We have developed Corleone, a hands-off crowdsourcing system for EM. Corleone crowdsources the entire EM pipeline thereby requiring no developer. As such it can scale to growing EM needs at enterprises and startups, and open up crowdsourcing for the masses. We have shown that Corleone achieves comparable or better accuracy than traditional solutions and published results, at a reasonable crowdsourcing cost.

2. We have developed Rmony, a rule-based EM management system for the analysts. Rmony helps an analyst write EM rules manually. Rmony also automatically suggests EM rules to the analysts. We have evaluated Rmony with real-world users and data to demonstrate its effectiveness. We have described a case study to show how Rmony was instrumental in improving an EM system in production. We have described the main lessons learned during the course of development of Rmony, some of which have led to the inception of Magellan, an open-source EM management system in the Python ecosystem.

3. We have developed Falcon, a system that scales up Corleone, using RDBMS-style query execution and optimization over a Hadoop cluster. Falcon's MapReduce solution to execute complex rules over the Cartesian product of two tables significantly advances the state of the art. Extensive experiments show that Falcon can efficiently match tables of millions of tuples. We are currently in the process of deploying Falcon as an EM service on the cloud for data scientists.

Future Research Directions: This dissertation suggests several interesting future research directions. First, our systems can be extended with more capabilities. For example, Corleone can be extended with better algorithms (e.g., sampling pairs during blocking, stopping criteria during active learning), Rmony can be extended with crowdsourcing; Falcon can be extended with more operators (e.g., accuracy estimator), cost-based optimization, etc. Second, we handle only the EM scenario of matching two tables. Exploring other EM scenarios (e.g., linking tables to a knowledge base, matching entity mentions in text with a structured catalog) will be interesting. Third, a next logical research direction for EM can be EM services on the cloud, which will bring with itself its own novel challenges (e.g., pricing, minimizing resource utilization, security, etc.). Finally, this dissertation presents key ideas that are applicable in many research problems, not just EM. For example, hands-off crowdsourcing (in Corleone), system building for analysts (in Rmony), and using RDBMS-style approaches to scale solutions (in Falcon) are applicable to many data management problems, including information extraction, schema matching, and data cleaning.

Bibliography

[1] *Google Knowledge Graph*. http://googleblog.blogspot.com/2012/05/introducing-knowledge-graph-things-not.html.

[2] *Google Shopping*. http://www.google.com/shopping.

[3] *IBM InfoSphere QualityStage*. http://www-03.ibm.com/software/products/en/ibminfoqual.

[4] *Informatica Data Quality*. http://www.informatica.com/us/products/data-quality/.

[5] *Nextag*. http://www.nextag.com/.

[6] *Oracle Enterprise Data Quality*. http://www.oracle.com/technetwork/middleware/oedq/overview/index.html.

[7] *PriceGrabber*. http://www.pricegrabber.com/.

[8] *SAS Data Quality*. http://www.sas.com/en_us/software/data-management/data-quality.html.

[9] *SlickDeals*. http://slickdeals.net/.

[10] *Tamr*. http://www.tamr.com.

[11] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD*, 2013.

[12] M. Ankerst, C. Elsen, M. Ester, and H.-P. Kriegel. Visual classification: An interactive approach to decision tree construction. In *SIGKDD*, 1999.

[13] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *SIGMOD*, 2010.

[14] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, 2009.

[15] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.

[16] B. Becker, R. Kohavi, and D. Sommerfield. Visualizing the simple Bayesian classifier. *Information Visualization in Data Mining and Knowledge Discovery*, 18:237–249, 2001.

[17] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. In *SIGKDD*, 2012.

[18] O. Benjelloun, H. Garcia-Molina, H. Kawai, T. E. Larson, D. Menestrina, Q. Su, S. Thavisomboon, and J. Widom. Generic entity resolution in the SERF project. *Data Engineering Bulletin*, 2006.

[19] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *TKDD*, 2007.

[20] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, 2003.

[21] M. Bilgic, L. Licamele, L. Getoor, and B. Shneiderman. D-dupe: An interactive tool for entity resolution in social networks. In *VAST*, 2006.

[22] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[23] D. Caragea, D. Cook, and V. G. Honavar. Gaining insights into support vector machine pattern classifiers using projection-based tour methods. In *SIGKDD*, 2001.

[24] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[25] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *TODS*, 24(2):177–228, 1999.

[26] P. Christen. Febrl: A freely available record linkage system with a graphical user interface. In *HDKM*, 2008.

[27] P. Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer, 2012.

[28] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *TKDE*, 24(9):1537–1555, 2012.

[29] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. In *VLDB*, 2016.

[30] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *TOIS*, 18(3):288–321, 2000.

[31] M. W. Craven and J. W. Shavlik. Extracting tree-structured representations of trained networks. *NIPS*, 1996.

[32] S. Das, A. Doan, P. S. G. C., C. Gokhale, and P. Konda. The Magellan data repository. `https://sites.google.com/site/anhaidgroup/projects/data`.

[33] A. Das Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, 2012.

[34] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *ICDT*, 2013.

[35] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. ZenCrowd: Leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW*, 2012.

[36] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. In *VLDB*, 2016.

[37] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Elsevier Science, 2012.

[38] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the world-wide web. *CACM*, 54(4):86–96, 2011.

[39] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.

[40] U. Draisbach and F. Naumann. Dude: The duplicate detection toolkit. In *QDB*, 2010.

[41] H. L. Dunn. Record linkage*. *American Journal of Public Health and the Nations Health*, 36(12):1412–1416, 1946.

[42] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *Big Data*, 2015.

[43] M. G. Elfeky, A. K. Elmagarmid, and V. S. Verykios. TAILOR: A record linkage tool box. In *ICDE*, 2002.

[44] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

[45] J. Fan, M. Zhang, S. Kok, M. Lu, and B. C. Ooi. CrowdOp: Query optimization for declarative crowdsourcing systems. *TKDE*, 27(8):2078–2092, 2015.

[46] U. Feige, L. Lovász, and P. Tetali. Approximating min sum set cover. *Algorithmica*, 40(4):219–234, 2004.

[47] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[48] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD*, 2011.

[49] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Declarative data cleaning: Language, model, and algorithms. 2001.

[50] C. L. Giles, K. D. Bollacker, and S. Lawrence. Citeseer: An automatic citation indexing system. In *International Conference on Digital Libraries*, 1998.

[51] L. Gill. OX-LINK: The oxford medical record linkage system, 1997.

[52] L. Gill. *Methods for automatic record matching and linkage and their use in national statistics*. Number 25. Office for National Statistics, 2001.

[53] I. Gog, M. Schwarzkopf, N. Crooks, M. P. Grosvenor, A. Clement, and S. Hand. Musketeer: all for one, one for all in data processing systems. In *EuroSys*, 2015.

[54] C. Gokhale, S. Das, A. Doan, J. F. Naughton, R. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

[55] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: Dynamic max discovery with the crowd. In *SIGMOD*, 2012.

[56] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. In *VLDB*, 2015.

[57] D. Haas, J. Wang, E. Wu, and M. J. Franklin. CLAMShell: Speeding up crowds for low-latency data labeling. In *VLDB*, 2016.

[58] P. Hanrahan. Analytic DB technology for the data enthusiast. SIGMOD Keynote, 2012.

[59] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *TODS*, 23(2):113–157, 1998.

[60] M. Hernández, G. Koutrika, R. Krishnamurthy, L. Popa, and R. Wisnesky. Hil: a high-level scripting language for entity integration. In *EDBT*, 2013.

[61] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, 1995.

[62] F. Hueske, M. Peters, A. Krettek, M. Ringwald, K. Tzoumas, V. Markl, and J.-C. Freytag. Peeking into the optimization of data flow programs with mapreduce-style udfs. In *ICDE*, 2013.

[63] N. Katariya, A. Iyer, and S. Sarawagi. Active evaluation of classifiers on large datasets. In *ICDM*, 2012.

[64] Z. Khayyat, I. F. Ilyas, A. Jindal, S. Madden, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, and S. Yin. BigDansing: A system for big data cleansing. In *SIGMOD*, 2015.

[65] L. Kolb, H. Köpcke, A. Thor, and E. Rahm. Learning-based entity resolution with mapreduce. In *CloudDb*, 2011.

[66] L. Kolb, A. Thor, and E. Rahm. Parallel Sorted Neighborhood Blocking with MapReduce. In *BTW*, 2011.

[67] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient Deduplication with Hadoop. *PVLDB*, 5(12):1878–1881, 2012.

[68] P. Konda, S. Das, P. S. G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems over data science stacks. *VLDB*, 2016.

[69] P. Konda, S. Das, P. Suganthan G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *VLDB*, 2016.

[70] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowledge Engineering*, 69(2):197–210, 2010.

[71] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1-2):484–493, 2010.

[72] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *VLDB*, 1986.

[73] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In *SIGMOD*, 2015.

[74] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. CDAS: a crowdsourcing data analytics system. *PVLDB*, 5(10):1040–1051, 2012.

[75] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A. H. Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[76] A. Marcus, D. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. In *VLDB*, 2013.

[77] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *VLDB*, 2011.

[78] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.

[79] A. E. Monge. Matching algorithms within a duplicate detection system. *Data Engineering Bulletin*, 23(4):14–20, 2000.

[80] B. Mozafari, P. Sarkar, M. Franklin, M. Jordan, and S. Madden. Scaling up crowd-sourcing to very large datasets: A case for active learning. *VLDB*, 2014.

[81] K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. In *ICDT*. 2005.

[82] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, 2011.

[83] A. Parameswaran, A. D. Sarma, H. Garcia-Molina, N. Polyzotis, and J. Widom. Human-assisted graph search: It's okay to ask questions. In *VLDB*, 2011.

[84] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. CrowdScreen: Algorithms for filtering data with humans. In *SIGMOD*, 2012.

[85] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *CIKM*, 2012.

[86] A. G. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR*, 2011.

[87] H. Park and J. Widom. Query optimization over crowdsourced data. In *VLDB*, 2013.

[88] G. Paul Suganthan, C. Sun, K. Krishna Gayatri, H. Zhang, F. Yang, N. Rampalli, S. Prasad, E. Arcaute, G. Krishnan, R. Deep, et al. Why big data industrial systems need rules and what we can do about it. In *SIGMOD*, 2015.

[89] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, 2001.

[90] V. Rastogi, N. Dalvi, and M. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011.

[91] P. Ravikumar and W. W. Cohen. A hierarchical graphical model for record linkage. In *UAI*, 2004.

[92] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: An extensible logical optimizer for udf-heavy data flows. *Information Systems*, 52:96–125, 2015.

[93] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. Tung. Efficient and scalable processing of string similarity join. *TKDE*, 25(10), 2013.

[94] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *SIGKDD*, 2002.

[95] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, 2004.

[96] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. In *VLDB*, 2014.

[97] C. Sawade, N. Landwehr, and T. Scheffer. Active estimation of f-measures. In *NIPS*, 2010.

[98] B. Settles. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114, 2012.

[99] J. Talbot, B. Lee, A. Kapoor, and D. S. Tan. EnsembleMatrix: Interactive visualization to support machine learning with multiple classifiers. In *SIGCHI*, 2009.

[100] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, 2013.

[101] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.

[102] J. Wang, J. Feng, and G. Li. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *VLDB*, 2010.

[103] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *VLDB*, 2012.

[104] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.

[105] L. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2010.

[106] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.

[107] S. E. Whang, J. McAuley, and H. Garcia-Molina. Compare me maybe: Crowd entity resolution interfaces. Technical report, Stanford University, 2012.

[108] W. E. Winkler and Y. Thibaudeau. An application of the fellegi-sunter model of record linkage to the 1990 u.s. decennial census. Technical Report Statistical Research Report Series RR91/09, U.S. Bureau of the Census, Washington, D.C., 1991.

[109] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *VLDB*, 2008.

[110] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15, 2011.

[111] C. Yan, Y. Song, J. Wang, and W. Guo. Eliminating the redundancy in mapreduce-based entity resolution. In *CCGRID*, 2015.

[112] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: A survey. *Frontiers of Computer Science*, 10(3):399–417, 2016.

[113] C. J. Zhang, L. Chen, H. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6(9):757–768, 2013.