TOWARD BUILDING END-TO-END ENTITY MATCHING SOLUTIONS

by

Paul Suganthan Gnanaprakash Christopher

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: 12/11/2017

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

AnHai Doan, Professor, Computer Sciences

Aditya Akella, Professor, Computer Sciences

Paris Koutris, Assistant Professor, Computer Sciences

Theodoros Rekatsinas, Assistant Professor, Computer Sciences

Paul Hanson, Professor, Center for Limnology

© Copyright by Paul Suganthan Gnanaprakash Christopher 2018

All Rights Reserved

To my parents.

ACKNOWLEDGMENTS

This Ph.D. dissertation would not have been possible without the support and encouragement of a number of people. In this section, I would like to express my heart-felt gratitude to these individuals.

First and foremost, I would like to express my infinite gratitude to my advisor, AnHai Doan, without whom this dissertation would not have been possible. His honest feedback over the years has helped me improve my thinking, communication, and writing. I have learned a lot from AnHai. In particular, I have learned to look at the high-level picture, yet pay attention to details. I have fond memories of AnHai teaching me how to write a paper and how a paper should flow.

Next, I would like to thank my thesis committee members, Professors Aditya Akella, Paul Hanson, Paris Koutris, and Theodoros Rekatsinas, for their invaluable input. I would also like to thank my mentor at WalmartLabs, Chong Sun, for helping me evaluate my research at WalmartLabs.

I am thankful to all of my other co-authors and research collaborators throughout my graduate school life. An incomplete list includes Adel Ardalan, Han Li, Haojun Zhang, Jeff Ballard, Pradap Konda, Sanjib Das, and Yash Govind. Thank you Adel for being a wonderful office mate. Thank you Sanjib for being a superb peer over the course of my Ph.D. and always being there to help. Thank you Pradap for always being my go-to person for research discussions. I would never forget the countless discussions we have had on Ph.D., research, and life after graduate school. I would also like to thank my friends in the database group for their critical feedback on my research and for their friendly support. An incomplete list includes Bruhathi Sundarmurthy, Harshad Deshmukh, Mukilan Ashok, and Shaleen Deep.

I am grateful to my other close friends who were always there to support and help me. An incomplete list includes Anand Krishnamurthy, Adalbert Gerald, Kumaresh Visakan, Laxman Deepakraj, Sanketh Nalli, Shriram Sridharan, and Vivek Venkatesh. Thank you Anand, Kumaresh, and Shriram for being wonderful roommates during my initial years at Madison. Thank you Gerald for making sure I have some fun in between research. Thank you Sanketh for the countless food exploits on State Street. Thank you Vivek for always being there for me since high school.

Finally, I would like to thank my family for their continued encouragement during my Ph.D. Thank you amma, appa, and Saro for your continued love and support.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Entity matching (EM) finds data records that refer to the same real-world entity. Numerous EM solutions have been proposed. These solutions however suffer from two main problems. First, they are not end-to-end. That is, the EM workflow consists of multiple steps, such as cleaning, blocking, matching, sampling, labeling, debugging, etc. Current work however has focused mostly on blocking and matching, ignoring the remaining steps. Second, most current works are designed primarily for power users. They are very difficult for lay users to use. In this dissertation I develop solutions to address the above two problems. For the first problem, I work together with several colleagues to develop Magellan, an end-to-end EM solution approach. Within the context of Magellan, I develop a solution to help users extract missing attribute values from textual data (so that EM can be performed more accurately). For the second problem, I develop a solution that lay users can use to perform EM end-to-end easily on the cloud, using a cluster of machines, and optionally using crowdsourcing. I then focus on string matching, a special case of EM, and develop an effective end-to-end solution for lay users. Finally, I describe how the above solutions have been implemented (mostly as open-source software) and deployed to solve real-world applications. The open-source implementation of several solutions in particular has been deployed on Kaggle, a large and well-known data science and competition platform with well over 0.5M users.

Chapter 1

Introduction

Entity matching (EM) identifies data instances that refer to the same real-world entity, such as (David Smith, UW-Madison) and (D. M. Smith, UWM). This is a critical problem in many application domains such as e-commerce, biomedical, scientific data, military intelligence, etc. [3, 5, 37, 2, 38, 30, 39, 97]. It will become even more critical in the age of Big Data and data science.

Many types of EM tasks exist, such as matching tuples across two tables, matching within a single table, matching into a knowledge base, matching XML data, etc [29, 19]. In this dissertation, I will consider the EM task of matching tuples across two tables, which is a very common scenario in practice. Specifically, given two tables A and B, find all tuple pairs $(a \in A, b \in B)$ that refer to the same real-world entity. Figure 1.1 illustrates this scenario.

Table A

	Name	City	State
a_1	Dave Smith	Madison	WI
a_2	Joe Wilson	San Jose	CA
a_3	Dan Smith	Middleton	WI

Table B

	Name	City	State
b_1	David D. Smith	Madison	WI
b_2	Daniel W. Smith	Middleton	WI

Matches
(a_1, b_1)
(a_3, b_2)

Figure 1.1: An example of matching two tables.

For this EM scenario, numerous solutions have been proposed over the past few decades [29, 19]. While significantly advancing the state of the art, these solutions still suffer from two major problems. First, they are not *end-to-end*. That is, the EM workflow consists of multiple steps, such as cleaning, blocking, matching, sampling, labeling, debugging, etc. Current work however

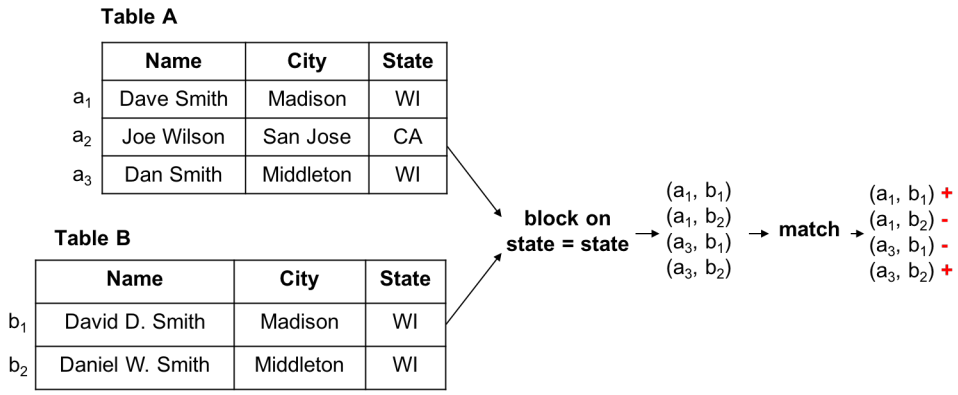

Figure 1.2: Two main steps in EM.

has focused mostly on blocking and matching, ignoring the remaining steps. Second, most current works are designed primarily for power users. They are very difficult for lay users to use. In what follows I elaborate on these two problems and then discuss my solution approaches.

1.1 Developing End-to-End EM Solutions

Two fundamental steps in the EM process are blocking and matching, and current work has focused mostly on these two steps. Below I briefly explain these two steps. Given two tables A and B to match, considering matching all pairs in the Cartesian product of A and B is often impractical, because there are too many of them. So the goal of blocking is to remove obvious non-matching tuple pairs, to reduce the number of pairs to be considered. Then matching is performed over the tuple pairs that survive blocking, to predict each tuple pair as a match/non-match. For example, in Figure 1.2, to match two tables describing persons, we first block on state attribute (i.e., considering only tuples that agree on the value of state) to reduce the number of tuple pairs considered for matching from 6 to 4. Then we perform matching only over these 4 pairs, to predict each pair as a match/non-match. In practice, however, EM is a long and complex process involving many more steps, such as cleaning, sampling, labeling, debugging, etc. Current work has mostly ignored these steps.

To address this problem, I work together with several colleagues to develop Magellan, an end-to-end EM solution approach. Within the context of Magellan, I develop a solution to help users

extract missing attribute values from textual data (so that EM can be performed more accurately). I now briefly elaborate on these solutions.

Developing an End-to-End EM Solution Approach: In collaboration with several colleagues, most notably Pradap Konda, I have contributed to developing Magellan, an end-to-end EM management system that focuses on all steps in the EM workflow [55]. The key ideas underlying Magellan are as follows:

- develop how-to guides, which is a procedure telling the user what to do step-by-step from the two input tables to matches,
- examine the procedure to identify all the pain points, and
- develop solutions for the pain points and implement the solutions as open-source tools in the Python data science ecosystem.

I describe Magellan in more details in Chapter 2. Within the context of Magellan, I develop a solution to help users extract missing attribute values from textual data (so that EM can be performed more accurately), as described next.

Helping Users Extract Missing Attribute Values: A problem often encountered when users develop EM workflows using Magellan is missing attribute values. For example, consider the tuple in Figure 1.3 which describes a product. Observe that the value for the attribute "Size" of the product is missing. Instead, the "Title" attribute contains the size value. In such cases the user often needs to extract such "sprinkled" attribute values, and then uses them to perform more accurate EM.

Title	Size	Brand
trademark poker NASCAR 24 inch cushioned folding stool		nascar

Figure 1.3: Example of a tuple describing a product with a missing attribute value.

To do so, users often write rule-based extractors (e.g., regexes). For example, one such regex is shown below:

$$\backslash d + \backslash s(dia|inches|feet|foot|ft|inch|in|meter|m|mm|cm|yards|yard|yd|yds).$$

The regex is then matched over the product title and the matched phrase is extracted as the size value. A common problem when writing such regexes is that an regex is often not "expansive" enough, i.e., a disjunction of a regex in the rule contains too few terms. Finding all or most such terms is time consuming (often taking hours in practice) and error-prone. For example, the above regex contains 15 such terms in the disjunction.

To address this, I develop an interactive tool that helps users efficiently expand a disjunction in a regex in minutes instead of hours. The user starts by writing a initial regex with a few terms (at least one) in the disjunction, and provides the initial regex and a dataset as input to the tool. Next, the tool processes the given dataset to find a set of candidate terms to be added to the disjunction. Next, it ranks these candidates, and shows the top-k candidates to the user. The user provides feedback on which candidates are correct. The tool uses the feedback to re-rank the remaining candidates. This repeats until either all candidates have been verified by the user, or when the user thinks he or she has found enough terms.

Realizing the above idea raises two challenges. First, we need to find candidate terms that can be added to the disjunction. Second, we need a method to evaluate these candidates in order to rank them. I address these challenges in Chapter 2.

1.2 Developing Effective EM Solutions for Lay Users

Name	Generic name	Strength
Maki-DM	Guaifenesin	500MG/300MG

Name	Generic name	Strength
Q-Tussin DM	Guaifenesin/Dextromethorphan	100-10MG/5

Figure 1.4: An example of matching drugs.

Most current EM works are designed primarily for power users (e.g., those who know how to program). But increasingly more and more lay users, who do not know how to program and may not know much about EM, also want to do EM. For example, consider biomedical scientists matching drugs across two tables (see Figure 1.4). Such users do not know much about EM, e.g., knowing about string similarity measures (e.g., edit distance, Jaccard, TF/IDF, etc.) and when to

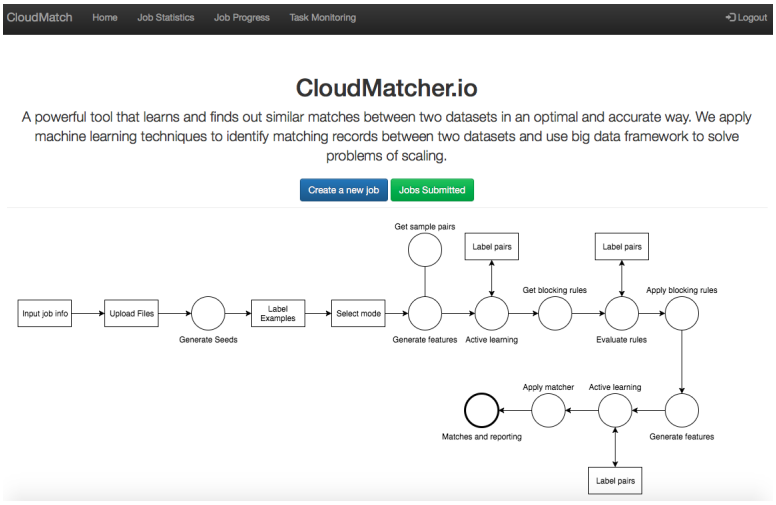

Figure 1.5: A screenshot of CloudMatcher's homepage.

use which measure, about machine learning models and when to use which model, etc. Current solutions however are very difficult for such lay users to use.

To address the above problem, I develop a solution that lay users can use to perform EM end-to-end easily on the cloud, using a cluster of machines, and optionally using crowdsourcing. I then focus on string matching, a special case of EM, and develop an effective end-to-end solution for lay users. In what follows I elaborate on the above two solutions.

Helping Lay Users Perform End-to-End EM on the Cloud: To address the EM needs of lay users, in collaboration with Sanjib Das, I propose Falcon, a end-to-end crowdsourced EM solution on the cloud for lay users. Specifically, a lay user simply needs to go to Falcon's Web site, uploads two tables to be matched, performs some basic pre-processing, then pushes a button. Falcon will perform EM end-to-end. To do so, it will use crowd workers on Amazon's Mechanical Turk (or some other crowdsourcing platform) to label tuple pairs (as match / non-match). The user just has to pay for the labeling. Alternatively, instead of using crowdsourcing, the user can just label these tuple pairs. At the end, Falcon will return the desired matches. Recently, in collaboration with Yash Govind, I have deployed Falcon as a cloud-based service, CloudMatcher, thereby making EM for lay users a reality. Figure 1.5 shows a screenshot of CloudMatcher's homepage.

Falcon often needs to scale the execution of crowdsourced EM workflows over tables of millions of tuples. Realizing this raises three major challenges:

- First, I do not want to scale up a monolithic standalone EM workflow. Rather I want a solution that is modular and extensible so that I can focus on scaling up pieces of it, and can easily extend it later to more complex EM workflows. To address this, I 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. I describe a set of implementations in Hadoop that significantly advances the state of the art. In particular, I 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. I develop a solution that can efficiently process complex rules over large tables. The solution uses indexes to avoid enumerating the Cartesian product, but faces the problem of what to do when indexes do not fit in memory. I show how the solution can nimbly adapt to these situations by redistributing the indexes and the associated workloads across the mappers and reducers.

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

I describe Falcon in more details in Chapter 3. I then focus on string matching, a special case of EM, and develop an effective end-to-end solution for lay users, as described next.

Helping Lay Users Perform End-to-End String Matching: Most current EM solutions focus on matching relational tuples. These solutions are not optimized for matching strings (e.g., matching

	A
a_1	Michael J. Williams
a_2	Michael J. Smith
a_3	Chen Y. Li

	B
b_1	Williams, Michael
b_2	Li, Chen

Matches
(a_1, b_1)
(a_3, b_2)

Figure 1.6: An example of matching two sets of strings.

two sets of names), which is ubiquitous in practice. As a result, in this direction I focus on this specialized yet common case of EM. I develop an end-to-end string matching solution that lay users can easily use yet obtain significantly higher matching accuracy than current string matching solutions.

String matching (SM) is the problem of finding strings from two given sets A and B that refer to the same real-world entity, such as "Michael J. Williams" and "Williams, Michael" (see Figure 1.6). Current SM works define match conditions using string similarity measures, of form $sim\_measure(a, b) \geq \delta$, which predicts a pair of strings $(a \in A, b \in B)$ as a match if their similarity score (e.g., edit distance, Jaccard) is at least δ. Numerous solutions have been proposed to efficiently execute such conditions over large sets of strings [104].

However, current SM approaches suffer from two major limitations. First, they are not *end-to-end*. That is, most of the current solutions only consider efficiently executing the match condition, ignoring the critical step of coming up with a good match condition. In practice, it is often error-prone and time consuming for the user to select a good similarity measure or pick a good threshold for the match condition.

Second, current solutions consider only match conditions that are *a single predicate*. In practice, using a single predicate for SM raises two serious problems. First, many real-world datasets are *heterogeneous*, in that different data regions exhibit different characteristics. They can best be matched using multiple predicates, each of which captures the characteristics of one data region.

Example 1.2.1. *Consider matching two sets of person names that contains both long names (e.g., Shivaram Venkataraman, Christos Papadimitriou) and short names (e.g., Dave Maier, Chen Li). A single predicate such as $jaccard\_2gram(a, b) \geq \epsilon$ does not work well because it is difficult to set the threshold ϵ properly. A high value for ϵ helps match long names accurately, but can be too high*

for short names, incorrectly predicting many matching short names as non-matches. Conversely, a low ϵ helps match short names accurately, but can be too low for long names. Intuitively, we should use two predicates of the form $jaccard\_2gram(a, b) \geq \epsilon$, but one with a high ϵ for long names, and the other with a lower ϵ for short names. We can check if a name is long using a predicate such as $length(a) > 9$, which returns true if the length of string a (i.e., the number of characters in a excluding space characters) exceeds 9.

Such heterogeneity arises naturally in a single dataset (e.g., large datasets of person names often contain a mixture of long and short names), or arises because a dataset to be matched is being created by integrating several smaller datasets, each of which contains data of a different nature.

Another serious problem is that real-world strings often contain *substrings with special meaning*. Treating such substrings differently from the rest of the strings can significantly improve the matching accuracy. To do so, however, we need to use multiple predicates.

Example 1.2.2. *Consider matching house addresses. Using a single predicate such as $jaccard\_3gram(a, b) \geq \epsilon$ does not work well. A high ϵ (e.g., 0.9) can match addresses correctly, but exclude many matches with lower Jaccard scores, e.g., "522 Wilson St Austin TX 78704" and "522 Wilson Street Austin TX 78704". A lower ϵ (e.g., 0.8) helps identify matches such as the above one, but incorrectly matches "522 Wilson St Austin TX 78704" and "422 Wilson St Austin TX 78704", which differ only in the house numbers. To address this problem, we can extract all numbers from each string, then declare two strings match if the strings are highly similar (e.g., using $jaccard\_3gram(a, b) \geq 0.8$) and their numbers are also highly similar (e.g., using a predicate such as $cosine\_num(a, b) \geq 0.8$).*

To address the above limitations, I propose Smurf, an end-to-end string matching solution that interacts with the user to learn a random forest (which uses a rich set of predicates) as the match condition, then efficiently executes the random forest over the two sets of strings. Specifically, to use Smurf, a lay user simply needs to upload two sets of strings to Smurf's Web site, and labels the set of pairs of strings shown by Smurf. Smurf uses these labeled string pairs to learn a random

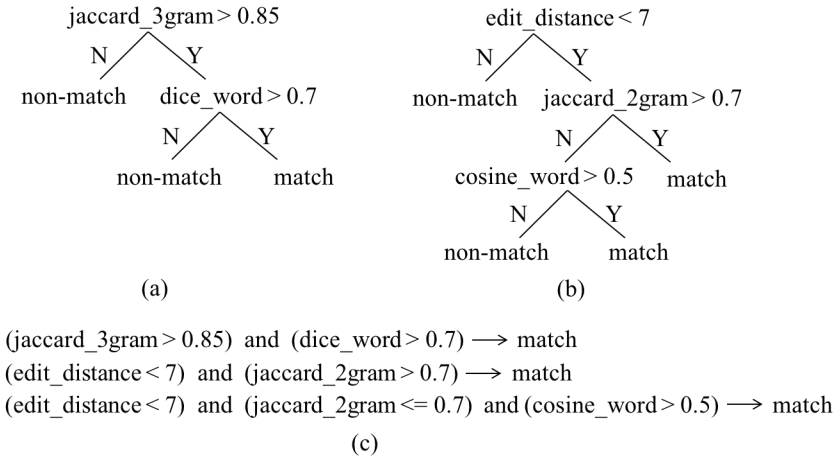

Figure 1.7: (a)-(b) A toy random forest consisting of two decision trees, and (c) matching rules extracted from the forest.

forest (in an active-learning fashion), executes the forest over the two sets of strings, then return the matches. Smurf uses random forests (RFs) as match conditions. A random forest F is a set of n decision trees [13]. It declares a string pair a match if at least αn trees in F declare the pair a match (where α is pre-specified). Random forests are widely used in practice (e.g., [82, 23, 42]), often give very competitive performance, are relatively easy to understand, and are amenable to optimization (as we will see). Figures 1.7.a-b show a toy forest with just two trees.

Realizing Smurf raises the major challenge of efficiently executing a random forest F over two sets of input strings A and B. Our solution to this forms the key technical contribution of this work. Specifically, consider a random forest F of n trees. Naively, we can execute each tree on A and B, then combine their outputs (e.g., predicting a string pair a match if at least αn trees predict the pair a match). This however is very time consuming. To address this problem, I propose to (a) execute only a subset of trees on A and B to obtain a relatively small set J of string pairs that are likely to be matches, then (b) execute the remaining trees only on J (instead of on A and B). I show that this solution is guaranteed to be correct, yet takes far less time. I call the above two steps *blocking* and *matching*, as they are similar in spirit to the blocking and matching steps commonly used in EM [29].

At the heart of both blocking and matching is the need to efficiently execute a set of decision trees (DTs) over two sets of strings (or over a set of string pairs). A DT can be viewed as a disjunction of matching rules (each being a conjunction of predicates). Figure 1.7.c shows all three matching rules extracted from the toy forest. Thus, executing a set of DTs reduces to executing a set of rules. Current work has optimized the execution of *individual matching rules* [23, 57]. But as far as we know, no work has yet optimized the execution of *a set of rules*. Our work develops such a solution. We observe that the matching rules often share a lot of computation, as illustrated below:

Example 1.2.3. *Suppose that a rule contains* $edit\_dist(a, b) < 3$ *and that another rule contains* $edit\_dist(a, b) < 5$*. Then the (relatively expensive) edit distance computation is performed twice. As another example, suppose that two rules contains* $overlap\_word(a, b) > 3$ *and* $jaccard\_word(a, b) > 0.6$*, respectively. Then the overlap computation (i.e., finding the number of words that are common to both* a *and* b*) is performed twice (because computing Jaccard scores also requires computing the overlap).*

To address this problem, I execute the rules jointly, in a way reminiscent of multi-query optimization in RDBMSs [86]. Specifically, I define a small set of core operators that are specific to string contexts. Given a set of rules to be executed, I show how to combine them into a plan (composed of these operators). I define four optimization techniques to remove redundant computations in such a plan. I show how to estimate the runtime of a plan, then how to efficiently search a large space of plans to find one that employs the above optimization techniques to minimize runtime. Finally, I show how to efficiently execute the selected plan. I describe Smurf in more details in Chapter 4.

1.3 Contributions and Outline of This Dissertation

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

- First, I work together with several colleagues to develop Magellan, an end-to-end EM solution approach that focuses on all steps in the EM workflow. Within the context of Magellan,

I develop a solution to help users extract missing attribute values from textual data (so that EM can be performed more accurately). As far as we can tell, no current work has considered this problem for EM.

- Second, in collaboration with Sanjib Das, I propose Falcon, an end-to-end crowdsourced EM solution on the cloud for lay users. Falcon often needs to scale the execution of crowdsourced EM workflows over tables of millions of tuples. To address this, I use RDBMS-style query execution and optimization over a Hadoop cluster. The Hadoop-based solution in Falcon to execute complex rules over the Cartesian product of the two tables 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. Falcon can efficiently perform crowdsourced EM over tables of 1.0M - 2.5M tuples at the cost of $54 - $65.5.

- Third, I propose Smurf, an end-to-end string matching solution that lay users can easily use yet obtains significantly higher matching accuracy than current string matching solutions. Smurf learns random forests (which uses a rich set of predicates) as match conditions, and efficiently executes the random forest over the two sets of strings. To execute the random forest fast, Smurf decomposes it into a blocking step and a matching step, then uses RDBMS-style plan generation and optimization to execute sets of decision trees efficiently in both steps, by reusing computation across trees.

- Finally, I implement the above solutions (mostly as open-source software) and deploy them to solve real-world problems. The open-source implementation of several solutions in particular has been deployed on Kaggle, a large and well-known data science and competition platform with well over 0.5M users.

The rest of this dissertation is organized as follows. Chapter 2 introduces Magellan and describes the solution to help users extract missing attribute values. Chapters 3 and 4 describe Falcon and Smurf respectively. Chapter 5 describes software development and deployment experience, and Chapter 6 concludes this dissertation.

Parts of this dissertation have been published in database conferences. Magellan (Chapter 2) is described in two VLDB-2016 papers [55, 54]. Our solution to help users extract missing attribute values (Chapter 2) is described in a SIGMOD-2015 paper [75]. Falcon (Chapter 3) is described in a SIGMOD-2017 paper [23], and Smurf (Chapter 4) is currently under submission.

Chapter 2

Developing End-to-End EM Solutions

In this chapter I develop Magellan, an end-to-end EM solution approach. Within the context of Magellan, I develop a solution to help users extract missing attribute values from textual data (so that EM can be performed more accurately).

2.1 Developing an End-to-End EM Solution Approach

In this section I begin by describing how EM is done in practice and then build on that to develop an end-to-end EM solution approach.

2.1.1 EM in Practice

In practice, EM often involves many more steps than just blocking and matching, and is often carried out in two stages: development and production. The goal of the development stage is to discover a good *EM workflow*, e.g., one with high matching accuracy. This is typically done using data samples. The production stage then applies the workflow discovered in the development stage to the entirety of the data. Since this data is often large, a major concern here is to scale up the workflow. Other concerns include quality monitoring, logging, crash recovery, etc.

To illustrate, consider an example of trying to match two large tables A and B (of 1M tuples each) using supervised machine learning. To do this, we begin with the development stage, as shown in Figure 2.1. Here the user often has to do the following steps:

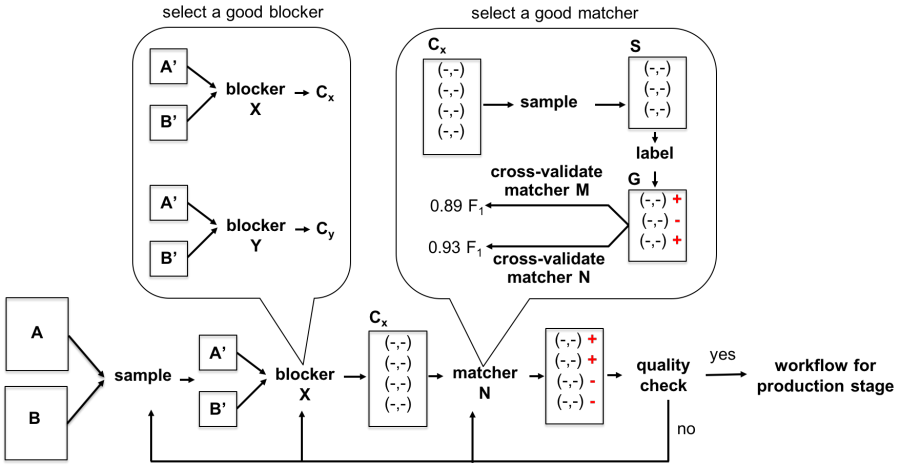

Figure 2.1: Development stage.

- The large input tables A and B cannot be used directly as any operation performed on them could be very time consuming. So the user will first downsample the tables to create smaller tables A' and B' (say of 100K tuples each).

- Then the user may explore the tables by performing visualization and do data cleaning if needed.

- Next, the user will perform blocking on A' and B', by trying out various blocking strategies to come up with what he or she judges to be the best. After applying blocking, the user will get a candidate set of tuple pairs C to be considered for matching.

- Next, the user has to somehow select and apply a matcher to C. Since a supervised learning based approach is used for matching, the user first needs to create training data by sampling a set S from C and manually label each tuple pair in S as a match or a non-match to obtain a labeled set G.

- The user will now use G to select a best matcher using cross validation. After applying the matcher, each pair in C will be predicted as a match or a non-match.

- Finally, the user will check the quality of the predictions. If the user is satisfied with the accuracy, then a workflow will be outputted for the production stage. Otherwise the user

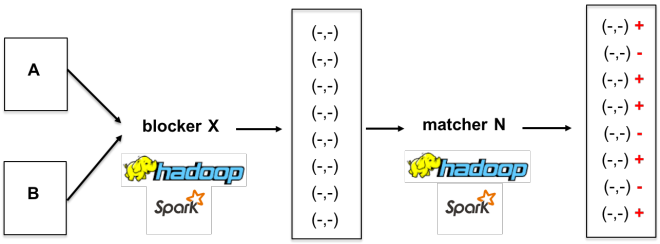

Figure 2.2: Production stage.

may need to go back, debug, and redo the previous steps. For example, after debugging, the user may perform data cleaning, do information extraction, add new features, etc.

After the development stage, the user has obtained a workflow which consists of the selected blocker and matcher. Then in the production stage the user will apply this workflow to the original tables A and B. Since these tables are large, scaling is a major issue. Thus, the workflow is typically executed in a big data system such as Hadoop, Spark, etc. (see Figure 2.2).

Current EM solutions provide support only for the blocking and matching steps in this EM workflow, while ignoring less well-known yet equally critical steps such as extraction, debugging, sampling, labeling, etc. To address this, in collaboration with several colleagues, I have contributed to developing Magellan, an end-to-end EM management system that focuses on all steps in the EM workflow [55]. In what follows I describe Magellan, an attempt to build an end-to-end EM solution .

2.1.2 The Magellan Approach

Figure 2.3 shows the Magellan architecture. The system targets a set of EM scenarios. For each EM scenario it provides a how-to guide. The guide proposes that the user solve the scenario in two stages: development and production.

In the development stage, the user seeks to develop a good EM workflow (e.g., one with high matching accuracy). The guide tells the user what to do, step by step. For each step the user can use a set of supporting tools, each of which is in turn a set of Python commands. This stage is typically done using data samples.

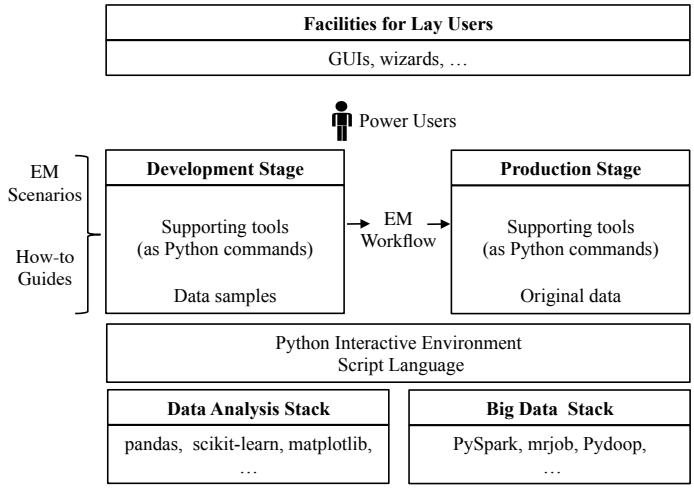

Figure 2.3: The Magellan architecture.

In the production stage, the guide tells the user how to implement and execute the EM workflow on the entirety of data, again using a set of supporting tools. Both stages have access to the Python script language and interactive environment (e.g., IPython). Further, tools for these stages are built on top of the Python data analysis stack and the Python Big Data stack, respectively. Thus, Magellan is an "open-world" system, as it often has to borrow functionalities (e.g., cleaning, extraction, visualization) from other Python packages on these stacks.

Finally, the current Magellan is geared toward power users (who can program). We envision that in the future facilities for lay users (e.g., GUIs, wizards) can be laid on top (see Figure 2), and lay user actions can be translated into sequences of commands in the underlying Magellan.

Within the context of Magellan, I now develop a solution to help users extract missing attribute values from textual data (so that EM can be performed more accurately), as described next.

2.2 Helping Users Extract Missing Attribute Values

A problem often encountered when users develop EM workflows using Magellan is missing attribute values. For example, consider the tuple in Figure 2.4 which describes a product. Observe that the value for the attribute "Size" of the product is missing. Instead, the "Title" attribute contains the size value. In such cases the user often needs to extract such "sprinkled" attribute values,

Title	Size	Brand
trademark poker NASCAR 24 inch cushioned folding stool		nascar

Figure 2.4: Example of a tuple describing a product with a missing attribute value.

and then uses them to perform more accurate EM.

To extract such attribute values, users often write rule-based extractors (e.g., regexes). But writing such rules can be very error-prone and time consuming. In particular, a common problem is that an rule is often not "expansive" enough, i.e., a disjunction of a regex in the rule contains too few terms. For example, to extract product type attribute from the product title, a user may write a rule such as:

$$R_1 : (motor \mid engine)\ oils? \rightarrow motor\ oil,$$

which states that if a product title contains the word "motor" or "engine", followed by "oil" or "oils", then extract the product type as "motor oil".

Later the user may realize that this rule is not expansive enough, in that the disjunction should contain more terms such as "car", "truck", "vehicle", and "scooter" as well. In fact, there can be tens of terms indicating "motor" and they should all be in this disjunction. Eventually, the rule may become:

$$R_2 : (motor \mid engine \mid auto(motive)? \mid car \mid truck \mid suv \mid van \mid vehicle \mid motorcycle \mid pick[-]?up \mid scooter \mid atv \mid boat)\ (oil \mid lubricant)s? \rightarrow motor\ oil.$$

The first disjunction of the regex in the above rule contains 13 terms (e.g., "motor", "engine", etc.).

Clearly, finding all or most such terms is error-prone and time consuming (often taking hours in our experience). To find these terms, the user often has to painstakingly "comb" a very large set of product titles, in order to maximize recall and avoid false positives. EM users have indicated to us that a solution that helps them to efficiently expand such regexes would be highly desirable.

2.2.1 Our Solution

Thus, I will develop a tool that helps users find such terms, which we call "synonyms", in minutes instead of hours. The user starts by writing a short rule such as Rule R_1 described above.

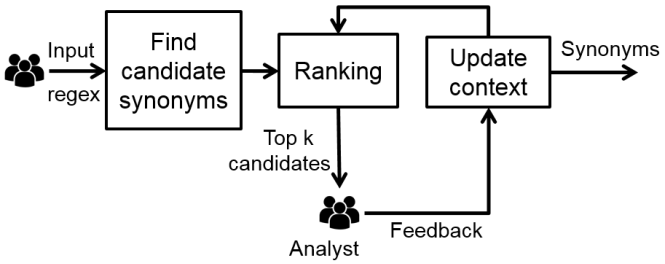

Figure 2.5: The architecture of the tool that supports users in creating rules.

Next, suppose the user wants to expand the disjunction in R_1, given a data set of product titles D. Then he or she provides the following rule to the tool:

$$R_3 : (motor \mid engine \mid \backslash syn)\ oils? \rightarrow motor\ oil,$$

where the string $\backslash syn$ means that the user wants the tool to find all synonyms for the corresponding disjunction (this is necessary because a regex may contain multiple disjunctions, and currently for performance and manage-ability reasons the tool focuses on finding synonyms for just one disjunction at a time).

Next, the tool processes the given data set D to find a set of synonym candidates C. Next, it ranks these synonym candidates, and shows the top k candidates. The user provides feedback on which candidates are correct. The tool uses the feedback to re-rank the remaining candidates. This repeats until either all candidates in C have been verified by the user, or when the user thinks he or she has found enough synonyms (see Figure 2.5). I now describe the main steps of the tool in more details.

Finding Candidate Synonyms: Given an input regex R, we begin by generating a set of generalized regexes, by allowing any phrase up to a pre-specified size k in place of the disjunction marked with the $\backslash syn$ tag in R. Intuitively, we are only looking for synonyms of the length up to k words (currently set to 3). Thus, if R is $(motor \mid engine \mid \backslash syn) \quad oils?$, then the following generalized regexes will be generated:

$(\backslash w+) \quad oils?$
$(\backslash w+\backslash s+\backslash w+) \quad oils?$

$$(\backslash w+\backslash s+\backslash w+\backslash s+\backslash w+) \quad oils?$$

We then match the generalized regexes over the given data D to extract a set of candidate synonyms. In particular, we represent each match as a tuple <*candidate synonym, prefix, suffix*> , where *candidate synonym* is the phrase that appears in place of the marked disjunction in the current match, and *prefix* and *suffix* are the text appearing before and after the candidate synonym in the product title, respectively.

For example, applying the generalized regex $(\backslash w+) \quad (jean \mid jeans)$ to the title *"big men's regular fit carpenter jeans, 2 pack value bundle"* produces the candidate synonym *carpenter*, the prefix *"big men's regular fit"*, and the suffix *"2 pack value bundle"*. We use the *prefix* and *suffix* (currently set to be 5 words before and after the candidate synonym, respectively) to define the context in which the candidate synonym is used.

The set of all extracted candidate synonyms contains the "golden synonyms", those that have been specified by the analyst in the input regex (e.g., "motor" and "engine" in the "motor oil" example). We remove such synonyms, then return the remaining set as the set of candidate synonyms. Let this set be C.

Ranking the Candidate Synonyms: Next we rank synonyms in C based on the similarities between their contexts and those of the golden synonyms, using the intuition that if a candidate synonym appears in contexts that are similar to those of the golden synonyms, then it is more likely to be a correct synonym. To do this, we use a TF/IDF weighting scheme [80]. This scheme assigns higher scores to contexts that share tokens, except where the tokens are very common (and thus having a low IDF score).

Specifically, given a match m, we first compute a prefix vector $\vec{P_m} = (pw_{1,m}, pw_{2,m}, ..., pw_{n,m})$, where $pw_{t,m}$ is the weight associated with prefix token t in match m, and is computed as $pw_{t,m} = tf_{t,m} * idf_t$. Here, $tf_{t,m}$ is the number of times token t occurs in the prefix of match m, and idf_t is computed as $idf_t = log(\frac{|M|}{df_t})$, where $|M|$ is the total number of matches.

Next, we normalize the prefix vector $\vec{P_m}$ into $\hat{P_m}$. We compute a normalized suffix vector $\hat{S_m}$ for match m in a similar fashion.

In the next step, for each candidate synonym $c \in C$, we compute, $\vec{M_{p,c}}$, the mean of the normalized prefix vectors of all of its matches. Similarly, we compute the mean suffix vector $\vec{M_{s,c}}$.

Next, we compute $\vec{M_p}$ and $\vec{M_s}$, the means of the normalized prefix and suffix vectors of the matches corresponding to *all* golden synonyms, respectively, in a similar fashion.

We are now in a position to compute the similarity score between each candidate synonym $c \in C$ and the golden synonyms. First we compute the prefix similarity and suffix similarity for c as: $prefix\_sim(c) = \frac{\vec{M_{p,c}} \cdot \vec{M_p}}{|\vec{M_{p,c}}||\vec{M_p}|}$, and $suffix\_sim(c) = \frac{\vec{M_{s,c}} \cdot \vec{M_s}}{|\vec{M_{s,c}}||\vec{M_s}|}$. The similarity score of c is then a linear combination of its prefix and suffix similarities:

$$score(c) = w_p * prefix\_sim(c) + w_s * suffix\_sim(c)$$

where w_p and w_s are balancing weights (currently set at 0.5).

Incorporating User Feedback: Once we have ranked the candidate synonyms, we start by showing the top k candidates to the user (currently $k = 10$). For each candidate synonym, we also show a small set of sample product titles in which the synonym appears, to help the user verify. Suppose the user has verified l candidates as correct, then he or she will select these candidates (to be added to the disjunction in the regex), and reject the remaining $(k - l)$ candidates. We use this information to rerank the remaining candidates (i.e., those not in the top k), then show the user the next top k, and so on.

Specifically, once the user has "labeled" the top k candidates in each iteration, we refine the contexts of the golden synonyms based on the feedback, by adjusting the weights of the tokens in the mean prefix vector $\vec{M_p}$ and the mean suffix vector $\vec{M_s}$ to take into account the labeled candidates. In particular, we use the Rocchio algorithm [77], which increases the weight of those tokens that appear in the prefixes/suffixes of correct candidates, and decreases the weight of those tokens that appear in the prefixes/suffixes of incorrect candidates. Specifically, after each iteration, we update the mean prefix and suffix vectors as follows:

$$\vec{M_p'} = \alpha * \vec{M_p} + \frac{\beta}{\mid C_r \mid} \sum_{c \in C_r} \vec{M_{p,c}} - \frac{\gamma}{\mid C_{nr} \mid} \sum_{c \in C_{nr}} \vec{M_{p,c}} \tag{2.1}$$

$$\vec{M_s'} = \alpha * \vec{M_s} + \frac{\beta}{\mid C_r \mid} \sum_{c \in C_r} \vec{M_{s,c}} - \frac{\gamma}{\mid C_{nr} \mid} \sum_{c \in C_{nr}} \vec{M_{s,c}} \tag{2.2}$$

Product Type	Input Regex	Sample Synonyms Found
Area rugs	(area | \syn) rugs?	shaw, oriental, drive, novelty, braided, royal, casual, ivory, tufted, contemporary, floral
Athletic gloves	(athletic | \syn) gloves?	impact, football, training, boxing, golf, workout
Shorts	(boys? | \syn) shorts?	denim, knit, cotton blend, elastic, loose fit, classic mesh, cargo, carpenter
Abrasive wheels & discs	(abrasive | \syn) (wheels? | discs?)	flap, grinding, fiber, sanding, zirconia fiber, abrasive grinding, cutter, knot, twisted knot

Table 2.1: Sample regexes provided by the analyst to the tool, and synonyms found.

where C_r is the set of correct candidate synonyms and C_{nr} is the set of incorrect candidate synonyms labeled by the analyst in the current iteration, and α, β and γ are pre-specified balancing weights.

The user iterates until all candidate synonyms have been exhausted, or he or she has found sufficient synonyms. At this point the tool terminates, returning an expanded rule where the target disjunction has been expanded with all new found synonyms.

2.2.2 Empirical Evaluation

Overall Performance: We have evaluated the tool using 25 input regexes randomly selected from those being worked on at the experiment time by the WalmartLabs analysts. Table 4.7.1 shows examples of input regexes and sample synonyms found. Out of the 25 selected regexes, the tool found synonyms for 24 regexes, within three iterations (of working with the analyst). The largest and smallest number of synonyms found are 24 and 2, respectively, with an average number

Rule ID	# Cand.	Method	Iter 1	Iter 2	Iter 3	Iter 4	Iter 5	Total
1	155	TF-IDF	8	4	3	2	2	19
		NB	3	4	3	4	4	18
2	9	TF-IDF	8	0	0	0	0	8
		NB	8	0	0	0	0	8
3	501	TF-IDF	4	1	2	1	0	8
		NB	3	3	0	1	0	7
4	313	TF-IDF	3	1	1	0	0	5
		NB	1	1	1	4	0	7
5	1721	TF-IDF	5	4	2	2	1	14
		NB	0	2	0	0	0	2
6	86	TF-IDF	4	2	1	2	1	10
		NB	1	4	4	0	0	9
7	53	TF-IDF	5	4	1	4	1	15
		NB	1	3	3	5	2	14
8	215	TF-IDF	10	4	4	1	0	19
		NB	4	3	3	2	3	15
9	746	TF-IDF	7	4	1	1	0	13
		NB	6	2	0	2	0	10
10	123	TF-IDF	3	3	2	0	0	8
		NB	0	1	2	1	0	4

Rules used in the above experiment

1 : (abrasive|\syn)[-]wheels? → Abrasive wheels & discs
2 : (crimped|\syn) wire wheels? → Abrasive wheels & discs
3 : (artificial|\syn) flowers? → Artificial plants & flowers
4 : artificial (flower|\syn) → Artificial plants & flowers
5 : (area|\syn) rugs? → Rugs
6 : (audio|\syn) compressors? → Audio compressors
7 : (audio|\syn) mixers? → Audio mixers
8 : (composite|\syn) cables? → Consumer electronics cables
9 : (art|\syn) (paper|pad)s? → Paper pads
10 : (arm|\syn) ?chairs? → Arm chairs & accent chairs

Figure 2.6: Comparison of the TF-IDF and NB (i.e., Naive Bayes) approaches to finding synonyms for regex expansion.

of 7 per regex. The average time spent by the analyst per regex is 4 minutes, a significant reduction from hours spent in such cases. This tool has been in production at WalmartLabs since June 2014.

Effectiveness of TF-IDF Technique: Our goal is to examine the effectiveness of the TF/IDF technique underlying this tool, which henceforth we will call SynFinder for brevity.

To examine this effectiveness, we can try to show that for each of the 25 regexes in our experiment, SynFinder ranks the correct synonyms higher than the incorrect ones. To do this, we would need to label all candidate synonyms found by SynFinder as correct/incorrect (so that we have the golden data to perform the above experiment). Unfortunately SynFinder often returns a

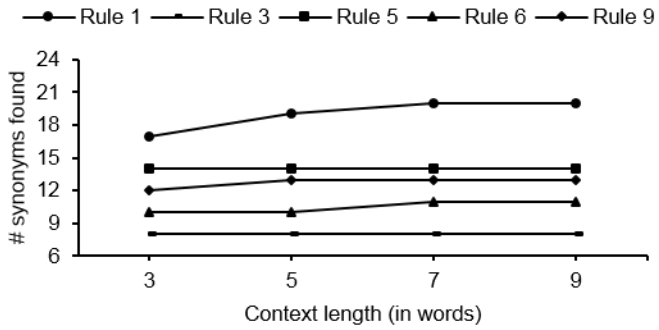

Figure 2.7: Number of synonyms found after five iterations as we vary the context window size.

very large set of candidate synonyms (from hundreds to over a thousand, see the second column in Table 4.7.1). Manually labeling such large sets is highly impractical.

As a result, we opted for a second approach, where we compare SynFinder with some other reasonable method, and show that SynFinder finds more correct synonyms than that method in the first several iterations (thus proving more effective for the analyst). Specifically, we compare SynFinder with a method that uses a Naive Bayes classifier to label a candidate synonym as correct or incorrect. We trained the Naive Bayes classifier on the contexts surrounding golden synonyms, using as the default 5 tokens to each side of the synonyms.

Figure 2.6 compares the two methods. The first column shows 10 rules that we randomly sampled for this experiment. The second column shows the number of candidate synonyms. The names "TF-IDF" and "NB" in the third column stand for SynFinder and the Naive Bayes method, respectively. Subsequent columns show the number of correct synonyms returned by each method in the first five iterations (most WalmartLabs analysts typically stopped after 3-5 iterations).

The table in the figure shows that on nine out of ten rules, SynFinder finds as many or significantly more synonyms than the Naive Bayes method, sometimes as many as 7 times more (e.g., 14 vs. 2 for Rule #5, see the last column). It also shows that SynFinder often finds more synonyms faster (e.g., in the first few iterations) than the Naive Bayes method. These results suggest the effectiveness of the SynFinder approach.

Effect of Context Length: We examined the sensitivity of SynFinder as we vary the size of the contexts (set at 5 as default). Figure 2.7 shows the total number of synonyms found by SynFinder after 5 iterations, as we vary the context size from 3 to 9. The figure shows no change or a minimal improvement from size 3 to 9, thus suggesting that SynFinder is relatively robust to small changes in the context window size.

2.3 Related Work

Developing an End-to-End EM Solution Approach: Numerous EM algorithms have been proposed [29]. But far fewer EM systems have been developed [19]. None of the existing systems provides support for steps such as extraction, sampling, labeling, selecting and debugging blockers and matchers, as Magellan does.

Some recent works have discussed desirable properties for EM systems, e.g., being extensible and easy-to-deploy [21], being flexible and open source [18], and the ability to construct complex EM workflow consisting of distinct phases, each requiring a specific technique depending on the given application and data requirements [34]. These works do not discuss covering the entire EM pipeline, how-to guides, building on top of data analysis and Big Data stacks, and open-world systems, as we do in Magellan.

Finally, the notion of "open world" has been discussed in [35], but in the context of crowd workers' manipulating data inside an RDBMS. Here we discuss a related but different notion of open-world systems that often interact with and manipulate each other's data. In this vein, the work [15] is related in that it discusses the API design of the scikit-learn package and its design choices to seamlessly tie in with other packages in Python.

Helping Users Extract Missing Attribute Values: Several works have addressed the problem of finding synonyms or similar phrases [40, 59]. Godbole et al. [40] consider building generic synonym dictionaries. Lin [59] considers automatically clustering similar words based on the distributional pattern of words. But our problem is different in that we focus on finding "synonyms" that can be used to extend a regex. Thus the notion of synonym here is defined by the regex.

Many interactive regex development tools exist (e.g., [6, 7]). But they focus on helping users interactively test a regex, rather than extending it with "synonyms". Li et al. [58] address the problem of transforming an input regex into a regex with better extraction quality. They use a greedy hill climbing search procedure that chooses at each iteration the regex with the highest F-measure. But this approach again focuses on refining a given regex, rather than extending it with new phrases.

2.4 Conclusion

In this chapter, I described Magellan, a novel end-to-end EM management system. Then I proposed a solution to help users extract missing attribute values from textual data (so that EM can be performed more accurately). Finally, I presented our evaluation with analysts at WalmartLabs. As far as we can tell, no current work has considered this problem for EM.

Chapter 3

Helping Lay Users Perform End-to-End EM on the Cloud

3.1 Introduction

Name	Generic name	Strength
Maki-DM	Guaifenesin	500MG/300MG

Name	Generic name	Strength
Q-Tussin DM	Guaifenesin/Dextromethorphan	100-10MG/5

Figure 3.1: An example of matching drugs.

Most current EM works are designed primarily for power users (e.g., those who know how to program). But increasingly more and more lay users, who do not know how to program and may not know much about EM, also want to do EM. For example, consider biomedical scientists matching drugs across two tables (see Figure 3.1). Such users do not know much about EM, e.g., knowing about string similarity measures (e.g., edit distance, Jaccard, TF/IDF, etc.) and when to use which measure, about machine learning models and when to use which model, etc. Current solutions however are very difficult for such lay users to use.

To address this problem, a recent work has introduced Corleone [42], a solution that crowdsources the *entire* EM workflow. Specifically, a user only needs to supply the two tables to be matched and pay the crowdsourcing cost. Corleone will perform the end-to-end EM in an hands-off manner, making it easier for lay users to do EM.

As described, Corleone is highly promising. But it suffers from a major limitation: it does not scale to large tables, as the following example illustrates.

Example 3.1.1. *Back in 2016, we wanted 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 wanted 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 I introduce Falcon (fast large-table Corleone), a solution that scales up Corleone to tables of millions of tuples.

I 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.

I 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, I do not want to scale up a monolithic stand-alone EM workflow. Rather, I 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, I 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. I describe a set of implementations in Hadoop that significantly advances the state of the art. I 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. I develop a solution that can efficiently process complex rules over large tables. The 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. I show how the solution can nimbly adapt to these situations by redistributing the indexes and associated workloads across the mappers and reducers.

Finally, I consider the challenge of optimizing EM plans. I show that combining machine operations with crowdsourcing introduces novel optimization opportunities, such as using crowd time to mask machine time. I develop masking techniques that use the crowd time to build indexes and to speculatively execute machine operations. I 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, my main contributions are:

- I 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.

- I show that an RDBMS-style execution and optimization framework is a good way to address scaling for crowdsourced EM, and I develop the first end-to-end solution to scale up hands-off crowdsourced EM.
- I define a set of operators and plans for crowdsourced EM that uses machine learning.
- I 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.
- I 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.

3.2 Problem Definition

I now briefly describe Corleone [42], analyze its limitations, and define the problem considered in this work.

3.2.1 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 3.2 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 3.3.b shows two such rules). Since $A \times B$ is often very large, considering all tuple

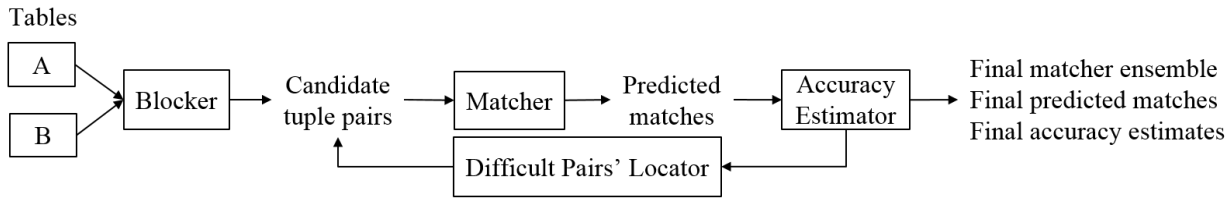

Figure 3.2: The EM workflow of Corleone.

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 [13], 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.

3.2.2 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.

I now briefly describe the Blocker and the Matcher, focusing only on the aspects necessary to understand Falcon (see [42] for a complete description).

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 [13], 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.

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 3.3.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 3.3.b shows two such rules extracted from the tree in Figure 3.3.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 [42]).

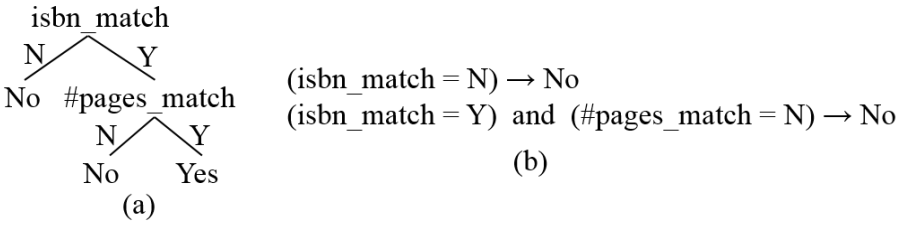

Figure 3.3: (a) A decision tree learned by Corleone and (b) blocking rules extracted from the tree.

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 3.3.b. As we will see in Section 3.4, 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 4.8, 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 3.4). In particular, it can be shown that when a blocking rule performs key-based blocking (e.g., the first rule in Figure 3.3.b, "(isbn_match = N) $\rightarrow$ No", considers only tuples that share the same ISBN), our solution in Section 3.4 reduces to current key-based blocking solutions on MapReduce.

3.2.3 Limitations of Corleone

Corleone is highly promising because it uses only crowdsourcing to achieve high EM accuracy at a relatively low cost [42]. However it suffers from a major limitation: it does not scale to large tables. The largest table pair in [42] is 2.6K tuples $\times$ 64K tuples.

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 3.5.2, this raises very interesting optimization opportunities, and can significantly reduce the total execution time.

3.2.4 Goals of Falcon

It is difficult to address all of the above performance factors simultaneously. So as a first step, in Falcon I 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 the procedure of evaluating blocking rules described in [42] is guaranteed to execute at most 20 iterations per rule (see [1] for a proof). As a result, we can estimate an upper bound on the total crowd time (regardless of the table sizes):

Proposition 1. *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 [42]. 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 4.8 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 I discuss in Section 3.6.1, crowd time can vary widely depending on the platform. In fact, I 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 [47].

3.3 The Falcon Solution

3.3.1 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, I propose that Falcon adopt an RDBMS approach. Specifically, (1) I 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). I will compose these operators to form EM workflows. (2) I will develop efficient implementations of these operators, using Hadoop. And (3) I 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).

I now define a set of operators and show how to compose them to form EM workflows, henceforth called *EM plans*. Sections 3.4-3.5 describe efficient implementations of operators, then plan generation, execution, and optimization.

3.3.2 Operators

I 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 3.5.1 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 3.3.b. Each rule R_i is of the form

$$p_1^i(a, b) \wedge \ldots \wedge p_{m_i}^i(a, b) \rightarrow \ drop\ (a, b), \tag{3.1}$$

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.

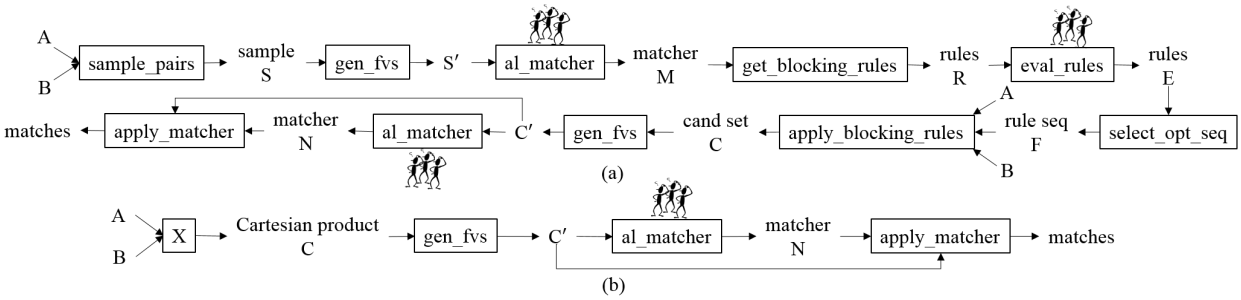

Figure 3.4: The two plan templates used in Falcon.

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., 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 3.4.3).

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.

3.3.3 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 3.4, which correspond to the EM workflows that use both the Blocker and Matcher, and just the Matcher, respectively.

The first plan template (Figure 3.4.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 3.4.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 3.5.1).

3.4 Efficient Implementations of Operators

I have developed an efficient Hadoop solution for the $apply\_blocking\_rules$ operator (the remaining operators have been implemented by Sanjib Das and are described in [1]). Among the operators, $apply\_blocking\_rules$ 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 3.1, then outputs all tuple pairs $(a, b) \in A \times B$ that satisfy at least one rule in $\bar{R}$.

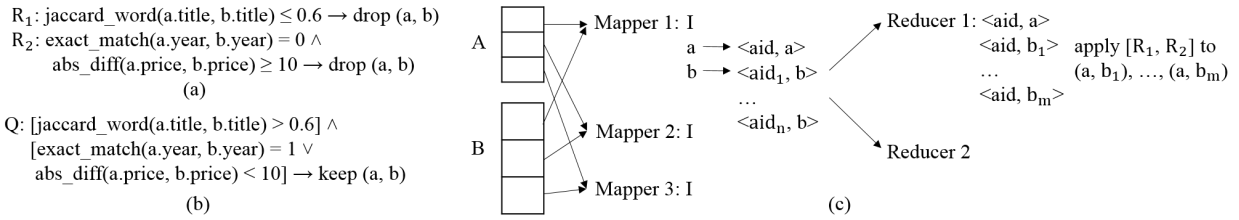

Figure 3.5: (a) A rule sequence, (b) the same rule sequence converted into a single "positive" rule, and (c) an illustration of how $apply\_all$ works.

Example 3.4.1. *Consider the sequence of two rules $[R_1, R_2]$ in Figure 3.5.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 I describe the limitations of the current solutions for this operator, the key ideas underlying our solution, then the implementation of these ideas.

3.4.1 Limitations of Current Solutions

Two MapReduce solutions to apply rules to tuple pairs in $A \times B$ have been proposed: $MapSide$ and $ReduceSplit$ [52].

$MapSide$ 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 I can tell, these are state-of-the-art solutions that can be applied to our setting. (The works [90, 101, 57] 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).

3.4.2 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 3.5.1), and these features in turn often use well-known similarity functions, e.g., edit distance, Jaccard, exact match, etc. (see Example 3.4.1). Thus, we can exploit certain properties of these functions to build index-based filters, then use them to avoid enumerating $A \times B$.

Example 3.4.2. *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 $|y|/t \geq |x| \geq |y| \cdot t$ [104]. 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 I 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.

3.4.3 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 3.4.3. *The rule sequence $[R_1, R_2]$ in Figure 3.5.a is converted into the "positive" rule Q in CNF in Figure 3.5.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., [83, 17]). 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 3.4.4.

Example 3.4.4. *Consider again rule Q in Figure 3.5.b.* Falcon *assigns three filters to predicate $jaccard\_word(a.title, b.title) > 0.6$: length filter, prefix filter, and position filter [104].* 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 3.4.5).

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 3.5.1 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 3.1).

Example 3.4.5. *Consider three mappers into whose memory we already load indexes $\mathcal{I}$ (Figure 3.5.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 $\langle aid, a\rangle$, 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 $\langle aid_1, b\rangle, \ldots, \langle aid_n, b\rangle$ (I 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 3.5.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 $\bar{R}$. See [1] on 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

Algorithm 3.1 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

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. First, in the default mode some mappers process only tuples from A and some process only tuples from B. This incurs highly unbalanced loads. We have optimized so that each mapper processes both A's and B's tuples in a way that evens out the loads. Second, we have minimized the intermediate output size, e.g., by passing only the IDs of the tuples from B to the reducers, instead of passing the whole tuples, whenever possible (this is in addition to compressing the intermediate output.) Finally, we extensively optimized processing rule sequences. 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 $p < 0.5\ AND\ p < 0.2$ into $p < 0.2$ (see [1] for more details).

3.4.4 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 [29] 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 [29] for more details. Next I describe how we construct indexes in Falcon to implement the various filters.

3.4.5 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.

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.

3.5 Plan Generation, Execution, and Optimization

3.5.1 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. Later, operators such as gen_fvs 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 3.4.a. Otherwise we generate the plan in Figure 3.4.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 3.4.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 3.4). 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 3.4.

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 [1] 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 $ReduceSplit$.

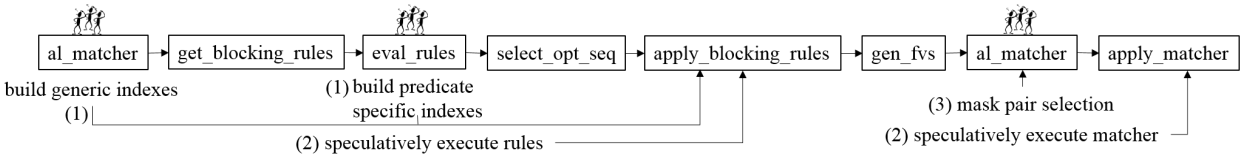

Figure 3.6: Three types of optimization solutions that use crowd time to mask machine time.

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.

3.5.2 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 3.6.

- 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.

I 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 [104].

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.

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 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, I 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

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 3.1: Data sets for our experiments.

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 4.8).

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.

3.6 Empirical Evaluation

We now empirically evaluate Falcon. We consider three real-world data sets in Table 3.1, which describe electronics products, songs within a single table, and citations in Citeseer and DBLP, respectively. Songs and Citations have 1-2.5M tuples in each table, and are far larger than those used in crowdsourced EM experiments so far. See [1] for more details on these data sets.

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. 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.

3.6.1 Overall Performance

We begin by examining the overall performance of Falcon. The first few columns of Table 3.2 show that Falcon achieves high accuracy, 81.9% F_1 on Products and 95.2-97.6% F_1 on Songs

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

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 [42]). Note that each row of Table 3.2 is averaged over three runs. (Table 3.3 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).

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 3.3).

Drug Matching: Recently we have successfully deployed Falcon to match drug descriptions across two tables for a major medical research center. The tables have 453K and 451K tuples.

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 3.3: All runs of Falcon on the data sets.

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.

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.

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

3.6.2 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 3.4 shows the time of the first run on each data set, broken down by operator.

Table 3.4 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, $al\_matcher$ 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 3.4 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 3.2 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 3.4 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 3.4 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 3.5.2). 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).

Operator apply_blocking_rules: The numbers in parentheses in column "apply_block_rules" of Table 3.4 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 $ReduceSplit$ [52]. 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, 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).

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 3.5: Effect of optimizations on machine time.

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 $A \times B$.

3.6.3 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 4.4 shows this unoptimized time, 18m - 2h 12m, for the first run of each data set. 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).

3.6.4 Sensitivity Analysis

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) [42]. Figure 3.7 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

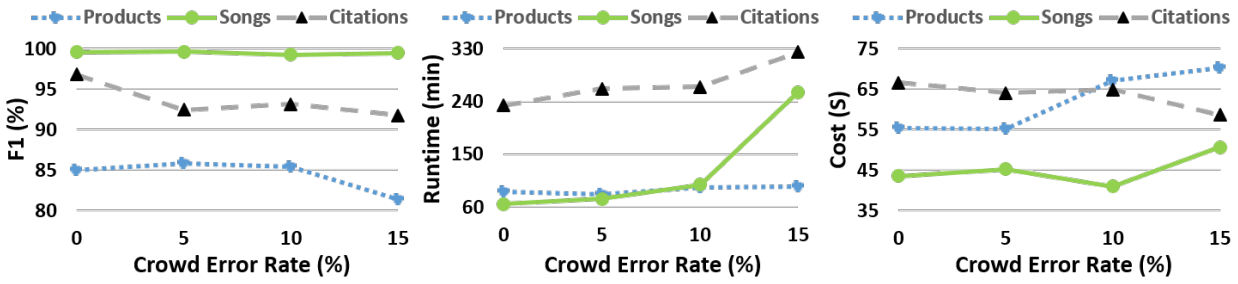

Figure 3.7: Effect of crowd error rate on F_1, runtime, and cost.

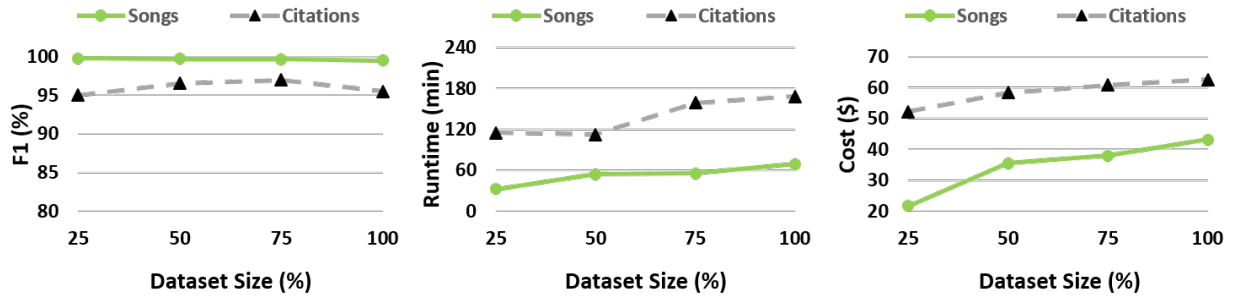

Figure 3.8: Performance of Falcon across varying sizes of Songs and Citations data.

crowdsourcing costs. In any case, recall that there is a cap on crowdsourcing cost (Section 3.2.4) and the costs in Figure 3.7 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 3.8 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 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 3.6.2. 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.

3.7 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., [51, 76, 48, 41]).

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 [51]. 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 [36, 51, 46]. 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 [51], or define operators at granularity levels that are too coarse for the above purposes [36, 46]. 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 [51]. However, such blocking types are not accurate for many real-world data sets, due to dirty/missing data (see Section 3.2.2). 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 [51, 31, 102, 25, 20]. 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 3.2.2). 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 4.8), 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 [53] and meta-blocking [31, 102], 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) [90, 101, 78, 99, 92] and theta joins [68]. To avoid examining all tuple pairs in the Cartesian product, work on SJs uses inverted indexes [83], prefix filtering [17], partition-based filtering [28], and other pruning techniques [101] (see [104] for a recent survey). Some have considered special similarity functions such as Euclidean distance [84] and edit distance [99, 92]. Most works however consider join conditions of just a single predicate [90, 101] or a conjunction of predicates [57], 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, [66] 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 [73]. 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 [64, 74], minimizing crowd latency [47], or studying the trade-offs between the two [32]. 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 3.5.2.

Crowdsourced RDBMSs: Finally, works have proposed crowdsourced RDBMSs [35, 64, 72, 73, 62] and have addressed crowdsourcing enumeration, select, max, count, and top-k queries, among others (e.g., [89, 71, 45, 61, 26, 70, 8]). Crowdsourced joins (CSJs) which at the heart solve the EM problem, have been addressed in [35, 63, 64, 32, 95, 16, 91, 93, 96]. Initial CSJ works [35, 63] however crowdsource all tuple pairs in the Cartesian product of the two tables and hence do not scale. Recent CSJ works [32, 64] 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.

3.8 Conclusion

In this chapter I 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.

I 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. I am 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 4

Helping Lay Users Perform End-to-End String Matching

Most current EM solutions focus on matching relational tuples [29]. These solutions are not optimized for matching strings (e.g., matching two sets of names), which is ubiquitous in practice. As a result, in this direction I focus on this specialized yet common case of EM. I develop an end-to-end string matching solution that lay users can easily use yet obtain significantly higher matching accuracy than current string matching solutions.

4.1 Introduction

String similarity join (also called string matching) finds strings from two given sets that refer to the same real-world entity, such as "Michael J. Williams" and "Williams, Michael" in Figure 4.1. This problem plays a fundamental role in many data management applications, including schema matching, entity matching, value normalization, etc. [104].

	A		B	Matches
a_1	Michael J. Williams	b_1	Williams, Michael	(a_1, b_1)
a_2	Michael J. Smith	b_2	Li, Chen	(a_3, b_2)
a_3	Chen Y. Li			

Figure 4.1: An example of matching two sets of strings.

As a result, over the past few decades, string similarity join (SSJ) has received significant attention. Tremendous progress has been made [104]. Current SSJ works however still suffer from two major limitations.

First, they are not *end-to-end*. That is, most of the current solutions only consider efficiently executing the join condition, ignoring the critical step of coming up with a good join condition. In practice, it is often error-prone and time consuming for the user to select a good similarity measure or pick a good threshold for the join condition.

Second, they consider only join conditions that are *a single predicate*, such as $jaccard\_2gram(a, b) > 0.8$, which tokenizes strings a and b into sets of 2-grams, computes their Jaccard score, then declares a and b matched if this score exceeds 0.8. In practice, using a single predicate for SSJ raises two serious problems. First, many real-world datasets are *heterogeneous*, in that different data regions exhibit different characteristics. They can best be matched using multiple predicates, each of which captures the characteristics of one data region.

Example 4.1.1. *Consider matching two sets of person names that contains both long names (e.g., Shivaram Venkataraman, Christos Papadimitriou) and short names (e.g., Dave Maier, Chen Li). A single predicate such as $jaccard\_2gram(a, b) \geq \epsilon$ does not work well because it is difficult to set the threshold ϵ properly. A high value for ϵ helps match long names accurately, but can be too high for short names, incorrectly predicting many matching short names as non-matches. Conversely, a low ϵ helps match short names accurately, but can be too low for long names. Intuitively, we should use two predicates of the form $jaccard\_2gram(a, b) \geq \epsilon$, but one with a high ϵ for long names, and the other with a lower ϵ for short names. We can check if a name is long using a predicate such as $length(a) > 9$, which returns true if the length of string a (i.e., the number of characters in a excluding space characters) exceeds 9.*

Such heterogeneity arises naturally in a single dataset (e.g., large datasets of person names often contain a mixture of long and short names), or arises because a dataset to be matched is being created by integrating several smaller datasets, each of which contains data of a different nature.

Another serious problem is that real-world strings often contain *substrings with special meaning*. Treating such substrings differently from the rest of the strings can significantly improve the matching accuracy. To do so, however, we need to use multiple predicates.

Example 4.1.2. *Consider matching house addresses. A single predicate such as* $jaccard\_3gram(a, b) \geq \epsilon$ *does not work well. A high* ϵ *(e.g., 0.9) can match addresses correctly, but exclude many matches with lower Jaccard scores, e.g., "522 Wilson St Austin TX 78704" and "522 Wilson Street Austin TX 78704". A lower* ϵ *(e.g., 0.8) helps identify matches such as the above one, but incorrectly matches "522 Wilson St Austin TX 78704" and "422 Wilson St Austin TX 78704", which differ only in the house numbers. To address this problem, we can extract all numbers from each string, then declare two strings match if the strings are highly similar (e.g., using* $jaccard\_3gram(a, b) \geq 0.8$*) and their numbers are also highly similar (e.g., using a predicate such as* $cosine\_num(a, b) \geq 0.8$*).*

To address the above two problems, in this work I describe Smurf (String matching using random forest), a solution that uses multiple predicates in the join condition for SSJ. Experiments in Section 4.8 show that Smurf significantly improves matching accuracy compared to the single-predicate case, by 22.4, 19.03, 10.96, 10.5, and 1.15% absolute F_1 over five datasets, thereby demonstrating the promise of this approach.

To realize Smurf, I identify properties of real-world strings that can be important for matching (e.g., those regarding length, capitalization, special substrings, numeric tokens, etc.), then define a rich set of features to capture these properties. This is so that given any two sets of strings to match, Smurf can automatically generate the features then use them to create multiple predicates.

Next, I consider how to combine the predicates to form join conditions for SSJs. The simplest method is to use a *single matching rule* which is a conjunction of multiple predicates [23, 57], such as

$$[length(a) > 9] \wedge [jaccard\_2gram(a, b) \geq 0.8] \rightarrow match \tag{4.1}$$

But a single rule cannot express "if-then-else" conditions necessary to handle data heterogeneities. To do so, a natural solution is to use a decision tree (DT), which can be viewed as a disjunction of multiple matching rules. But a single DT is often sensitive to noise. As a result, I use random forests (RFs) as join conditions. A random forest F is a set of n decision trees [13]. It declares a string pair a match if at least αn trees in F declare the pair a match (where α is pre-specified).

Random forests are widely used in practice (e.g., [82, 23, 42]), often give very competitive performance, are relatively easy to understand, and are amenable to optimization (as we will see). Section 4.8.1 shows that random forests achieve significantly higher matching accuracy than a single DT.

I then consider how to create a random forest for the join condition. Asking the user to *manually* write a random forest is unrealistic. Instead, I propose to ask him or her to perform *active learning* (by labeling string pairs as match/no-match) to learn a random forest. Several recent works [82, 23, 42] have described such solutions, but for entity matching. I adapt them to our SSJ context. Our solution asks the user to label no more than 400 string pairs. For the five datasets in Section 4.8, users typically need 1-3 seconds to label a pair, or under 20 minutes for 400 pairs.

Once I have learned a random forest F as the join condition, I need to execute it over the two sets of input strings A and B. This raises major scaling challenges. My solution to these forms the key technical contribution of this work.

Specifically, consider a random forest F of n trees. Naively, we can execute each tree on A and B, then combine their outputs (e.g., predicting a string pair a match if at least αn trees predict the pair a match). This however is very time consuming. To address this problem, I propose to (a) execute only a subset of trees on A and B to obtain a relatively small set J of string pairs that are likely to be matches, then (b) execute the remaining trees only on J (instead of on A and B). I show that this solution is guaranteed to be correct, yet takes far less time. I call the above two steps *blocking* and *matching*, as they are similar in spirit to the blocking and matching steps commonly used in entity matching [29]. I show how to select a good subset of trees for the blocking step.

At the heart of both blocking and matching is the need to efficiently execute a set of decision trees (DTs) over two sets of strings (or over a set of string pairs). A DT can be viewed as a disjunction of matching rules (each being a conjunction of predicates, e.g., see the rule in Formula 4.1). Thus, executing a set of DTs reduces to executing a set of rules. Current work has optimized the execution of *individual matching rules* [23, 57] (see Section 4.5.1). But as far as I know, no work has yet optimized the execution of *a set of rules*. Our work develops such a solution. I observe that the matching rules often share a lot of computation, as illustrated below:

Example 4.1.3. *Suppose that a rule contains* $edit\_dist(a, b) < 3$ *and that another rule contains* $edit\_dist(a, b) < 5$*. Then the (relatively expensive) edit distance computation is performed twice. As another example, suppose that two rules contains* $overlap\_word(a, b) > 3$ *and* $jaccard\_word(a, b) > 0.6$*, respectively. Then the overlap computation (i.e., finding the number of words that are common to both* a *and* b*) is performed twice (because computing Jaccard scores also requires computing the overlap).*

To address this problem, I execute the rules jointly, in a way reminiscent of multi-query optimization in RDBMSs [86]. Specifically, I define a small set of core operators that are specific to string contexts. Given a set of rules to be executed, I show how to combine them into a plan (composed of these operators). I define four optimization techniques to remove redundant computations in such a plan. I show how to estimate the runtime of a plan, then how to efficiently search a large space of plans to find one that employs the above optimization techniques to minimize runtime. Finally, I show how to efficiently execute the selected plan. Section 4.8.1 shows that this solution drastically outperforms existing solutions that execute the rules individually, by up to 32x.

In summary, I make the following contributions:

- I show that using multiple predicates, instead of a single predicate as in current SSJ work, can significantly improve SSJ accuracy.
- I present Smurf, a solution that learns a random forest as a join condition for SSJ, then efficiently executes it over two sets of strings.
- I show how to efficiently execute a set of matching rules (that are at the heart of a random forest) by executing them jointly to minimize redundant computation. Our solution significantly outperforms current solutions that only optimize the execution of a single matching rule.
- I describe extensive experiments that show the utility of our approach.

Taken together, our work significantly advances the state of the art of SSJs, and can potentially be applied to other contexts requiring fast execution of decision trees, random forests, or sets of rules.

4.2 Problem Definition

I now provide some background information, then define the SSJ problem considered in this work. I begin by defining:

Definition 1. *[Features and predicates] A feature is a function that takes two strings a and b and returns a numeric value. A predicate $p(a, b)$ is of the form $f(a, b)$ op ϵ, where f is a feature, op is a comparison operator (e.g., $\geq, \leq$), and ϵ is a pre-specified threshold. Given two strings a and b, predicate $p(a, b)$ evaluates to true iff a and b satisfy the comparison, and evaluates to false otherwise.*

For example, feature $jaccard\_3gram(a, b)$ tokenizes strings a and b into sets of 3-grams S_a and S_b, then returns the Jaccard score $|S_a \cap S_b|/|S_a \cup S_b|$. Predicate $jaccard\_3gram(a, b) > 0.8$ evaluates to true iff the Jaccard score exceeds 0.8. In SSJ contexts, features often involve string similarity measures, e.g., edit distance, Jaccard, overlap, etc. [104].

Given two sets of strings A and B, *string similarity join* (SSJ, often also called string matching) is the problem of finding all pairs $(a \in A, b \in B)$ that match, i.e., refer to the same real-world entity. So far the most common solution is to apply a single-predicate join condition to A and B (e.g., $jaccard\_3gram(a, b) > 0.8$) to find all string pairs that satisfy the condition.

Using Indexes to Efficiently Execute a Single Predicate: Applying the SSJ predicate to all pairs in $A \times B$ is often impractical because $A \times B$ can be very large. To address this problem, current work typically builds an index I over a table, say A (sometimes multiple indexes may be built, over both tables). For each string $b \in B$, it then consults the index to locate only a (relatively small) set of strings in A that can potentially match with b. For example, suppose I is an inverted index that, given a token, returns the IDs of all strings in A that contain that token. Then for a string $b \in B$, we can consult I to find only those strings a in A that share at least a token with b, then apply the SSJ predicate only to these (a, b) pairs. The step of consulting the index to locate a small set of pairs is often called *blocking*, and the step of applying the SSJ predicate to these pairs is called *matching*. Numerous blocking techniques for SSJs have been developed, e.g., inverted index, size filtering, prefix filtering, etc. [104].

SSJs with Random-Forest Join Conditions: In this work I will consider the following problem:

Definition 2 (SSJs using random forests)**.** *Given two sets of strings A and B, perform active learning with a user U to learn a random forest F, such that given any two strings $a \in A, b \in B$, we can apply F to predict if they match. Then apply F to A and B to obtain all pairs $(a \in A, b \in B)$ predicted matched.*

In practice, many users (e.g., domain scientists, lay users) do not know how to use (or want to use or have access to) a machine cluster [50, 43]. Thus in this work, as a first step, I will consider solving the above problem on a single machine, deferring solving it on a cluster to future work. I now describe Smurf, our solution to this problem.

4.3 Defining Features

I now define a rich set of features and show how they capture string properties that are important for matching. I show later how Smurf uses the features to create random-forest join conditions.

(a) Combining Tokenizers and Similarity Measures: The most obvious way to create many features for a string pair is to exploit the entire range of similarity measures and tokenizers. First, we can create features that use sequence-based similarity measures, e.g., edit distance, hamming distance. (these measures do not tokenize the strings). Second, we can create features that combine a tokenizer type (e.g., word-level, q-gram) with a set-based similarity measure (e.g., Jaccard, overlap). Third, we can create features that use phonetic measures (e.g., soundex). Finally, we can combine these similarity measures to create more features (e.g., Jaccard over edit distance) [29].

(b) Computing Basic Properties of String Pairs: Examples of such features include computing the lengths of the individual strings, the sum of the lengths, the absolute difference of the lengths, whether a string is capitalized (e.g., the first character of each word, all characters), whether both strings are capitalized, etc.

(c) Exploiting Special Character Sequences: Strings often contain substrings that are sequences of special characters, e.g., numeric, all caps, alphanumeric, such as “326”, “TX”, and “78704” in string “326 Main St Austin TX 78704”, “D246-34” and “41in” in “Sony TV D246-34 41in”.

We can create features that extract sets of such substrings then compare them. For example, one feature may compare two sets of numeric substrings using Jaccard, another feature may compare two sets of alphanumeric substrings, etc.

(d) Exploiting Substrings at Certain Positions: Certain substrings may hold special significance, e.g., the last word in a string may be last name (of a person), the first two words in a string which is a product title may be the product name. As a result, we can create features that extract the k-word prefixes and suffixes of strings, say, then compare them.

(e) User-Defined Features: Smurf automatically creates features of the above four types. But it also gives the user the option to "plug in" blackbox features. For example, if the strings often contain dates, phone numbers, etc., then the user can write "quick-and-dirty" extractors (e.g., regex-based ones) to extract these entities, then create features to compare them.

As described, features of type (a) compute *similarity scores*, and thus are clearly necessary for matching strings. Most current works use these features, but not those of types (b)-(e). Features of type (b) help differentiate different data regions, thus handling *data heterogeneity*. Features of types (c)-(d) help handle *special substrings*. Finally, features of types (e) can be used to handle both.

Exact vs. Approximate Extraction: One may wonder why not ask the user to write features that extract the *exact* special substrings, e.g., the exact last name or house number, then use them to match more accurately. Smurf can naturally use such features (and it should, whenever available). But such *exact extraction* is well-known to be labor-intensive and error-prone [81]. For example, extracting last names is highly non-trivial. There is no clear procedure even for a human to determine just by looking at a full person name (especially foreign names) that which part of this name is the last name, e.g., the last word or the last two words?

An important point that I make in this work is that even when such *exact extraction* is not possible, due to constraints on time and labor, with just the generic features of types (a)-(d), we can already significantly improve matching accuracy, compared to using a single predicate.

Type	Features
Similarity	Jaccard_word, Jaccard_2gram, Jaccard_3gram, Cosine_word, Cosine_2gram, Cosine_3gram, Dice_word, Dice_2gram, Dice_3gram, Overlap_word, Overlap_2gram, Overlap_3gram, Overlap_coeff_word, Overlap_coeff_2gram, Overlap_coeff_3gram, Edit_distance
Heterogeneity	Lengths of individual strings, Sum of lengths, Absolute difference of lengths
Special character sequences	Jaccard_alphabetic, Jaccard_alphanumeric, Jaccard_numeric, Cosine_alphabetic, Cosine_alphanumeric, Cosine_numeric, Dice_alphabetic, Dice_alphanumeric, Dice_numeric, Overlap_alphabetic, Overlap_alphanumeric, Overlap_numeric, Overlap_coeff_alphabetic, Overlap_coeff_alphanumeric, Overlap_coeff_numeric

Table 4.1: The set of features used by Smurf to learn a random forest.

Indeed, the current Smurf uses only 35 features of types (a)-(c). Smurf automatically generates a set of features F defined over a pair of strings. Table 4.1 shows the features of different types used by Smurf. Specifically, Smurf generates 16 features that captures similarity between strings by computing a similarity score for the strings, 4 features that exploit the properties of strings (to handle heterogeneity) and 15 features that extract special character sequences from the strings and computes similarity between them.

Similarity-based features use either sequence-based measure such as edit distance, or combine a tokenizer (such as word-level, q-gram) with a set-based similarity measure such as cosine, Dice, Jaccard, overlap, overlap coefficient, etc. For example, the feature $Jaccard\_word$ takes in two strings, tokenizes each string into a set of words, and then computes the Jaccard score between the two sets of words. The different set-based similarity measures taking as input two sets of tokens X and Y can be computed as follows:

$$Jaccard(X, Y) = |X \cap Y|/|X \cup Y|$$

$$Dice(X, Y) = 2|X \cap Y|/(|X| + |Y|)$$

$$overlap(X, Y) = |X \cap Y|$$

$$overlap\_coefficient(X, Y) = |X \cap Y|/min(|X|, |Y|)$$

$$cosine(X, Y) = |X \cap Y|/\sqrt{|X| \cdot |Y|}.$$

Also Smurf generates features that exploit properties of strings such as lengths of individual strings, absolute difference of lengths etc. Finally, Smurf generates features that extract special substrings from the input strings, and compare them. For example, the feature $Jaccard\_numeric$

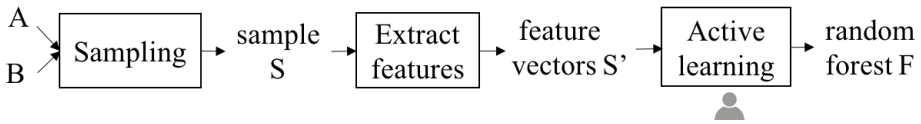

Figure 4.2: Learning a random forest via active learning.

takes in two strings, extracts the numeric substrings in each input string to obtain two sets of numeric substrings, and then computes the Jaccard score between them.

Section 4.8.1 shows that with just this set of generic built-in features (which require no effort from the user), we can already improve SSJ accuracy by 1.15-22.4% F_1 on five datasets. As Section 4.8.1 shows, this is due to using certain features to perform *approximate extraction* (e.g., extracting all numbers from an address can be viewed as an "approximate extraction" of house numbers), and to using certain features to differentiate data regions and match each region differently.

4.4 Learning a Random Forest

Given two sets of strings A and B, I now describe how Smurf learns a random forest (RF) as the join condition. Existing work has developed solutions to learn RFs via active learning [23, 42, 82]. Smurf uses Falcon, the solution in [23], as it is the closest to our SSJ context. As such, this part uses existing solutions and is not viewed as a contribution of our current work.

The goal of Falcon is to match *records* across two tables A and B. To do so, it performs active learning on A and B to learn a RF, as illustrated in Figure 4.2. Specifically, Falcon begins by taking a small sample S of tuple pairs from the Cartesian product of A and B (without materializing this product). This is because learning directly on $A \times B$ is difficult as $A \times B$ is often too large.

Next, Falcon creates a set of features based on the schemas of A and B, then uses them to convert each tuple pair in S into a feature vector. Let S' be the resulting set of feature vectors. Next, Falcon trains an initial random forest F (by asking the user to supply two positive and two negative examples), uses F to select the k "most informative" examples in S', asks the user to label these examples, uses them to retrain F, and so on. This repeats until a convergence condition is met, or the number of iterations has reached a pre-specified n. Falcon then outputs the final random forest F.

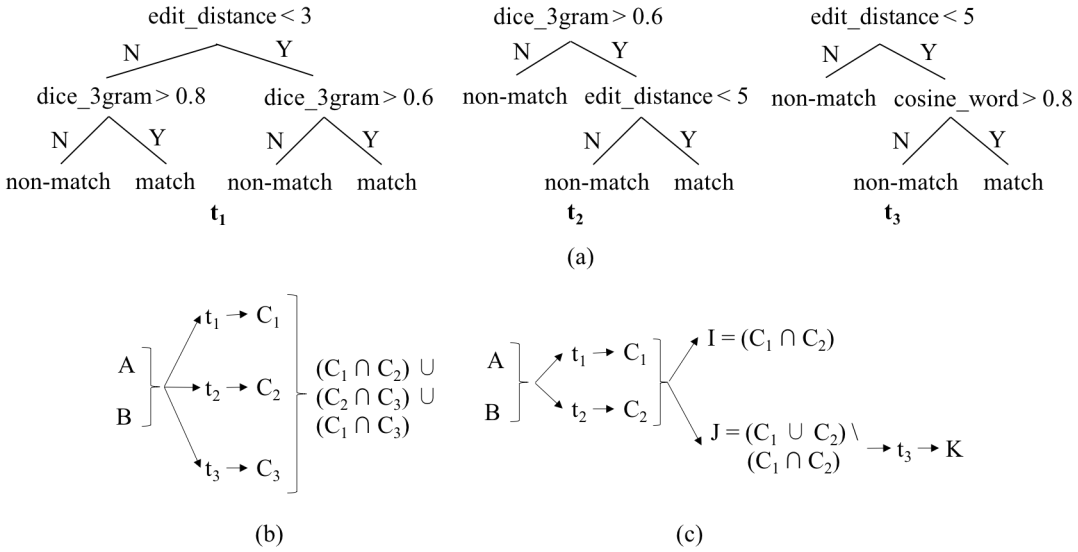

Figure 4.3: An example to motivate the blocking and matching steps.

Falcon can be straightforwardly applied to our context. Specifically, it uses the set of features defined in the previous section, asks the user to label $k = 20$ string pairs in each iteration, runs for at most $n = 20$ iterations (it could terminate earlier if a convergence condition is met), and learn a random forest of 10 trees. These values are selected based on experimental results with Falcon in [23]. Thus the user labels at most 400 string pairs (taking under 20 mins in our experiments). Section 4.8.3 shows that Smurf is robust with respect to varying n, sample sizes, maximal tree depth, and the number of trees in the forest. For further details, including how to take a good sample from A and B, see [23].

4.5 Executing a Random Forest

I now provide an overview of our solution to efficiently execute the learned random forest over two sets of strings, as well as the solution architecture. Subsequent sections describe the main components of the solution in detail.

4.5.1 Solution Overview

Suppose we have learned the random forest F of three decision trees (DTs) t_1, t_2, t_3 in Figure 4.3.a (here, for simplicity, we show each predicate such as $edit\_dist(a, b) < 3$ only as $edit\_dist < 3$). We now consider how to efficiently execute F over two sets of strings A and B. Note that given a string pair $(a \in A, b \in B)$, each tree in F will predict the pair as match or non-match (see Figure 4.3.a). We refer to the set of all string pairs a tree or random forest predicts to be matches as *the output* of that tree or random forest.

Blocking and Matching: Suppose the above random forest F outputs a pair (i.e., predicting it to be a match) only if at least two out of it three trees also output the pair. Naively, we can execute F on two sets of strings A and B by executing each tree t_i on A and B to obtain an output C_i, then output all pairs that appear in the outputs of at least two trees (see Figure 4.3.b). This however is very time consuming.

A better idea is to execute just two trees, say t_1, t_2 on A and B, to obtain outputs C_1 and C_2 (see Figure 4.3.c). The set $I = C_1 \cap C_2$ consists of all pairs predicted match by both t_1 and t_2, and so can be output immediately as a part of output of the random forest F.

The set $J = (C_1 \cup C_2) \setminus (C_1 \cap C_2)$ consists of all pairs predicted match by only one tree (either t_1 or t_2). It is easy to see that we need to apply the remaining tree t_3 only to set J. Let K be the set of pairs in J predicted match by t_3. Clearly, any such pair is also a match for the random forest F, because it is matched by exactly two trees (either t_1 or t_2, together with t_3). The output of random forest F is thus $I \cup K$ (see Figure 4.3.c). Any other pair (i.e., neither in I nor in J) is *not* predicted match by both t_1 and t_2 and hence cannot be a match for F.

In practice, the set J tends to be relatively small (see Section 4.8.2). Thus, applying tree t_3 to J tends to be much faster than applying it to the (potentially large) sets of strings A and B. This time saving is significant when F is large, say 10 trees. Suppose in this case we need at least five trees to match in order for F to match. Then we can apply six trees to A and B to obtain sets I and J, then apply the remaining four trees to just the relatively small set J.

Smurf uses the above idea. I refer to the first step of applying a subset of trees as *blocking*, in the sense that it "blocks" a large portion of string pairs, only allowing a relatively small set of pairs J to continue in the SSJ pipeline. I refer to the second step of applying the remaining trees to J as *matching*.

Executing a Tree by Executing Its Matching Rules: The blocking step must execute a set of DTs over sets of strings A and B. Continuing with the example in Figure 4.3, let us consider how to execute the first tree t_1 (Figure 4.3.a).

I refer to each path from the root of t_1 to a "match" node as a *matching rule*. Figure 4.4.a shows the two rules extracted from t_1. Each rule is a conjunction of two predicates. Rule r_1 for instance captures the first path leading to a "match" node in the tree. Note that since on this path the predicate $(edit\_dist < 3)$ takes value "N", i.e, false, in rule r_1 we capture this as $(edit\_dist \geq 3)$.

Executing t_1 then reduces to executing these two rules on A and B, then combining their output. Figure 4.4.b illustrates this step. Note that the output of the tree is the *union* of the outputs of the rules, because any pair predicted match by a rule is also predicted match by the tree.

Current Work on Rule Execution: I now consider how to execute a matching rule efficiently. No current work has addressed this problem explicitly for string matching. But two recent works [23, 57] have addressed it for record matching and can be applied to this context.

The first work, Falcon [23], proposes two solutions: ApplyAll and ApplyGreedy. To execute a rule such as r_1 in Figure 4.4.a, ApplyAll builds indexes for all predicates in the rule, i.e., for both $(edit\_dist \geq 3)$ and $(dice\_3gram > 0.8)$. It then consults all indexes and take the intersection of the outputs of these indexes to be the set H of record pairs that can possibly be in the rule output. Finally, it evaluates the rule on H. ApplyGreedy builds an index for just one predicate in the rule, consults the index to find a set H of record pairs, then applies the rule to H. The key challenge is to select a predicate that is highly selective to build an index on, to minimize the size of H.

The second work [57] proposes a solution that I will refer to as RAR (Rule Applied to Record Pairs). RAR analyzes the entire rule, builds a single index covering all predicates in the rule (using prefix filtering ideas), uses the index to find a set H, then applies the original rule to H.

Limitations of Current Work & Our Solution: As described, current work has developed efficient solutions to execute a *single* matching rule. Executing a DT however requires executing a *set* of rules, e.g., rules r_1, r_2 for tree t_1 (Figure 4.4.a). As far as I can tell, no work has examined efficiently executing a set of matching rules. (Falcon [23] does consider a set of rules, but those are blocking rules, not matching rules as considered here. When adapting Falcon to matching rules, it is easy to show that Falcon is capable of executing only a single such rule.)

Executing a set of rules by executing each rule individually can be very time consuming. I observe that these rules *often share a lot of computation*. So I seek to execute them jointly, in a "multi-query optimization" style. Specifically, I define a set of core operators, use them to generate a plan that encodes the rules, then optimize and execute the plan.

For example, consider again the two rules r_1, r_2 in Figure 4.4.a. Predicates $(dice\_3gram > 0.8)$ of r_1 and $(dice\_3gram > 0.6)$ of r_2 share computation. The plan in Figure 4.4.c executes both rules jointly, by reusing this computation. It uses three operators: join, select, and filter (see Section 4.6.1). Specifically, the plan first performs a single-predicate SSJ between A and B, using join condition $dice\_3gram > 0.6$. Next, it selects from the output of this join all pairs where feature $dice\_3gram$ (which has been computed) exceeds 0.8 (see the left path of the plan), then it computes feature $edit\_dist$ for these pairs and selects only those with $edit\_dist \geq 3$. This produces the set D_1, the output of rule r_1. Similarly, the right path of the plan produces D_2, the output of rule r_2. The plan returns $C_1 = D_1 \cup D_2$, the output of tree t_1. Note how computing the feature $dice\_3gram$ is done only once in this plan (in the "join" node).

So far I have discussed executing a single DT. The blocking step executes a set of DTs. It is easy to see that the above idea generalizes to this case: we simply extract all matching rules from the trees, then execute them in a joint fashion. (We do need to make sure that we know which output pair comes from which tree.) For example, to execute trees t_1, t_2 (Figure 4.3.a), we extract their three matching rules, then combine the rules to form the plan in Figure 4.4.d. Note how this plan returns both C_1 and C_2, the outputs of trees t_1 and t_2, respectively. The matching step also executes DTs and thus will also use the above joint execution idea.

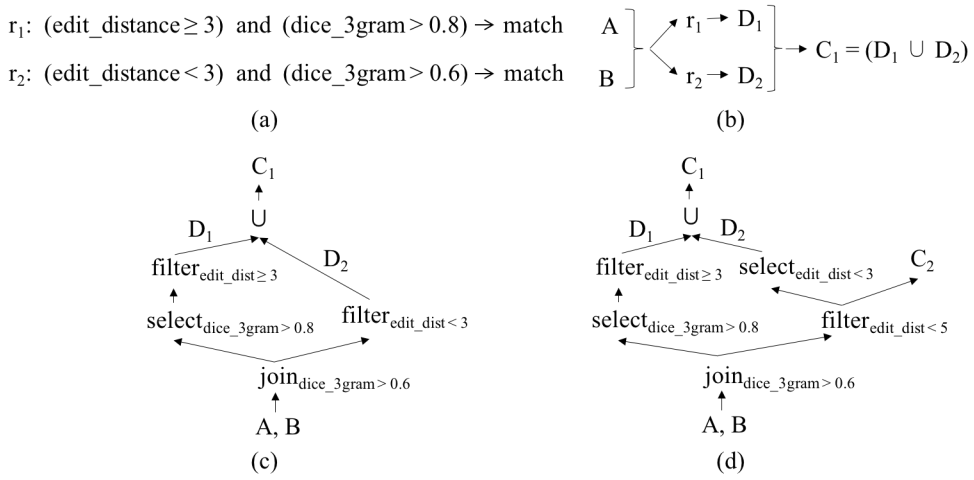

Figure 4.4: Executing rules in a joint fashion.

4.5.2 Solution Architecture

The above ideas lead to the overall architecture for executing a random forest F in Figure 4.5. Given the set of trees T in F, I first select a subset T' of trees to perform blocking. This produces a set of predicted matches I and a set of candidate pairs J. Next, in the matching step, I apply the remaining trees $T \setminus T'$ to J, to obtain a set of matches K. The output of F is then $I \cup K$. Both the blocking and matching steps rely on a module that provides efficient execution of a set of DTs, using operators, indexes, cache, plan generation, optimization, and execution.

Realizing this architecture raises three challenges: (1) how to select a subset of trees for the blocking step? (2) how to execute the remaining trees in the matching step? and (3) how to execute a set of DTs, by extracting the rules, defining operators, then generating, optimizing, and executing a plan? I now discuss our solutions to these challenges. First I discuss (3), then build on it to discuss (1) and (2).

4.6 Optimizing and Executing a Set of Decision Trees

I now describe how to efficiently executing a set of DTs. I consider the following concrete problem (and show later how it can be used for blocking and matching):

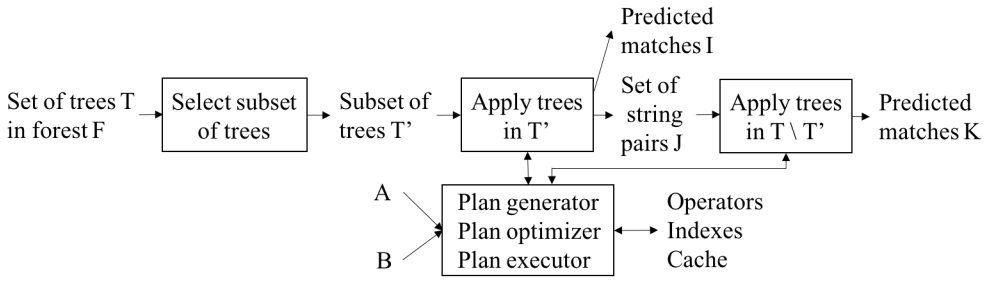

Figure 4.5: The process of executing a random forest in Smurf.

Definition 3 (Executing a set of DTs over two sets of strings)**.** *Let G be a set of decision trees. Given two sets of strings A and B, return all pairs $(a, b) \in A \times B$ that is a match output by at least a tree in G, and associate with each such pair a set E of the IDs of all trees in G that output that pair.*

To solve the above problem, I begin by extracting each path from the root of a tree in G to a "match" node as a *matching rule*. Each rule r_i is of the form $p_1^i(a, b) \wedge \ldots \wedge p_{m_i}^i(a, b) \rightarrow predict\ (a, b)\ as\ match$, where each predicate $p_j^i(a, b)$ is of the form described in Definition 1. Figure 4.4.a shows two matching rules extracted from tree t_1 in Figure 4.3.a.

Let R be the set of all matching rules extracted from the trees in G. Executing G reduces to executing the rules in R, then union their outputs (but annotate each pair in the output with the IDs of all the rules that predict that pair to be a match).

As discussed earlier, executing the rules in isolation is inefficient. So I seek to execute them jointly, by sharing computation. To do so, I define a set of operators, convert the set of rules into a plan composed of these operators, develop optimization techniques, then search a large plan space to select a plan that uses these techniques to minimize runtime. I now discuss these steps.

4.6.1 Operators and Default Plan Generation

I define the following four operators (and motivate them below):

$\boldsymbol{join_p(A, B)}$**:** This operator takes two sets of strings A and B and a predicate p, and returns all pairs $(a, b) \in A \times B$ that satisfies p. For example, given predicate $jaccard\_word(a, b) > 0.5$, this operator returns all pairs (a, b) with Jaccard score above 0.5.

$\boldsymbol{filter_p(C)}$**:** This operator returns all string pairs $c \in C$ that satisfies predicate p. It assumes that feature f in p has *not* been computed for the pairs in C. So given a pair $c \in C$, it computes f for c, then outputs c if c satisfies p. For example, given $jaccard\_word(a, b) > 0.5$, this operator computes feature $jaccard\_word$ for each $c \in C$, then outputs c if c satisfies the predicate.

$\boldsymbol{select_p(C)}$**:** This operator is the same as $filter_p(C)$, but it assumes feature f in p has been computed for all pairs in C. So it simply evaluates p for each $c \in C$ and outputs c if p evaluates to true.

$\boldsymbol{feature_f(C)}$**:** This operator assumes feature f has not been computed for pairs in C. So it computes f for all pairs in C then returns those pairs.

Operator $join_p(A, B)$ performs a single-predicate SSJ, and has been studied intensively [104]. Operator $filter_p(C)$ is typically applied to string pairs coming out of a $join_q(A, B)$ operator, as we will see below. To motivate operators $select_p(C)$ and $feature_f(C)$, suppose in a plan (defined below) we execute a $join_q(A, B)$ operation to obtain a set of pairs C, then execute both operations $filter_{jac\_word>0.6}$ and $filter_{jac\_word<0.8}$ on C. Then we would compute feature jac_word twice. To avoid this, we can execute operation $feature_{jac\_word}$ once, followed by two select operations $select_{jac\_word>0.6}$ and $select_{jac\_word<0.8}$.

Default Plan: I now discuss how to convert a set of rules into a default plan (later I show how to rewrite this plan into a set of plans, then select the best one). First, we convert each rule into a plan. Specifically, given each rule $p_1(a, b) \wedge \ldots \wedge p_m(a, b) \rightarrow match$, we construct a plan $A, B \rightarrow join_{p_1} \rightarrow filter_{p_2} \rightarrow \ldots \rightarrow filter_{p_m} \rightarrow C$. This plan performs $join_{p_1}$ on sets of strings A and B (using indexes, see Section 4.2), applies $filter_{p_2}$ to the output of the join, applies $filter_{p_3}$ to the output of $filter_{p_2}$, etc., until producing the output C. We then merge the individual plan by adding a node to union their outputs, to obtain a "global" default plan.

Example 4.6.1. *We convert the set of two rules* $r_1 : (edit\_dist < 5) \wedge (jac\_2g > 0.5) \rightarrow match$ *and* $r_2 : (dice\_3g > 0.7) \wedge (edit\_dist < 7) \rightarrow match$ *into the default plan in Figure 4.6.a (ignore the dotted boxes and the notations* P_1, P_2 *for now).*

A plan is thus a directed acyclic graph (DAG), where the root nodes (those with no incoming edges) denote input data (e.g., sets A and B), the leaf nodes (those with no outgoing edges) denote output data (e.g., C), the remaining nodes denote the four operators described above plus the set union operator, and the edges denote the flow of data among the operators.

4.6.2 Strategies for Reusing Computation

Given a plan G many possible strategies exist for reusing computation within G. As a first step, in this work I propose four such strategies. The key idea is to identify plan fragments that often share computation, then analyze how to merge them to enable reuse. We focus in particular on a common kind of fragment called reusable paths:

Definition 4 (Reusable path)**.** *Given a plan* G*, which is a DAG, a reusable path* P *is a path in graph* G *of the form* $o_1 \rightarrow o_2 \rightarrow \ldots \rightarrow o_n$*, such that (a) each node* o_i *is an operator and has exactly one incoming edge and one outgoing edge in graph* G*, and (b)* P *is the longest such path, i.e., we cannot extend path* P *before* o_1 *or after* o_m *to obtain a longer path that still satisfies (a).*

When there is no ambiguity, I will use "path" instead of "reusable path". I refer to nodes o_1 and o_m as the root and leaf nodes of a path, respectively, and the node with an edge leading to o_1 as the parent node of the path. Figure 4.6.a shows two reusable paths P_1 and P_2 (denoted with dotted boxes).

I now describe four reuse strategies for paths: join reuse, inter-path filter reuse, intra-path filter reuse, and filter ordering.

1. Join Reuse: This strategy merges two paths with joins to enable join reuse. Consider the two paths P_1 and P_2 in Figure 4.6.a. Path P_1 performs $join_{edit\_dist<5}$, while P_2 performs $join_{dice\_3g>0.7}$. These two joins are different and cannot be shared. Observe however that P_2 contains a node $filter_{edit\_dist<7}$. We can push this node down to become the root node of P_2, then merge it with

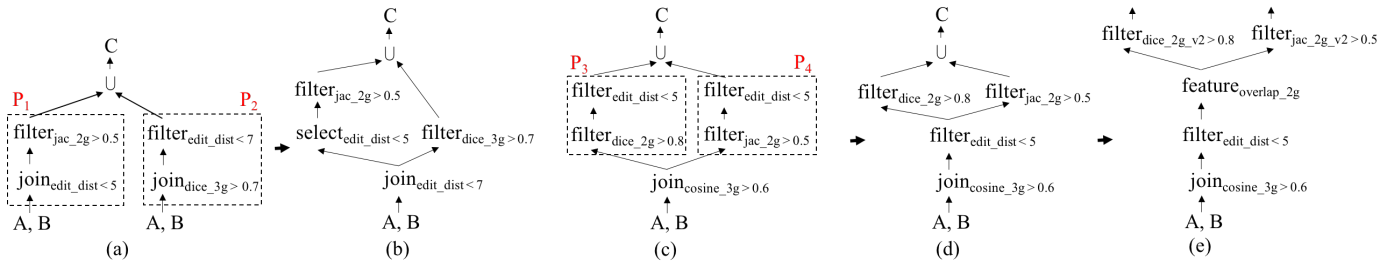

Figure 4.6: Default plan generation, join reuse, and inter-path filter reuse.

the root node $join_{edit\_dist<5}$ of P_1, to obtain the plan fragment in Figure 4.6.b, which reduces the number of joins from two to one. To realize this idea, I first define

Definition 5 (Predicate containment)**.** *Let p_1 and p_2 be two predicates defined over the same feature f, E denote a set of string pairs and $p_i(E)$ denote the result of applying p_i over E. I say that (a) p_1 is contained in p_2, denoted $p_1 \sqsubseteq p_2$, iff for any E, $p_1(E) \subseteq p_2(E)$, and (b) p_1 is equivalent to p_2, denoted $p_1 \equiv p_2$, iff $p_1 \sqsubseteq p_2$ and $p_2 \sqsubseteq p_1$.*

For example, for predicates $p_1 : jac\_3g(a, b) > 0.5$ and $p_2 : jac\_3g(a, b) > 0.7$, we have $p_2 \sqsubseteq p_1$.

Join reuse then works as follows. It takes as input two paths P_1 and P_2 such that (a) both roots are join operations, and (b) the parents are the same input (e.g., two sets of strings A and B). We first find a node n_i containing predicate $p(n_i)$ in P_1, and a node n_j containing predicate $p(n_j)$ in P_2 such that either $p(n_i) \equiv p(n_j)$, $p(n_i) \sqsubseteq p(n_j)$ or $p(n_j) \sqsubseteq p(n_i)$. If n_i, n_j exist, then we push them down the paths to become the two new roots (the old roots become new filter nodes). Then we merge the two paths. Specifically, if $p(n_i) \equiv p(n_j)$, then we delete n_j and append the rest of path $P2$ to n_i. If $p(n_i) \sqsubseteq p(n_j)$, then we modify n_i to be a selection operator and append it as a child of n_j (see Figure 4.6.b), and so on (see Algorithm 4.1 for the pseudo code).

Note that the above describes *one* join reuse rule. If multiple combinations of n_i, n_j exist, then each combination gives rise to a join reuse rule. Later we use these rewrite rules to generate a space of alternative plans.

2. Inter-path Filter Reuse: In this strategy I consider two paths that share the same parent, identify filters (across the paths) that perform common computation, then merge/modify them to

Algorithm 4.1 Join Reuse

1: **Input:** Two paths P_1 and P_2 such that the root of P_1 and P_2 are join operators and their parents are the same input. Nodes n_1 in P_1 and n_2 in P_2 such that $p(n_1) \equiv p(n_2)$, $p(n_1) \sqsubseteq p(n_2)$ or $p(n_2) \sqsubseteq p(n_1)$.
2: **Output:** Root of the merged plan fragment
3:
4: Move n_1 as root of P_1
5: Move n_2 as root of P_2
6: **if** $p(n_1) \equiv p(n_2)$ **then**
7: Move path rooted at child of n_2 as child of n_1
8: Delete n_2
9: **return** n_1
10: **else if** $p(n_1) \sqsubseteq p(n_2)$ **then**
11: Modify n_1 to be select operator
12: Move path rooted at n_1 as child of n_2
13: **return** n_2
14: **else if** $p(n_2) \sqsubseteq p(n_1)$ **then**
15: Modify n_2 to be select operator
16: Move path rooted at n_2 as child of n_1
17: **return** n_1
18: **end if**

reuse the computation. I distinguish two cases:

(a) Reusing filters with the same feature: To motivate, consider the two paths P_3 and P_4 in Figure 4.6.c, which share the same parent $join_{cosine\_3g>0.6}$. Both paths execute $filter_{edit\_dist<5}$. So we can push this filter down to be the root of each path, then merge them, to produce the plan fragment in Figure 4.6.d, which performs the above filter only once.

More generally, this strategy works as follows. Given two paths P_1 and P_2 sharing the same parent, if we find a filter node n_i containing predicate $p(n_i)$ in P_1, and a filter node n_j containing predicate $p(n_j)$ in P_2 such that $p(n_i)$ and $p(n_j)$ are defined over the same feature f, then we rewrite P_1 and P_2 by pushing n_i and n_j down to be the root nodes of the paths. Next, we merge n_i and n_j. If $p(n_i) \equiv p(n_j)$, then we delete n_j and append the rest of P_2 to n_i. If $p(n_i) \sqsubseteq p(n_j)$, then we

modify n_i to be a selection operator and append it as a child of n_j. If none of these holds, then we add a new feature node n_f that computes the feature f as a child of the parent node, move n_i and n_j to be n_f's children, then make n_i and n_j into select nodes.

(b) Reusing filters with correlated features: To motivate, consider again the plan in Figure 4.6.d. Consider the path P_5 consisting of the sole operation $filter_{dice\_2g>0.8}$ and the path P_6 consisting of the sole operation $filter_{jac\_2g>0.5}$. These two filters do not share the same feature, and hence cannot benefit from the reuse strategy in Case (a). However, these features are *correlated*, in that they perform some common computation. Indeed, $dice(X,Y) = 2|X \cap Y|/(|X| + |Y|)$ and $jac(X,Y) = |X \cap Y|/|X \cup Y|$. So they both compute the overlap feature $|X \cap Y|$.

To reuse this computation, we can modify the plan fragment in Figure 4.6.d into that in Figure 4.6.e. In this new plan fragment, we first execute $feature_{overlap\_2g}$, then execute the above two filters. However, we rewrite these filters with new features. Consider filter $filter_{dice\_2g>0.8}$. Feature $dice\_2g$ of this filter performs a full computation of the Dice score, i.e., computing the overlap, among others. But now $feature_{overlap\_2g}$ already computes the overlap. As a result, we define a new feature $dice\_2g\_v2$, which also computes the Dice score, but assumes that the overlap information already exists (and stored with the incoming string pair). As a result, it does not compute the overlap again, thereby saving time compared to the old feature $dice\_2g$. Thus, we rewrite $filter_{dice\_2g>0.8}$ into $filter_{dice\_2g\_2v>0.8}$, and similarly rewrite $filter_{jac\_2g>0.5}$ into $filter_{jac\_2g\_2v>0.5}$ (see Figure 4.6.e). Algorithm 4.2 provides the pseudo code of this reuse strategy.

3. Intra-path Filter Reuse: This strategy is similar to inter-path filter reuse, but applies to filters within a single path. Here I can also distinguish two cases:

(a) Reusing filters with the same feature: Within a single path, we also often have multiple filters with the same feature. (Such paths encode rules extracted from decision trees, and these rules often have multiple predicates with the same feature.) In such cases, we can reuse computation across these filters. For example, the path in Figure 4.7.a has two filters involving feature $edit\_dist$. Clearly we can rewrite the second filter as a select operation, because $edit\_dist$ has been computed in the first filter (see Figure 4.7.b).

Algorithm 4.2 Inter-path Filter Reuse

Input: Two paths P_1 and P_2 with same parent. Nodes n_1 in P_1 and n_2 in P_2 such that n_1 and n_2 are filter operations such that $p(n_1)$ and $p(n_2)$ are defined over the same or correlated features.
Output: Root of the merged plan fragment

Move n_1 as root of P_1
Move n_2 as root of P_2
set_measures $\Leftarrow \{cosine, Dice, Jaccard, overlap, overlap\ Coeff.\}$
if $p(n_1) \equiv p(n_2)$ **then**
    Move path rooted at child of n_2 as child of n_1
    Delete n_2
    return n_1
else if $p(n_1) \sqsubseteq p(n_2)$ **then**
    Modify n_1 to be select operator
    Move path rooted at n_1 as child of n_2
    return n_2
else if $p(n_2) \sqsubseteq p(n_1)$ **then**
    Modify n_2 to be select operator
    Move path rooted at n_2 as child of n_1
    return n_1
else if $sim(f(n_1)) \in$ set_measures and $sim(f(n_2)) \in$ set_measures and $tok(f(n_1)) == tok(f(n_2))$ **then**
    Create a feature node n_f that computes $overlap$ feature
    Modify $f(n_1)$ and $f(n_2)$ to be their corresponding new feature that assumes that $overlap$ is precomputed
    Modify n_1 and n_2 to be select operators
    Move paths rooted at n_1 and n_2 as children of n_f
    return n_f
else
    Create a feature node n_f that computes feature f
    Modify n_1 and n_2 to be select operators
    Move paths rooted at n_1 and n_2 as children of n_f
    return n_f
end if

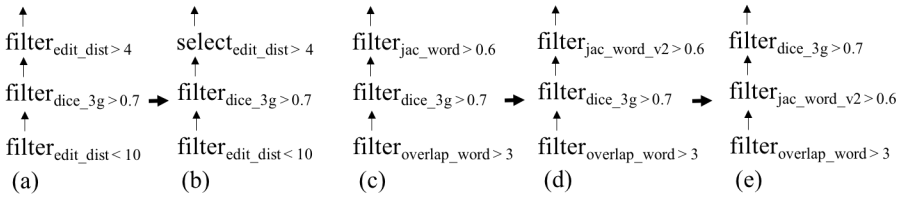

Figure 4.7: Intra-path filter reuse and ordering.

(b) Reusing filters with correlated features: Within a single path, we also often have filters that have different, but *correlated* features. We can also share computation among these filters, in a way similar to the case of inter-path filter reuse. Consider for example the path in Figure 4.7.c. Here features $overlap\_word$ and jac_word are different, but correlated: computing the Jaccard score requires computing the overlap. As a result, we can rewrite operation $filter_{jac\_word>0.6}$ as $filter_{jac\_word\_v2>0.6}$ (see Figure 4.7.d), where feature jac_word_v2 is a new feature that also computes the Jaccard score, but assumes that the overlap has been computed and stored with the incoming string pair.

4. Filter Ordering: Within a path the filters can be re-ordered (i.e., moved around) without affecting the output of the path. Different orderings however can significantly affect the runtime of the path. Consider again the path in Figure 4.7.d. Here $filter_{jac\_word\_v2>0.6}$ is quite fast, because it assumes the overlap information has been computed (by the upstream $filter_{overlap\_word>3}$). On the other hand, $filter_{dice\_3g>0.7}$ is slow. If we re-order these two filters, to obtain the path in Figure 4.7.e, then the slow $filter_{dice\_3g>0.7}$ is applied to fewer string pairs, and thus the entire path may execute much faster. As a result, in this strategy given a path we seek to find a good ordering of its filters. This raises two challenges: how to estimate the runtime of an ordering and how to search the large space of possible orderings. I now describe our approach.

Given a set of filters $U = \{u_1, ..., u_m\}$ in path P, our goal is to find an optimal sequence of the filters such that the time taken to execute the sequence over a set of input string pairs is minimized. This problem is NP-hard as shown in [11]. Specifically, [11] 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 [67].

Algorithm 4.3 Intra-path Filter Reuse

Input: Path P. Nodes n_1 and n_2 in P such that n_2 appears after n_1 in P and are defined over the same or correlated features .
Output: Root of the rewritten path

if $f(n_2)$ assumes overlap is precomputed **then**
    return P
end if
if $f(n_1)$ == $f(n_2)$ **then**
    Modify n_2 to be select operator
else if $f(n_1)$ assumes overlap is precomputed or $f(n_1)$ is $overlap$ feature **then**
    Modify $f(n_2)$ to be the new feature that assumes overlap is precomputed
else
    Create a feature node n_f that computes $overlap$ feature
    Move the path rooted at n_1 as child of n_f
    Modify $f(n_1)$ and $f(n_2)$ to be their corresponding new feature that assumes overlap is precomputed
end if
return P

The problem in [11] is equivalent to ours. To elaborate, [11] considers the problem of optimally ordering a set of n filters $\{u_1, u_2, ..., u_n\}$ in conjunction, where each filter u_i takes a tuple t in a stream as input and returns either true or false. If u_i returns false for tuple t, then u_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 finding an optimal sequence of filter operators where each filter operator either drops or allows a string pair.

The work [67] 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 [11] proposes a 4-approximation greedy algorithm to optimally order pipelined filters for data streams. We apply this greedy algorithm to our problem. Specifically, let $w(u_i)$ be the average time to apply filter u_i over a string pair and $sel(\{u_1, .., u_i\})$ be a selectivity factor of the set of filters $u_1, ..., u_i$.

We begin by choosing filter u_i that maximizes $(1 - sel(\{u_i\}))/w(u_i)$, then choose filter $u_j (j \neq i)$ that maximizes $(1 - sel(\{u_i, u_j\}))/w(u_j)$, and so on. The chosen filters form the selected sequence (in that order). [11] shows that this solution finds a sequence whose estimated runtime is at most four times the estimated runtime of the optimal sequence.

Extending Reuse Strategies to Feature Operations: Finally, it is easy to see that the reuse strategies described above can also be easily extended to cover feature operations, because each feature operation can be viewed as a filter operation with a trivial selection that lets all input tuples go through. We have

Proposition 2. *The above reuse strategies are correct in that if we have applied any of them to transform a plan G_1 into a new plan G_2, then G_1 and G_2 are equivalent, i.e., they produce the same output on any two sets of strings A and B.*

4.6.3 Searching for the Best Plan

I now describe how to search for the best plan. Note that the reuse strategies in the previous section give rise to a set of rewrite rules, each of which rewrites a plan into a potentially better plan.

So a simple search strategy is to start with the default plan G (see Section 4.6.1), apply all possible rewrite rules repeatedly, until we cannot apply any more rules, to obtain a place space $\mathcal{G}$.

Example 4.6.2. *Suppose that (a) we can apply only two rules to G, producing two new plans G_1 and G_2, (b) we can apply only one rule to G_1, producing a new plan G_3, and (c) we can apply no other rules to any plan. Then $\mathcal{G} = \{G, G_1, G_2, G_3\}$.*

In the next step, we estimate the runtime of each plan in $\mathcal{G}$ (see Section 4.6.4), and select the fastest plan. $\mathcal{G}$ however is often huge (e.g., hundreds of millions to billions of plans for our experiments), rendering this strategy impractical.

Staged Search: As a result, I explore the following staged search strategy. Given the default plan G, we apply (a) all possible *join reuse* rules repeatedly (until we cannot apply any further), then

(b) all possible inter-path filter reuse rewrite rules, then (c) all possible filter ordering rules, and finally (d) all possible intra-path filter reuse rules.

The rationale for this ordering of the rules is as follows. First, joins are very expensive. So we want to do (a) first, to consider all possible join reuse opportunities. We can delay (c) and (d) to the end, because they are *local* rules and do not increase the estimated runtime of any target plan (as I discuss below). This leaves inter-path filter reuse rules to be executed in (b). Finally, we do (c) before (d) because it is not difficult to prove that if there is any intra-path filter reuse we want to perform for a path, we can always perform it (or another reuse with equivalent effect) *after* we have performed filter ordering for the path.

Let $\mathcal{U}$ be the resulting plan space. We can reduce $\mathcal{U}$ somewhat, by observing that applying filter ordering or intra-path filter reuse rules does *not* increase the estimated time of the plan. Formally,

Proposition 3. *Suppose applying a filter ordering or intra-path filter reuse rule as described in Section 4.6.2 to a plan P yields a new plan P'. Then the runtime of P' does not exceed that of P, where the runtimes are estimated using the procedure in Section 4.6.4.*

As a result, if we rewrite a plan P into P' using one of the above rules, we can drop P from $\mathcal{U}$. We then estimate the runtime for each plan in $\mathcal{U}$ and select the fastest plan.

Incremental Staged Search: Unfortunately, the plan space $\mathcal{U}$ is still huge (e.g., 100+M plans in our experiments). As a result, we perform an incremental staged search that explores a much smaller space yet finds good plans (see Section 4.8).

Specifically, let R be a set of n matching rules to be executed. We first sort the rules in R in some order $r_1, \ldots, r_n$ (discussed below). Next, we convert the set of the first two rules r_1 and r_2 into a default plan P_{12}, then perform staged search (as described earlier) on it to find the best plan P^*_{12}. Next, we convert rule r_3 into a default plan P_3, merge it with plan P^*_{12} (by adding a node that unions their output), to form a new plan P_{123}. Then we perform staged search on P_{123}, to find the best plan P^*_{123}. During this search, however, we fix the plan fragment P^*_{12}, applying rewrite rules only to the rest of plan P_{123}. Next, we convert rule r_4 into a default plan P_4, then merge it with P^*_{123}, etc., until we have processed the last rule r_n.

I now discuss how to sort the rules in R. The key idea is to give the maximal amount of freedom in selecting a join operator to the rule whose minimal estimated runtime is higher than that of other rules. As a result, *we sort the rules in the decreasing order of their minimal estimated runtime*. Specifically, for each rule $r_i \in R$, we first enumerate all plans where a predicate in r_i becomes a join operator and the remaining predicates become filter operators. Then for each such plan P we perform all filter ordering and intra-path filter reuse rewriting, which can only help reduce P's runtime (see Proposition 3). Finally, we estimate the runtimes of these plans (see Section 4.6.4), then select the lowest runtime to be the minimal estimated runtime of rule r_i.

4.6.4 Plan Cost Estimation

As the final piece in the optimization puzzle, I describe estimating plan costs, i.e., runtimes. In our context, a plan P is a DAG of operators (see Section 4.6.1). To execute P, we read the sets of strings A and B from disk into memory, execute the DAG in memory, then write the output to disk. As a result, we will only estimate the CPU time of executing the DAG (the I/O time is the same for all plans), which is the sum of the CPU times of all operations in the DAG.

Thus, we need to estimate the CPU time for each operation in the DAG. There are five types of operator: select, feature, filter, join, and union. Since unions take negligible time, we only need to estimate time for the first four types of operator. For each operator type, we need to estimate its runtime as well as the size of the output relative to the size of the input (which we need for estimating the runtime of any operator that consumes the output of this operator).

In what follows I briefly describe the cost model that estimates these two quantities for each operator type.

$\boldsymbol{select_p(C)}$**:** applies a predicate p to each pair in C to obtain an output C_{out}. Then we estimate the output size as $|C_{out}| = \rho_p \cdot |C|$, where ρ_p is a selectivity factor for predicate p. We estimate the runtime of this operator as $\alpha \cdot |C|$, where α is the average time to apply p to a pair (this time involves just a single comparison, hence it is very small and assumed to be the same regardless of p). The cost model of this operator thus requires estimating ρ_p and α.

$\boldsymbol{feature_f(C)}$**:** computes feature f for each pair in C. Thus the output size is the same as the input size. The runtime is estimated as $\beta_f \cdot |C|$, where β_f is the average time to compute feature f for a string pair.

$\boldsymbol{filter_p(C)}$**:** computes a feature f (specified by predicate p) and applies p to each pair in C, then output only those pairs satisfying p. We estimate the output size to be $\rho_p \cdot |C|$, where ρ_p is a selectivity factor for predicate p, and the runtime to be $(\beta_f + \alpha) \cdot |C|$ (because for each pair in C it takes time β_f to compute feature f and time α to apply p.

$\boldsymbol{join_p(A, B)}$**:** returns all pairs in $A \times B$ that satisfy predicate p. Thus, we estimate the output size as $\rho_p \cdot |A \times B|$, where ρ_p is the selectivity factor of predicate p. Estimating the runtime of this operator is more involved. Given two sets of strings A and B, this operator first builds an index I on A. Then for each string $b \in B$, it probes I to obtain a relatively small set of strings $Q(b)$ in A. Finally, it processes each pair (b, q), where $q \in Q(b)$, by computing feature f for the pair (the feature mentioned in predicate p), applying predicate p, then outputting the pair if it satisfies p.

Thus, the runtime of this operator consists of the times for index building, index probing, and processing of string pairs. We estimate the index building time to be $\delta_p \cdot |A|$ and the index probing time to be $\mu_p \cdot |B|$. Let $Q = \cup_{b \in B}\, Q(b)$. Then the processing time is $(\beta_f + \alpha) \cdot |Q|$ (because for each pair in Q it takes β_f time to compute feature f, then α time to apply predicate p). Finally, we estimate $Q = \gamma_p \cdot |A \times B|$, where γ_p is a reduction factor showing how much the index-based probing "shrinks" the set of string pairs $A \times B$.

I now describe how Smurf estimates cost model parameters $\alpha, \beta_f, \gamma_p, \rho_p, \delta_p$, and μ_p. We begin by taking a small random sample of string pairs X (of size currently set at 30K) from the sample S used when learning the join condition.

Estimating α: The average selection time per string pair α is a constant which is independent of the predicate being applied. We estimate α by measuring the time to apply an arbitrary predicate (with feature precomputed) over each pair in X, and taking the average.

Estimating β_f: We estimate the time factor β_f for each feature f by measuring the time to apply f to each pair in X, and taking the average.

Estimating γ_p: We estimate the reduction factor due to index-based probing γ_p for each predicate p as follows. We begin by applying the prefix filter for p to each pair in X (i.e., for a given pair (a, b), check if the prefix of a and b share at least one token) to obtain a set of string pairs Y (that satisfy the filter). We then estimate γ_p as $\mid Y \mid / \mid X \mid$.

Estimating ρ_p: We now discuss how to estimate the selectivity of each predicate p, ρ_p. We do not precompute ρ_p for each predicate p, as we do not assume the predicates are independent. For example, if we apply a sequence of two filters containing predicates p_1 and p_2, respectively, over an input set of pairs C, we cannot compute the output size of applying the sequence of filters as $\rho_{p_1} * \rho_{p_2} * \mid C \mid$ since p_1 and p_2 may not be independent.

To address this, for each predicate p_i we compute the coverage of p_i over sample X, $cov(p_i, X)$, which is the set of pairs in X that p_i would satisfy. Then we can estimate the selectivity of p_i, ρ_p, to be $\mid cov(p_i, X) \mid / \mid X \mid$. And we can compute the selectivity of applying a sequence of predicates p_1, p_2, to be $\mid cov(p_1, X) \cap cov(p_2, X) \mid / \mid X \mid$. Hence we only keep track of the coverage of each predicate and estimate the selectivities of predicates on the fly. To estimate selectivities efficiently, Smurf maintains the coverages of predicates in the form of bitmaps.

Estimating δ_p: To estimate the index building time, we need to estimate δ_p which is the average time spent per string in A when building the index. To do so, we take a small random sample of strings Y from A (where $\mid Y \mid$ is $min\{0.1 * \mid A \mid, 1K\}$), then measure the time it takes to index each string $a \in Y$ (i.e., the time taken to insert each token in the prefix of a into the index), and take the average.

Estimating μ_p: To estimate the index probing time, we need to estimate μ_p which is the average time spent per string in B when probing the index. To do so, we take a small random sample of strings Z from B (where $\mid Z \mid$ is $min\{0.1 * \mid B \mid, 1K\}$), then measure the time it takes to probe each string $b \in Z$ (i.e., the time taken to probe each token in the prefix of b) in the index built over strings in sample Y (used for estimating δ_p above), and take the average.

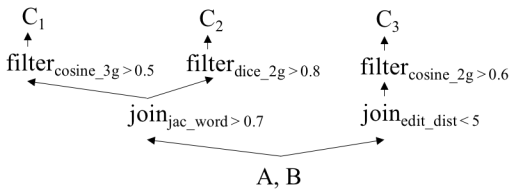

Figure 4.8: An example of a final execution plan.

4.6.5 Plan Execution

I now describe how a plan G is executed over two sets of input strings A and B. First, we perform pre-processing, e.g., traversing G to collect information, tokenizing the strings in A and B, materializing the tokens to disk, etc.

Next, we traverse G depth first, and execute its nodes in that order. For the plan in Figure 4.8, for example, starting with the inputs, we first execute node $join_{jac\_word>0.7}$, then node $filter_{cosine\_3g>0.5}$, then $filter_{dice\_2g>0.8}$, and so on. For a node with more than one children, we cache its output until all child nodes (i.e., those needing this output) have been executed. Executing nodes of types $filter$, $feature$ and $select$ is straightforward, as we just apply such operations to each string pair. Executing a $join$ node over two sets of strings A and B is more involved. Here we use the index-based filtering approaches discussed in Section 4.2 to avoid enumerating $A \times B$. Recall that prior work has developed many efficient filtering techniques [98, 94, 99]. The current Smurf implements the ppjoin filtering algorithm [98] for set-based similarity measures (e.g., Jaccard, overlap, Dice, etc.) and edjoin algorithm [99] for edit distance.

Caching: In certain nodes of the plan G, which compute a set-based similarity feature, the input strings need to be tokenized into sets of tokens. We can tokenize the strings when executing each node in G containing such a feature. But this is expensive as we repeatedly tokenize the strings. To address this, we traverse through G to identify a set of tokenization types W needed to execute G (e.g., 3-gram, whitespace, etc.), then for each type $w \in W$ we tokenize the strings in A and B and materialize the tokens to disk.

Then each node in G loads the appropriate tokens from disk, if needed. A better solution however is to cache the tokens in memory if possible, thereby avoiding I/O when executing each

node. To do so, we proceed in this order: (1) If the tokenized output needed to execute G fits in the available memory, we load them into the memory. (2) If the tokenized output needed to execute a subtree in G fits in the available memory, we load them into the memory before executing the subtree. (3) Otherwise we do not cache any tokenized output.

4.7 Efficient Blocking and Matching

Recall that to perform SSJ over two sets of strings A and B, we first interact with the user to learn a random forest F, then execute F over A and B. In particular, I break the execution of F into two steps: blocking and matching. I now describe how to perform these two steps efficiently (using the solution to efficiently execute a set of DTs described in the previous section). Algorithm 4.4 describes the pseudo code for executing a random forest over two sets of strings.

4.7.1 The Blocking Step

Suppose the learned random forest F has n trees. For ease of exposition, suppose we need at least $\lceil n/2 \rceil$ trees in F to match, in order for F to match (our solution generalizes straightforwardly to the case where we need αn trees to match, where α is pre-specified). Then we can easily prove that

Proposition 4. *If the blocking step executes at least* $(\lfloor n/2 \rfloor + 1)$ *trees, then any string pair not output by this step (i.e., not output by any of these trees) cannot be a match.*

To minimize the run time of blocking, we will execute exactly $(\lfloor n/2 \rfloor + 1)$ trees in this step (Section 4.8.1 shows that executing more trees actually incurs longer total join execution time.)

Specifically, let T be the set of all trees in random forest F. We will select a subset $T' \subseteq T$ of $(\lfloor n/2 \rfloor + 1)$ trees for blocking, such that the time taken to apply the trees in T' to A and B to produce a set of pairs J, plus the time taken to apply the remaining trees in $T \setminus T'$ to set J is minimized. Let these two times be $time(T')$ and $time(T \setminus T')$, respectively. We estimate $time(T)$ by applying the procedure in Section 4.6.3 to generate a good execution plan P for the set of trees

Algorithm 4.4 Executing a Random Forest

Input: Sets of strings *A* and *B*, random forest F of n trees
Output: Set of predicted matches C

$T \Leftarrow$ Set of decision trees in F
for tree $t \in T$ **do**
    $R_t \Leftarrow$ Extract matching rules from t
end for
$T' \Leftarrow$ Select a set of $(\lfloor n/2 \rfloor + 1)$ trees from T
$R \Leftarrow \bigcup_{t \in T'} R_t$
$G \Leftarrow$ Generate a plan to execute R
$D \Leftarrow$ Execute G over A and B
Initialize $C \Leftarrow \{c \in D \mid c$ already has at least $\lceil n/2 \rceil$ votes$\}$
$D \Leftarrow D \setminus C$
$\bar{S} \Leftarrow$ Order the trees in $T \setminus T'$
$ubound \Leftarrow \mid \bar{S} \mid$
for tree $t \in \bar{S}$ **do**
    if D is empty **then** return C
    $G_t \Leftarrow$ Generate a plan to execute R_t
    $E_t \Leftarrow$ Execute G_t over D
    $ubound \Leftarrow ubound$ - 1
    for pair $c \in E_t$ **do** $v_c \Leftarrow v_c$ + 1 **end for**
    for pair $c \in D$ **do**
        if $v_c \geq \lceil n/2 \rceil$ **then**
            $D \Leftarrow D \setminus \{c\}$
            $C \Leftarrow C \cup \{c\}$
        else if $v_c + ubound < \lceil n/2 \rceil$ **then**
            $D \Leftarrow D \setminus \{c\}$
        end if
    end for
end for
return C

T, then take the estimated runtime of P (using the cost estimation procedure in Section 4.6.4) to be $time(T)$. We estimate $time(T \setminus T')$, the matching time, as described in the next subsection.

The problem is that there are too many possible subsets of trees of size $(\lfloor n/2 \rfloor + 1)$ ($\binom{n}{\lfloor n/2 \rfloor + 1}$ such subsets). So we cannot enumerate and estimate $time(T') + time(T \setminus T')$ for all of them, then select the one with the lowest total time. As a result, we select T' using greedy search. First, we assign T' to be the set of all trees T. Then in each iteration we remove the tree t from T' that results in the largest reduction of $time(T') + time(T \setminus T')$, until we have removed $(\lceil n/2 \rceil - 1)$ trees. Let T'_* be the remaining set of trees. We perform blocking using T'_*, i.e., we generate an efficient execution plan for T'_* (see Section 4.6) then execute it on A and B.

4.7.2 The Matching Step

Suppose that executing T'_* trees in the blocking step produces a set of pairs J. I now consider how to execute the remaining trees on J. We begin by noting that the optimization procedure in Section 4.6, which finds a good plan to execute a set of trees over *two sets of strings A and B*, can easily be adapted to find a good plan to execute a set of trees over *a set of string pairs J*.

Now let U be the set of the remaining trees to be executed on set J. Similar to how we execute trees in the blocking step, here we can simply use the above optimization procedure to generate a single plan P that executes all the trees in U in a *combined* fashion (i.e., reusing computation). A better solution however is to apply the trees *sequentially* to avoid applying all trees in U to all pairs in J.

Example 4.7.1. *Consider a forest F of 10 trees, where at least 5 trees must match in order for F to match. Then the blocking step executes 6 trees to produce a set of pairs J. Consider a pair $p_1 \in J$ matched by 4 trees in blocking. Then we can declare p_1 a match as soon as one of the remaining 4 trees matches p_1. Consider a pair $p_2 \in J$ matched by just one tree in blocking. Then we can declare p_2 a non-match as soon as one of the remaining 4 trees predicts it a non-match.*

Thus we will order and execute the trees in U sequentially. In particular, we want to find the tree sequence that minimizes the total execution time. This problem is NP-hard. As a result, we employ a greedy approach.

Specifically, let M be the output of applying a tree sequence $\langle t_1, \ldots, t_i \rangle$ to set J, i.e., (a) M contains all pairs in J for which we still cannot make a match/non-match decision, and (b) $J \setminus M$ contains all pairs in J for which we have already made a match/non-match decision (after executing sequence $\langle t_1, \ldots, t_i \rangle$, see Example 4.7.1). Then we refer to $|J \setminus M|/|J|$ as the pruning rate of the sequence $\langle t_1, \ldots, t_i \rangle$ and denote this rate as $d(\langle t_1, \ldots, t_i \rangle)$. Let $w(t_i)$ be the average runtime of tree t_i on a string pair.

Intuitively, we want to be able to make match/non-match decisions as soon as possible, for as many pairs as possible. So we want to start with the trees with the highest pruning rates. But we need to balance this against the runtimes of those trees. Thus, we select the tree sequence as follows. First, we select a tree t_i that maximizes $d(\langle t_i \rangle)/w(t_i)$. Then we select another tree t_j that maximizes $d(\langle t_i, t_j \rangle)/w(t_j)$, etc., until we have selected all trees in U. This forms the sequence $\bar{U}$ to be executed in the matching step.

Finally, recall from Section 4.7.1 that when considering whether to select a subset of trees T' for blocking, we need to estimate the total runtime of executing the remaining trees, $U = T \setminus T'$, in the matching step. To do this, we first find a good tree sequence $\bar{U}$, as described above. Let $\bar{U} = \langle t_1, \ldots, t_k \rangle$. Then we estimate the runtime of $\bar{U}$ on set J as $|J| \cdot z$, where $z = w(t_1) + (1 - d(\langle t_1 \rangle)) \cdot w(t_2) + (1 - d(\langle t_1, t_2 \rangle)) \cdot w(t_3) + \cdots + (1 - d(\langle t_1, \ldots, t_{k-1} \rangle)) \cdot w(t_k)$.

Recall from Section 4.5.1 that we perform blocking on two sets A and B to obtain a set I of matches and a set J of candidates. Then we perform matching on J to obtain a set K of matches. The following theorem shows the correctness of the blocking and matching algorithms, as described above:

Theorem 1. *Let $C*$ be the set of all pairs in $A \times B$ predicted match by a random forest F. Then $I \cup K = C*$.*

4.8 Empirical Evaluation

I evaluate Smurf using the five datasets described in the first four columns of Table 4.2. Addresses describes street addresses extracted from Yelp and Yellow Pages. Researchers describes the names of researchers at a university. Citations and Products are derived from the datasets used

Dataset	A	B	# Matches	Smurf				Best single predicate				Increase in F1 (in %)
				P	R	F1	# Pairs	P	R	F1	Predicate	
Addresses	24,650	29,531	9,850	96.97	99.95	**98.43**	400	61.33	100	**76.03**	jaccard_numeric(a, b) ≥ 0.84	22.4 (29.46)
Researchers	8,342	43,549	4,556	96.31	97.58	**96.94**	374	99.91	75.46	**85.98**	jaccard_word(a, b) ≥ 0.76	10.96 (12.75)
Citations	2,616	64,263	5,347	87.41	90.99	**89.16**	400	86.28	89.83	**88.01**	cosine_qg3(a, b) ≥ 0.59	1.15 (1.31)
Names	10,341	15,396	5,132	90.7	99.82	**95.05**	400	89.96	65.82	**76.02**	jaccard_qg2(a, b) ≥ 0.46	19.03 (25.03)
Products	2,554	22,074	1,154	66.16	64.04	**65.08**	400	70.26	44.63	**54.58**	overlap_coeff_qg2(a, b) ≥ 0.93	10.5 (19.24)

Table 4.2: Accuracy of Smurf vs. the best single predicate on five datasets.

in [22], and Names describes full names from the US Census Bureau [4]. The column "# Matches" lists the number of gold matches in each dataset.

Best Single-Predicate Join Conditions: For each dataset, we find the best single-predicate join condition by an exhaustive search. Specifically, we consider 25 features, each created by pairing one of six common tokenization method (e.g., 2-gram, 3-gram, word, numeric, etc.) with one of five common similarity measures (e.g., Jaccard, edit distance, cosine, etc.). For each feature f, we consider all predicates of the form $f \geq t$, where t ranges from 0.1 to 1 in increments of 0.01 (edit distance was converted into a similarity measure for this purpose). We then find the predicate with the highest F_1 accuracy (shown in Column "Predicate" of Table 4.2).

Smurf: Smurf was implemented in Cython. We consider random forests with ten trees (the default in many learning packages, e.g., scikit-learn), but experiment with a varying number of trees below, and learn the random forest on each dataset by simulating a user who labels the string pairs. All experiments were run on a machine with Ubuntu 14.04.4 with two Intel Xeon E5-2630 CPUs (8 cores, 2.4GHz) and 32GB memory.

4.8.1 Accuracy and Runtime

Accuracy: In Table 4.2, the columns under "Smurf" and "Best single predicate" show the accuracy in P, R, F_1 of Smurf and the best single predicate (BSP), respectively. The results show that Smurf significantly outperforms BSP, by 10.5-22.4% absolute increase in F_1 in four cases, and by 1.15% in one case (see the last column of the table). Column "# Pairs" (under "Smurf") shows the number of pairs 374-400 that the user has to label to learn the join condition using active learning

(for Researchers, active learning stopped earlier, when all unlabeled pairs have entropy 0). We found that real-world users typically need 1-3 seconds to label a pair of strings in our datasets, or 15-20 minutes to label 400 pairs.

Extensive examination reveals that the accuracy improvement was indeed due to Smurf's ability to use more than one predicate. For example, in Addresses, the best single predicate (shown in Table 4.2) uses Jaccard score over numeric tokens extracted from the address string (i.e., house number, zip code). But this means it also matches address pairs with the same house numbers and zip codes, *but different streets* (e.g., 100 E Main St, Austin, TX 83703 and 100 S Doty St, Austin, TX 83703). In contrast, Smurf learns rules such as $cosine\_num(a, b) > 0.93 \wedge dice\_qg3(a, b) > 0.83 \rightarrow match$.

Here, $dice\_qg3(a, b) > 0.83$ computes the Dice score over 3-gram tokens of the addresses, and $cosine\_num(a, b) > 0.93$ computes the cosine score over the numeric tokens of the address. Together, they state that two addresses match if their strings are highly similar (3-gram token-wise) and their numbers are also highly similar. This dramatically increases the precision from 61.33% to 96.97%.

In Names, about 40% of the names are short names (i.e., less than 10 characters). The best single predicate is $jaccard\_qg2(a, b) \geq 0.46$, which fails to match most short names (by setting a "too high" threshold). In contrast, Smurf uses two rules:

$$r_1 : jaccard\_qg2(a, b) > 0.47 \wedge len(a) > 9.5 \rightarrow match$$
$$r_2 : jaccard\_qg2(a, b) > 0.28 \wedge len(a) \leq 9.5 \rightarrow match$$

Rule r_1 states that if the first name is long ($len(a) > 9.5$), then use a high threshold (0.47) for $jaccard\_qg2(a, b)$. Rule r_2 states that if the first name is short, then lower that threshold to 0.28. This allows Smurf to correctly match many short names, increasing recall from 65.82% to 99.82%.

In Products, the best single predicate (very conservatively) declares two product titles matched if their 2-gram tokens are very similar (using $overlap\_coeff\_qg2(a, b) \geq 0.93$). In contrast, Smurf states that if two titles are only somewhat similar in their alphabetic tokens (using $jaccard\_alph(a, b$

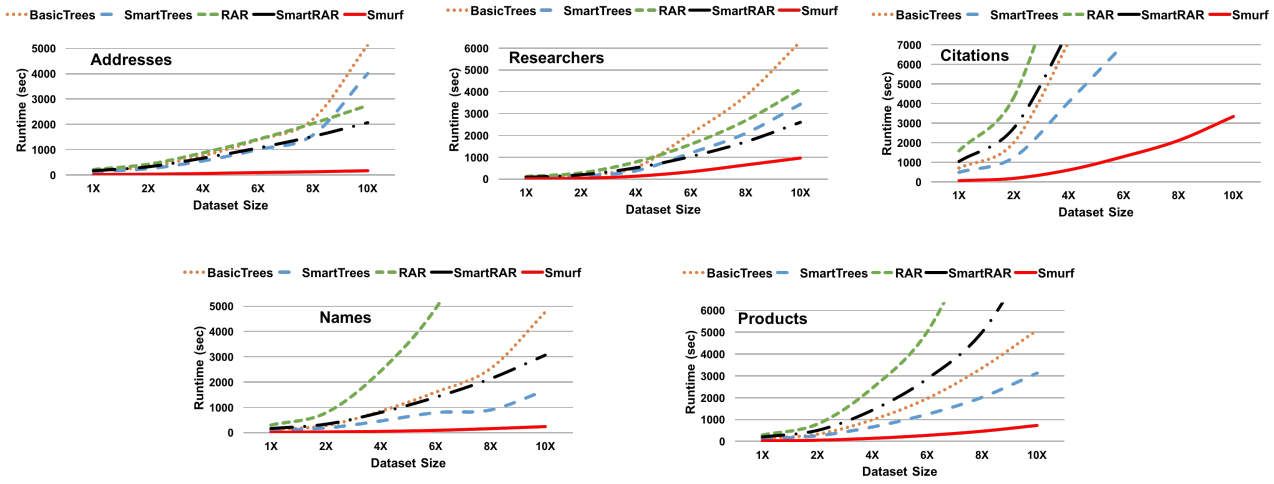

Figure 4.9: Runtimes of Smurf versus baselines that use existing solutions on rule execution.

$) > 0.42$), but the numbers extracted from them (which are typically model numbers) are very similar (using predicate $overlap\_coeff\_num(a, b) > 0.92$), then they also match. This increases recall from 44.63% to 64.04%.

Runtime: As discussed in Section 4.5.1, no published work has optimized the execution of *a set of matching rules*. But two recent works, Falcon and RAR [23, 57], have optimized the execution of *a single rule*. We use these works to build four baselines for comparison. **BasicTrees** does not do blocking. It executes all trees in the random forest on A and B. To do so, it extracts the matching rules of the trees, executes all of them, then merges their outputs. To execute a rule r, it creates and executes an optimized plan for r as described in Section 4.6.1. As such, BasicTrees is equivalent to Falcon variation that uses ApplyGreedy [23]. (I also experimented with the ApplyAll variation [23] but it was outperformed by ApplyGreedy in our settings and hence is not discussed further.)

RAR is similar to BasicTrees, but when executing an individual rule it uses the holistic prefix index solution of [57]. **SmartTrees** and **SmartRAR** are versions of BasicTrees and RAR that use blocking. They first execute a set of $(\lfloor n/2 \rfloor + 1)$ trees (the same set of trees used by Smurf for blocking) to A and B to obtain a set of pairs J, then apply the remaining trees to J. They differ from Smurf only in that they do not execute the rules of the trees in an optimized fashion (i.e., no reuse, see Section 4.5.1).

Dataset	Smurf	$\text{Smurf}_{\text{rand}}$	$\text{Smurf}_{\text{sel}}$	$\text{Smurf}_{\text{time}}$
Addresses	158	771	158	164
Researchers	956	1123	1146	1050
Citations	3333	6167	6899	5391
Names	236	785	490	319
Products	727	733	922	931

Table 4.3: Selecting a subset of trees for blocking.

Figure 4.9 compares the runtimes as we increase the dataset size. Here a value 4x on the x-axis means that we replicate the original dataset 4 times, by using random perturbations (e.g., inserting/deleting characters) of the original strings.

The results show that Smurf significantly outperforms the four baselines, and this gap increases as the dataset size increases, e.g., at dataset size of 10x, Smurf performs 6-32 times better than BasicTrees (i.e., Falcon), 3-25 times better than SmartTrees, 4-54 times better than RAR, and 2-13 times better than SmartRAR. It is clear that executing the trees in a joint fashion (to reuse computations), as Smurf does, is absolutely critical for scaling.

4.8.2 Performance of the Components

Blocking: Blocking drastically reduces the number of pairs to be considered, from 56M-727M for $A \times B$ to 4,887-25,763 pairs. Using blocking, SmartTrees significantly outperforms BasicTrees, and similarly SmartRAR outperforms RAR (see Figure 4.9).

Table 4.3 examines how well Smurf selects a subset of trees for blocking. It shows the runtime (in secs) of Smurf vs three Smurf variations. Smurf_{rand} uses a random subset of $(\lfloor n/2 \rfloor + 1)$ trees for blocking. Smurf_{sel} selects the first $(\lfloor n/2 \rfloor + 1)$ trees in decreasing order of their pruning power. Smurf_{time} selects the first $(\lfloor n/2 \rfloor + 1)$ trees in increasing order of their average execution time (as we want to reduce the blocking time). The results show that Smurf always outperforms the three variants, often by a large margin (e.g., by 38-51% for Citations), suggesting that Smurf selects good subsets of trees for blocking.

Dataset	BT	ST	O	O - O_1	O – O_2	O – O_3	O – O_4
Addresses	5145	4007	158	956	178	169	185
Researchers	6302	3428	956	1164	1015	1211	1020
Citations	43890	23526	3333	6270	3544	4076	3553
Names	4763	1671	236	511	248	381	556
Products	5103	3129	727	808	791	756	729

Table 4.4: Runtimes of the components (in seconds).

Matching: I found that executing the trees in the matching step in a sequential, instead of combined, fashion reduces runtime by 8-20%, suggesting that sequential execution is effective. To examine how well Smurf orders the trees, I compare it with three variations that order the trees (a) randomly, (b) in decreasing order of their pruning power, and (c) in increasing order of average execution time. For Addresses and Citations, Smurf is the best (9-15% faster compared to the second best). For Researchers Smurf is the second best (11% slower than the best). This suggests that Smurf selects a reasonable sequence of trees.

Optimization: Table 4.4 examines the effect of executing a set of trees in a joint optimized fashion, on the 10x versions of the datasets. Columns "BT" and "ST" show that BasicTrees and SmartTrees incur significant runtimes, and that optimization (i.e., Smurf) drastically reduces these times to 158-3,333 secs (see Column "O"), a major reduction of 72-96%.

The next four columns show the runtimes when I turn off each type of optimization: join reuse (O_1), inter-path filter reuse (O_2), ordering filters (O_3), and intra-path filter reuse (O_4). Comparison with Column "O" shows that all four optimization types are useful, and that the effects of some are quite significant (e.g., O_1 on all the data sets, O_3 on Researchers and Citations).

4.8.3 Sensitivity Analysis

I now examine the main factors affecting the performance of Smurf on the three data sets: Addresses, Researchers, and Citations. The results are similar for the remaining two data sets.

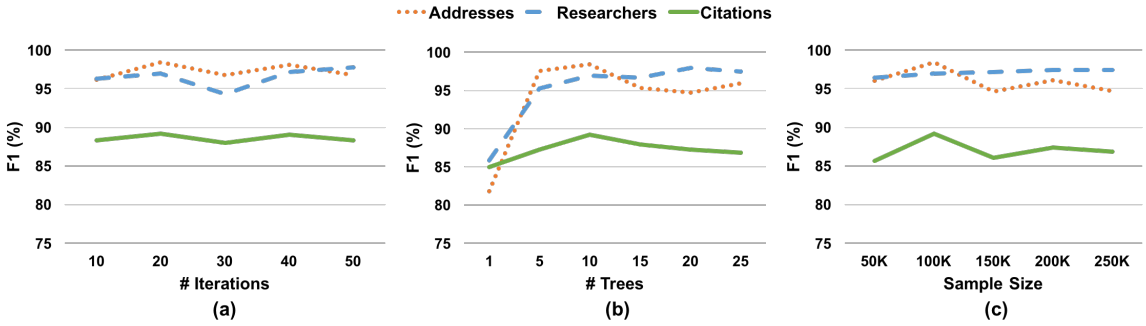

Figure 4.10: Effect of number of iterations, number of trees and sample size on F_1.

Number of Iterations: I now examine how varying the maximum number of iterations during active learning affects the accuracy of Smurf. Figure 4.10.a shows that as we increase this number from 10 to 50, F_1 fluctuates in a small range for all three data sets. This suggests that capping the number of iterations at 20, as Smurf does, is a reasonable solution to avoid a large number of pairs being labeled and yet achieve good accuracy.

Number of Trees: Figure 4.10.b shows that as we increase the number of trees in the random forest from 1 to 25, F_1 increases significantly from 1 to 5 trees, and then fluctuates in a small range for all three datasets. This suggests that using a random forest outperforms using a single decision tree, and that using a forest of ten trees (as Smurf does) is a reasonable default choice.

Sample Size: Figure 4.10.c shows that as we increase the sample size (used for learning the random forest) from 50K to 250K pairs, F_1 increases slightly then fluctuates in a small range. This suggests that a sample size of 100K (as used in Smurf) is a good default size.

Number of Trees Used for Blocking: Recall that blocking must use at least $(\lfloor n/2 \rfloor + 1)$ trees. So in our context it can use 6-10 trees (as $n = 10$). As we increase the number of trees from 6 to 10, Smurf's runtime increases significantly, e.g., by 398% for Addresses, 28% for Researchers, and 121% for Citations. This suggests that using the minimally required number of trees for blocking (as Smurf does) is the best strategy.

Maximum Depth of a Decision Tree: We found that as we increase the maximum depth of decision trees from 1 to 10, F_1 accuracy remains stable on Citations. But on Addresses and Researchers it increases significantly until the depth of 3 (from 77% to 90% for Addresses, and 73% to 95% for Researchers), and then fluctuates in a small range.

4.9 Related Work

String Similarity Joins: SSJs have been widely studied [98, 12, 49, 60] (see [104] for a survey). To avoid examining all pairs of strings, prior works use inverted indexes [83], prefix filter [17], size filter [9, 12], position filter and suffix filter [98], among others. Work has examined SSJs within a database [44, 17, 10] and developed scalable parallel solutions (e.g., using MapReduce [90, 65, 24, 27]). Recent work has also examined top-k SSJs [100, 105]. Current SSJ work however has only examined single-predicate join conditions (e.g., [98, 90]). In contrast, Smurf considers more powerful multi-predicate join conditions in form of random forests.

Learning Join Conditions: Few works address learning join conditions for SSJ. The work [14] learns a single-predicate join condition using active learning. In contrast, Smurf learns a random forest join condition. The works [23, 42] use active learning to learn a random-forest blocker for entity matching. Smurf adapts their solution to SSJ contexts. Recent works [56, 85] have studied how to learn other ML models such as generalized linear models over a join without having to materialize the join output.

Scaling up Random Forests: Most works on scaling RFs focus on efficiently *learning* RFs over large datasets [69, 87, 103]. In contrast, Smurf considers efficiently *executing* random forests. The work [33] develops pruning techniques for reducing the prediction time of ensemble models, but assumes a set of feature vectors as input.

Execution of Matching Rules and Multi-Query Optimization: The works [23, 57] have examined how to efficiently execute *a single matching rule* (for entity matching). In contrast, Smurf examines how to efficiently execute *a set of rules*, by reusing computation. This combined execution of rules is reminiscent of multi-query optimization in RDBMSs, which optimizes the execution

of a set of queries [86]. Similar to Smurf, prior works on multi-query optimization also represent each query as a DAG and combine the DAGs into a single DAG by exploiting the common sub-expressions in the queries [86, 79]. But the two contexts are sufficiently different, so that the ideas underlying multi-query optimization cannot be straightforwardly adapted to our SSJ contexts.

Additional Related Work: The problem of selecting a subset of trees for blocking is reminiscent of the problem of selecting a optimal set of SSJ filters (such as size filter, prefix filter) when executing a single-predicate join condition [88]. However, the SSJ filters considered in [88] form a conjunction, whereas in our blocking step the trees form a disjunction (i.e., we need to output the string pairs predicted as a match by at least one tree). Finally, the problem of finding an optimal tree sequence for matching is similar to the problem of ordering pipelined filters [11]. However, our problem is more complex, a special case of which is the problem in [11].

4.10 Conclusion

Current SSJ work is limited in that it has considered only single-predicate join conditions. I have shown that using multiple predicates for join conditions can significantly improve SSJ accuracy. I have described Smurf, a solution that uses active learning to learn a join condition that is a random forest containing multiple predicates, then executes the random forest on the input sets to match strings. Our key technical contribution is a solution to efficiently execute such random forests over two sets of strings.

Going forward, I plan to explore better and more optimization/execution techniques for random forests, and to consider the machine cluster setting. I also plan to explore how the techniques developed here can be applied to other settings, such as entity matching (e.g., over multi-attribute tuples). Finally, it would be interesting to consider join conditions that involve other types of learning, e.g., SVM, deep learning, among others.

Chapter 5

Tool Development and Deployment

In this chapter, I describe the development and deployment experience of our solutions.

5.1 Tool for Expanding Regexes

I have implemented the solution for finding synonyms to add to disjunctions in a regex (described in Chapter 2) as a Web-based app. The app was implemented in Java.

A user will provide an initial regex and a dataset as input to the app. The app will then process and return a ranked list of top-k terms to add to the disjunction along with evidence in the dataset. The user marks the terms as relevant or not, which is then used by the app to re-rank the remaining terms and shows the next list of top-k terms. Figure 5.1 shows a screenshot of the app's homepage and Figure 5.2 shows a screenshot of the ranked list of synonyms shown by the app.

Deployment: The app has been used by analysts at WalmartLabs since June 2014. The time spent by analysts reduced from hours to few minutes. Further it also resulted in a new direction of work. Specifically, we extended the current approach to generate new product classification rules.

The existing product classification system at WalmartLabs consisted of a set of rules for each product type, where each rule is of the form $r \rightarrow t$ which assigns the product type t to any product whose title matches the regular expression r. One of the problems with the existing system was that the coverage of the rules was low which resulted in many products not being classified. To address this, we extended our solution to expand the existing classification rules to generate new rules. We evaluated our approach on a set of training data that consists of roughly 885K labeled products, covering 3707 types. Our approach generated 63K new rules, which was added to the

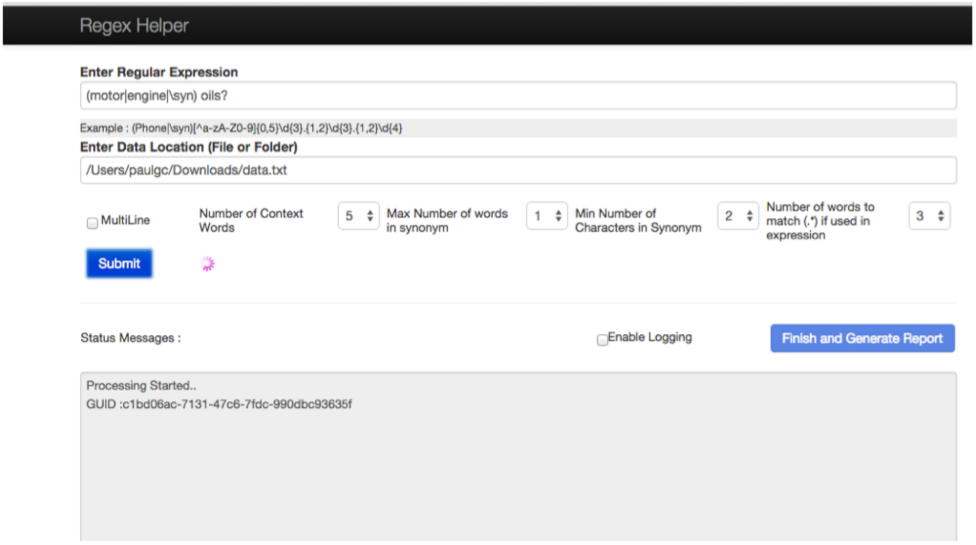

Figure 5.1: A screenshot of the app's homepage.

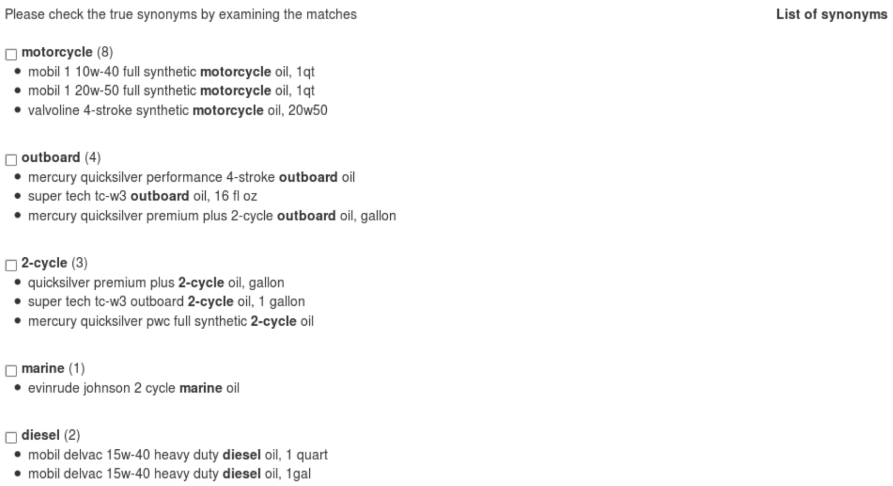

Figure 5.2: A screenshot of the ranked list shown by the app.

existing system. The new system was operational since June 2014, and the addition of these rules resulted in an 18% reduction in the number of items that the system declines to classify.

5.2 Deploying Falcon as a Cloud Service

In collaboration with Yash Govind, I have deployed the Falcon system (described in Chapter 3) as a cloud service and have open sourced the code. Specifically, I have deployed Falcon as CloudMatcher, a cloud/crowd service for EM.

CloudMatcher is a fast, easy-to-use, scalable, and highly available service on the Web. Specifically, to use this service, a user simply needs to go to CloudMatcher's Web site, uploads two tables to be matched, performs some basic pre-processing, then pushes a button. CloudMatcher will perform EM end-to-end. To do so, it will use crowd workers on Amazon's Mechanical Turk (or some other crowdsourcing platform) to label tuple pairs (as match / non-match). The user just has to pay for the labeling. Alternatively, instead of using crowdsourcing, the user can just label these tuple pairs. At the end, CloudMatcher will return the desired matches. In the backend, CloudMatcher performs EM using a machine cluster that we will maintain.

As described, when using CloudMatcher, the user does not need to install or learn how to use any complicated system (using CloudMatcher should be very straightforward). The user does not have to know EM (e.g., knowing string similarity measures). He or she will only perform simple actions such as labeling a tuple pair as match / non-match. Alternatively, if the user is not even willing to label the tuple pairs, then he or she can pay to "outsource" that work to a crowd of workers (assuming that the data is not sensitive and that crowd workers can be quickly trained to label tuple pairs). Finally, the system can scale to tables of millions of tuples and can automatically add more machine resources as necessary. Figure 5.3 shows a screenshot of CloudMatcher's homepage.

Deployment: CloudMatcher has been developed for over 1.5 years, in a combination of Python and Java, at cloudmatcher.io. It is not yet available to the general public (we still need to work out issues such as how to let a "public" user pay easily and how to securely store his/her data).

CloudMatcher however has been applied to many datasets at UW-Madison, Johnson Controls Inc. (JCI), and WalmartLabs, and has been opened to several other users, including biomedical

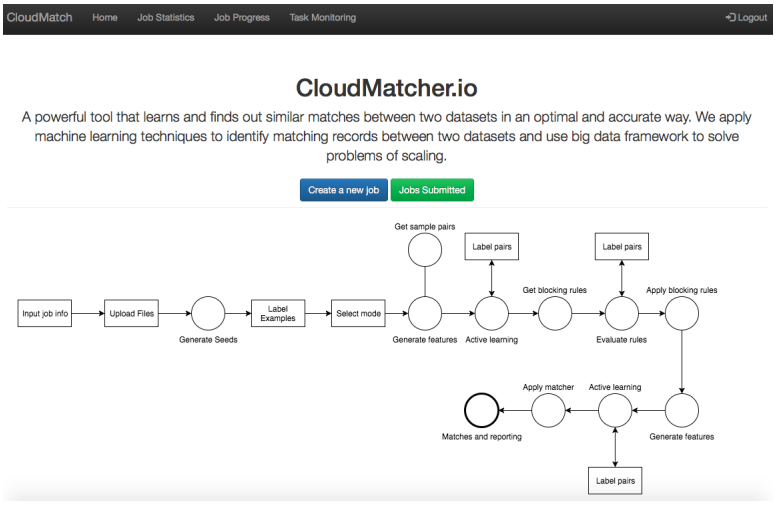

Figure 5.3: A screenshot of CloudMatcher's homepage.

researchers in a joint project between Marshfield Clinic and UW-Madison, and users at a non-profit organization (NPO) tracking Wisconsin politics [43].

5.3 Tools for Matching Strings

I have developed two Python packages: *py_stringmatching* and *py_stringsimjoin*, which provide string matching capabilities. Specifically, *py_stringmatching* consists of a variety of string similarity measures, and *py_stringsimjoin* consists of commands to perform string similarity joins. In what follows I describe these two packages.

5.3.1 Tool for Computing String Similarity Measures

Many applications such as EM require string matching capabilities, and yet today there are very few packages in the Python data eco-system that provide such capabilities. To address this, I have implemented the package *py_stringmatching*.

py_stringmatching is a Python package that consists of a variety of string tokenizers (e.g., whitespace tokenizer, qgram tokenizer) and string similarity measures (e.g., edit distance, Jaccard measure, TF/IDF) [1]. The goal is to build a comprehensive and scalable set of string tokenizers and similarity measures for the Python data management eco-system.

5.3.1.1 Limitations of Current Tools

In contrast to the wealth of research work, there has been relatively few implemented tokenizing and string similarity measure packages. As of June 2016, we counted 2 non-Python packages and 7 Python packages for string matching. The main limitations of these packages are:

- *Language*: The two non-Python packages are in Java, and thus are hard to use in Python. To use them, users would need to install and run JVM (Java Virtual Machine), a cumbersome process.

- *Coverage*: The packages do not provide a comprehensive coverage of the most common string similarity measures, and it is not clear if they have plans to provide comprehensive coverage. For example, 4 out of 7 Python packages cover only edit distance variations, while the remaining 3 packages do not implement certain common similarity measures (e.g., TF/IDF, Jaccard).

- *Runtime Performance*: The performance of certain measures is unsatisfactory. For example, our experiments show that the runtime of edit distance can vary by as much as 180 times across the packages. In particular, it appears that it is difficult to obtain satisfactory performance for edit distance (say) using just Python (C and Cython appear to provide a far better performance).

- *Installation*: Some of the packages are cumbersome to use. For example, to use Abydos, the user needs to install the complete package, of which the module on similarity measure constitutes just a small part.

- *Licensing*: Some of the packages use restrictive copyright licenses. For example, python-Levenshtein (which has the best runtime performance for edit distance in our experiments) uses GPL. Roughly speaking, any code using this code would also become GPL open-source code. In contrast, we want to allow any code to use *py_stringmatching* with acknowledgment. For these reasons, we plan to use BSD 3-Clause license (which is also used by well-known packages such as pandas and sklearn).

- *Extensibility*: It is not clear if the existing packages are designed for extension, and if so, how to extend them. In contrast, we envision that py stringmatching will have to be extended to handle more tokenizers and similarity measures, and to be adapted to various domains. So we plan to design *py_stringmatching* to be extensible from the scratch. For example, we design tokenizers and string similarity measures as classes as opposed to functions.

5.3.1.2 Goals

As a result, we want to develop a new Python package for string tokenizers and similarity measures. Our goals are as follows:

- Develop a comprehensive package. It should at least cover all common tokenizers and similarity measures, and then expand over time. Ideally it should subsume current packages.
- The package should be in Python with minimal dependencies, so that many other Python packages can use it easily. For example, if some of our code is in Java, then we may have to require the user to install and start the Java Virtual Machine (JVM) before using the package, a cumbersome process that may not work on certain machines.
- The tokenizers and similarity measures should be as fast as possible.
- The licensing should allow liberal usage (with acknowledgment) yet shield us from legal responsibilities.
- The package should be designed such that it can be easily extended.

5.3.1.3 Overview of py_stringmatching

Currently we have implemented five tokenizer classes (organized into a class taxonomy): alphabetic-, alphanumeric-, delimiter-, qgram-, and whitespace tokenizers. We have also implemented 23 similarity measures, organized into five groups:

- *Sequence-based Measures:* affine gap, bag distance, editex, Hamming distance, Jaro, Jaro Winkler, Levenshtein, Needleman Wunsch, partial ratio, partial token sort, ratio, Smith Waterman, token sort.
- *Set-based Measures:* cosine, Dice, Jaccard, overlap coefficient, Tversky Index.
- *Bag-based Measures:* TF/IDF.
- *Phonetic-based Measures:* soundex.
- *Hybrid Measures:* Generalized Jaccard, Monge Elkan, Soft TF/IDF.

5.3.2 Tool for String Similarity Joins

String similarity join (SSJ) finds all pairs of strings that refer to the same real-world entity, between two collection of strings. For example, the string David Smith in one database may refer to the same person as David R. Smith in another database. Similarly, the strings 1210 W. Dayton St Madison WI and 1210 West Dayton Madison WI 53706 refer to the same physical address. SSJ plays a critical role in many data integration tasks, including schema matching, entity matching and information extraction.

In contrast to the wealth of research work on similarity joins, there are no packages available in Python to perform similarity joins. To address this gap, I implemented $py\_stringsimjoin$, a Python package that provides scalable implementation of string similarity joins over two tables, for commonly used similarity measures such as Jaccard, Dice, cosine, overlap, overlap coefficient and edit distance.

5.3.2.1 Overview

Given two tables A and B, this package provides commands to perform string similarity joins between two columns of these tables, such as $A.name$ and $B.name$, or $A.city$ and $B.city$. An example of such joins is to return all pairs (x, y) of tuples from the Cartesian product of Tables A and B such that,

- x is a tuple in Table A and y is a tuple in Table B.
- $Jaccard(3gram(x.name), 3gram(y.name)) > 0.7$. That is, first tokenize the value of the attribute "name" of x into a set P of 3grams, and tokenize the value of the attribute "name" of y into a set Q of 3grams. Then compute the Jaccard score between P and Q. This score must exceed 0.7. This is often called the "join condition".

Such joins are challenging because a naive implementation would consider all tuple pairs in the Cartesian product of Tables A and B, an often enormous number (for example, 10 billion pairs if each table has 100K tuples). The package provides efficient implementations of such joins, by using a filter-verification approach which includes two steps: (1) Filter step: using effective filtering algorithms to prune large numbers of dissimilar pairs and generating a set of candidate pairs; and (2) Verification step: verifying each candidate pair by computing the real similarity and outputting the final results.

Currently, $py\_stringsimjoin$ supports similarity joins using five similarity measures: cosine, Dice, edit distance, Jaccard, overlap and overlap coefficient. And, it implements 5 filtering algorithms: overlap filter, size filter, prefix filter, position filter and suffix filter. Further, it also contains profiling tools and utilities to convert columns between data types.

5.3.2.2 How-to Guide

$py\_stringsimjoin$ provides an how-to guide for the users specifying a step by step procedure to join two tables using a similarity measure. The guide states that to join two tables A and B, the user should load the tables (Step 1), profile the tables (Step 2), create a tokenizer (Step 3) and then perform the join (Step 4).

5.3.2.3 Challenges and Design Decisions

We now discuss some of the challenges encountered while implementing the package and the various design alternatives considered.

- *Handling missing values:* By "missing values" we mean cases where the values of the strings are missing (e.g., represented as None or NaN in Python). For example, consider the row "David,,38" in a CSV file. The value for the second cell of this row is missing. So when reading this file into a data frame, the corresponding cell will have the value NaN. Note that missing values are different from empty strings, which are represented as "".

  To handle missing values, we can first define multiple policies. One policy would be to always throw an error if encountering a missing value. Another policy is to quietly return a missing value. Two more common policies are "optimistic" and "pessimistic". Consider computing the similarity score between a missing value u and a string v. The optimistic policy would return 1, on the ground that the missing string u can be the same as the string v in the optimistic case. Similarly, the pessimistic policy would return 0.

  In $py\_stringsimjoin$, we provide flags that the user can set to select between optimistic and pessimistic policies to handle missing values, depending on the application.

- *Interplay between filters and join:* There are different types of filtering techniques such as size filtering, prefix filtering, suffix filtering, etc., that can be employed to perform similarity join. Specifically, any combination of filters can be employed to perform a join. Note that, a different combination of filters does not affect the join output, it only alters the execution time of the join.

  Hence different combinations of filters can affect runtime of the join significantly. There are two possible design choices for implementing filters. First option would be to implement filtering algorithms within the join methods. Second option would to implement filters as separate classes and the join methods can create filter objects as needed.

  From the perspective of flexibility, the first option is bad because it does not allow the user to specify which filters to use for performing the join. For example, if the default behavior of a join method is to employ prefix and suffix filter for performing the join, but the user knows that using position filter will be efficient for the data, then the user has no way to control which filters are being used.

Whereas, with the second option, the user can create a custom join workflow by creating a position filter object, then apply the filter to the input tables and, finally applying a matcher which computes the actual join condition. For this reason, we take the second option, implementing filters as classes and making the join methods create filter objects as needed.

5.3.3 Deployment

The packages have been used in education, science and at companies. Specifically, it has been used as a teaching tool for data science classes at UW-Madison, used for matching drugs in biomedical field and used extensively at companies such as WalmartLabs, Johnson Controls, Marshfield Clinic and Recruit Institute of Technology. Recently, $py\_stringmatching$ and $py\_stringsimjoin$ have been deployed on Kaggle, a large and well-known data science and competition platform with well over 0.5M users.

Chapter 6

Conclusions

Entity matching (EM) identifies data instances that refer to the same real-world entity. This is a critical problem in many application domains such as e-commerce, biomedical, scientific data, military intelligence, etc.

Numerous EM solutions have been proposed over the past few decades [29, 19]. These solutions however suffer from two main problems. First, they are not end-to-end. That is, the EM workflow consists of multiple steps, such as cleaning, blocking, matching, sampling, labeling, debugging, etc. Current work however has focused mostly on blocking and matching, ignoring the remaining steps. Second, most current works are designed primarily for power users. They are very difficult for lay users to use. In this dissertation I develop solutions to address the above two problems. Specifically, I make the following contributions:

- First, I work together with several colleagues to develop Magellan, an end-to-end EM solution approach that focuses on all steps in the EM workflow. Within the context of Magellan, I develop a solution to help users extract missing attribute values from textual data (so that EM can be performed more accurately). As far as we can tell, no current work has considered this problem for EM.

- Second, in collaboration with Sanjib Das, I propose Falcon, an *end-to-end* crowdsourced EM solution on the cloud for lay users. Recently, in collaboration with Yash Govind, I have deployed Falcon as a cloud-based service, CloudMatcher, thereby making EM for lay users a reality. Falcon often needs to scale the execution of crowdsourced EM workflows over tables of millions of tuples. To address this, I use RDBMS-style query execution and

optimization over a Hadoop cluster. The Hadoop-based solution in Falcon to execute complex rules over the Cartesian product of the two tables 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. Falcon can efficiently perform crowdsourced EM over tables of 1.0M - 2.5M tuples at the cost of $54 - $65.5.

- Third, I propose Smurf, an end-to-end string matching solution that lay users can easily use yet obtain significantly higher matching accuracy than current string matching solutions. Smurf learns random forests (which uses a rich set of predicates) as match conditions, and efficiently executes the random forest over the two sets of strings. To execute the random forest fast, Smurf decomposes it into a blocking step and a matching step, then uses RDBMS-style plan generation and optimization to execute sets of decision trees efficiently in both steps, by reusing computation across trees.

- Finally, I implement the above solutions (mostly as open-source software) and deploy them to solve real-world problems. The open-source implementation of several solutions in particular has been deployed on Kaggle, a large and well-known data science and competition platform with well over 0.5M users.

Future Research Directions: This dissertation suggests several interesting future research directions. First, our solutions can be extended with more capabilities. For example, Falcon and Smurf can be extended with better sampling algorithms, accuracy estimation capabilities, etc. Second, a next logical research direction for EM is EM services on the cloud, which will raise novel challenges (e.g., pricing, resource allocation, scaling to large number of EM tasks, etc.). Third, we handle only the EM scenario of matching two tables or two sets of strings. Exploring other EM scenarios (e.g., linking tables to a knowledge base) will be interesting. Finally, using RDBMS-style approaches to scale execution of a random forest (as in Smurf) can be extended to scale other machine learning models.

Bibliography

[1] Falcon: Scaling up hands-off crowdsourced entity matching https://1drv.ms/b/s!Ahp0BfrE2HjfbEw_dPtrzIcO4ro. Technical Report.

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

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

[4] *Names dataset*. https://catalog.data.gov/dataset/.

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

[6] Regex Buddy http://www.regexbuddy.com/.

[7] Regex Magic http://www.regexmagic.com/.

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

[9] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, 2006.

[10] N. Augsten, A. Miraglia, T. Neumann, and A. Kemper. On-the-fly token similarity joins in relational databases. In *SIGMOD*, 2014.

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

[12] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

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

[14] L. Büch and A. Andrzejak. Approximate string matching by end-users using active learning. In *CIKM*, 2015.

[15] L. Buitinck et al. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

[16] C. Chai, G. Li, J. Li, D. Deng, and J. Feng. Cost-effective crowdsourced entity resolution: A partial-order approach. In *SIGMOD*, 2016.

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

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

[19] P. Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Publishing Company, Incorporated, 2012.

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

[21] M. Dallachiesa et al. Nadeef: A commodity data cleaning system. 2013.

[22] 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`.

[23] S. Das et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, 2017.

[24] A. Das Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014.

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

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

[27] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, 2014.

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

[29] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

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

[31] 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.

[32] 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.

[33] W. Fan et al. Pruning and dynamic scheduling of cost-sensitive ensembles. In *AAAI*, 2002.

[34] M. Fortini et al. Towards an open source toolkit for building record linkage workflows. In *SIGMOD Workshop on Information Quality in Information Systems*, 2006.

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

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

[37] C. L. Giles, K. D. Bollacker, and S. Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the Third ACM Conference on Digital Libraries*, DL '98, pages 89–98, New York, NY, USA, 1998. ACM.

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

[39] L. Gill and G. B. O. for National Statistics. *Methods for automatic record matching and linkage and their use in national statistics*. London : National Statistics, 2001. Bibliography: p. 139 - 151.

[40] S. Godbole, I. Bhattacharya, A. Gupta, and A. Verma. Building re-usable dictionary repositories for real-world text mining. In *CIKM '10*.

[41] 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.

[42] 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.

[43] Y. Govind et al. Cloudmatcher: A cloud/crowd service for entity matching. BigDAS, 2017.

[44] L. Gravano et al. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

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

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

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

[48] 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.

[49] Y. Jiang, G. Li, J. Feng, and W.-S. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[50] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, 2012.

[51] 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.

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

[53] L. Kolb, A. Thor, and E. Rahm. Parallel sorted neighborhood blocking with MapReduce. In *BTW*, 2011.

[54] 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.

[55] 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. *Proc. VLDB Endow.*, 9(12):1197–1208, Aug. 2016.

[56] A. Kumar et al. Learning generalized linear models over normalized data. In *SIGMOD*, 2015.

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

[58] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *EMNLP '08*.

[59] D. Lin. Automatic retrieval and clustering of similar words. In *COLING '98*.

[60] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *PVLDB*, 9(9):636–647, 2016.

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

[62] A. Marcus and A. Parameswaran. Crowdsourced data management: Industry and academic perspectives. *Foundations and Trends in Databases*, 6(1-2):1–161, 2015.

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

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

[65] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

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

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

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

[69] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. Planet: Massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426–1437, 2009.

[70] 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.

[71] 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.

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

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

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

[75] 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.

[76] 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.

[77] J. Rocchio. Relevance feedback in information retrieval. 1971.

[78] 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):2217–2230, 2013.

[79] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, 2000.

[80] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Inf. Process. Manage.*

[81] S. Sarawagi. Information extraction. *Found. Trends databases*, 1(3):261–377, 2008.

[82] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. KDD, 2002.

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

[84] A. D. Sarma, Y. He, and S. Chaudhuri. ClusterJoin: A similarity joins framework using Map-Reduce. In *VLDB*, 2014.

[85] M. Schleich et al. Learning linear regression models over factorized joins. In *SIGMOD*, 2016.

[86] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.

[87] J. C. Shafer et al. Sprint: A scalable parallel classifier for data mining. In *VLDB*, 1996.

[88] C. Sun, J. F. Naughton, and S. Barman. Approximate string membership checking: A multiple filter, optimization-based approach. In *ICDE*, 2012.

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

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

[91] N. Vesdapunt, K. Bellare, and N. N. Dalvi. Crowdsourcing algorithms for entity resolution. In *VLDB*, 2014.

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

[93] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. In *VLDB*, 2012.

[94] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: An adaptive framework for similarity join and search. In *SIGMOD*, 2012.

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

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

[97] W. E. Winkler and Y. Thibaudeau. An application of the fellegi-sunter model of record linkage to the 1990 u.s. decennial census. In *U.S. Decennial Census?. Technical report, US Bureau of the Census*, 1987.

[98] C. Xiao et al. Efficient similarity joins for near duplicate detection. In *WWW*, 2008.

[99] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[100] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, 2009.

[101] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *TODS*, 36(3):15:1–15:41, 2011.

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

[103] J. Ye et al. Stochastic gradient boosted distributed decision trees. In *CIKM*, 2009.

[104] 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.

[105] Z. Zhang et al. Bed-tree: An all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, 2010.