MAGELLAN: TOWARD BUILDING ENTITY MATCHING MANAGEMENT SYSTEMS

by

Pradap Venkatramanan Konda

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: 06/26/2018

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

AnHai Doan, Professor, Computer Sciences, UW-Madison (advisor)
Brent Hueth, Professor, Agricultural and Applied Economics, UW-Madison
Paraschos Koutris, Assistant Professor, Computer Sciences, UW-Madison
Ben Liblit, Professor, Computer Sciences, UW-Madison
Theodoros Rekatsinas, Assistant Professor, Computer Sciences, UW-Madison

ACKNOWLEDGMENTS

This Ph.D. 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. His honest feedback over the years has greatly improved my thinking, communication, and writing. I have learned a lot from AnHai, including how to look at the high-level picture and paying attention to details. I have fond memories of AnHai teaching me how to build systems by putting the user at the center of it.

Next, I thank my thesis committee members, Professors Ben Liblit, Brent Hueth, Paris Koutris, and Theodoros Rekatsinas, for their invaluable inputs. I would also like to thank my mentor Young-choon Park for helping me evaluate my research at Johnson Controls. I am thankful to all of my research collaborators throughout my graduate school life. An incomplete list includes Adel Ardalan, Han Li, Haojun Zhang, Jeff Ballard, Paul Suganthan, Sanjib Das, Phil Martinkus and Yash Govind. Thank you Sanjib and Phil for being my wonderful office mates. Thank you Paul for being a superb peer over the course of my Ph.D. and always being there to help. 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 support. An incomplete list includes Arun Kumar, Ce Zhang, Aaron Feng, Victor Bittorf, Vidhya Govindaraju, Bruhathi Sundarmurthy, Harshad Deshmukh, Mukilan Ashok, and Shaleen Deep.

I am grateful to my other close friends who were always there to support me. An incomplete list includes Balaji, Gerald, Mahesh, Narayanan, Praveen Kumar, Raajay, Sashi Kanth, Shiva, Sundar, Vijay, Vijaya Ganesh, and Yogesh. Thank you Raajay for being a wonderful roommate during my initial years in Madison.

Finally, I thank my family for their continued support and encouragement. Thank you amma, appa, mama, mami, Ramar, Hareni, Vicky, Renga, Kumaran, Priya Kumaran for your love and support. This Ph.D. would not have been possible with out my wife Priya and my daughter Vamshika. Thank you Priya for your love and thank you Vamshi kutty for bringing so much joy into my life.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Entity matching (EM) identifies data instances that refer to the same real-world entity, such as (David Smith, UWMadison) and (D. M. Smith, UWM). This problem has been a long-standing challenge in data management and will be even more critical in the age of data science. Most current EM works focus only on developing *matching algorithms*. Going forward, we argue that far more efforts should be devoted to building *EM systems*, in order to significantly advance the field.

We discuss the limitations of current EM systems, then present as a solution Magellan, a new kind of EM systems. Magellan is novel in four important aspects. (1) It provides how-to guides that tell users what to do in each EM scenario, step by step. (2) It provides tools to help users do these steps; the tools seek to cover the entire EM pipeline, not just matching and blocking as current EM systems do. (3) Tools are built on top of the data analysis and Big Data stacks in Python, allowing Magellan to naturally integrate into the Python ecosystem of data science tools, and to borrow a rich set of capabilities in data cleaning, IE, visualization, learning, etc. (4) Magellan provides a powerful scripting environment to facilitate interactive experimentation and quick "patching" of the system.

While promising, realizing the above novelties raises major challenges. First, it turns out that developing effective how-to guides, even for very simple EM scenarios such as applying supervised learning to match, is already quite difficult and complex. Second, developing tools to support these guides is equally difficult. In particular, current EM work may have dismissed many steps in the EM pipeline as engineering. But here we show that many such steps (e.g., loading the data, sampling and labeling, debugging, etc.) do raise difficult research challenges. Finally, while most current EM systems are stand-alone monoliths, Magellan is designed to be placed within an ecosystem and is expected to play well with others (e.g., other Python packages). We distinguish this by saying that current EM systems are *closed-world systems* whereas Magellan is an *open-world system*, because it relies on many other systems in the eco-system in order to provide the fullest amount of support to the user doing EM. It turns out that building open-world systems raises non-trivial challenges, such as designing the right data structures and managing metadata, among many others.

In this dissertation we discuss how we have addressed these challenges, built, and open sourced Magellan. As far as we can tell, Magellan is the most comprehensive open-source EM system today (August 2018), in terms of the number of features it supports. Magellan has been successfully used in several domain science projects in academia and projects in industry. We describe these "in the wild" experience with Magellan, as well as extensive experiments in controlled settings. Finally, we discuss lessons learned and many possible future research directions. Beside concrete contributions, this dissertation also introduces a new template of research, system development, and education for EM, with many potential impacts.

Chapter 1

Introduction

This dissertation studies *entity matching:* the problem of finding data instances that refer to the same real-world entity, such as (David Smith, UW-Madison) versus (D. M. Smith, UWM), and (The Return of the King, 2003) versus (LOTR: Return of the King, 2003).

We begin this chapter by showing that entity matching (EM) is a fundamental step in numerous data management applications, and will become even more critical in the age of data science. Next, we discuss the challenges of EM, state-of-the-art solutions, and their limitations. We then outline our Magellan solution to these limitations. Finally, we list the contributions and give a road map to the rest of the dissertation.

1.1 Applications of Entity Matching

Entity matching has had a long history, dating back to the late 50s (see [102, 55, 128, 82, 83, 63, 43, 37, 56, 51] for recent books, surveys, and tutorials; see also the related work chapter). One of the earliest applications of EM arose from the need to match persons for census purposes, among others [104, 57, 127, 129].

These early works originated from the statistics community. Starting in the 1980s, however, EM also received increasing attention in the database, data mining, Web, and AI communities, due to the need to integrate data that come from multiple disparate sources. This problem and its variants are often known as *data integration* [51]. For example, when an e-retailer acquires another e-retailer, it often ends up with two databases that keep track of their customers (e.g., their names, addresses, phones, etc.), and has to integrate these two databases into a single unified database. To do so, it must be able to match, i.e., identify the same customer across the two databases. Other important applications include integrating datasets for mining [70, 69], matching entities on the World-Wide Web (e.g., publications) [97], and matching entities on the Semantic Web [51].

Today, as data science grows, EM is receiving even more attention. This is because many data science applications must first perform data integration (DI) to combine the raw data from multiple sources, before analysis can be carried out to extract insights. The DI process in turn often requires EM [49].

1.2 Challenges of EM, Current Solutions, and Their Limitations

Entity matching has long been known to raise two major challenges: accuracy and scalability. First, EM is difficult to perform accurately, because the same entity may appear in many different formats. For example, a person name such as “David Smith” may appear as “D. Smith”, “Dave M. Smith”, “Prof. D. Smith”, etc. Second, EM is difficult to perform on a large scale. Consider for example matching two tables A and B, each having 100,000 tuples. Suppose each tuple describes a person and suppose that our goal is to find all tuple pairs $(a \in A, b \in B)$ that match. Then a naive solution would enumerate and consider matching all 10 billions of tuple pairs in the entire Cartesian product $A \times B$, which is practically infeasible.

Numerous solutions have been proposed to address these two challenges ([37, 56], see the related work chapter). Despite all this attention, however, today we do not really know whether the field is making good progress. The vast majority of EM works have focused on developing *algorithmic solutions*. But we know very little about whether these (ever-more-complex) algorithms are indeed useful in practice.

The field has also built mostly isolated EM system prototypes, which are hard to use and combine, and are often not powerful enough for real-world applications. This makes it difficult to decide what to teach in data science and data integration classes. Teaching complex EM algorithms and asking students to do projects using our prototype systems can train them well for doing EM research, but are not likely to train them well for solving real-world EM problems in later jobs.

Similarly, outreach to real users (e.g., domain scientists) is difficult. Given that we have mostly focused on "point EM problems" (as we discuss later), we do not know how to help them solve end-to-end EM tasks. That is, we cannot tell them how to start, what algorithms to consider, what systems to use, and what they need to do manually in each step of the EM process.

In short, today our EM effort in research, system development, education, and outreach seem disjointed from one another, and disconnected from real-world applications. As data science grows, this state of affairs makes it hard to figure out how we can best relate and contribute to this major new field.

1.3 Goals of the Dissertation

In this dissertation we take the first steps in addressing the above problems. We begin by arguing that the key to move forward (and indeed, to tie everything together) is to devote far more effort to building EM systems.

EM is engineering by nature. We cannot just keep developing EM algorithmic solutions in a vacuum. This is akin to continuing to develop join algorithms without building the rest of the RDBMSs. At some point we need to build end-to-end EM systems and work with real users to evaluate these algorithms, to integrate disparate R&D efforts, and to make practical impacts.

In this aspect, EM can take inspiration from RDBMSs and Big Data systems. Pioneering systems such as System R, Ingres, Hadoop, and Spark have really helped push these fields forward, by helping to evaluate research ideas, providing an architectural blueprint for the entire community to focus on, facilitating more advanced systems, and making widespread real-world impacts.

The question then is what kinds of EM systems we should build, and how? The goal of this dissertation is to explore answers to this question. Toward this goal, we proceed with the following steps:

1. Analyze current academic and industrial EM systems to understand the limitations that prevent them from being extensively in practice.

2. Propose the Magellan system architecture, which seeks to manage the entire EM process. Magellan manages this process in two stages: development and production. For each stage, it provides a how-to guide to the user, identifies the pain points in the guide, and provides (semi-)automatic tools to address these pain points. Tools are being built into the ecosystem of open-source data science tools in Python.

3. Discuss how to develop the development stage and the production stage of Magellan from a *user's perspective*. That is, how to create how-to guides for users, to identify pain points, and to develop tools for the pain points.

4. Discuss how to develop both stages from a *developer's perspective*. That is, what kinds of challenges developers face as they build guidance and tools for the stages.

5. Evaluate Magellan in both controlled settings as well as "in the wild", via teaching and outreach to domain science projects in academia and projects in industry.

1.4 Overview of the Solutions

We now outline our solutions to the above problem steps.

1.4.1 The Case for Entity Matching Management Systems

Limitations of Current EM Systems: We begin by analyzing 18 major non-commercial systems (e.g., D-Dupe, DuDe, Febrl, Dedoop, Nadeef), and 15 major commercial ones (e.g., Tamr, Data Ladder, Informatica Data Quality). Then we show that these systems suffer from four limitations that prevent them from being used extensively in practice.

- First, when performing EM users often must execute many steps, e.g., blocking, matching, exploring, cleaning, debugging, sampling, labeling, estimating accuracy, etc. Current systems however do not cover the entire EM pipeline, providing support for only a few steps (e.g., blocking, matching), while ignoring less well-known yet equally critical steps (e.g., debugging, sampling).

- Second, EM steps often exploit many techniques, e.g., learning, mining, visualization, outlier detection, information extraction (IE), crowdsourcing, etc. Today however it is very difficult to exploit a wide range of such techniques. Incorporating all such techniques into a single EM system is extremely difficult. EM is often an iterative process. So the alternate solution of moving data repeatedly among an EM system, a data cleaning system, an IE system, etc. does not work either, as it is tedious and time consuming. A major problem here is that most current EM systems are standalone monoliths that are not designed from the scratch to "play well" with other systems.

- Third, users often have to write code to "patch" the system, either to implement a lacking functionality (e.g., extracting product weights) or to glue together system components. Ideally such coding should be done using a script language in an interactive environment, to enable rapid prototyping and iteration. Most current EM systems however do not provide such facilities.

- Finally, in many EM scenarios users often do not know how to proceed end to end. Suppose a user wants to perform EM with at least 95% precision and 80% recall. Should he or she start out using a learning or rule-based EM approach? If learning-based, then which technique to select among the many existing ones? How to debug? What to do if after many tries the user still cannot reach 80% recall? Current EM systems provide no answers to such questions.

The Magellan Solution: To address these limitations, we propose Magellan, a new kind of EM systems. Magellan (named after Ferdinand Magellan, the first end-to-end explorer of the globe) is novel in several important aspects.

- First, Magellan provides how-to guides that tell users what to do in each EM scenario, step by step.

- Second, Magellan identifies the pain points of the steps (in the guide), then provides tools that help users do those pain points. These tools seek to cover the entire EM pipeline (e.g., debugging, sampling), not just the matching and blocking steps.

- Third, the tools are being built on top of the Python data analysis and Big Data stacks. Specifically, we propose that users solve an EM scenario in two stages. In the development stage users find an accurate EM workflow using data samples. Then in the production stage users execute this workflow on the entirety of data.

  We observe that the development stage basically performs data analysis. So we develop tools for this stage on top of the well-known Python data analysis stack, which provide a rich set of tools such as pandas, scikit-learn, matplotlib, etc. Similarly, we develop tools for the production stage on top of the Python Big Data stack (e.g., Pydoop, mrjob, PySpark, etc.). Thus, Magellan is well integrated with the Python data eco-system, allowing users to easily exploit a wide range of techniques in learning, mining, visualization, IE, etc.

- Finally, an added benefit of integration with Python is that Magellan is situated in a powerful interactive scripting environment that users can use to prototype code to "patch" the system.

As described, it is important to note that the how-to guides are *not* user manuals. Instead, they are *detailed algorithms* for the human user. A "rule of thumb" is that if the user knows how to code, he or she should be able to use the guide to execute the end-to-end EM scenario, even without utilizing any tool (of course, this can take a long time, but the key is that the user should be able to do it). In practice, the guide can utilize any appropriate (semi-)automatic existing tools.

Another important point to note is that unlike current EM systems, which often focus on providing implementations for just a few important problems in the EM process (e.g., blocking, matching, see Chapter 2), Magellan seeks to manage all aspects of the end-to-end EM process. We refer to this kind of systems as *entity matching management systems (EMMSs)*. Further, since Magellan heavily involves the human user (as a "first-class citizen" of the EM process), it is an example of *"human-in-the-loop" data management systems*, which have received significant recent attention [48].

EM Scenarios Considered: In practice, there is a wide variety of EM scenarios (e.g., matching two tables, matching a table to a knowledge base, matching within a single table, etc.) [51]. As a first step, in this dissertation, we will build Magellan to handle a few common scenarios, and hope that future work can extend it to more scenarios (over time). Specifically, we focus on (1) matching two tables using supervised learning, (2) matching two tables using rules, and (3) matching two tables using both supervised learning and rules.

1.4.2 The Development Stage of Magellan

With the high-level architecture of Magellan sketched out, we turn our attention to developing the development stage of Magellan, from a user's perspective. That is, what does a user expect to have in this stage, and how is that useful to the user?

To answer these questions, we begin by developing a how-to guide for the scenario of matching two tables using supervised learning. The guide states that to match two tables A and B, the user should load the tables into Magellan, do blocking, label a sample of tuple pairs, use the sample to iteratively find and debug a learning-based matcher, then return this matcher and its estimated matching accuracy.

For each of the above steps, we identify pain points, and discuss tools that we have developed for those pain points, as well as desirable future tools. We then build on the above how-to guide to develop guides for the other two EM scenarios (matching two tables using rules and matching using both learning and rules).

We show that even for relatively simple EM scenarios such as those discussed above, a good guide can already be quite complex. Thus developing how-to guides is a major challenge, but such guides are absolutely critical in order to successfully guide the user through the EM process.

We also show that each step of the guide, including those that prior work may have viewed as trivial or engineering (e.g., sampling, labeling), can raise many interesting re- search challenges. We provide preliminary solutions to several such challenges in this work. But much more remains to be done.

Finally, we show that the current guides and tools are already highly promising, in that users can already use them to achieve high matching accuracy on diverse data sets. Specifically, we report on extensive experiments with graduate students at UW-Madison in a data science class project. (Later we describe our experience applying Magellan "in the wild", to real-world EM scenarios in domain science projects and at companies. That experience further demonstrates the above promise and highlights challenges for future work.)

1.4.3 The Production Stage of Magellan

Earlier we have considered the development stage. Specifically, we considered developing how-to guides and tools to help the user experiment and find an accurate EM workflow, using data samples. In the next step, we consider the production stage, in which the user executes this workflow on the entirety of data, i.e., on the two original tables to be matched.

There are numerous challenges in the production stage, such as scaling, crash recovery, logging, etc. We will focus on scaling. We first motivate our scaling considerations and define our problem settings. In particular, we argue for the need to develop a how-to guide for users on how to handle scaling challenges in the production stage.

Next, we describe such preliminary guides for several scaling scenarios. We then identify pain points in the guides and develop tools for the pain points. Specifically, we develop tools to tune the parameters of the implementation versions of several EM commands. We present experiments showing that even preliminary tuning tools can already be useful to the user. Overall, our work in this part, even though still preliminary, shows the promise of following the "how-to guide / pain points / tools" template that we have successfully used for the development stage.

1.4.4 Building Magellan

In the next step, we discuss the above two stages from a developers perspective. That is, how developers should build these stages, what challenges they will face, and how they can address those challenges.

We begin by discussing the development stage. Here, we note that while most current EM systems are stand-alone monoliths, Magellan is designed to be placed within an "ecosystem" and is expected to "play well" with others (e.g., other Python packages). We say that Magellan is an "open-world system", because it relies on many other systems in the ecosystem in order to provide the fullest amount of support to the user doing EM.

In this context, the biggest challenges developers face arise from the need to make the Python packages interoperate. Example challenges include how to design appropriate data structures (for interoperability purposes), how to manage metadata (when packages can manage one anothers metadata), how to handle missing values across the packages, and more. We discuss how these challenges arise in the context of open-world systems, to which Magellan belongs, and our preliminary solutions. This is in contrast to "close-world systems" such as RDBMSs.

We then discuss the production stage. Here a major challenge is scaling the Python commands that users may want to execute. We discuss current common scaling solutions (such as custom optimizations for a single core or multiple cores), analyze them, then create a how-to guide for developers (on how to scale Magellans commands). We used this guide to scale a subset of current commands. Finally, we discuss the current status of the open-source implementation of the Magellan system.

1.4.5 Magellan "in the Wild": Successes and Lessons Learned

In the past three years we have started to address the above challenges. Specifically, we have open sourced Magellan [13]. As far as we can tell, Magellan is the most comprehensive open-source EM system today, in terms of the number of features it supports.

Magellan has been successfully used in five domain science projects at UW-Madison (in economics, biomedicine, environmental science [80, 89, 107, 28]), and at several companies (e.g., Johnson Controls, Marshfield Clinic, Recruit Holdings [4], WalmartLabs). For example, at WalmartLabs it improved the recall of a deployed EM solution by 34%, while reducing precision slightly by 0.65%. It has also been used by 400+ students to match real-world data in five data science classes at UW-Madison (e.g., [5]).

Applying Magellan to the above real-world applications raised many research challenges. Examples include helping users finalize their matching definition [80, 48], debugging blocking [94], debugging rule-based EM [105], human-in-the-loop EM [48], applying deep learning to match textual data [101], hands-off string matching, data cleaning, and more.

We have started to address some of these research challenges [94, 101, 105, 48], describe case studies [80], and summarize the lessons learned [48, 80]. Magellan and the data generated in this project have also been used by other research groups (e.g., [53, 62]).

In the last part of the dissertation, we briefly describe how we have successfully applied Magellan “in the wild” to a number of EM projects in domain science and industry, as described above. We then describe a case study of end-to- end EM in applied economics in detail. The goal is to make very concrete many challenges that arise during EM in practice. As far as we can tell, no academic publication has discussed a detailed execution of end-to-end EM in practice. Finally, based on our experience, including working on the above applied economics case, we discuss a set of lessons learned.

1.5 Contributions of the Dissertation

1.5.1 Concrete Contributions

As described, this dissertation makes the following concrete contributions:

- First, we analyze 33 commercial and open-source EM systems and identify four major limitations that prevent these systems from being used extensively in practice.

- Second, we propose Magellan, a new kind of EM systems that addresses the above limitations.

- Third, we address the development stage in Magellan for matching two tables using supervised learning and rules. Specifically, we develop how-to guides and tools for the pain points in the guide, and extensively evaluate them using real users.

- Fourth, we address the production stage in Magellan for scaling a single step in an EM workflow that gets executed on a single machine with multiple cores. Specifically, we develop how-to guide for developers to scale a single step and develop how-to guides for users to use these scaled implementations. We then develop tools for the pain points and evaluate them using real-world data sets.

- Fifth, we have developed and open-sourced Magellan system. As far as we can tell, Magellan is the most comprehensive open-source EM system today, in terms of the number of features it supports.

- Finally, we have successfully applied Magellan to multiple EM projects in academia and industry, and described in detail one such case, end to end. Based on this experience, we discuss a number of lessons learned.

1.5.2 Potential Broader Impacts

Beside the above concrete contributions, this dissertation introduces a new template of research, system development, and education for EM, with potential research and practical impacts.

Specifically, it introduces a new research template for EM. The current paradigm focuses largely on developing algorithmic solutions for blocking and matching, two important steps in the EM process. However, this dissertation argues that EM processes often involve many other "pain points". It proposes a new research template in which we move beyond examining just blocking and matching. Instead, we would (a) develop a step-by-step how-to guide that captures the entire end-to-end EM process, (b) examine the guide to identify all true pain points of the EM process, then (c) develop solutions and tools for the pain points. The dissertation confirms that there are indeed many other pain points (e.g., data labeling, debugging, EM-centric cleaning, and defining the notion of match), that solving these pain points is critical for developing practical EM tools, and that addressing them raises many novel research challenges.

The dissertation then introduces a new system building template for EM. Most current EM systems are built as stand-alone monolithic systems. However, the dissertation makes the case that such systems are very difficult to extend, customize, and combined. It observes that many EM steps essentially perform data science tasks, and that there exist already vibrant ecosystems of open-source data science tools (e.g., those in Python and R), which are being used heavily by data scientists to solve these tasks.

Thus, the dissertation proposes to develop EM tools within such data science ecosystems. This way, the EM tools can easily exploit other tools in the ecosystems, and at the same time make such ecosystems better at solving DI problems. Compared to current system building practices, this is different in two ways. First, it suggests that instead of building isolated stand-alone systems for EM (the way we built RDBMSs for relational data management), we should focus on building ecosystems of tools. Second, it suggests that researchers working on EM in our community should "connect" with the vibrant and expanding ecosystems of open-source data science tools and build our EM tools directly into those ecosystems.

The new research and system development template articulated by this dissertation suggests a new way to teach and train our students in EM. Today, we teach our students isolated research problems, and ask them to do projects using mostly stand-alone research-prototype EM tools that industry is often unfamiliar with. In this new way, first we teach our students to solve EM problems end-to-end, to identify pain points, and to find or develop new tools to solve the pain points. Thus, they are solving EM problems grounded in practice. Second, we train them in using tools in the open-source data science ecosystems, which they are likely to use again in industry. Finally, if we develop new research tools, they will be parts of such ecosystems and thus can be naturally evaluated by students in their class projects.

Finally, while this dissertation focuses on EM, its templates can potentially be applied to solve other DI problems as well, such as schema matching, information extraction, data cleaning, etc. In fact, a recent work of ours [49] has proposed such a system building agenda for data integration, based on the work in this dissertation.

1.6 Outline

The rest of this dissertation is organized as follows. Chapter 2 discusses the limitations of current EM systems and then describes Magellan. Chapters 3 and 4 discuss the development and production stages in Magellan for matching two tables using supervised learning and rules, from a user's perspective. Chapter 5 discusses building the Magellan system as a part of the Python data ecosystem, from a developer's perspective. Chapter 6 describes applying Magellan "in the wild": successes and lessons learned. It also describes a case study of solving an EM problem end to end using Magellan. Chapter 7 discusses the related work, and Chapter 9 concludes.

Parts of this dissertation have been published in database conferences. Magellan (Chapters 2, 3, 4) is described in two VLDB-16 papers [77, 79] and in SIGMOD Record [81].Our experience of matching grant descriptions (Chapter 6) is included as a part of Magellan case studies [80]. Many of the lessons we have learned in the course of developing Magellan have been discussed in [48, 49].

Chapter 2

The Case for Entity Matching Management Systems

In this chapter, we first describe entity matching (EM). Next, we discuss current EM systems and their limitations. We then make a case for entity matching management systems and discuss what these systems should do. As an example of such systems, we propose Magellan, which we will build into the Python ecosystem of open-source data science tools.

2.1 Entity Matching

Entity Matching (EM), also known as record linkage, data matching, etc., has received much attention in the past few decades [37, 56]. A common EM scenario finds all tuple pairs (a, b) that match, i.e., refer to the same real-world entity, between two tables A and B (see Figure 2.1). Other EM scenarios include matching tuples within a single table, matching into a knowledge base, matching XML data, etc. [37, 50].

Most EM works have developed matching algorithms, exploiting rules, learning, clustering, crowdsourcing, among others [37, 56]. The focus is on improving the matching accuracy and reducing costs (e.g., run time). Trying to match all pairs in $A \times B$ often takes very long. So users often employ heuristics to remove obviously non-matched pairs (e.g., products with different colors), in a step called *blocking*, before matching the remaining pairs. Several works have studied this step, focusing on scaling it up to large amounts of data (see the chapter on related work).

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 2.1: An example of matching two tables.

2.2 Current Entity Matching Systems

In contrast to the extensive effort on matching algorithms (e.g., 96 papers were published on this topic in 2009-2014 alone, in SIGMOD, VLDB, ICDE, KDD, and WWW), there has been relatively little work on building EM systems. As of early 2016 we counted 18 major non-commercial systems (e.g., D-Dupe, DuDe, Febrl, Dedoop, Nadeef), and 15 major commercial ones (e.g., Tamr, Data Ladder, Informatica Data Quality). In what follows we examine these two types of systems in detail.

2.2.1 Non-Commercial EM Systems

Table 2.1 summarizes the characteristics of 18 non-commercial systems (see [37] for a discussion of such systems up to 2012). Empty cells mean reliable information cannot be gleaned from the documentation and system examination. This table shows that

- The systems focus on the scenarios of matching within a single table or across two tables.
- They provide a wide range of methods for the well-known blocking and matching steps, but little guidance on how to select appropriate blockers and matchers.
- Eight systems provide limited data exploration capabilities (e.g., browsing, showing statistics about the data) and cleaning capabilities (mostly ways to perform relatively simple transformations such as regex-based ones and to clean certain common attributes such as person names). No system provides support for less well-known but critical steps such as debugging, sampling, and labeling.

Name	Affiliation	Scenarios	Blocking	Matching	Exploration, cleaning	User interface	Language	Open source	Scaling
Active Atlas	University of Southern California	Single table, two tables	Hash-based	ML-based (decision tree)	No	GUI, commandline	Java	No	No
BigMatch	US Census Bureau	Single table, two tables	Attribute equivalence, rule-based	Not supported	No	Commandline	C	No	Yes (supports parallelism on a single node)
D-Dupe	University of Maryland	Single table, two tables	Attribute equivalence	Relational clustering		GUI	C#	No	No
Dedoop	University of Leipzig	Single table	Attribute equivalence, sorted neighborhood	ML-based (decision tree, logistic regression, SVM etc.)	No	GUI	Java	No	Yes (Hadoop)
Dedupe	Datamade	Single table, two tables	Canopy clustering, predicate-based	Agglomerative hierarchical clustering-based	Browsing, statistics, basic transformation, cleaning certain attribute types	Commandline	Python	Yes	Yes
DuDe	University of Potsdam	Single table, two tables	Sorted neighborhood	Rule-based	Statistics	Commandline	Java	Yes	No
Febrl	Australian National University	Single table, two tables	Full index, blocking index, sorting index, suffixarray index, qgram index, canopy index, stringmap index	Fellegi-Sunter, optimal threshold, k-means, FarthestFirst, SVM, TwoStep	Browsing, statistics, basic transformation, cleaning certain attribute types	GUI, commandline	Python	Yes	No
FRIL	Emory University	Single table, two tables	Attribute equivalence, sorted neighborhood	Expectation maximization	Basic transformation, cleaning certain attribute types	GUI	Java	Yes	Yes (supports parallelism on a single node)
MARLIN	University of Texas at Austin		Canopy clustering	ML-based (decision tree, SVM)					No
Merge Toolbox	University of Duisburg-Eissen	Single table, two tables	Attribute equivalence, canopy clustering	Probabilistic, expectation maximization	No	GUI	Java	No	No
NADEEF	Qatar Computing Research Institute	Single table, two tables		Rule-based	No	GUI	Java	No	No
OYSTER	University of Arkansas	Single table, two tables	Attribute equivalence	Rule-based	Statistics	Commandline	Java	Yes	No
pydedupe	GPoulter (GitHub username)	Single table, two tables	Attribute equivalence	ML-based, rule-based	Browsing, statistics, basic transformation, cleaning certain data types	Commandline	Python	Yes	No
RecordLinkage	Institute of Medical Biostatistics, Germany	Single table, two tables	Attribute equivalence	ML-based, probabilistic	Browsing, statistics, basic transformation, cleaning certain attribute types	Commandline	R	Yes	No
SERF	Stanford University	Single table		R-Swoosh algorithm	No	Commandline	Java	No	No
Silk	Free University of Berlin	RDF data		Rule-based	Browsing, basic transformation	GUI	Java	Yes	Yes (supports parallelism on a single node, Hadoop)
TAILOR	Purdue University	Single table, two tables	Attribute equivalence, sorted neighborhood	Probabilisitic, clustering, hybrid, induction	No	GUI	Java	No	No
WHIRL	William Cohen			Vector space model		Commandline	C++	No	No

Table 2.1: Characteristics of 18 non-commercial EM systems.

- No system provides how-to guides that tell users how to do EM, step by step. And no system emphasizes a clear distinction between the development stage and the production stage (i.e., guiding users to develop a good EM workflow in the development stage and then execute the workflow in the production stage).

- Less than half of the systems are open source. No system provides any easy interfacing with data science stacks (and is not intentionally designed to interface with such stacks).

- Thirteen systems are written in languages such as C, C#, C++, and Java, and thus are not situated in a powerful scripting environment that facilitates rapid and iterative experimentation (e.g., examining the effect of a data cleaning operation, trying out a different blocker or matcher).

- About half of the systems provide just commandline interfaces, while the remaining half also provide GUIs. A few systems provide limited scaling capabilities.

2.2.2 Commercial EM Systems

We compiled a list of 15 commercial EM systems from our experience working in industry, and from examining quarterly reports such as "The Forrester Wave: Data Quality Solutions" [64] and other trade literature. Tables 2.2-2.3 summarize the characteristics of these systems. Again, the empty cells in the tables mean reliable information cannot be gleaned from the documentation and system examination.

Table 2.2 summarizes the general characteristics of the commercial systems. It shows that

- Five systems focus exclusively on EM. The remaining ten systems provide EM as a part of data integration or cleaning pipelines.

- The systems focus on the scenarios of matching within a single table or across two tables. Unlike non-commercial systems, these systems have very sophisticated GUI or Web-based user interfaces.

	Purpose and how EM fits in	**Supported EM scenarios**	**Main user interface**	**Distinction between dev. and prod. stages**	**Language**	**Scripting environment**
DataMatch from Data Ladder	Data cleaning, data matching. EM forms the core of their solution	Multiple tables	GUI	No		No
Dedupe.io	Record linkage, deduplication. EM forms the core of their solution	Single table, two tables	Web-based	No		No
FuzzyDupes	Duplicate detection, data cleaning. EM forms the core of their solution	Single table, two tables	GUI	No		No
Graphlab Create	EM is offered as a service on top of their GraphLab platform	Single table, two tables, linking records to a KB	Web-based		C++	Yes
IBM InfoSphere	Customer data analytics. EM is supported by a component (BigMatch) in the product	Single table, two tables	Web-based		Java	No
Informatica Data Quality	Improve data quality. EM forms a part of data quality pipeline	Single table, two tables	GUI			No
LinkageWiz	Data matching and data cleaning. EM forms the core of their solution	Single table, two tables	GUI	No		No
Oracle Enterprise Data Quality	Improve data quality. EM forms a part of data quality pipeline	Single table, two tables	GUI			No
Pentaho Data Integration	ETL, data integration. EM forms a part of ETL/data integration pipe line	Single table, two tables	GUI		Java	No
SAP Data Services	Improve data quality, data integration. EM forms a part of data integration pipeline	Single table, two tables	GUI	No		
SAS Data Quality	Improve data quality. EM forms a part of data quality pipeline	Single table, multiple tables	Web-based			Limited support
Strategic Matching	Data matching and data cleaning. EM forms the core of their solution	Single table, two tables	GUI	No		No
Talend Data Quality	Improve data quality. EM forms a part of data quality pipeline	Single table, two tables	GUI			No
Tamr	Data curation. EM forms a part of data curation pipeline	Multiple tables	Web-based	No	Java	No
Trillium Data Quality	Improve data quality. EM forms a part of data quality pipeline	Single table, multiple tables	GUI			No

Table 2.2: Characteristics of 15 commercial EM systems (Part 1).

- There is no how-to guide that tells users how to do EM, step by step. Instead, the vendors sell consulting services (sometimes called "data stewarding") that presumably help users use the

	Supported data formats/sources	**Data exploration support**	**Data cleaning support**	**Down sampling input table(s)**	**Blocking**	**Support to combine multiple blockers**	**Debugging blocker output**	**Labeling data**	**Matching**	**Debugging matcher output**	**Scaling**
DataMatch from Data Ladder	Relational databases, XLS, DB2, CSV, delimited text files	Browsing, statistics	Yes	No	Not supported	No	No	No	Rule-based	Limited support	Yes
Dedupe.io	Relational databases (Postgres), CSV, XLS,				Canopy clustering, predicate-based blocking	No	No	Yes	Clustering-based (AHC)	Limited support	Yes
FuzzyDupes	Relational databases, XLS, CSV, delimited text files			No		No	No	No			Yes
Graphlab Create	Relational databases, CSV, Pandas dataframes , HDFS, Amazon S3, JSON	Browsing, statistics			Attribute equivalence	No			Clustering-based (KNN)		Yes (Hadoop, Spark)
IBM InfoSphere	Relational databases, XLS, delimited text files, XML, JSON, HDFS, text files	Browsing, statistics	Yes		Attribute equivalence, blocking based on first 3 characters, phonetic codes				Rule-based		Yes (Hadoop)
Informatica Data Quality	Relational databases, CSV, excel, XML, delimited text files, HDFS	Browsing, statistics	Yes	No	Attribute equivalence	No	No	No	Rule-based		Yes
LinkageWiz	XLS, delimited text files, SPSS	Browsing, statistics	Yes	No	Attribute equivalence	No	No	No	Rule-based	Limited support	
Oracle Enterprise Data Quality	Relational databases, XLS, delimited text files	Browsing, statistics	Yes	No		No	No	No	Rule-based	Limited support	Yes (Hadoop, Hive, HBase, Pig, Sqoop, Spark)
Pentaho Data Integration	Relation databases, CSV, XML, JSON, MongoDB, NuoDB, Couchbase, Avro	Browsing, statistics	Yes				No		Rule-based		Yes (Hadoop, Spark, Mongo DB, Splunk, Cassandra)
SAP Data Services	Relational databases, CSV, XLS, JSON, XML, HDFS	Browsing, statistics	Yes	No	Attribute equivalence	No	No	No	Rule-based		Yes (Hadoop, Spark)
SAS Data Quality	Relational databases, XLS and delimited text files, XML	Browsing, statistics	Yes		Not supported				Hash-based		Yes (Hadoop)
Strategic Matching	Relational databases (SQL server), MS Access, SAS	Browsing, statistics	Yes	No			No	No	Rule-based	Limited support	
Talend Data Quality	Relational databases, CSV, XLS, XML, JSON, EBCDIC	Browsing, statistics	Yes	No	Attribute equivalence	No	No	No	Rule-based	Limited support	Yes (Hadoop, Spark)
Tamr	Relational databases, JSON, XML, YAML, RDF, HDFS, Hive, Amazon/redshift, Google cloud storage, MongoDB, Cloudant, Cassandra, CSV, XLS			No	Modified k-means	No	No	Yes	Rule-based	Limited support	Yes
Trillium Data Quality	Relational databases, CSV, XLS, JSON, HDFS, NoSQL	Browsing, statistics	Yes						Rule-based		Yes (Hadoop, Spark)

Table 2.3: Characteristics of 15 commercial EM systems (Part 2).

systems. Seven systems make no distinction between the development stage and the production stage. For the remaining eight systems we cannot reliably tell from the documentation, but they do not seem to make such a distinction either.

- Many systems use languages such as C++ and Java. As far as we can tell, no system (except GraphLab Create) is situated in a powerful scripting environment for rapid and iterative experimentation.

- No system is open source and designed to interface well with tools in a data science stack.

Table 2.3 summarizes the support for the entire EM pipeline in these systems. It shows that

- These systems support far more types of input data (e.g., relational tables, JSON, CSV, XML) than the non-commercial systems.

- There seems to be more support for data exploration and cleaning (compared to non-commercial systems), though still limited. Data exploration is typically accomplished via GUIs that display statistics about the data (e.g., the percentage of missing values of an attribute). Many systems provide tools to clean common kinds of attributes (e.g., addresses, phone numbers, person names). But powerful general-purpose data cleaning tools are typically missing.

- Interestingly, these systems do not seem to provide as many different types of blocking and matching as the non-commercial systems. For example, the most common type of supported blocking is attribute equivalence, and the most common type of supported matching is rule-based. It is possible that these systems need to scale EM to very large amounts of data and so they intentionally limit the set of blocking and matching techniques considered for now, to ensure scalability. Indeed, virtually all systems provide capabilities to scale, using Hadoop and Spark.

- There is very limited or no support for other critical steps of the EM pipeline, such as sampling, debugging, and labeling. For example, there is no support for debugging blockers, and support for debugging matchers is typically limited to showing which EM rule fires on a given tuple pair.

We now describe a few selected commercial systems, specifically SAS Data Quality, Informatica Data Quality, DataMatch, and Tamr.

SAS Data Quality: This system (henceforth SAS for short) provides EM as a part of their data quality pipeline. SAS focuses on the scenarios of matching within a single table or across multiple tables. The EM workflow supported in SAS consists of five major steps.

First, the user loads the data into SAS. SAS supports various data formats and sources, such as Excel, CSV, XML, delimited text files, relational databases, and HDFS.

Second, the user explores the loaded data. SAS lets the user perform pattern analysis, column analysis, and domain analysis. In pattern analysis the user can verify if the data values in an attribute match the expected pattern (e.g., 9-digits for SSN, 10-digits for phone numbers), and visualize the distribution and frequency for various patterns, e.g., how many phone numbers were of the form (xxx) xxx-xxxx). In column analysis, the user can explore various statistics (e.g., cardinality, number of missing values, range, min, mean, median) of a column in a table. In domain analysis, the user can verify if the data conforms to the expected or accepted data values and ranges (e.g., age is between 0 and 150 years).

Third, the user cleans and standardizes the data. In cleaning, the user can fix capitalization in data values, remove punctuations, break a “full name” column into “first name” and “last name” columns by specifying a delimiter, etc. In standardization, the user specifies that an attribute is of the type “name”, “address”, “phone”, etc. and SAS makes sure that names are capitalized consistently, addresses use “st.” as an abbreviation for street names, etc.

Fourth, the user performs hash-based matching in a single table or across multiple tables. Specifically, the user first selects the attributes (say a_1, a_2, a_3) to consider for matching. For every tuple t, SAS will then generate a hash code, $h(t)$, which is a concatenation of multiple smaller hash codes, one per attribute, i.e., $h(t) = h(t.a_1)!h(t.a_2)!h(t.a_3)$, where ! is the concatenating delimiter. SAS generates the hash code per attribute by taking two inputs from the user: (a) *type* value for the attribute from a pre-defined set, comprising standard types such as name, address, organization, date, zip, and (b) a sensitivity value for the attribute telling SAS how sensitive the hashing function should be to variations in values (e.g., a low sensitivity will result in same hash code for Rob, Robert, Bob, Bobby; a moderate sensitivity will result in same hash code for Rob and Robert, but a different hash code for Bob and Bobby; a high sensitivity will result in different hash codes for each of them).

Finally, after the hash codes have been generated for each tuple in a table (or multiple tables), SAS will show the tuples grouped into clusters, each cluster having tuples with the same hash code. The user then consolidates the data by taking one of the three actions of deleting (i.e., physically deleting duplicate tuples), merging (i.e., keeping the best information across multiple tuples), or retaining all the tuples.

Informatica Data Quality: This system provides EM as a part of its data quality pipeline. Specifically, it supports matching within a single table or across two tables. The supported EM workflow consists of six steps.

First, the user loads the data into the system. The system supports various data formats such as CSV, Excel, XML, delimited text files etc.

Second, the user explores the data to identify attributes to use for blocking and matching. The system provides tools to analyze individual attributes and explore various statistics about the attributes.

Third, the user cleans and standardizes the data. Specifically, the user can fix variations in format or spelling, remove punctuations, fix capitalization etc. Further, the system also provides support to standardize certain attribute types like address, phone number etc.

Fourth, the user performs blocking by selecting an attribute to be used as a blocking key. Records with the same blocking key are grouped together.

Fifth, the user performs matching within each group. Specifically, the system supports four types of matchers: Hamming distance, edit distance, Jaro distance, and bigram. The user needs to specify which matchers to use, along with a matching threshold and weights for different matchers. Record pairs whose aggregate score is greater than or equal to the matching threshold are considered duplicates. The system groups the matching record pairs into clusters.

Finally, the user examines the clusters of records and decides to either consolidate the duplicate records into a master record or delete the duplicate records.

DataMatch: DataMatch from Data Ladder provides a software suite for data cleansing, matching, and deduplication. Entity matching is the core of their solution. Specifically, the tool supports deduplicating a single table or matching multiple tables. The matching workflow consists of the following six steps: (1) loading the data, (2) profiling, (3) cleaning and standardizing, (4) matching, (5) viewing and consolidating the results, and (6) exporting the results.

The user begins by loading the data into the tool (the tool supports various data formats/sources such as XLS, SQL server, MySQL, MS Access, CSV, DB2, and delimited text file). Next, the user can explore the data to assess the data quality and get some useful statistics (e.g., missing values, presence of non-printable characters, mean, median, mode). Next, the user can clean and standardize the data. The tool provides support for basic transformations such as making strings uppercase/lowercase/proper case, removing non-printable characters, removing characters specified by the user, and cleaning email using predefined regular expressions. Further, the tool also provides support to standardize certain attribute types such as person names, address, etc.

After cleaning, the user will perform matching. The tool supports only rule-based matching. Specifically, the user will specify the features (using a predefined list of similarity functions) to be computed for the attributes from the tables, and provide a matching threshold. Tuple pairs with the aggregate score greater than or equal to the matching threshold are considered matches. Next, the user can view and consolidate the matched tuple pairs. The user can manually review and clean the matches by flagging tuple pairs as non-matches.

Next, the matched tuple pairs are clustered by the system into groups, where all tuples in a group match and tuples across groups do not. Next, the user can specify how the group should be merged to form a canonical tuple. Specifically, for each attribute the user can specify whether the longest string should be taken, the average value (in the case of numerical values) should be taken, etc. Also, the user can control this decision per tuple pair. Finally, the user can export the results. The tool provides exporting the results to various file formats/sinks such as XLS, SQL server, MySQL, MS Access, CSV, DB2, and delimited text file.

Tamr: This system has entity matching as a component in a data curation pipeline. This EM component effectively does deduplication and merging: given a set of tuples D, clusters them into groups of matching tuples, and then merges each group into a super tuple. Toward this goal, Tamr starts by performing blocking on the set of tuples D. Specifically, it creates a set of categories, then use some relatively inexpensive similarity measure to assign each tuple in D to one or several categories. Only tuples within each category will be matched against one another.

Next, Tamr obtains a set of tuple pairs and asks users to manually label them as matched / non-matched. Tamr takes care to ensure that there are a sufficient number of matched tuple pairs in this set. Next, Tamr uses the labeled data to learn a set of matching rules. These rules use the similarity scores among the attributes of a tuple pair, or the probability distributions of attribute similarities for matching and non-matching pairs (these probabilities in turn are learned using a Naive Bayes classifier). Next, the matching rules are applied to find matching tuple pairs. Tamr then runs a correlation clustering algorithm that uses the matching information to group tuples into matching group. Finally, all tuples within each group are consolidated using user-defined rules to form a super tuple.

2.3 Key Limitations of Current Systems

Overall, we found that commercial EM systems are better than non-commercial EM systems in terms of support for the types of input data, user interfaces, data exploration and cleaning, and scaling. They appear less powerful than the non-commercial ones in terms of the types of supported blocking and matching techniques. Both types of systems however suffer from the following four major problems that we believe prevent these systems from being used widely in practice:

1. Systems Do Not Cover the Entire EM Pipeline: When performing EM users often must execute many steps, e.g., blocking, matching, exploration, cleaning, extraction (IE), debugging, sampling, labeling, etc. Current systems provide support for only a few steps in this pipeline, while ignoring less well-known yet equally critical steps. For example, all 33 systems that we have examined provide support for blocking and matching. Twenty systems provide limited support for data exploration and cleaning. There is no meaningful support for any other steps (e.g., debugging, sampling, etc.). Even for blocking the systems merely provide a set of blockers that users can call; there is no support for selecting and debugging blockers, and for combining multiple blockers.

2. Difficult to Exploit a Wide Range of Techniques: Practical EM often requires a wide range of techniques, e.g., learning, mining, visualization, data cleaning, IE, SQL querying, crowdsourcing, keyword search, etc. For example, to improve matching accuracy, a user may want to clean the values of attribute “Publisher” in a table, or extract brand names from “Product Title”, or build a histogram for “Price”. The user may also want to build a matcher that uses learning, crowdsourcing, or some statistical techniques.

Current EM systems do not provide enough support for these techniques, and there is no easy way to do so. Incorporating all such techniques into a single system is extremely difficult. But the alternate solution of just moving data among a current EM system and systems that do cleaning, IE, visualization, etc. is also difficult and time consuming. A fundamental reason is that most current EM systems are stand-alone monoliths that are not designed from the scratch to “play well” with other systems. For example, many current EM systems were written in C, C++, C#, and Java, using proprietary data structures. Since EM is often iterative, we need to repeatedly move data among these EM systems and cleaning/IE/etc systems. But this requires repeated reading/writing of data to disk followed by complicated data conversion.

3. Difficult to Write Code to “Patch” the System: In practice users often have to write code, either to implement a lacking functionality (e.g., to extract product weights, or to clean the dates), or to tie together system components. It is difficult to write such code correctly in “one shot”. Thus ideally such coding should be done using an interactive scripting environment, to enable rapid prototyping and iteration. This code often needs access to the rest of the system, so ideally the system should be in such an environment too. Unfortunately only 5 out of 33 systems provide such settings (using Python and R).

4. Little Guidance for Users on How to Match: In our experience this is by far the most serious problem with using current EM systems in practice. In many EM scenarios users simply do not know what to do: how to start, what to do next? Interestingly, even the simple task of taking a sample and labeling it (to train a learning-based matcher) can be quite complicated in practice, as we show in Section 3.2.3. Thus, it is not enough to just build a system consisting of a set of tools. It is also critical to provide step-by-step guidance to users on how to use the tools to handle a particular EM scenario. No EM system that we have examined provides such guidance.

2.4 Entity Matching Management Systems

To address the above limitations, we propose to build a new kind of EM systems. In contrast to current EM systems, which mostly provide a set of implemented matchers/blockers, these new systems are far more advanced. First and foremost, they seek to handle a wide variety of EM scenarios. These scenarios can use very different EM workflows. So it is difficult to build a single system to handle all EM scenarios. Instead, we should build a set of systems, each handling a well-defined set of similar EM scenarios. Each system should target the following goals:

1. **How-to Guide:** Users will have to be "in the loop". So it is critical that the system provides a how-to guide that tells users what to do and how to do it. The guide is a detailed algorithm for the human user. A "rule of thumb" is that if the user knows how to code, he or she should be able to use the guide to execute the EM scenario, even without utilizing any tool (of course, this can take a long time, but the key is that the user should be able to do it).

2. **User Burden:** The system should minimize the user burden. It should provide a rich set of tools to help users easily do each EM step (focusing for example on the pain points of the step), and do so for all steps of the EM pipeline, not just matching and blocking. Special attention should be paid to debugging, which is critical in practice.

3. **Runtime:** The system should minimize tool runtimes and scale tools up to large amounts of data.

4. **Expandability:** It should be easy to extend the system with any existing or future techniques that can be useful for EM (e.g., cleaning, IE, learning, crowdsourcing). Users should be able to easily "patch" the system using an interactive scripting environment.

Of these goals, "expandability" deserves more discussion. If we can build a single "super-system" for EM, do we need expandability? We believe it is very difficult to build such a system. First, it would be immensely complex to build just an initial system that incorporates all of the techniques mentioned in Goal 4. Indeed, despite decades of development, today no EM system comes close to achieving this.

Second, it would be very time consuming to maintain and keep this initial system up-to-date, especially with the latest advances (e.g., crowdsourcing, deep learning).

Third, and most importantly, a generic EM system is unlikely to perform equally well for multiple domains (e.g., biomedicine, social media, payroll). Hence we often need to extend and customize it to a particular target domain, e.g., adding a data cleaning package specifically designed for biomedical data (written by biomedical researchers). For the above three reasons, we believe that EM systems should be fundamentally expandable.

Clearly, systems that target the above goals seek to *manage* all aspects of the end-to-end EM process. So we refer to this kind of systems as *entity matching management systems (EMMSs)*. Building EMMSs is difficult, long-term, and will require a new kind of architecture compared to current EM systems. In the rest of this chapter we describe Magellan, an attempt to build such an EMMS.

2.5 The Magellan Approach

Figure 2.2 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. 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.

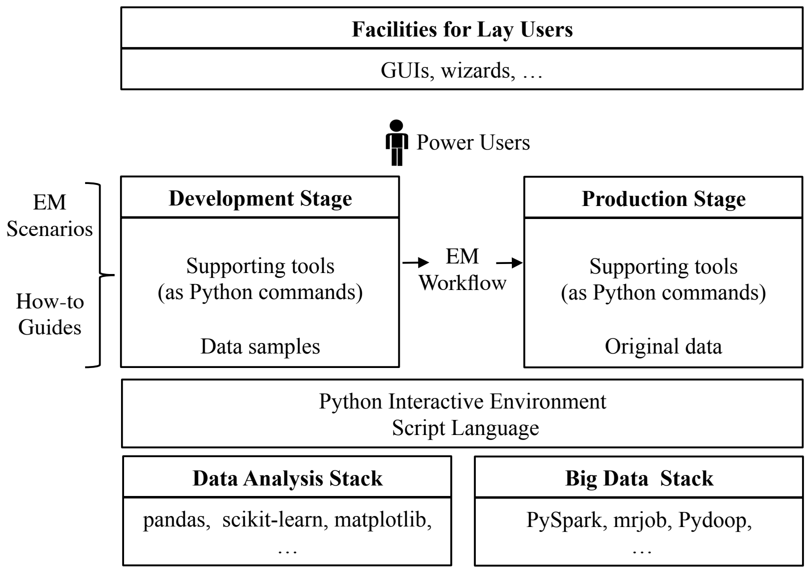

Figure 2.2: The Magellan architecture.

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 2.2), and lay user actions can be translated into sequences of commands in the underlying Magellan.

2.5.1 EM Scenarios and Workflows

We classify EM scenarios along four dimensions:

- **Problems:** Matching two tables; matching within a table; matching a table into a knowledge base; etc.
- **Solutions:** Using learning; using learning and rules; performing data cleaning, blocking, then matching; performing IE, then cleaning, blocking, and matching; etc.

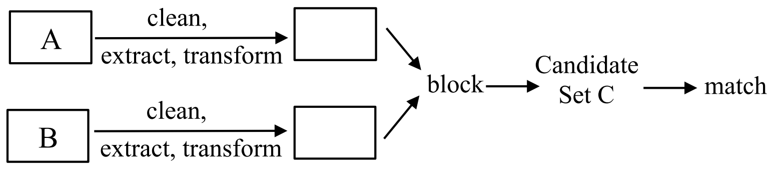

Figure 2.3: The EM workflow for the learning-based matching scenario.

- **Domains:** Matching two tables of biomedical data; matching e-commerce products given a large product taxonomy as background knowledge; etc.

- **Performance:** Precision must be at least 92%, while maximizing recall as much as possible; both precision and recall must be at least 80%, and run time under four hours; etc.

An EM scenario can constrain multiple dimensions, e.g., matching two tables of e-commerce products using a rule-based approach with desired precision of at least 95%.

Clearly there is a wide variety of EM scenarios. So we will build Magellan to handle a few common scenarios, and then extend it to more similar scenarios over time. Specifically, for now we will consider the three scenarios that match two given relational tables A and B using (1) supervised learning, (2) rules, and (3) learning plus rules, respectively. These scenarios are very common. In practice, users often try Scenario 1 or 2, and if neither works, then a combination of them (Scenario 3).

EM Workflows: As discussed earlier, to handle an EM scenario, a user often has to execute many steps, such as cleaning, IE, blocking, matching, etc. The combination of these steps form an *EM workflow*. Figure 2.3 shows a sample workflow of matching two tables using supervised learning.

2.5.2 The Development Stage versus the Production Stage

From our experience with real-world users' doing EM, we propose that the how-to guide tell the user to solve the EM scenario in two stages: *development* and *production*. In the development stage the user tries to find a good EM workflow, e.g., one with high matching accuracy. This is typically done using data samples. In the production stage the user applies the workflow to the entirety of 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. The following example illustrates these two stages.

Example 2.5.1. *Consider matching two tables A and B each having 1M tuples. Working with such large tables will be very time consuming in the development stage, especially given the iterative nature of this stage. Thus, in the development stage the user U starts by sampling two smaller tables A' and B' from A and B, respectively. Next, U performs blocking on A' and B'. The goal is to remove as many obviously non-matched tuple pairs as possible, while minimizing the number of matching pairs accidentally removed. U may need to try various blocking strategies to come up with what he or she judges to be the best.*

The blocking step can be viewed as removing tuple pairs from $A' \times B'$. Let C be the set of remaining tuple pairs. Next, U may take a sample S from C, examine S, and manually write matching rules, e.g., "If titles match and the numbers of pages match then the two books match". U may need to try out these rules on S and adjust them as necessary. The goal is to develop matching rules that are as accurate as possible.

Once U has been satisfied with the accuracy of the matching rules, the production stage begins. In this stage, U executes the EM workflow that consists of the developed blocking strategy and matching rules on the original tables A and B. To scale, U may need to rewrite the code for blocking and matching to use Hadoop or Spark. $\square$

As described, these two stages are very different in nature: one goes for accuracy and the other goes for scaling (among others). Consequently, they will require very different sets of tools. We now discuss developing tools for these stages.

Development Stage on a Data Analysis Stack: We observe that what users try to do in the development stage is very similar in nature to data analysis tasks, which analyze data to discover insights. Indeed, creating EM rules can be viewed as analyzing (or mining) the data to discover accurate EM rules. Conversely, to create EM rules, users also often have to perform many data analysis tasks, e.g., cleaning, visualizing, finding outliers, IE, etc.

As a result, if we are to develop tools for the development stage in isolation, within a stand-alone monolithic system, as current work has done, we would need to somehow provide a powerful data analysis environment, in order for these tools to be effective. This is clearly very difficult to do.

So instead, we propose that tools for the development stage be developed on top of an open-source data analysis stack, so that they can take full advantage of all the data analysis tools already (or will be) available in that stack. In particular, two major data analysis stacks have recently been developed, based on R and Python (new stacks such as the Berkeley Data Analytics Stack are also being proposed). The Python stack for example includes the general-purpose Python language, numpy and scipy packages for numerical/array computing, pandas for relational data management, scikit-learn for machine learning, among others. More tools are being added all the time, in the form of Python packages. By Oct 2015, there were 490 packages available in the popular Anaconda distribution. There is a vibrant community of contributors to continuously improve this stack.

For Magellan, since our initial target audience is the IT community, where we believe Python is more familiar, we have been developing tools for the development stage on the Python data analysis stack.

Production Stage on a Big Data Stack: In a similar vein, we propose that tools for the production stage, where scaling is a major focus, be developed on top of a Big Data stack. Magellan uses the Python Big Data stack, which consists of many software packages to run MapReduce (e.g., Pydoop, mrjob), Spark (e.g., PySpark), and parallel and distributed computing in general (e.g., pp, dispy).

Together, the data analysis stack and the Big Data stack in Python form PyData, an ecosystem of open-source data science tools. We discuss this ecosystem in detail in Section 7.3, to further motivate our decision for building Magellan into this ecosystem. We also discuss developing the development stage and the production stage for Magellan in detail in the next two chapters.

Expandability Revisited: We are now in a position to discuss how Magellan addresses the expandability requirement outlined in Section 2.4. Current EM systems address expandability in two ways: adding external libraries or moving data among a set of stand-alone systems (e.g., an EM system, an IE system, a visualization system, etc.).

Both methods are problematic. To add an external library we need to write extra code to convert between the data structures used by the system and the library. This is time consuming and may not even be feasible if we do not have access to the system code. Moving data repeatedly among a set of stand-alone systems is very cumbersome as it requires repeatedly writing data to disk, reading data from disk, and converting between the various data formats.

As discussed in Section 2.3, the root of these problems is that most current EM systems are not designed from the scratch to support expandability. In contrast, Magellan assumes that there is already an ecosystem of "systems" (in form of Python packages) that have been designed to expand (i.e., "play well" with one another) and that Magellan will have to be in that ecosystem and to "play well" too.

In sum, the Magellan solution for expandability is to design the system such that it can be easily "plugged" into an existing and expanding data management ecosystem, and that it can combine well with tools in this ecosystem. As an aside, this approach also brings the non-trivial benefit that we are filling in "gaps" in the Python data management ecosystem. This ecosystem is important because more and more users are using its tools to analyze data, but so far good EM tools (and good data integration tools in general) have been missing, seriously hampering user efforts.

Chapter 3

The Development Stage of Magellan

In the previous chapter we have proposed to develop the Magellan system, a new kind of EM system to address the limitations of current EM systems. We have described the high-level architecture of Magellan. Specifically, this architecture consists of two stages: development stage and production stage.

In the *development stage*, the user tries to develop an accurate EM workflow, using data sample. In the *production stage*, the user executes this workflow over the original tables. Magellan must provide *how-to guides* to both stages, which tell the user what to do, step by step, end to end. It should also identify the *pain points* of the guides and provide *semi-automated tools* to help address these pain points. Finally, the tools should be built into *PyData*, i.e., the Python ecosystem of open-source data science tools.

We now elaborate on these two stages.

- This chapter discusses the development stage from a *user's perspective*. That is, what does a user expect to have in this stage, and how is that useful to the user?

- The next chapter (Chapter 4) discusses the production stage, also from a user's perspective.

- Chapter 5 then discusses building both stages from a *developer's perspective*. That is, what challenges do developers face in building these two stages, and what can they do?

In the rest of this chapter we discuss the development stage from a user's perspective. Specifically, we discuss considering a small set of EM scenarios, creating the how-to guides for these scenarios, identifying the pain points, and developing the tools to address the pain point.

1. Load tables A and B into Magellan. Downsample if necessary.
2. Perform blocking on the tables to obtain a set of candidate tuple pairs C.
3. Take a random sample S from C and label pairs in S as matched / non-matched.
4. Create a set of features then convert S into a set of feature vectors H. Split H into a development set I and an evaluation set J.
5. Repeat until out of debugging ideas or out of time:
   (a) Perform cross validation on I to select the best matcher. Let this matcher be X.
   (b) Debug X using I. This may change the matcher X, the data, labels, and the set of features, thus changing I and J.
6. Let Y be the best matcher obtained in Step 5. Train Y on I, then apply to J and report the matching accuracy on J.

Figure 3.1: The top-level steps of the how-to guide for the EM scenario of matching using supervised learning.

Our work here is not complete, in the sense that the guides and tools can, and should be, further extended and refined in future work. We show however that the current guides and tools are already highly promising, in that users can already use them to achieve high matching accuracy on diverse data sets.

3.1 How-to Guides and Tools

Recall from Chapter 2 that in the current Magellan system we target three EM scenarios: matching two tables A and B using (1) supervised learning, (2) rules, and (3) both learning and rules. In what follows we will focus on Scenario 1, briefly discussing Scenarios 2-3 in Section 3.2.7.

We now discuss developing how-to guides and tools to support Scenario 1. Our goal is twofold:

- First, we show that even for relatively simple EM scenarios (e.g., matching using supervised learning), a good guide can already be quite complex. Thus developing how-to guides is a major challenge, but such guides are absolutely critical in order to successfully guide the user through the EM process.

```
c1: down_sample_tables (A, B, B_size, k)
c2: debug_blocker (A, B, C, output_size = 200)
c3: get_features_for_matching (A, B)
c4: select_matcher (matchers, table, exclude_attrs, target_attr, k = 5)
c5: vis_debug_dt (matcher, train, test, exclude_attrs, target_attr)
```

Figure 3.2: Sample commands from Magellan.

- Second, we show that each step of the guide, including those that prior work may have viewed as trivial or engineering (e.g., sampling, labeling), can raise many interesting research challenges. We provide preliminary solutions to several such challenges in this work. But much more remains to be done.

The Current Guide for Learning-Based EM: Figure 3.1 shows the current guide for Scenario 1: matching using supervised learning. The figure lists only the top six steps. While each step may sound like fairly informal advice (e.g., "create a set of features"), the full guide itself (available with Magellan's release) is considerably more complex and actually spells out in detail what to do (e.g., run a Magellan command to automatically create the features). We developed this guide based on observing how real-world users (e.g., at WalmartLabs and Johnson Control) as well as students in several UW-Madison classes handled this scenario.

The guide states that to match two tables A and B, the user should load the tables into Magellan (Step 1), do blocking (Step 2), label a sample of tuple pairs (Step 3), use the sample to iteratively find and debug a learning-based matcher (Steps 4-5), then return this matcher and its estimated matching accuracy (Step 6).

3.2 Steps of the How-To Guide

We now discuss the steps of the above how-to guide, possible tools to support them, and tools that we have actually developed. Our goal is to automate each step as much as possible, and where it is not possible, then to provide detailed guidance to the user. We focus on discussing problems with current solutions, the design alternatives, and opportunities for automation. For ease of exposition, we will assume that tables A and B share the same schema.

3.2.1 Loading and Downsampling Tables

Downsampling Tables: We begin by loading the two tables A and B into memory. If these tables are large (e.g., each having 100K+ tuples), we should sample smaller tables A' and B' from A and B respectively, then do the development stage with these smaller tables. Since this stage is iterative by nature, working with large tables can be very time consuming and frustrating to the user. Random sampling however does not work, because tables A' and B' may end up sharing very few matches, i.e., matching tuples (especially if the number of matches between A and B is small to begin with). Thus we need a tool that samples more intelligently, to ensure a reasonable number of matches between A' and B'.

We have developed such a tool, shown as the Magellan command c_1 in Figure 3.2. This command first randomly selects B_size tuples from table B to be table B'. For each tuple $x \in B'$, it finds a set P of $k/2$ tuples from A that may match x (using the heuristic that if a tuple in A shares many tokens with x, then it is more likely to match x), and a set Q of $k/2$ tuples randomly selected from $A \setminus P$. Table A' will consist of all tuples in such Ps and Qs. The idea is for A' and B' to share some matches yet be as representative of A and B as possible. To find P, the command relies on the heuristic that if two tuples share many tokens, then they are likely to match. Thus, it builds an inverted index I of $(token, tuple\_id)$ over table A, probes I to find all tuples in A that share tokens with x, rank these tuples in decreasing number of shared tokens, then take (up to) the top $k/2$ tuples to be the set P. Note that index I is built only once, at the start of the command. The command then randomly samples $k - |P|$ tuples in $A \setminus P$ to be the set Q.

More Sophisticated Downsampling Solutions: The above command was fast and quite effective in our experiments. However it has a limitation: it may not get all important matching categories into A' and B'. If so, the EM workflow created using A' and B' may not work well on the original tables A and B. For example, consider matching companies. Tables A and B may contain two matching categories: (1) tuples with similar company names and addresses match because they refer to the same company, and (2) tuples with similar company names but different addresses may still match because they refer to different branches of the same company. Using the above command, tables A' and B' may contain many tuple pairs of Case 1, but no or very few pairs of Case 2. To address this problem, we are working on a better "downsampler". Our idea is to use clustering to create groups of matching tuples, then analyze these groups to infer matching categories, then sample from the categories. Major challenges here include how to effectively cluster tuples from the large tables A and B, and how to define and infer matching categories accurately.

3.2.2 Blocking to Create Candidate Tuple Pairs

In the next step, we apply blocking to the two tables A' and B' to remove obviously non-matched tuple pairs. Ideally, this step should be automated (as much as possible). Toward this goal, we distinguish three cases.

(1) We already know which matcher we want to use. Then it may be possible to analyze the matcher to infer a blocker, thereby completely automating the blocking step. For example, when matching two sets of strings (a special case of EM [37]), often we already know the matcher we want to use (e.g., $jaccard(x, y) > 0.8$, i.e., predicting two strings x and y matched if their Jaccard score exceeds 0.8). Prior work [37] has analyzed such matchers to infer efficient blockers that do not remove true matches. Thus, debugging the blocker is also not necessary.

(2) We do not know yet which matcher we want to use, but we have a set T of tuple pairs labeled matched / no-matched. Then it may be possible to partially automate the blocking step. Specifically, the system can use T to learn a blocker and propose it to the user (e.g., training a random forest then extracting the negative rules of the forest as blocker candidates [66]). The user still has to debug the blocker to check that it does not accidentally remove too many true matches.

(3) We do not know yet which matcher we want to use, and we have no labeled data. This is the case considered in this work, since all we have so far are the two tables A' and B'. In this case the user often faces three problems (which have not been addressed by current work): (a) how to select the best blocker, (b) how to debug a given blocker, and (c) how to know when to stop? Among these, the first problem is open to partial automation.

Selecting the Best Blocker: A straightforward solution is to label a set of tuple pairs (e.g., selected using active learning [66]), then use it to automatically propose a blocker, as in Case 2. To propose good blockers, however, this solution may require labeling hundreds of tuple pairs [66], incurring a sizable burden on the user. This solution may also be unnecessarily complex. In practice, a user often can use domain knowledge to quickly propose good blockers, e.g., "matching books must share the same ISBN", in a matter of minutes. Hence, our how-to guide tries to help the user identify these "low-hanging fruits" first.

Specifically, many blocking solutions have been developed, e.g., overlap, attribute equivalence (AE), sorted neighborhood (SNB), hash-based, rule-based, etc. [37]. From our experience, we recommend that the user try successively more complex blockers, and stop when the number of the tuple pairs surviving blocking is already sufficiently small. Specifically, the user can try overlap blocking first (e.g., "matching tuples must share at least k tokens in an attribute x"), then AE (e.g., "matching tuples must share the same value for an attribute y"). These blockers are very fast, and can significantly cut down on the number of candidate tuple pairs. Next, the user can try other well-known blocking methods (e.g., SNB, hash) if appropriate. This means the user can use multiple blockers and combine them in a flexible fashion (e.g., applying AE to the output of overlap blocking). Finally, if the user still wants to reduce the number of candidate tuple pairs further, then he or she can try rule-based blocking. It is difficult to manually come up with good blocking rules. So we will develop a tool to automatically propose rules, as in Case 2, using the technique in [66], which uses active learning to select tuple pairs for the user to label.

Debugging Blockers: Given a blocker L, how do we know if it does not remove too many matches? We have developed a debugger to answer this question [94], shown as command c_2 in Figure 3.2. Suppose applying L to A' and B' produces a set C of tuple pairs $(a \in A', b \in B')$. Then $D = A' \times B' \setminus C$ is the set of all tuple pairs removed by L. The debugger examines D to return a list of k tuple pairs in D that are most likely to match ($k = 200$ is the default). The user U examines this list. If U finds many matches in the list, then that means blocker L has removed too many matches. U would need to modify L to be less "aggressive", then apply the debugger again.

Eventually if U finds no or very few matches in the list, U can assume that L has removed no or very few matches, and thus is good enough.

Developing this debugger raises two challenges. First, how can it judge that a tuple pair is likely to match? Second, how can it search D very fast (given that debugging is interactive by nature)? To address the first challenge, we first select a set of attributes judged to be discriminative, in that if two tuples $(a \in A', b \in B')$ share similar or identical values for most of these attributes, then they are likely to match. Let x be an attribute, we compute

- $unique(x, A')$ to be the number of unique values of x in A' divided by the number of non-empty values of x in A',
- $missing(x, A')$ to be the number of missing values of x in A' divided by the number of tuples in A', and
- $s(x, A') = unique(x, A') + 1 - missing(x, A')$.

The score $s(x, A')$ indicates how discriminative attribute x is in table A'. Intuitively, the higher $unique(x, A')$, the more likely that a value of x can uniquely identify a tuple in A', unless x has a lot of missing values, which is taken into account using $1 - missing(x, A')$. Defining $s(x, B')$ similarly, we can define a discriminativeness score for x across both tables: $s(x) = s(x, A') \cdot s(x, B')$. We then select the top k attributes with the highest $s(x)$ scores (where k is pre-specified), to be used in the debugger.

Let the set of selected attributes be T. For each tuple $a \in A'$, let $t(a)$ be the string resulting from concatenating the values of the selected attributes. Define $t(b)$ similarly for each tuple $b \in B'$. Let $J(t(a), t(b))$ be the Jaccard score between $t(a)$ and $t(b)$, assuming each of these strings have been tokenized into a set of 3-grams. Then the debugger returns the top k tuple pairs (a, b) in $D = A' \times B' \setminus C$ with the highest $J(t(a), t(b))$ scores. Intuitively, the debugger states that these pairs are likely to be matches, so the user should check them. To find these pairs fast, the debugger uses indexes on the tables.

Knowing When to Stop Modifying the Blockers: How do we know when to stop tuning a blocker L? Suppose applying L to A' and B' produces the set of tuple pairs $block(L, A', B')$.

The conventional wisdom is to stop when $block(L, A', B')$ fits into memory or is already small enough so that the matching step can process it efficiently. In practice, however, this often does not work. For example, since we work with A' and B', *samples* from the original tables, monitoring $|block(L, A', B')|$ does not make sense. Instead, we want to monitor $|block(L, A, B)|$. But applying L to the large tables A and B can be very time consuming, making the iterative process of tuning L impractical. Further, in many practical scenarios (e.g., e-commerce), the data to be matched can arrive in batches, over weeks, rendering moot the question of estimating $|block(L, A, B)|$.

As a result, in many practical settings users want blockers that have (1) high pruning power, i.e., maximizing $1 - |block(L, A', B')|/|A' \times B'|$, and (2) high recall, i.e., maximizing the ratio of the number of matches in $block(L, A', B')$ divided by the number of matches in $A' \times B'$. Users can measure the pruning power, but so far they have had no way to estimate recall. This is where our debugger comes in. In our experiments (see Section 3.3) users reported they had used our debugger to find matches that the blocker L had removed, and when they found no or only a few matches, they concluded that L had achieved high recall and stopped tuning the blocker.

3.2.3 Sampling and Labeling Tuple Pairs

Let L be the blocker we have created. Suppose applying L to tables A' and B' produces a set of tuple pairs C. In the next step, user U should take a sample S from C, then label the pairs in S as matched / no-matched, to be used later for training matchers, among others.

At a first glance, this step seems very simple: why not just take a random sample and label it? Unfortunately in practice this is far more complicated. For example, suppose C contains relatively few matches (either because there are few matches between A' and B', or because blocking was too liberal, resulting in a large C). Then a random sample S from C may contain no or few matches. But the user U often does not recognize this until U has labeled most of the pairs in S. This is a waste of U's time and can be quite serious in cases where labeling is time consuming or requires expensive domain experts (e.g., labeling drug pairs when we worked with Marshfield Clinic). Taking another random sample does not solve the problem because it is likely to also contain no or few matches.

To address this problem, our guide builds on [66] to propose that user U sample and label in iterations. Specifically, suppose U wants a sample S of size n. In the first iteration, U takes and labels a random sample S_1 of size k from C, where k is a small number. If there are enough matches in S_1, then U can conclude that the "density" of matches in C is high, and just randomly sample $n-k$ more pairs from C. Otherwise, the "density" of matches in C is low. So U must re-do the blocking step, perhaps by creating new blocking rules that remove more non-matching tuple pairs in C, thereby increasing the density of matches in C. After blocking, U can take another random sample S_2 also of size k from C, then label S_2. If there are enough matches in S_2, then U can conclude that the density of matches in C has become high, and just randomly sample $n - 2k$ more pairs from C, and so on.

3.2.4 Selecting a Matcher

Once user U has labeled a sample S, U uses S to select a good initial learning-based matcher. Today most EM systems supply the user with a set of such matchers, e.g., decision tree, Naive Bayes, SVM, etc., but do not tell the user how to select a good one. Our guide addresses this problem. Specifically, user U first calls the command c_3 in Figure 3.2 to create a set of features $F = \{f_1, \ldots, f_m\}$, where each feature f_i is a function that maps a tuple pair (a, b) into a value. This command creates all possible features between the attributes of tables A' and B', using a set of heuristics. For example, if attribute $name$ is textual, then the command creates feature $name\_3gram\_jac$ that returns the Jaccard score between the 3-gram sets of the two names (of tuples a and b). Next, U converts each tuple pair in the labeled set S into a feature vector (using features in F), thus converting S into a set H of feature vectors. Next, U splits H into a development set I and an evaluation set J.

Let M be the set of all learning-based matchers supplied by the EM system. Next, U uses command c_4 in Figure 3.2 to perform cross validation on I for all matchers in M, then examines the results to select a good matcher. Command c_4 highlights the matcher with the highest accuracy. However, if a matcher achieves just slightly lower accuracy (than the best one) but produces results that are easier to explain and debug (e.g., a decision tree), then c_4 highlights that matcher as well, for the user's consideration. Thus, the entire process of selecting a matcher can be automated (if the user does not want to be involved), and in fact Magellan does provide a single command to execute the entire process.

3.2.5 Debugging a Matcher

Let the selected matcher be X. In the next step user U debugs X to improve its accuracy. Such debugging is critical in practice, yet has received very little attention in the research community. Our guide suggests that user U debug in three steps: (1) identify and understand the matching mistakes made by X, (2) categorize these mistakes, and (3) take actions to fix common categories of mistakes.

Identifying and Understanding Matching Mistakes: U should split the development set I into two sets P and Q, train X on P then apply it to Q. Since U knows the labels of the pairs in Q, he or she knows the matching mistakes made by X in Q. These are *false positives* (non-matching pairs predicted matching) and *false negatives* (matching pairs predicted not). Addressing them helps improve precision and recall, respectively.

Next U should try to understand why X makes each mistake. For example, let $(a, b) \in Q$ be a pair labeled "matched" for which X has predicted "not matched". To understand why, U can start by using a debugger that explains how X comes to that prediction. For example, if X is a decision tree then the debugger (invoked using command c_5 in Figure 3.2) can show the path from the root of the tree to the leaf that (a, b) has traversed. Examining this path, as well as the pair (a, b) and its label, can reveal where things go wrong. In general things can go wrong in four ways:

- The data can be dirty, e.g., the price value is incorrect.
- The label can be wrong, e.g., (a, b) should have been labeled "not matched".
- The feature set is problematic. A feature is misleading, or a new feature is desired, e.g., we need a new feature that extracts and compares the publishers.

- The learning algorithm employed by X is problematic, e.g., a parameter such as "maximal depth to be searched" is set to be too small.

Currently Magellan has debuggers for a set of learning-based matchers, e.g., decision tree, random forest. We are working on improving these debuggers and developing debuggers for more learning algorithms.

Categorizing Matching Mistakes: After U has examined all or a large number of matching mistakes, he or she can categorize them, based on problems with data, label, feature, and the learning algorithm. Examining all or most mistakes is very time consuming. Thus a consistent feedback we have received from real-world users is that they would love a tool that can automatically examine and give a preliminary categorization of the types of the matching mistakes. As far as we can tell, no such tool exists today.

Handling Common Categories of Mistakes: Next U should try to fix common categories of mistakes by modifying the data, labels, set of features, and the learning algorithm. This part often involves data cleaning and extraction (IE), e.g., normalizing all values of attribute "affiliation", or extracting publishers from attribute "desc" then creating a new feature comparing the publishers. This part is often also very time consuming. Real-world users have consistently indicated needing support in at least two areas. First, they want to know exactly what kinds of data cleaning and IE operations they need to do to fix the mistakes. Naturally they want to do as minimally as possible. Second, re-executing the entire EM process after each tiny change to see if it "fixes" the mistakes is very time consuming. Hence, users want an "what-if" tool that can quickly show the effect of a hypothetical change.

Proxy Debugging: Suppose we need to debug a matcher X but there is no debugger for X, or the debugger exists but is not very informative. In this case X is effectively a "blackbox". To address this problem, in Magellan we have introduced a novel debugging method. In particular, we propose to train another matcher X' for which there is a debugger, then use that debugger to debug X', instead of X. This "proxy debugging" process cannot fix problems with the learning algorithm of X, but it can reveal problems with the data, labels, features, and fixing them can

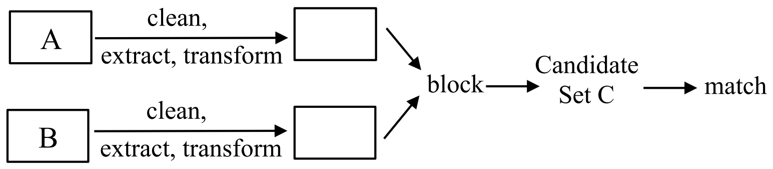

Figure 3.3: The EM workflow for the learning-based matching scenario.

potentially improve the accuracy of X itself. Section 3.3 shows cases of proxy debugging working quite well in practice.

Selecting a Matcher Again: So far we have discussed selecting a good initial learning-based matcher X, then debugging X using the development set I. To debug, user U splits I into training set P and testing set Q, then identifies and fixes mistakes in Q. Note that this splitting of I into P and Q can be done multiple times. Subsequently, since the data, labels, and features may have changed, U would want to do cross validation again to select a new "best matcher", and so on (see Step 5 in Figure 3.1).

3.2.6 The Resulting EM Workflow

After executing the above steps, user U has in effect created an EM workflow, as shown in Figure 3.3. Since this workflow will be used in the production stage, it takes as input the two original tables A and B. Next, it performs a set of data cleaning, IE, and transformation operations on these tables. These operations are derived from the debugging step discussed in Section 3.2.5. Next, the workflow applies the blockers created in Section 3.2.2 to obtain a set of candidate tuple pairs C. Finally, the workflow applies the learning-based matcher created in Section 3.2.5 to the pairs in C. Note that the steps of sampling and labeling a sample S do not appear in this workflow, because we need them only in the development stage, in order to create, debug, and train matchers. Once we have found a good learning-based matcher (and have trained it using S), we do not have to execute those steps again in the production stage.

3.2.7 How-to Guides for Scenarios with Rules

Recall that Magellan currently targets three EM scenarios. So far we have discussed a how-to guide and tools for Scenario 1: matching using supervised learning. We now briefly discuss Scenarios 2 and 3.

Scenario 2 uses only rules to match. This is desirable in practice for various reasons (e.g., when matching medicine it is often important that we can explain the matching decision). For this scenario, we have developed guides and tools to help users (a) create matching rules manually, (b) create rules using a set of labeled tuple pairs, or (c) create rules using active learning. Scenario 3 uses both supervised learning and rules. Users often want this when using neither learning nor rules alone gives them the desired accuracy. For this scenario, we have also developed a guide and tools to help users. Our guide suggests that users do learning-based EM first, as described earlier for Scenario 1, then add matching rules “on top” of the learning-based matcher, to improve matching accuracy.

Team	Domain	Size of Table A	Size of Table B	Cand. Set Size	Initial Learning-Based Matcher (A)			Final Learning-Based Matcher (B)			Num. of Iterations (C)	Final Learning + Rules Matcher (D)			Num. of Iterations (E)	Diff. in F_1 between (D) and (A) in %
					P	R	F1	P	R	F1		P	R	F1		
1	Vehicles	4786	9003	8009	71.2	71.2	71.2	91.43	94.12	92.75	4	100	100	100	2	30.27
2	Movies	7391	6408	78079	99.28	95.13	97.04	98.21	100	99.1	2	100	100	100	1	2.12
3	Movies	3000	3000	1000000	98.9	99.44	99.5	98.63	98.63	98.63	1	98.63	98.63	98.63	0	-0.87
4	Movies	3000	3000	36000	68.2	69.16	68.6	98	100	98.99	3	98	100	98.99	1	44.3
5	Movies	6225	6392	54028	100	95.23	97.44	100	100	100	3	100	100	100	1	2.63
6	Restaurants	6960	3897	10630	100	37.5	54.55	100	88.89	94.12	3	100	88.89	94.12	1	72.54
7	Electronic Products	4559	5001	823832	73	51	59	73.3	64.71	68.75	2	100	64.71	78.57	1	33.17
8	Music	6907	55923	58692	92	79.31	85.19	90.48	82.61	86.36	2	100	92.16	95.92	2	1.37
9	Restaurants	9947	28787	400000	100	78.5	87.6	94.44	97.14	95.77	4	94.44	97.14	95.77	0	9.33
10	Cosmetic	11026	6445	36026	56	56	56	96.67	87.88	92.06	3	96.43	87.1	91.53	4	64.39
11	E-Books	6482	14110	13652	96.67	96.67	96.67	100	95.65	97.78	4	100	98.33	99.13	1	1.15
12	Beer	4346	3000	4334961	84.5	59.6	65.7	100	60.87	75.68	4	91.3	91.3	91.3	4	15.19
13	Books	3506	3508	2016	93.46	100	96.67	91.6	100	95.65	2	91.6	100	95.65	0	-1.06
14	Books	3967	3701	4029	74.17	82.2	82.5	100	84.85	91.8	3	100	84.85	91.8	5	11.27
15	Anime	4000	4000	138344	95.9	88.9	92.2	100	100	100	2	100	100	100	1	8.46
16	Books	3021	3098	931	74.2	100	85.2	96.34	84.95	90.29	2	94.51	92.47	93.48	1	5.97
17	Movies	3556	6913	504	94.2	99.33	96.6	95.04	94.26	94.65	2	95.04	94.26	94.65	1	-2.02
18	Books	8600	9000	492	91.6	100	84.8	94.8	100	90.2	3	100	92.31	96	1	6.37
19	Restaurants	11840	5223	5278	98.6	93.8	96.1	95.6	94.02	95.57	2	100	94.12	96.97	1	-0.55
20	Books	3000	3000	257183	94.24	72.88	81.71	90.91	83.33	86.96	2	92.31	100	96	1	6.43
21	Literature	3885	3123	1590633	84.4	86.9	85.5	100	95.65	97.83	3	100	95.65	97.83	0	14.42
22	Restaurants	3014	5883	78190	100	93.59	96.55	100	100	100	5	100	100	100	0	3.57
23	E-Books	6501	14110	18381	94.6	92.5	93.4	94.6	97.22	95.89	2	100	100	100	1	2.67
24	Baby Products	10000	5000	11000	78.6	44.8	57.7	96.43	72.97	83.08	5	100	72.97	84.37	2	43.99

Table 3.1: Large-scale experiments with Magellan on Web data.

3.3 Empirical Evaluation

We now show that the current guide and tools are already highly promising, in that users can already use them to achieve high matching accuracy on diverse data sets. Specifically, in what follows we report on experiments with graduate students at UW-Madison. Chapter 6 describes our experience applying Magellan "in the wild", to real-world EM scenarios in domain science projects and at companies. That experience further demonstrates the above promise (and highlights challenges for future work).

As of March 2018, Magellan [13] consists of 6 Python packages, 37K lines of code, and 104 commands. It has been developed over 3 years by 13 developers. So far 400+ students (including 90+ undergraduates) have used Magellan in 5 data science classes at UW-Madison. These students can be considered the equivalents of power users at organizations. They know Python but are not experts in EM.

In each class, we asked the students to form team of 2-3 students, then asked each team to find two data-rich Web sites, extract and convert data from them into two relational tables, then apply Magellan to match tuples across the tables [45]. We typically asked each team to do the EM scenario of supervised learning followed by rules, and aim for precision of at least 90% with recall as high as possible (a very common scenario in practice).

We now describe in detail our experience with the Fall 2015 class, which consisted of 44 students divided into 24 teams. The first four columns of Table 3.1 show the teams, domains, and the sizes of the two tables, respectively. Note that two teams may cover the same domain, e.g., "Movies", but extract from different sites. Overall, there are 12 domains, and the tables have 7,313 tuples on average, with 5-17 attributes. We asked each team to do the EM scenario of supervised learning followed by rules, and aim for precision of at least 90% with recall as high as possible. This is a very common scenario in practice.

The Baseline Performance: The columns under "Initial Learning-Based Matcher (A)" show the matching accuracies achieved by the best learning-based matcher (after cross validation, see Section 3.2.4): $P = 56 - 100\%, R = 37.5 - 100\%, F_1 = 56 - 99.5\%$. These results show that many of these tables are not easy to match, as the best learning-based matcher selected after cross validation does not achieve high accuracy. In what follows we will see how Magellan was able to significantly improve these accuracies.

Using the How-to Guide: The columns under "Final Learning+Rule Matcher (D)" show the final matching accuracies that the teams obtained: $P = 91.3 - 100\%, R = 64.7 - 100\%, F_1 = 78.6 - 100\%$. All 24 teams achieved precision exceeding 90%, and 20 teams also achieved recall exceeding 90%. (Four teams had recall below 90% because their data were quite dirty, with many missing values.) All teams reported being able to follow the how-to guide. Together with qualitative feedback from the teams, this suggests that users can follow Magellan's how-to guide to achieve high matching accuracy on diverse data sets. We elaborate on these results below, broken down by blocking and matching.

Blocking and Debugging Blockers: All teams used 1-5 blockers (e.g., attribute equivalence, overlap, rule-based), for an average of 3. On average 3 different types of blockers were used per team. This suggests that it is relatively easy to create a blocking pipeline with diverse blocker types. All teams debugged their blockers, in 1-10 iterations, for an average of 5. 18 out of 24 teams used our debugger (see Section 3.2.2), and reported that it helped in four ways.

- **Cleaning data:** By examining tuple pairs (returned by the debugger) that are matches accidentally removed by blocking, 12 teams discovered data that should be cleaned. For example, one team removed the edition information from book titles, and another team normalized the date formats in the input tables.

- **Finding the correct blocker types and attributes:** 12 teams were able to use the debugger for these purposes. For example, one team found that using attribute equivalence (AE) blocker over "phone" removed many matches, because the phone numbers were not updated. So they decided to use "zipcode" instead. Another team started with AE over "name" then realized that the blocker did not work well because many names were misspelled. So they decided to use a rule-based blocker instead.

- **Tuning blocker parameters:** 18 teams used the debugger for this purpose, e.g., to change the overlap size for "address" in an overlap blocker, or to use a different threshold for a Jaccard measure in a rule-based blocker.

- **Knowing when to stop:**12 teams explicitly mentioned in their reports that when the debugger returned no or very few matches, they concluded that the blocking pipeline had done well, and stopped tuning this pipeline.

Teams reported spending 4-32 hours on blocking (including reading documentations). Overall, 21 out of 24 teams were able to prune away more than 95% of $|A \times B|$, with an average reduction of 97.3%, suggesting that they were able to construct blocking with high pruning rate. Feedback-wise, teams reported liking (a) the ability to create rich and flexible blocking sequences with different types of blockers, (b) the diverse range of blocker types provided by Magellan, and (c) the debugger. They complained that certain types of blockers (e.g., rule-based ones) were still slow (an issue that we are currently addressing).

Matching and Debugging Matchers: Recall from Section 3.2.5 that after cross validation on labeled data to select the best learning-based matcher X, user U iteratively debugged X to improve its accuracy. Teams performed 1-5 debugging iterations, for an average of 3 (see Column "Num of Iterations (C)" in Table 3.1). The actions they took were:

- **Feature selection:** 21 teams added and deleted features, e.g., adding more phone related features, removing style related features.

- **Data cleaning:** 12 teams cleaned data based on the debugging result, e.g., normalizing colors using a dictionary, detecting that the tables have different date formats. 16 teams found and fixed incorrect labels during debugging.

- **Parameter tuning:** 3 teams tuned the parameters of the learning algorithm, e.g., modifying the maximum depth of decision tree based on debugging results.

These debugging actions helped improve accuracies significantly, from 56-100% to 73.3-100% precision, and 37.5-100% to 61-100% recall (compare columns under "A" with those under "B" in Table 3.1).

Adding rules further improves accuracy. 19 teams added 1-5 rules, found in 1-5 iterations (see column "E"). This improved precision from 73.3-100% to 91.3-100% and recall from 61-100% to 64.7-100% (compare columns under "D" with those under "B"). Overall, Magellan improved the baseline accuracy in columns "A" significantly, by as much as 72.5% F_1, for an average of 18.8% F_1. For 3 teams, however, accuracy dropped by 0.87-2.02% F_1. This is because the baseline F_1s already exceeded 94%, and when teams tried to add rules to increase F_1 further, they overfit the development set.

Teams reported spending 5-50 hours, for an average of 12 hours (including reading documentation and labeling samples) on matching. They reported liking debugger support, ease of creating custom features for matchers, and support for rules to improve learning-based matching. They would like to have more debugger support, including better ordering and visualization of matching mistakes.

Summary: Our experience with the development stage of Magellan suggest that users can successfully follow the how-to guide to achieve high EM accuracy on diverse data sets. In fact, we consider the how-to guide to be the single most important component of the system. Without it, users are lost: they do not even know where to start, when to use what tools, and how.

Our experience further suggests that the various tools developed for the development stage (e.g., debuggers) can be highly effective in helping the users. It also clearly shows that practical EM requires a wide range of capabilities, e.g., cleaning, extraction, visualization, underscoring the importance of placing Magellan in an ecosystem that provides such capabilities. (In fact, Magellan currently uses 12 packages in the Python ecosystem to provide such capabilities.)

Chapter 4

The Production Stage of Magellan

In the previous chapter we have considered the development stage. Specifically, we considered developing how-to guides and tools to help the user experiment and find an accurate EM workflow, using data samples. In this chapter we consider the production stage, in which the user executes this workflow on the entirety of data, i.e., on the two original tables to be matched.

There are numerous challenges in the production stage, such as scaling, crash recovery, logging, etc. We will focus on scaling. We first motivate our scaling considerations and define our problem settings. In particular, we argue for the need to develop a how-to guide for users on how to handle scaling challenges in the production stage. Next, we describe such preliminary guides for several scaling scenarios. We then identify pain points in the guides and develop tools for the pain points. Specifically, we develop tools to tune the parameters of the implementation versions of several EM commands. We present experiments showing that even preliminary tuning tools can already be very useful to the user.

Overall, our work in this chapter, even though still preliminary, shows the promise of following the "how-to guide / pain points / tools" template that we have successfully used for the development stage.

4.1 Motivations and Our Problem Setting

Recall that in the production stage our goal is to execute a workflow W (which was discovered in the development stage) on the entirety of data, e.g., on the two original tables. This raises many challenges, including scaling, crash recovery, logging, etc. In this work we will focus on scaling, i.e., executing the workflow W fast.

Focusing on Scaling Each Individual Python Command: A common way to scale is to optimize the run time of the entire workflow W, e.g., we can define a set of operators, compile W into a DAG of these operators, optimize and then execute the DAG. This is reminiscent of the scaling approach of RDBMSs.

The above approach however assumes that the user is interested in executing the entire workflow W in one shot. In many cases, we observe that the user may want to execute only fragments of W (perhaps one fragment after another). There are many reasons for this. The user may want to execute a fragment, inspect the output, debug, and repeat if necessary, before executing the next fragment. Different user groups may be in charge of executing different parts of the workflow. The user may want to interleave the development stage with the production stage. For example, after discovering a good blocking method in the development stage, the user may want to execute that blocking method on the entirety of data (thus being in the production stage), before continuing with the development stage by sampling from the blocking output, and so on.

As a result, we will not focus on scaling the entire workflow W in this thesis. Rather, we focus on scaling its fragments. In particular, the workflow W often consists of multiple Python commands (each performing an action, such as blocking, matching, etc.). So as a first step, we will focus on scaling each individual Python command (that Magellan provides for the production stage).

Focusing on a Single Machine Setting: Each Python command in the production stage can be executed on a single machine or a cluster of machines, depending on what computing environments the user has and what implementations of the command are available to the user. As a first step, in this work we will focus on scaling such Python commands on a single machine setting. Within this setting, we will consider both the scenario of using a single core and using multiple cores.

Limitations of Current Scaling Approaches: Now that we have scoped our settings, we consider how to scale a single Python command. A straightforward baseline is to just develop an implementation that is highly optimized for time (in a way similar to developing an highly optimized implementation for a join operation). This however does not work because a different implementation may work best for different datasets. For example, a simple Python implementation targeting a single core (on a single machine) may be fastest for small datasets, but will not scale to large datasets. This is again reminiscent of the fact that different join implementations exist (e.g., hash join, nested loop join, index join, etc.), and each implementation is best in a different context.

As a result, we start by asking what implementations of a single Python command are commonly available (for the single-machine setting). From now on, we also refer to such implementations as command versions, where there is no ambiguity. We observe that the following versions are commonly available in practice (see Chapter 5 fore more discussions on these versions):

- **Single-core version:** This version executes using a single core in the machine. The single-core versions are implemented using Python. This is typically the first version that developers implement for a Python command, as it is the simplest version to implement. Later if this version (in Python) appears not fast enough, developers may rewrite certain parts of it in Cython.

- **Multi-core version developed using automatic parallel frameworks:** This version uses multiple cores and is implemented using the primitives from an automatic parallel framework (e.g., Dask [113]). This version often includes multiple scaling parameters such as the number of partitions of the input tables, choice of the scheduler, etc. If the single-core version is not fast enough (e.g., based on feedback from many users), then developers are likely to develop this version, because developing it by using a framework such as Dask to modify the single-core version code is relatively painless.

- **Custom multi-core version:** This version uses multiple cores and is implemented using Python multiprocessing libraries [7, 14]. It often requires the developers to have a deep knowledge about how the command works and then use that knowledge to provide customized optimizations. Like the previous version, this version also often include multiple parameters that the user needs to set, such as the number of partitions of the original tables. Typically this version is developed when it appears that both the single-core version and the automatic multi-core version are not fast enough for many users.

In practice, the above versions often are developed over time, and either all three or a subset of them are available. This raises a major challenge. In the production stage, when the user has to execute a Python command, suppose this command has all three versions available, which one should the user chose?

The above question is reminiscent of how to select a join implementation (e.g., hash-, index-, or nested loop join) to execute a join operation in RDBMSs. The common solution there is to use an optimizer. Roughly speaking, this optimizer estimates the runtime of each join implementation and then selects the fastest one.

In our context, however, it is not clear how to develop such an optimizer. The main reason is that it is very difficult to estimate the runtime of each version. For example, a version such as "multi-core automatic" often has many parameters that can be tuned (e.g., how many partitions for Table A and how many partitions for Table B, assuming that we want to match A and B). Depending on how we tune these parameters, the runtime can vary drastically. We cannot however assume tuning will be a part of the optimization process, because tuning can take a significant amount of time, e.g., minutes or tens of minutes. So the total optimization time can end up being higher than the time it takes to execute a command version.

In short, today it is still unclear how to optimize in our contexts such that the optimization time is negligible. For example, any optimization method appears to require tuning, and tuning can still take a very long time.

Another problem is that in practice, users often do not always need to run a command in the least amount of time. Suppose the single-core command version takes 20 minutes, but a user does not really need the result until a few days from now, then this command version is perfectly fine for him or her. So the goal of optimizing to select the fastest command version is a bit of an "overkill".

Our Proposed Solution: To address the above issues, we propose to develop a how-to guide for users on which command version to use in which context, and what to do next if that version is too slow. We then analyze the how-to guide to identify pain points, then develop tools for the pain points. This is the same solution approach we have used for the development stage, as discussed in Chapter 3.

It is important to note that our solution here will be somewhat preliminary, because we focus on the development stage in this dissertation. However, it will provide evidence to suggest that the same "how-to guide / pain points / tools" approach is also highly promising for the production stage.

4.2 Developing a How-to Guide for Users

We observe that there are many different possible scenarios based on the available versions of a command. In this dissertation, we target three scenarios: (1) only a single-core version is available, (2) a single-core version and a multi-core version using Dask are available, and (3) a single-core version, a multi-core version using Dask, and a custom multi-core version are available. Developing how-to guides for these scenarios is important as they commonly occur in practice. In what follows we describe an initial guide for these scenarios.

Guide for Scenario 1: Since only a single-core version available in this scenario, the user should execute it. If it does not appear fast enough (for example, the user may abort the command in the middle because it already takes too long, or the command finishes but the user wants to run it again, in less time), then the user should check if there are parameters to tune, tune those, and execute again. If even this is not fast enough (or if there is no parameter to tune), then the user should try a more powerful machine with more RAM. (More RAM ensures that any memory-intensive operations will likely have enough memory to operate, especially when other users also run jobs on the same machine.)

Guide for Scenario 2: In this scenario a single-core version and a multi-core version using Dask are available. If the tables are small (e.g., under 300 tuples each), the user should execute the single-core version. Running a multi-core version here would incur an overhead of initial setup cost and serialization/deserialization time while moving the data between the cores, and thus may be slower than the single-core version.

If the tables are not small, or the single-core version is not fast enough, the user should execute the multi-core version using Dask (using the default values for the scaling parameters). If this version is not fast enough, the user should tune the parameters and execute the version again. If this is still not fast enough, the user should try using a more powerful machine with more cores and RAM.

Guide for Scenario 3: In this scenario a single-core version, multi-core version using Dask, and multi-core custom version are available. If the tables are small (e.g., under 300 tuples each), the user should execute the single-core version.

If the tables are not small, or the single-core version is not fast enough, the user should execute the multi-core custom version (using the default values for the scaling parameters). This version is executed before the multi-core Dask version because the custom multi-core version is expected to include custom optimizations that make the execution faster. If this is not fast enough, the user should tune the scaling parameters and execute the command again.

If this is still not fast enough, the user should execute the multi-core Dask version. This version is executed because it may be the case that the custom multi-core version is not suited for the current input tables. If this is not fast enough, the user should tune the parameters and execute the command again. If this is still not fast enough, the user should try using a more powerful workstation with more cores and RAM. In the new workstation, the user should follow the same guide and start by executing the custom multi-core version using the default values for the scaling parameters.

4.3 Identifying Pain Points and Developing Tools

We now consider the pain points of the above how-to guide and developing tools to address the pain points. A clear pain point is tuning the parameters of a command version. If a user uses the

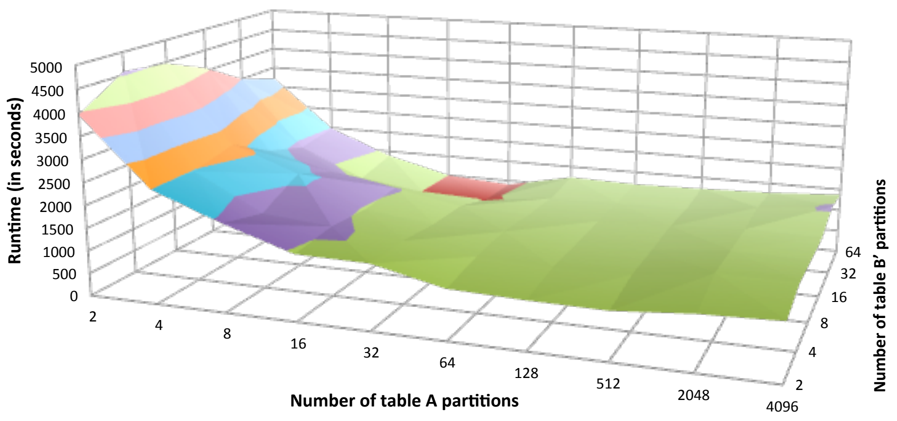

Figure 4.1: Runtime vs the number of table partitions for the downsampling command.

default values or does not set the right values for the parameters, the runtime performance may be poor and it may not meet his or her goals. For example, Figure 4.1 shows the impact of the number of partitions of input tables on the runtimes of the downsampling command. The figure shows that the runtimes vary a lot and bad parameter settings will result in significantly increased runtimes. It is hard however for a user to try different possible combinations of parameter values and choose the best one. To address this, we focus on developing tools to tune parameters of command versions.

There are two options to develop tools to tune parameters: (1) develop a generic tool that can be used for *all* commands, and (2) develop command-specific tools, i.e., for each time-intensive command we will develop a tool just for tuning that command.

In this dissertation, we focus on developing command-specific tools. First, it is not clear yet how to develop a generic tool that works well for all commands. By developing command-specific tools, we hope that we will learn lessons that can be applied to developing generic tuning tools in the future. Second, the goal of this dissertation is not to develop the best tuning tools. Rather, for this part of the dissertation, the goal is to demonstrate that even basic tuning tools can already be very useful for the users.

More concretely, we will focus on developing tools to tune the parameters of the multi-core Dask versions for two command: downsampler and overlap blocker.

There has been a lot of work on parameter tuning [87, 88, 72, 122, 90]. Our goal is to develop a preliminary tool to tune the parameters of commands. (Again, our goal is to demonstrate that even basic tuning tools can already be very useful for the users.) So we will focus on developing a tool using the "staged tuning" approach and leave other approaches for future work. Staged tuning is a sequential, greedy approach. The key idea of this approach is to tune one parameter at a time. For example, if there are three parameters to tune, we tune the first parameter with the other two parameters set to some default values. Next, we tune the second parameter with the first parameter set to the tuned value and the third parameter set to a default value. Finally, we tune the third parameter with the first two parameters set to the tuned values.

4.3.1 Developing a Tuning Tool for the Downsampling Command

To develop a tool to tune the parameters of the downsampling command, we will first describe the procedure used for the multi-core Dask version of this command. Then we will use this procedure to develop a tool to tune the parameters.

4.3.1.1 The Dask Implementation Version

We will first describe the procedure used for the single-core version of the command, followed by the changes made to create the multi-core Dask version. Conceptually, given two large tables A and B, the goal of this command is to create two smaller tables, A' and B'. In the single-core version, the command first randomly selects B_size tuples from B to make table B'. Next, it concatenates the string attributes in table A, preprocesses (e.g., removes punctuations, special characters, etc.) and tokenizes these concatenated strings, and uses these tokens to build an inverted index. Then for each $x \in B'$, it concatenates the string attributes, preprocesses and tokenizes the concatenated string, and finds a set P of $k/2$ tuples that may match with x using the inverted index, and a set Q of $k/2$ tuples randomly selected from $A \setminus P$. Table A' will thus consist of all tuples in such Ps and Qs.

In the multi-core Dask version of this command, the same sequence of steps is followed as in the single-core version, except that the tables A and B' are split into multiple partitions of equal size. These partitions are used to build and probe the inverted index in parallel. Specifically, the multi-core Dask version of the command first randomly selects B_size tuples from B to make table B'. Next, it splits table A into n_A partitions of equal size. Then for each partition, it concatenates the string attributes, preprocesses and tokenizes these concatenated strings in parallel. Next, it takes a union of these tokens and builds an inverted index. Then it splits table B' into n_B partitions of equal size. Next, for each partition and each tuple x in this partition, it concatenates the string attributes, preprocesses and tokenizes this concatenated string, and finds a set P of $k/2$ tuples that may match with x using the inverted index, and a set Q of $k/2$ tuples randomly selected from $A \setminus P$, in parallel. Table A' will then consist of all tuples in such Ps and Qs.

Based on the procedure for the multi-core Dask version, there are three parameters to tune: (1) whether input tables A and B must be swapped, (2) n_A, the number of partitions into which to split table A, and (3) n_B, the number of partitions into which to split table B'. These parameters have a significant impact on the runtime performance. For example, the number of partitions in tables A and B' determine the data size that gets processed in parallel. If there is a computation skew in these partitions (i.e., if some partitions require more time to complete), then the number of partitions will have a significant impact on the overall runtime.

4.3.1.2 Tuning the Dask Version

We will now describe the algorithm used to tune the parameters of the downsampling command. Conceptually, the algorithm takes two tables A and B' as input and performs the following steps. First, it takes a sample of tables A and B'. Next, it decides the order in which the parameters must be tuned. Next, it decides the range of values for each of these parameters. Then it selects the best configuration for these parameters using staged tuning. Finally, it returns the best configuration. Though the procedure is simple, it includes at least three challenges. First, how should we sample the input tables? Second, in what order must the parameters be tuned? Third, what should be the range of values for each parameter? We will now discuss each of these challenges and proposed solutions.

Sampling: In the Dask procedure, we observed that the input table B is sampled to get B'. Our goal is to sample the tables A and B', as these tables are used in the subsequent steps. A naive solution would be to sample them randomly. However, this solution does not work, as the sampled tables may not be representative of the input tables with respect to the processing of tuples. To address this, we use a stratified sampling approach to sample the input tables. Specifically, we sample the table A by first concatenating the string attributes in each tuple, then taking a stratified sample using the lengths of these concatenated strings, and finally ordering these tuples as in the original table A. The intuition here is that while building the inverted index using table A, most of the compute time is spent in preprocessing and tokenizing the concatenated strings. The runtimes of these two operations depend on the lengths of the concatenated strings. So taking a stratified sample using the lengths of these concatenated strings will include a representative set of tuples from table A. We reorder these sampled tuples as in the input table A, to capture the computation skew among the partitions in table A. Specifically, in the multi-core Dask version, the table is partitioned in the following manner. If there are 1M tuples in table A and if n_A is 10, then the first partition will include the first 100K tuples from table A, the second partition will include the next 100K tuples, and so on. These partitions can have different computation times, and it is important to capture these variations in the sampled tables. So we reorder the sampled tuples as in the original table in order to mimic the computation skew (as much as possible) of the original table. Similarly, we sample table B' by taking a stratified sample using the number of tuples in A by which each tuple in B' must be probed. Currently, the size of the sampled tables is set to 10 percent of the input table sizes. For example, if table A includes 1000 tuples, then the sampled table will include 100 tuples.

Ordering of Parameters: There are three parameters to tune in the multi-core Dask version of the downsampling command: (1) whether input tables A and B must be swapped, (2) n_A, the number of partitions into which to split table A, and (3) n_B, the number of partitions into which to split table B'. It is obvious from the procedure for the multi-core Dask version that the decision, whether or not swap the input tables has to be made first, as it determines what table should be used to build the inverted index and what table should be used to probe. So this parameter is tuned first. Between n_A and n_B, the parameter n_B is tuned first, as the probing time is typically longer than the index-building time. Finally, n_A is tuned.

Choice of Parameter Values: The parameter for swapping the input tables can take two values: True or False. If the tables have to be swapped, then we return True; otherwise, we return False. The number of partitions in table A (i.e., n_A) can ideally take any value in the range $[1, N]$, where N is the number of tuples in A. If n_A is set to N, then each partition will include a single tuple. However, N can be large (in the order of millions), so selecting the best value in this large range is hard. To address this, currently, we have set N to be 128. Even in this range, it will be very time-consuming to consider every value between 1 and N. So currently, we consider only a subset of values in this range. Specifically, we start with the number of cores, n_c, in the machine, and then double this value until it reaches 128. The intuition here is that increased runtimes are often caused by the computation skew in the partitions of table A. In such cases, ideally, we must find the partition that takes longer to finish processing and then split that partition further. However, this requires modifying the existing implementation to collect more diagnostic information. So as a workaround, we split all the partitions, essentially doubling the number of partitions. Thus, the value range for the number partitions in table A (i.e. n_A) is set to [n_c, 128]. Similarly, the value range for the number of partitions in table B' (i.e. n_B) is set to [n_c, 128].

Selecting the Best Configuration: To select the best configuration, first we decide if the input tables need to be swapped. To do this, we first randomly sample B_size tuples from B to get B'. Next, we sample the tables A and B', using stratified sampling, to get the sampled tables S_A and S_B. Then we execute the multi-core Dask version using the sampled tables. While executing the multi-core Dask version, we set the parameters, B_size equal to the size of S_B, so that S_B is not sampled in the command. Next, we repeat the above procedure by swapping the input tables A and B. While we execute the multi-core Dask version, the number of partitions, n_A, n_B, are set to the number of cores in the machine. We choose whether or not to swap the input tables based on the runtimes of the multi-core Dask version using the sampled tables. If the runtime is lower when the sampled tables are swapped, then we choose to swap the input tables; otherwise, we retain the same order given in the input. Based on this decision, we use the appropriate sampled tables, S_A and S_B, for tuning the other parameters.

Next, we select the number of partitions for table B'. Specifically, we iterate through a subset of values in the range [n_c, 128], execute the multi-core Dask version for each of the different values (starting with n_c and doubling each time till it reaches 128), and select the maximum value before which the runtime starts increasing. While running this, we set the number of partitions for table A to the number of cores in the machine. Next, we select the number of partitions for table A. We follow the same procedure that we followed while selecting the number of partitions for table B'. While running this, the number of partitions for B' is assigned to the value found in the previous step. Finally, we return the selected configuration to the user as an ordered list (X, Y, Z), where X tells the user whether the tables need to be swapped, Y is the number of partitions for table A, and Z is the number of partitions for table B'.

4.3.2 Developing a Tuning Tool for the Overlap Blocker Command

To develop a tool to tune the parameters of the overlap blocker command, we will first describe the procedure used for the multi-core Dask version of this command. Then we will use this procedure to develop a tool to tune the parameters.

4.3.2.1 The Dask Implementation Version

We will first describe the procedure for the single-core version of the command, followed by the changes made to create the multi-core Dask version. Conceptually, given two tables A and B, a blocking attribute $block\_attr$, and an overlap threshold K, the goal of the overlap blocker is to return the tuple pairs from these input tables that share at least K tokens in $block\_attr$. We assume that both tables A and B include the $block\_attr$. In the single-core version, the overlap blocker first preprocesses and tokenizes the strings in the $block\_attr$ column in table A. Then it uses these tokens to build an inverted index. Next, for each string in the $block\_attr$ column in table B, it preprocesses and tokenizes the string, finds a set of tuples from A that share at least K tokens (using the inverted index), and creates a set P of such satisfying tuple pairs. Finally, it creates a new table by taking a union of all such Ps and returns this table to the user.

In the multi-core Dask version of the overlap blocker, the same sequence of steps is followed as in the single-core version, except that the input tables A and B are split into multiple partitions of equal size and these partitions are used to build and probe the inverted index in parallel. Specifically, this implementation of the blocker first splits table A into n_A partitions of equal size. Then for each partition, it preprocesses and tokenizes the string in the $block\_attr$ column, in parallel. Next, it takes a union of all these tokens and builds an inverted index. Then it splits B into n_B partitions of equal size. Next, for each partition and each string in the $block\_attr$ column in this partition, it preprocesses and tokenizes the string, finds tuples in table A that share at least K tokens (using the inverted index), and then creates a set P of such satisfying tuple pairs in parallel. Then it creates a new table using all such Ps and returns this table to the user.

4.3.2.2 Tuning the Dask Version

We will now describe the algorithm for tuning the Dask procedure. This algorithm is similar to the algorithm for tuning the parameters of the downsampling command. Conceptually, the algorithm takes tables A and B as input and performs the following steps. First, it takes a sample of tables A and B. Next, it decides the order in which the parameters must be tuned. Next, it decides the range of values for each of the parameters. Then it selects the best configuration for these parameters using staged tuning. Finally, it returns the best configuration.

Sampling: A naive solution is to sample the input tables randomly. However, this solution does not work, as the sampled tables may not be representative of the input tables with respect to the processing of tuples. To address this, we use a stratified sampling approach to sample the input tables. Specifically, we sample table A by first concatenating the string attributes in each tuple, then taking a stratified sample using the lengths of the concatenated strings, and finally ordering the tuples as in the original table A. We reorder the sampled tuples as in the input table A in order to capture the computation skew among the partitions in table A. Currently, the size of the sampled tables is set to 10 percent of the input table sizes. For example, if table A includes 1000 tuples, then the sampled table will include 100 tuples.

Ordering of Parameters: There are three parameters to tune for the multi-core Dask version: (1) whether input tables A and B must be swapped, (2) n_A, the number of partitions into which to split table A, and (3) n_B, the number of partitions into which to split table B. It is obvious from the procedure of the multi-core Dask version that the decision, whether or not to swap the input tables must be made first, as it determines the table that should be used to build the inverted index and the table that should be used to probe. So this parameter is tuned first. Between n_A and n_B, the parameter n_B is tuned first, as the probing time is typically longer than the index-building time. Finally, n_A is tuned.

Choice of Parameter Values: The parameter for swapping the input tables can take two values: True or False. If the tables have to be swapped, then we return True; otherwise, we return False. We observe that the procedures for the downsampling command and for the overlap blocker are similar. Specifically, both include building an inverted index using one table and probing using another table. So we use the same range of values for n_A and n_B. Specifically, the range of the number of partitions for table A (i.e., n_A) is set to [n_c, 128], where n_c is the number of cores in the machine. Similarly, the range of the number partitions for table B is set to [n_c, 128].

Selecting the Best Configuration: To select the best configuration, we must first decide whether the input tables need to swapped. Specifically, first we sample the input tables A and B using stratified sampling to get the sampled tables S_A and S_B. Then we execute the multi-core Dask version using the sampled tables. Next, we repeat the above procedure by swapping the input tables A and B. While we execute the multi-core Dask version, the number of partitions n_A and n_B, are set to the number of cores in the machine. We select the best configuration between swapping and not swapping the input tables based on the runtimes of the command executed using the sampled tables. Based on this decision, we use the appropriate sampled tables S_A and S_B to tune other parameters.

Dataset	**Table A (num of rows, num of cols)**	**Table B (num of rows, num of cols)**
Citations	$(1823978,\ 7)$	$(2512927, 7)$
Drugs	$(446048,\ 10)$	$(440048,\ 10)$
Songs	$(1000000,\ 8)$	$(1000000,\ 8)$
Songs/Tracks	$(961594, 4)$	$(734486, 6)$
FEC	$(3000000, 21)$	$(3000000, 21)$

Table 4.1: Data sets for evaluating the tuning tools for the downsampling and overlap blocker commands.

Next, we select the number of partitions for table B. Specifically, we iterate through a subset of values in the range [n_c, 128], execute the multi-core Dask version for each of the different values, and choose the maximum value before which the runtime starts increasing. While running this, we set the number of partitions for table A to the number of cores in the machine. Next, we select the number of partitions for table A. We follow the same procedure we followed to select the number of partitions for table B. While running this, the number of partitions for B is assigned to the value found in the previous step. Finally, we return the selected configuration to the user as an ordered list (X, Y, Z), where X tells the user whether the tables need to be swapped, Y is the number of partitions for table A, and Z is the number of partitions for table B. The user will then use this configuration to execute the multi-core Dask version.

4.4 Empirical Evaluation

We now evaluate the tuning tools for the downsampling and overlap blocker commands. We consider five real-world data sets in Table 4.1, which describes Citations, Drugs, Songs, Songs & Tracks, and Federal Election Commission (FEC) data sets, respectively. We ran experiments on a Linux machine with a 4-core Intel i5-3570 3.1GHz processor with 16 GB of RAM.

Dataset	**Baseline Configuration**	**Baseline Config. Runtime**	**Configuration by Tuning Tool**	**Tuning Tool Config. Runtime**	**Best Configuration**	**Best Config. Runtime**	**Time taken by Tuning Tool**
Citations	(False, 4, 4)	2475s	(False, 16, 16)	1301s	(False, 64, 32)	899s	356s
Drugs	(False, 4, 4)	984s	(False, 8, 16)	895s	(True, 32, 16)	593s	183s
Songs	(False, 4, 4)	379s	(False, 4, 16)	309s	(False, 32, 32)	241s	132s
Songs/ Tracks	(False, 4, 4)	1339s	(True, 8, 16)	871s	(True, 64, 32)	673s	347s
FEC	(False, 4, 4)	1284s	(False, 8, 16)	1187s	(False, 2, 128)	920s	484s

Table 4.2: Overall performance of the tuning tool for downsampling.

4.4.1 Evaluating the Tuning Tool for Downsampling

We now examine the performance of the tool to tune the parameters of the downsampling command. The results are shown in Table 4.2. The first column identifies the data sets. The B_size parameter for all the experiments was set to 100K.

Baseline Performance: The column "Baseline Configuration" shows the baseline configuration of parameters. The configuration is shown as an ordered list (X, Y, Z) where X indicates whether the input tables are swapped, Y is the number of partitions of table A, and Z is the number of partitions of table B'. The baseline configuration includes the default values for these parameters. The column "Baseline Config. Runtime" shows the runtime of the command using the default configuration.

Performance of the Tuning Tool: The columns "Configuration by Tuning Tool" and "Tuning Tool Config. Runtime" show the configuration selected by the tool and the runtime of the multi-core Dask version using this configuration. The results show that the tuning tool is able to reduce the runtime significantly compared to the baseline, for all five datasets, by as much as 47.4% (for Citations).

Comparison with the Best Configuration: The columns "Best Configuration" and "Best Config. Runtime" show the best parameter configuration and the runtime of the multi-core Dask version using this best configuration. We obtained this configuration by doing an exhaustive search of all possible parameter values within a pre-specified range. We observe that the tuning tool provides a reasonable configuration but not the best configuration. For example, for the Citations data set the best configuration is 402 seconds better than the configuration found by the tuning tool, and on average, the best configuration is 247 seconds better than the runtime from using the configuration given by the tuning tool. This indicates that there is still room for improvement.

The Tuning Time: The column "Time Taken by Tuning Tool" shows the time taken by the tuning tool to return a configuration. This time ranges from relatively low (132 seconds for Songs) to relatively high (484 seconds for FEC).

The total times (tuning time plus execution time) for the five datasets are 1657, 1078, 441, 1218, and 1671 seconds, respectively. Thus, the tuning tool works well on two datasets (Citations and Songs/Tracks), where the total times are still quite a bit lower than the time of the baseline configuration. On the other three datasets, the tuning tool does not work as well, mainly due to the overhead of tuning.

Discussion: The experimental results suggest that

- Even the simplest tuning method of staged tuning can already produce configurations that are much faster than the default configurations (despite the fact that these default configurations have been careful selected by experienced developers). This underscores the importance and promise of tuning.
- Staged tuning however is still falling short of finding the best configuration. So more sophisticated tuning should be examined.
- Tuning however can incur a considerable overhead, and it is not yet clear when it will help (once we have factored in the tuning overhead). There are clearly cases where tuning can help a lot, such as in the case of Citations. But in other cases, the overhead of tuning may negate its effect. More research is necessary on this topic.

Dataset	Baseline Configuration	Baseline Config. Runtime	Configuration by Tuning Tool	Tuning Tool Config. Runtime	Best Configuration	Best Config. Runtime	Time taken by Tuning Tool
Citations	(False, 4, 4)	392s	(False, 8, 8)	314s	(True, 64, 32)	260s	170s
Drugs	(False, 4, 4)	41s	(False, 8, 16)	23.07s	(True, 16, 32)	18.06s	42s
Songs	(False, 4, 4)	58s	(False, 16, 16)	42s	(False, 2, 64)	35.1s	45s
Songs/ Tracks	(False, 4, 4)	124s	(True, 8, 16)	91s	(True, 64, 128)	68.1s	82s
FEC	(False, 4, 4)	15s	(False, 4, 4)	15s	(False, 64, 16)	14s	28s

Table 4.3: Overall performance of the tuning tool for the overlap blocker command.

4.4.2 Evaluating the Tuning Tool for Overlap Blocker

We now examine the performance of the tuning tool for the overlap blocker. The results are shown in Table 4.3. The input tables have 100K tuples each, and the blocking was performed with the overlap threshold set to 3.

Baseline Performance: The column "Baseline Configuration" shows the baseline configuration of parameters. The configuration is shown as an ordered list (X, Y, Z), where X indicates whether the input tables are swapped, Y is the number of partitions of table A, and Z is the number of partitions of table B. The baseline configuration includes the default values for these parameters. The column "Baseline Config. Runtime" shows the runtime of the command using the default configuration.

Performance of the Tuning Tool: The columns "Configuration by Tuning Tool" and "Tuning Tool Config. Runtime" show the configuration selected by the tuning tool and the runtime of the multi-core Dask version using this configuration. The results show that these runtimes are less or equal to those of the baseline configurations, sometimes by as much as 44% (e.g., for Drugs).

Comparison with the Best Configuration: The columns "Best Configuration" and "Best Config. Runtime" show the best configuration and the runtime of the multi-core Dask version using this configuration. These runtimes are less than those of the tuned configurations, suggesting that there is still room for tuning improvement.

Tuning Time: The column “Time Taken by Tuning Tool” shows the time taken by the tuning tool to return a configuration. This time ranges from 28 seconds to 170 seconds. For all five datasets, the tuning time plus the execution time is higher than the time of the baseline configuration. The main conclusions we can draw from this set of experiments are similar to those drawn in the case of tuning tool for the downsampling command.

4.4.3 Summary

Our experiments clearly show that tuning can make a command version run much faster than using the baseline (default) version. They also show that even a simple tuning method such as staged tuning can already achieve significant gains.

But the experiments also show that tuning overhead can be considerable, as suspected, and adding this to the execution time may result in more time than simply executing the baseline version.

This brings up the question of where the current tuning tool can be useful. We envision several scenarios, all involving the user running a Dask version of a command.

- If this version ran too slowly, but it has finished, and the user does not have to run it again several more times, then there is no need to tune nor to execute this version again.
- If the version has finished, is judged too slow by the user, and the user also needs to run it at least several more times, then tuning is highly recommended. In this case, the overhead of tuning will be offset by the savings over several runs (as can be seen from the experimental results).
- If the version has run for a while but seemed to be slow, the user may take a risk in stopping it, performing tuning, then rerunning it again.

Given the above considerations, one may ask why not just build an optimizer that also involves tuning? In other words, why not just always tune? The problem with this is that this means there is always an overhead of tuning every time the user executes a command version. We suspect that there are many scenarios where even an untuned version may already produce acceptable runtime for the user. In such cases, tuning is a waste of time. Hence, we do not advocate always tuning. Future research is clearly necessary to speed up the tuning process, perform better tuning, and examine scenarios where the tuning tools can be helpful.

Chapter 5

Building Magellan

In Chapters 3 and 4 we have discussed the development and production stages from a *user's perspective*. For example, we discussed creating how-to guides for users, identifying the pain points in the guides, and developing tools that users can use to address these pain points.

We now discuss these two stages from a *developer's perspective.* That is, how developers should build these stages, what challenges they will face, and how they can address those challenges.

We begin by discussing the development stage. Here, the biggest challenges developers face arise from the need to make the Python packages interoperate. Example challenges include how to design appropriate data structures (for interoperability purposes), how to manage metadata (when packages can manage one another's metadata), how to handle missing values across the packages, and more. We discuss how these challenges arise in the context of open-world systems, to which Magellan belongs. This is in contrast to close-world systems such as RDBMSs.

We then discuss the production stage. Here a major challenge is scaling the Python commands that users may want to execute. We discuss current common scaling solutions (such as custom optimizations for a single core or multiple cores), analyze them, then create a how-to guide for developers (on how to scale Magellan's commands). We used this guide to scale a subset of current commands. Finally, we discuss the current status of the open-source implementation of the Magellan system.

5.1 Challenges of the Development Stage

In Chapter 3 we discussed developing how-to guides and tools. We now turn to the challenge of designing these tools as commands in Python. This challenge turned out to be highly non-trivial. It raises a fundamental question: what do we mean by "building on top of a data analysis stack"? To answer, we introduce the notion of closed-world vs. open-world systems for EM contexts. We show that Magellan should be built as an open-world system, but building such systems raises difficult problems such as designing appropriate data structures and managing metadata. Finally, we discuss how Magellan addresses these problems. As we will see, at the core, the key challenge facing the developers for the development state is *interoperability*: how to build commands and packages such that they can easily interoperate.

5.1.1 Closed-World versus Open-World Systems

A closed-world system controls its own data. This data can only be manipulated by its own commands. For this system, its own world is the only world. There is nothing else out there and thus it does not have a notion of having to "play well" with other systems. It is often said that RDBMSs are such closed-world systems. Virtually all current EM systems can also be viewed as closed-world systems. In contrast, an open-world system K is aware that there is a whole world "out there", teeming with other systems, and that it will have to interact with them. The system therefore possesses the following characteristics:

- K expects other systems to be able to manipulate K's own data.
- K may also be called upon by other systems to manipulate their own data.
- K is designed in a way that facilitates such interaction.

Thus, by building Magellan *on the Python data analysis stack we mean building an open-world system as described above (where "other systems" are current and future Python packages in the stack).* This is necessary because, as discussed earlier, in order to do successful EM, Magellan will need to rely on a wide range of external systems to supply tools in learning, mining, visualization, cleaning, IE, etc. Building an open-world system however raises difficult problems. In what follows we discuss problems with data structures and metadata. (We have also encountered several other problems, such as missing values, data type mismatch, package version incompatibilities, etc., but will not discuss them in this dissertation.)

5.1.2 Designing Data Structures

At the heart of Magellan is a set of tables. The tuples to be matched are stored in two tables A and B. The intermediate and final results can also be stored in tables. Thus, an important question is how to implement the tables. A popular Python package called pandas has been developed to store and process tables, using a data structure called "data frame". Thus, the simplest solution is to implement Magellan's tables as data frames. A problem is that data frames cannot store metadata, e.g., a constraint that an attribute is a key of a table.

A second choice is to define a new Python class called MTable, say, where each MTable object has multiple fields, one field points to a data frame holding the tuples, another field points to the key attributes, and so on. Yet a third choice is to subclass the data frame class to define a new Python class called MDataFrame, say, which have fields such as "keys", "creation-date", etc. besides the inherited data frame holding the tuples.

From the perspective of building open-world systems, as discussed in Section 5.1.1, the last two choices are bad because they make it difficult for external systems to operate on Magellan's data. Specifically, MTable is a completely unfamiliar class to existing Python packages. So commands in these packages cannot operate on MTable objects directly. We would need to redefine these commands, a time-consuming and brittle process.

MDataFrame is somewhat better. Since it is a subclass of data frame, any existing command (external to Magellan) that knows data frames can operate on MDataFrame objects. Unfortunately the commands may return inappropriate types of objects. For example, a command deleting a row in an MDataFrame object would return a data frame object, because being an external command it is not aware of the MDataFrame class. This can be quite confusing to users, who want external commands to work smoothly on Magellan's objects.

For these reasons, we take the first choice: storing Magellan's tables as data frames. Since virtually any existing Python package that manipulates tables can manipulate data frames, this maximizes the chance that commands from these packages can work seamlessly on Magellan's tables. In general, we propose that an open-world system K use the data structures that are most common to other systems to store its data. This brings two important benefits: it is easier for other systems to operate on K's data, and there will be far more tools available to help K manipulate its own data. If it is not possible to use common data structures, K should provide procedures that convert between its own data structures and the ones commonly used by other open-world systems.

5.1.3 Managing Metadata

We have discussed storing Magellan's tables as data frames. Data frames however cannot hold metadata (e.g., key and foreign key constraints, date last modified, ownership). Thus we will store such metadata in a central catalog. Regardless of where we store the metadata, however, letting external commands directly manipulate Magellan's data leads to a problem: the metadata can become inconsistent. For example, suppose we have created a table A and stored in the central catalog that "sid" is a key for A. There is nothing to prevent a user U from invoking an external command (of a non-Magellan package) on A to remove "sid". This command however is not aware of the central catalog (which is internal to Magellan). So after its execution, the catalog still claims that "sid" is a key for A, even though A no longer contains "sid". As another example, an external command may delete a tuple from a table participating in a key-foreign key relationship, rendering this relationship invalid, while the catalog still claims that it is valid.

In principle we can rewrite the external commands to be metadata aware. But given the large number of external commands that Magellan users may want to use, and the rapid changes for these commands, rewriting all or most of them in one shot is impractical. In particular, if a user U discovers a new package that he or she wants to use, we do not want to force U to wait until Magellan's developers have had a chance to rewrite the commands in the package to be metadata aware. But allowing U to use the commands immediately, "as is", can lead to inconsistent metadata, as discussed above.

To address this problem, we design each Magellan's command c from the scratch to be metadata aware. Specifically, we write c such that at the start, it checks for all constraints that it requires to be true, in order for it to function properly. For example, c may know that in order to operate on table A, it needs a key attribute. So it looks up the central catalog to obtain the constraint that "sid" is a key for A. Command c then checks this constraint to the extent possible. If it finds this constraint invalid, then it alerts the user and asks him or her to fix this constraint.

Command c will not proceed until all required constraints have been verified. During its execution, it will try to manage metadata properly. In addition, if it encounters an invalid constraint it will alert the user, but will continue its execution, as this constraint is not critical for its correct execution (those constraints have been checked at the start of the command). For example, if it finds a dangling tuple due to a violation of a foreign key constraint, it may just alert the user, ignore the tuple, and then continue.

5.2 Challenges of the Production Stage

As we have discussed several times in this dissertation, the first and foremost challenge of the production stage is scaling. In particular, how should we, i.e., developers, scale the execution of the entire EM workflow or the execution of pieces of this workflow?

In this chapter we focus on answering the above question. Recall that an EM workflow consists of multiple Python commands, such as those that perform blocking and matching, among others. We will focus on scaling the execution of a single such Python command, leaving scaling larger units (such as a workflow fragment consisting of a sequence of commands) for future work.

In this context, note that in Chapter 4 we have discussed scaling from a *user's perspective.* That is, how can a user execute a Python command fast, given several implementations of this command? Here we approach this question from a developer's perspective. That is, how should developers scale Python commands? We will focus on the setting of a single machine with multiple cores, leaving other settings such as a cluster of machines for future work.

We will pursue the following attack plan:

- First, we will analyze the pros and cons of the main current methods to scale a Python command.
- Based on that analysis, we will develop a how-to guide for developers, which tell them how to scale a Python command in Magellan. This guide is important as we envision that multiple developers will want to further extend Magellan, such as trying to scale the commands in the production stage. If so, we want to have a guidance to them on how to do so.
- Finally, we will use this how-to guide ourselves to scale a set of time-itensive Python commands for the production stage.

As we will see, the end result is that for each command, we have developed several implementation versions. Chapter 4 already discussed how users can select among these versions, using a how-to guide (note that that guide is for users, and is distinct from the how-to guide mentioned above for developers).

Finally, we note that even though we focus on scaling Python commands for the production stage, much of what we discuss here can be applied to scaling commands for the development stage as well.

5.2.1 Analyzing Current Scaling Methods

There are currently four common methods to scale Python commands on a single machine with multiple cores: custom optimization for a single core, using a faster language for a single core, custom optimization for multiple cores, and using an automatic parallelization framework (e.g., Dask, PySpark) for multiple cores. We now briefly explain and then analyze these methods.

5.2.1.1 Four Common Scaling Methods

Custom Optimization for a Single Core: In this method, a command is modified by using optimized libraries (in place of default ones) [6, 20, 8], employing better data structures, applying algorithmic changes, and using platform-specific optimizations [86]. This method has the advantage that the modified Python code is relatively easy to reason with, as the faster libraries and data structures are often drop-in replacements for existing ones. But this method has the following disadvantages. First, the developer needs to know the implementation details of the command to understand the impact of these optimizations. Second, debugging becomes harder when platform-specific optimizations are used. Finally, platform-specific optimizations are applicable only to a subset of platforms, thereby limiting the broad applicability of the code.

Using a Faster Language for a Single Core: Here the command is rewritten using faster languages such as Cython [26]. Cython is a superset of the Python programming language. It supports calling C functions and declaring types for variables and class attributes. Code written in Cython is first translated to C, then compiled to CPython extension modules. When these modules are executed they provide C-like performance. However, this method has three disadvantages. First, it requires a significant rewriting of some parts of existing code. Second, it is harder to write bug-free code using Cython. Specifically, the Cython constructs that speed up the execution often resemble C language and writing a bug-free code using such constructs is hard. Finally, it is harder to debug code written in Cython compared to code written in Python.

Custom Optimization for Multiple Cores: Here the command is modified to execute in parallel using multiple cores of the machine. Specifically, their Python code is rewritten using the multiprocessing libraries in Python [7, 12, 14]. This method has two disadvantages. First, implementing a parallel version of a command is not straightforward. Specifically, the developer has to identify the tasks performed by the current code, find the dependencies between these tasks, identify those tasks that could be executed in parallel, and finally manually fix the execution order of these tasks using the multiprocessing libraries. Second, it is non-trivial to debug a command that is implemented to execute in parallel.

Using an Automatic Parallelization Framework for Multiple Cores: Here the command is modified using the primitives from an automatic parallelization framework [113, 16]. Specifically, the code is either annotated or rewritten using the primitives provided by such a framework without worrying about parallelism. When the command is executed using the framework, the framework will internally use these primitives to execute the command in parallel. The exact set of primitives

```
def block_tables(A, B, overlap_threshold):
    pre_processed_A = pre_process(A)
    pre_processed_B = pre_process(B)
    inv_index_A = build_inv_index(pre_processed_A)
    probe_result = probe(inv_index_A, pre_processed_B,
                            overlap_threshold)
    result = post_process(probe_result)
    return result
```

Figure 5.1: An example implementation of overlap blocker for a single core.

and the way to use them vary among different frameworks. However, the overall idea of using these primitives to implement a command and using the framework to automatically interpret these primitives to execute the command in parallel remains the same. Example frameworks include Dask and PySpark [113, 16].

In the rest of this section we will illustrate these frameworks by briefly discussing how Dask works. An example of modifying a command using Dask [113] is shown in Figures 5.1, 5.2, and 5.3. Figure 5.1 shows a single-core implementation of a command that performs overlap blocking over two tables. This command takes in two tables, a blocking attribute, and an overlap threshold as input. For ease of exposition, we assume that both the input tables include the blocking attribute in their columns. Next, the command preprocesses the input tables and builds an inverted index using the preprocessed first table. Then it probes the inverted index using the preprocessed second table and identifies the tuple pairs that satisfy the blocking threshold. Finally, it post-processes the tuple pairs and returns the result back to the user.

Suppose after writing this single-core implementation, the developer realizes that the code runs too slowly, then he or she can easily "daskify" it as shown in Figure 5.2. Specifically, the developer identifies parts of the code that can be converted to a DAG and annotates it with the "delayed" primitive. Afterward, when the user executes the annotated command, Dask first creates a directed acyclic graph (DAG) for this implementation using the "delayed" annotations as shown in Figure 5.3. In the DAG, a circle indicates a task in each line of this updated Python implementation.

```
def block_tables(A, B, overlap_threshold):
    pre_processed_A = delayed(pre_process)(A)
    pre_processed_B = delayed(pre_process)(B)
    inv_index_A = delayed(build_inv_index)(pre_processed_A)
    probe_result = delayed(probe)(inv_index_A, pre_processed_B,
                                  overlap_threshold)
    result = delayed(post_process)(probe_result)
    return result
```

Figure 5.2: The Python implementation for overlap blocker annotated using “delayed.” This is the “daskified” version of the single-core Python code shown earlier in Figure 5.1.

The rectangular boxes indicate outputs of the tasks. Dask then creates a schedule based on these dependencies in the DAG. Finally, Dask executes this schedule using multiple cores in the machine.

Automatic parallelization frameworks such as Dask have a disadvantage similar to that of the previous approach, i.e., debugging parallel executions of tasks is difficult. However, these frameworks often provide good monitoring and debugging tools that alleviate the debugging problems.

5.2.1.2 Dask versus PySpark

There are two well-known automatic parallelization frameworks: Dask and PySpark. For the developers of the production stage of Magellan, we would like to recommend using one of these frameworks. In what follows we discuss both frameworks, in order to make a selection.

Dask [113] is a parallel computing framework for analytical computing. It is composed of three main components: (1) the ability to create graphs with a light annotation of normal Python code, (2) the ability to schedule the tasks in the graph dynamically, and (3) the inclusion of big data collections such as parallel arrays and dataframes that extend numpy [124] and pandas [98].

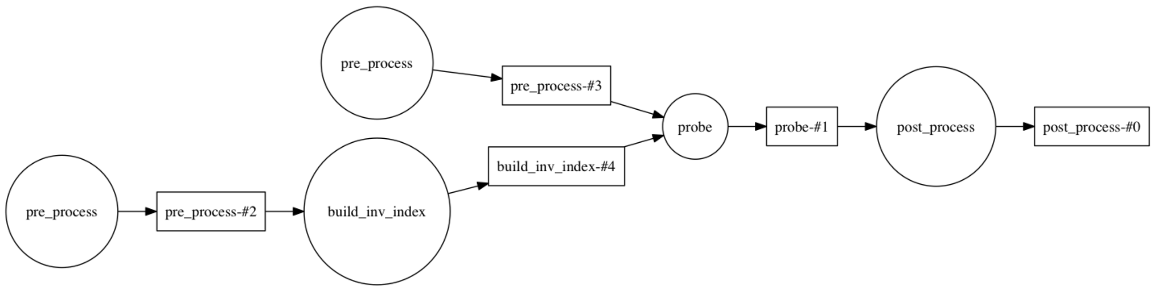

Figure 5.3: A DAG created by Dask when executing the code shown in Figure 5.2.

PySpark [16] is a Python library that provides a Python interface to Apache Spark [133]. Apache Spark is a general-purpose cluster computing framework. Similar to Dask, PySpark constructs a graph from the Python code (written using Spark primitives). It then dynamically schedules and executes the tasks in the graph. Spark also includes big data collections such as dataframes similar to Dask dataframes. We now discuss the above two frameworks from several perspectives.

Integration with the PyData Ecosystem: Dask is written in Python. Since it is Python native, it integrates well with the Python data ecosystem. Specifically, it couples with and scales libraries such as numpy, pandas, and scikit-learn. It also interoperates well with C, C++, or other natively compiled code linked through Python.

Spark was written in Scala with interfaces for Python and R. It interoperates well with other JVM code. Since it is not Python native, it does not integrate as well with the PyData ecosystem. Instead, Spark includes its own ecosystem. For example, it includes tools for machine learning, graphs, and streaming. [100, 3].

Maturity: Dask was started in 2014 and the software is still maturing. However, it extends mature packages in PyData ecosystem such as numpy and pandas. Spark was started in 2010 and the software is relatively well matured. It is used widely in the industry and it has become one of the dominant players in big data analytics space.

Scope: Dask aimed to support use cases that are more common with scientific and business users. Specifically, it focused on users who mainly use numpy, pandas, and scikit-learn, and required scaling a custom code written in Python. Spark is more focused on traditional business intelligence operations such as SQL and machine learning.

Scaling: Both Dask and Spark support scaling the same code from a single machine to a cluster of machines, but in different ways. Dask was designed to scale on a single machine first. Specifically, it supports executing the Python code on a single machine by using multiple threads or processes. Support for executing code on a cluster was added later. The main reason for prioritizing a single machine (over a cluster) was to support data scientists who want to analyze moderate-size data sets (for example 50 GB) that can fit in a single machine. Spark was designed primarily to scale to a cluster of machines. Executing on a single machine is typically used for the debugging purposes. In Spark, the design priority was to support processing big data (for example terabytes of data) using a cluster of machines.

Custom Parallelism: Dask allows users to specify an arbitrary task graph by simple annotations to the *existing* Python code and then executes this graph using a single machine or a cluster of machines. Spark does not allow for annotating existing Python code to specify an arbitrary task graph. It typically expects users to compose a workflow using their high-level primitives such as map, reduce, group by, and join. In other words, if the developer has written a single-code Python code and now is looking to scale it up, then using Dask would incur less change to the code than using PySpark.

5.2.1.3 Selecting Dask

Based on the above discussion, we chose to focus on Dask. First, since it is native to Python, it integrates well with the PyData ecosystem. Second, Dask can be used to scale single-core arbitrary Python code using simple annotations. This is a big advantage from a developer's perspective. We expect that in Magellan's context, many times a developer will start out writing single-core Python code for a command, as writing such code is much easier to start with. Then if the code is slow, the developer will look to scale it up, with minimal effort. Dask fits this requirement as it will require only light annotations of the code. In contrast, Spark will require a significant rewrite of the code.

Dataset	Table A (num. rows, num. cols)	Table B (num. rows, num. cols)
Citations	(1823978, 7)	(2512927, 7)
Songs	(1000000, 8)	(1000000, 8)

Table 5.1: Data sets for performance comparison between Dask and custom multi-core versions.

Indeed, our experience has been that Dask does not require too much annotation effort. Specifically, we took the single-core version of 12 Python commands that were developed as part of Magellan and annotated these commands using the Dask primitives. This required annotating less than 300 lines of code and minimal rewriting to handle data parallelism, which took 2 weeks of development effort with one developer.

5.2.1.4 Using Dask versus Custom Optimization for Multiple Cores

To scale up a Python command for multiple cores, we can use either Dask or custom optimization. Intuitively, custom optimization often takes far more time, but should be at least competitive with or can yield faster runtimes than Dask. Still, we are interested in knowing how much worse Dask would perform for Magellan commands, compared to custom optimization. This can inform our design of the how-to guide for developers.

In this section we examine this issue. We considered two real-world data sets: Citations and Songs [45], which contain 1M-2.5M tuples in each table (see Table 5.1). We considered 12 compute-intensive commands from Magellan. These commands are described in Table 5.2. The first column specifies the EM step that a command performs, the second column specifies the type of the operation in this EM step, and the third column specifies the exact command in Magellan.

For Dask implementation, we took the single-core version of the 12 commands (see Table 5.2) from Magellan and modified them using the Dask primitives. For custom multi-core implementation, we took the single-core version of the same 12 commands and reimplemented them using multiprocessing libraries [7, 14] in Python. We ran the experiments on a Linux machine with 4 cores, Intel i5-3570 3.1GHz processor, and 16GB of RAM. The specific input data set sizes varied based on the command semantics and they are described in their corresponding sections below.

Step	**Type**	**Command in Magellan**
Down sampling	NA	down_sample
Blocking	Attribute Equivalence Blocker	block_tables
	Attribute Equivalence Blocker	block_candset
	Overlap Blocker	block_tables
	Overlap Blocker	block_candset
	Rule-based Blocker	block_tables
	Rule-based Blocker	block_candset
	Blackbox Blocker	block_tables
	Blackbox Blocker	block_candset
Extract feature vectors	NA	extract_feat_vecs
Select matcher	NA	select_matcher
Predict matches	NA	predict_matches

Table 5.2: Magellan commands used for performance comparison between Dask and custom multi-core versions.

Dataset	Output Sample Size	Runtime for Custom Version	Runtime for DaskVersion	Memory Usage for Custom Version (GB)	Memory Usage for Dask Version (GB)
Songs	100K	2.37m	2.27m	1.53	1.6
	200K	4.65m	4.34m	1.57	1.73
Citations	100K	8.2m	7.9m	3.5	3.6
	200K	18.1m	16.3m	3.6	3.8

Table 5.3: Runtime and memory usage comparison for the downsampling command.

Downsampling: We consider two cases based on the sizes of the output tables: 100K tuples and 200K tuples. The runtime performance and the memory used in each of these cases is shown in Table 5.3.

We observe that both the Dask version and the custom multi-core version provide similar runtime performance, and that the Dask version consumes slightly more memory compared to the custom multi-core version. This is because of the extra metadata (such as the task dependencies and completed tasks) maintained by Dask scheduler while executing the DAG.

Blocker Type	Dataset	Input Table Size	Runtime for Custom Version	Runtime for DaskVersion	Memory Usage for Custom Version (GB)	Memory Usage for Dask Version (GB)
Attribute Equivalence	Songs	100K	49.7s	48.6s	6.3	6.4
		200K	DNC	DNC	DNC	DNC
	Citations	100K	32.2s	33.9s	6	6.2
		200K	DNC	DNC	DNC	DNC
Overlap	Songs	100K	26.09s	24.59s	2.04	2.3
		200K	102.8s	93.96s	6.8	7.31
	Citations	100K	41.67s	35.28s	1.64	2.04
		200K	146.38s	138	4.9	5.6

Table 5.4: Runtime and memory usage comparison for two blockers.

Blocker Type	Dataset	Input Table Size	Runtime for Custom Version	Runtime for DaskVersion	Memory Usage for Custom Version (GB)	Memory Usage for Dask Version (GB)
Rule-based	Songs	100K	22s	23s	1.9	2.32
		200K	73s	74.29s	2.8	3.44
	Citations	100K	34s	33.1s	1.8	2.08
		200K	94.2s	92.3s	2.8	3.2
Blackbox	Songs	100K	4.2h	4.1h	3.1	3.4
		200K	16.45h	15.5h	5.4	5.7
	Citations	100K	6.1h	5.33h	4.1	4.3
		200K	22.54h	21.32h	6.8	7.1

Table 5.5: Runtime and memory usage comparison for the rule-based and blackbox blockers.

Blocking: We consider four different blockers: attribute equivalence, overlap, rule-based, and black box. For each blocker, we consider two cases: blocking two tables and blocking a candidate set.

The blocking of two tables is run for sample sizes 100K and 200K for each of the Songs and Citations data sets. The results of blocking for different blockers and the sampled data sets are shown in Tables 5.5 and 5.4.

Overall, both the Dask version and the custom multi-core version provide similar runtime performance, and the Dask version uses more memory. Specifically, Dask uses 10.5% more memory on average compared to the custom multi-core version. But if we zoom in to the case of attribute equivalence blocker for the Citations data set with the sampled tables for size 200K, we see that the run did not complete (marked as "DNC"). This is because the number of output tuple pairs generated were too large and they consumed all the memory in the machine. The corresponding process was subsequently killed by the operating system.

Blocker Type	Dataset	Runtime for Custom Version	Runtime for DaskVersion	Memory Usage for Custom Version (GB)	Memory Usage for Dask Version (GB)
Attribute Equivalence	Songs	28s	27s	3.6	3.97
	Citations	25.8s	24s	3.9	4.1
Overlap	Songs	102s	103.46s	3.5	3.92
	Citations	172.8a	170.7s	3.95	4.25
Rule-based	Songs	650s	616s	3.5	3.9
	Citations	574.11s	556s	4	4.1
Blackbox	Songs	315s	278s	2.9	3.3
	Citations	257s	232s	2.8	3.2

Table 5.6: Runtime and memory usage comparison for blocking on a candidate set.

To compare the performance of blocking on a candidate set (a set of tuple pairs obtained from blocking over two tables), we considered two tables: one obtained from the Songs data set and the other obtained from the Citations data set. Specifically, these tables were obtained by applying overlap blocker over the sampled Songs and Citations data sets with 100K tuples.

The candidate set from Songs data set has 13M tuples and the candidate set from the Citations data set has 17M tuples. The results are shown in Table 5.6. We observe similar runtime performances between the Dask version and the custom multi-core version. But the Dask version uses 9.6% more memory on average compared to the custom multi-core version.

Dataset	Runtime for Custom Version	Runtime for DaskVersion	Memory Usage for Custom Version (GB)	Memory Usage for Dask Version (GB)
Songs	139.4s	133s	2.9	3.3
Citations	431.8s	423s	2.8	3.4

Table 5.7: Runtime and memory usage comparison for converting a candidate set to a set of feature vectors.

Converting a Candidate Set to a Set of Feature Vectors: This command was run for a candidate set of size 139K for Songs and 550K for Citations data sets. These candidate sets were obtained by applying a sequence of overlap blocker and rule-based blocker over the sampled Songs and Citations data sets (with 100K tuples). The number of features generated were 44 for Songs and 24 for Citations. The results are shown in Table 5.7. We observe similar runtime performances between the Dask version and the customized version. The Dask version uses 14.1% more memory on average compared to the custom multi-core version.

Dataset	Runtime for Custom Version	Runtime for DaskVersion	Memory Usage for Custom Version (GB)	Memory Usage for Dask Version (GB)
Songs	45.65s	41.81s	1.46	1.59
Citations	40.49s	35.84a	1.53	1.62

Table 5.8: Runtime and memory usage comparison for selecting a matcher.

Selecting a Matcher: This command was run for a sample of 500 tuples from the sets of feature vectors of Songs and Citations data sets. We considered five different matchers and used five-fold cross validation to select the best matcher. The results are shown in Table 5.8. We observe similar runtime performances for the Dask version and the customized version. The Dask version uses 6.8% more memory on average compared to the custom multi-core version.

Dataset	Runtime for Custom Version	Runtime for DaskVersion	Memory Usage for Custom Version (GB)	Memory Usage for Dask Version (GB)
Songs	1.5s	1.43s	2.1	2.4
Citations	2.89s	3.09s	2.2	2.34

Table 5.9: Runtime and memory usage comparison for predicting the matches.

Predicting the Matches: We predicted the matches for the feature vectors of sizes 139K and 550K from Songs and Citations data sets. We used a pre-trained random forest model for prediction purposes. The results are shown in Table 5.9. Similar to previous results, here the runtimes were similar and the Dask version used 9.8% more memory on average compared to the custom multi-core version.

Summary: Our results show that the Dask and custom multi-core versions provide similar run time performances. The Dask version uses more memory (specifically in the range 2.4%-18.7%) compared to the customized version, mainly because of the metadata that Dask stores for its internal purposes.

Some of our experiments (e.g., blocking two tables using attribute equivalence blocker) did not complete because of the memory error caused by too many tuple pairs produced. This issue was not specific to Dask. We observed a similar result for the customized version of the same command. This indicates that we should avoid keeping unnecessary data in the memory as much as possible. Dask provides such features, to some extent, and exploring these features is left for future work.

5.2.2 Creating a How-to Guide for Developers and Applying the Guide

Based on our analysis of current scaling methods for a single machine in Python, we have developed an initial guide for Magellan developers to scale Magellan commands on a single machine using multiple cores. We will now describe this guide. The guide states that the developer should first start with a single-core implementation using Python. This is simple to do in practice and this version often performs well for smaller data sets.

If the command (i.e., the above version) is not fast enough, apply simple custom optimizations (e.g., using better data structures) to the single-core code. If the command is still not fast enough, the developer should implement a multi-core version using Dask. This implementation typically requires minimal modifications to the Python code written for a single core.

Finally, if the command is still not fast enough, the developer should implement a custom multi-core version. This implementation would require the developer to rewrite the code using multiprocessing libraries in Python, write some parts of the code in Cython, apply custom optimizations such as integrating external modules written in C, perform speculative execution, using shared memory for updates, etc. This might require significant development effort.

Note that in the above guide, we do not ask the developers to implement the single-core Cython version. Based on our experience, it is difficult for developers to develop a Cython version of the commands. Further, after a Cython version has been developed, it is difficult to maintain, because it is often difficult for other developers to understand and update the code.

We have used the above how-to guide to scale 12 compute-intensive Python commands in Magellan (e.g., downsampling, blocking, generating feature vectors, etc.), by developing the Dask versions of these commands.

5.3 The Current Open-Source Implementation of Magellan

So far in this chapter we have discussed the challenges that developers face in implementing the development stage and the production stage of Magellan. We have also discussed preliminary solutions to some of these challenges. We have used these solutions to implement both stages.

In particular, Magellan has been in development since June 2015. We have developed how-to guides and implemented seven different tools for the pain points in those guides. These tools were implemented using 12 different packages in the Python data ecosystem. Over the past three years, 13 developers contributed to the Magellan system. As of June 2018, Magellan consists of 6 Python packages with 37K lines of code and 104 commands. We have open sourced Magellan

[13]. As far as we can tell, Magellan is the most comprehensive open-source EM system today, in terms of the number of features it supports.

Chapter 6

Magellan "in the Wild": Successes and Lessons Learned

In this chapter we first describe how we have successfully applied Magellan "in the wild" to a number of EM projects in domain science and industry. We then describe a case study of end-to-end EM in applied economics in detail. The goal is to make very concrete many challenges that arise during EM in practice. As far as we can tell, no academic publication has discussed a detailed execution of end-to-end EM in practice. Finally, based on our experience, including working on the above applied economics case, we discuss a set of lessons learned.

6.1 Applying Magellan to Projects in Domain Science and Industry

So far Magellan has been applied to five projects in three domain sciences at UW-Madison.

- A team of applied economists used Magellan to match two tables of 1,832 and 1,916 grant descriptions, respectively [80]. Magellan achieved significantly better accuracy, improving recall by 23% while achieving comparable precision, compared to a rule-based EM solution deployed at [80]. We describe the above EM project in detail below. The same team also used Magellan to match two tables of 1,851 and 13.5M organization descriptions, respectively. This project has resulted in a technical report, several planned publications, and a plan to deploy Magellan in production.

- A team in biomedicine used Magellan to match two tables of 453K and 451K of drug descriptions, achieving 99.1% precision and 95.2% recall [89, 107]. This project has resulted in a poster paper [89], a workshop paper [67], and several invited talks (e.g., see [107]).

- A team in geography used Magellan to match cattle ranches in Brazil, improving recall by more than 40% while achieving roughly the same precision, compared to a rule-based matching system in production. Magellan is currently being put into production for this team (as of August 2018).

- Another team in biomedicine used Magellan to match attribute names within a community data repository [28]. This project has resulted in a journal paper [28] and a community database available publicly on the Web (*metasra.biostat.wisc.edu*).

- Finally, a team in environmental sciences also used Magellan to match attribute names within a community data repository. These last two examples show how Magellan can also be used to match schema elements, not just data instances.

Magellan has also been used for EM at several companies, including WalmartLabs, Johnson Control, Marshfield Clinic, Recruit Holdings, and American Family Insurance. At WalmartLabs, Magellan was able to help improve the recall of a deployed EM solution by 34% while reducing precision slightly by 0.65%. Johnson Control has used Magellan to match addresses (between tables of size 90K vs. 231K) and vendors (within a single table of size 50K). Marshfield Clinic was involved in the drug matching project described earlier [89, 107]. Recruit Holdings used Magellan to match stores, companies, and properties (e.g., de-duplicating 10K store names with 98.9% accuracy) [4]. Finally, American Family Insurance, a Fortune 500 company, has used Magellan to match customers from multiple databases [2].

6.2 Entity Matching in Applied Economics: A Case Study

Though a lot of work has been done on EM, as far as we can tell, no work has described an end-to-end case study. Here we describe a case study that we worked on in collaboration with applied economists at UW-Madison. The goal is to make very concrete many challenges that arise during EM in practice.

6.2.1 The EM Problem at UMETRICS

UMETRICS is an abbreviation for "Universities: Measuring the Impacts of Research and Innovation, Competitiveness and Science" [19]. It is a consortium aimed to create independent statistical evidence about the value of university research and to provide answers regarding how the funds allocated from different government agencies such as the U.S. Department of Agriculture (USDA) and the National Institutes of Health (NIH) are used for research in the universities.

As a part of this effort, the Institute for Research on Innovation & Science (IRIS) [10] was created to manage the UMETRICS effort. It is a member-driven organization created by and for the universities. Specifically, IRIS's goal is to build a platform for the whole community involving funding agencies and universities and to make UMETRICS a trusted and national data source to quantify the impact of funding that supports university research.

To make this a reality, at the core, this involves matching the research projects for which the funds were granted (by the funding agencies) and the research projects in the universities. UMETRICS contains data on research projects, researchers, and graduate students supported by federal grants for each university. Similarly, funding agencies maintain grants given to the research projects in different universities. Specifically, the funding agencies typically maintain a record of the research project title, principal investigators of the project, funding duration, university name, etc. Similarly, UMETRICS maintains a record of the research project title, funding source, funding duration, etc., for each research project in a university.

Matching these research grants between the funding agencies and UMETRICS is an EM problem, which is not straightforward. For example, the same research project can have different research titles recorded in UMETRICS and funding agencies. As another example, a grant given by a funding agency to a research project may be distributed to many smaller projects in the university, and this information will be recorded in multiple entries in UMETRICS, so matching these entries is not trivial.

UW-Madison is a participant in the UMETRICS consortium. The team from UW-Madison included members from the UW-Madison Economics department. We will refer to this team as the UW-UMetrics team. The funding information from different agencies was stored across multiple data sets. The UW-UMetrics team's goal was to match these data sets to the UMETRICS records that are specific to UW-Madison. Specifically, their plan was to first create an initial matching solution, then measure the accuracy of this solution (as this will have an impact on the analysis done over the matches), and finally improve the matching solution to achieve better accuracy.

The UW-UMetrics team created a rule-based matching solution for each funding agency. Before we started working with them closely, we let them first implement their solution and decided to provide consulting help for specific problems such as debugging the matches and evaluating the match results. However, after multiple iterations of meetings, we observed that no streamlined matching process existed.

We then decided to get involved to solve the matching problem, and we decided to start by considering only a small part of the problem. Specifically, we considered matching the records from the USDA data set (which contained funding information from the USDA) to the UW-Madison records in the UMETRICS data set because we realized that this is already a complex enough problem. We worked closely with the UW-UMetrics team to do the matching. The UW-UMetrics team consisted of a professor, a PhD student, and an hourly paid student. Our team included a professor, a PhD student, and a professional master's program (PMP) student. We decided to use Magellan to perform the matching. Specifically, our goal was to follow the how-to guide and use the tools that were developed as a part of Magellan to perform the matching. In the following sections, we discuss the steps we followed. We intentionally discuss them in chronological order to show how zig-zag the process was. We will also discuss details such as where the files are stored, how the two teams communicated, so highlight the logistic aspects of executing such a data science project (in a distributed fashion).

6.2.2 Exploring and Understanding Input Tables

We received the raw data as flat files in CSV format from the UW-UMetrics team. These files were stored in a Google Drive folder, and the UW-UMetrics team gave read access permissions to our Gmail accounts. Once we gained access to the raw data, we started by exploring and understanding the tables in them. Our goal was to understand the "entities" in these tables and the relationships among them.

First, we downloaded the raw data from the Google Drive folder onto our local disk. There were six files with the "UMETRICS" prefix and one file with the "USDA" prefix. From the file names, we assumed that the names with the "UMETRICS" prefix would correspond to UMETRICS-related tables and the file with the "USDA" prefix would correspond to USDA table. Next, we explored each one of these tables to get a brief understanding of the information included in them and the data values in their columns. Specifically, for each table, we browsed over a few sample rows that were randomly selected from the table and printed some general statistics such as the number of unique values, number of missing values, mean, median, etc., for each column. To explore the tables, we used a combination of different tools. Specifically, we used pandas [98] and MS Excel to explore the smaller tables and SQLite to explore the larger tables (e.g., $>$ 300K records). To profile the tables, we used the pandas-profiling tool [15] and custom Python scripts.

Schemas of UMETRICS Tables: The schemas of the six UMETRICS tables are listed below:

```
UMETRICSAwardAggMatching(UniqueAwardNumber,AwardTitle, FundingSource,
FirstTransDate, LastTransDate,RecipientAccountNumber,
TotalOverheadCharged,
TotalExpenditures, NumberOfTransactions, DataFileYearEarliest, DataFileYearLatest,
SubOrgUnit, CampusID)

UMETRICSSubAwardMatching(UniqueAwardNumber, Address, BldgName, City, Country,
DUNS, DomesticZipCode, EIN, ForeignZipCode, ObjectCode, OrgName, OrganizationID,
POBox, PeriodEndDate, PeriodStartDate, RecipientAccountNumber, StrName, StrNumber,
SubAwardPaymentAmount, DataFileYear)

UMETRICSOrgUnitsMatching(CampusId, SubOrgUnit, CampusName, SubOrgUnitName,
DataFileYear)
```

```
UMETRICSEmployeeMatching(UniqueAwardNumber, PeriodStartDate, PeriodEndDate,
RecipientAccountNumber, DeidentifiedEmployeeIdNumber, FullName,
OccupationalClassification, JobTitle, ObjectCode, SOCCode, FteStatus,
ProportionOfEarningsAllocated, DataFileYear)

UMETRICSVendorMatching(UniqueAwardNumber, PeriodStartDate, PeriodEndDate,
RecipientAccountNumber, ObjectCode, OrganizationID, EIN, DUNS, VendorPaymentAmount,
OrgName, POBox, BldgNum, StrNumber, StrName, Address, City, State, DomesticZipCode,
ForeignZipCode, Country, DataFileYear)

UMETRICSObjectCodesMatching(ObjectCode, ObjectCodeText, DataFileYear)
```

Entities in UMETRICS Tables: Based on our exploration and profiling of the UMETRICS tables, we noted conceptually the entities (as much as we could) that each table corresponded to. “UMETRICSAwardAggMatching” included information about awards, i.e., research grants. It included funding information for the research projects in the university. “UMETRICSSubAward-Matching” included information about how the awards are split into multiple sub-awards to fund the research projects. “UMETRICSOrgUnitsMatching” included information about different organization units to which the awards were given. “UMETRICSEmployeeMatching” included information about the employees in the university, and “UMETRICSVendorMatching” included information about the vendors interacting with the university. We were not clear about the information included in the “UMETRICSObjectCodesMatching” table.

Relationships in UMETRICS Tables: We observed that “UMETRICSAwardAggMatching” was the central table to which most other tables (except “UMETRICSObjectCodesMatching”) could be joined using the key-foreign key relationship. The column “UniqueAwardNumber” was the primary key of the “UMETRICSAwardAggMatching” table. The tables “UMETRICSEmployeeMatching,” “UMETRICSVendorMatching,” and “UMETRICSSubAwardMatching” included the “UniqueAwardNumber” column, which could be joined with the ”UniqueAwardNumber” of the “UMETRICSAwardAggMatching” table. The table “UMETRICSOrgUnitsMatching” included a

UMETRICS

Table Name	Num. Rows	Num. Cols
UMETRICSAwardAggMatching	1336	13
UMETRICSEmployeesMatching	1454070	13
UMETRICSObjectCodesMatching	4574	3
UMETRICSOrgUnitMatching	264	5
UMETRICSSubAwardMatching	21470	23
UMETRICSVendorMatching	377746	21

USDA

Table Name	Num. Rows	Num. Cols
USDAAwardMatching	1915	78

Figure 6.1: Summary of original UMETRICS and USDA tables given by UW-UMetrics team.

"CampusId" column that could be joined with "CampusId" in the "UMETRICSAwardAggMatching" table. The table "UMETRICSObjectCodesMatching" included an "ObjectCode" column that could be joined with the "ObjectCode" column in the "UMETRICSEmployeeMatching" table.

The USDA Table: Only one table contained USDA information. The table included 78 columns. Because of space constraints, a partial schema of the table including the first few columns and the last column is shown below.

```
USDAAwardMatching(AccessionNumber, ProjectTitle, AwardNumber, ProjectStartDate,
ProjectEndDate, ProjectDirector, SponsoringAgency,.., Financial: USDAContracts,
Grants, Coop Agmt)
```

The table included a single entity containing USDA award-related information, and "AccessionNumber" was the primary key for this table. A summary including the number of rows and columns for each table is shown in Figure 6.1. A few example rows from UMETRICS tables are shown in Figures 6.2, 6.3, 6.4, and 6.5, and a few example rows from the USDA table are shown in Figure 6.6.

UMETRICSAwardAggMatching

UniqueAward Number	AwardTitle	Funding Source	FirstTrans Date	LastTrans Date	Recipient Account Number	Total Overhead Charged	Total Expenditures	NumberOf Transactions	DataFile YearEarliest	DataFile YearLatest	SubOrg Unit	Campus ID
00.070 03CS-11231300 -031	MAPPING THE 1990 WUI AND 1990-2000 WUI CHNAGE AT THE CENSUS BLOCK LEVEL ACROSS THE UNITED STATES	USDA, FOREST SERVICE	2007-07-01	2008-07-31	MSN106198	0	1296.43	5	2008	2009	7	UWMSN
10.203 WIS01717	Pathogen and host variability in alfalfa with regard to Aphanomyces root rot	HATCH ACT FORMULA FUND	2014-01-01	2016-06-30	MSN158789	0	81171.5	30	2014	2016	7	UWMSN

UMETRICSEmployeeMatching

UniqueAward Number	PeriodStart Date	PeriodEnd Date	Recipient Account Number	Deidentified EmployeeId Number	FullName	Occupaitional Classification	JobTitle	Object Code	SOC Code	Fte Status	ProportionOf Earnings Allocated	Data File Year
81.049 DE-FG02-95ER40896	2007-07-01	2007-07-31	MSN103534	B6D6B9B8C09CC8993C7A7EEDFA70970523B2A8BC	CARLSMITH, DUNCAN L	Faculty	Professor	1003	25-1000.0	1	1	2008
00.600 MSN108110	2007-10-01	2007-10-31	MSN108110	37CCA6327EEC785343CE2AF84C02B725534F2127	FIORE, MICHAEL C	Faculty	Professor	1001	25-1000.0	1	0.05	2008

Figure 6.2: Example rows from UMETRICS tables (part 1).

6.2.3 Understanding Match Definition

After exploring the tables, we have obtained an understanding of the entities and their relationships. However, we did not know how to use these tables to match the awards. We were not clear about which tables were relevant for matching and what it meant to be a match for a record pair from UMETRICS and USDA. We contacted the UW-UMetrics team to gain a clear understanding of the matching problem. They provided us with a document [54] that included information about the most relevant tables, matching definition, and a few examples of matching and non-matching record pairs. The document stated that only three tables (among seven) were most relevant for matching and provided the following matching definition.

- M_1. If a part of "UniqueAwardNumber" in UMETRICS matches "Award Number" in USDA, then the record pair can be considered a match. Specifically, "UniqueAwardNumber" can take the form "XX.XXX YYYY-YYYY-YYYYY-YYYYY". If "YYYY-YYYY-YYYYY-YYYYY" matches the "Award Number," then the record pair is a match. An example of such a match is shown in Figure 6.7.

UMETRICSVendorMatching

Unique Award Number	PeriodStart Date	PeriodEnd Date	Recipient Account Number	Object Code	Organization ID	EIN	DUNS	Vendor Payment Amount	Org Name	POBox	Bldg Num	Str Number	Str Name	Address	City	State	Domestic ZipCode	Foreign ZipCode	Country	Data File Year
00.500 MSN102793	2008-02-20	2008-02-20	MSN102793	3740	0000000066-1	NaN	NaN	698.8	CAPITAL NEWSPAPERS INC	NaN	NaN	NaN	NaN	PO BOX 9069	MADISON	WI	53708	NaN	US	2008
81.049 DE-FG02-95ER40896	2007-07-13	2007-07-13	MSN103534	3105	0000000066-1	NaN	NaN	49.58	ACE HARDWARE CENTER	NaN	NaN	NaN	NaN	1398 WILLIAMSON ST	MADISON	WI	53703	NaN	US	2008

Figure 6.3: Example rows from UMETRICS tables (part 2).

- M_2. A number of records in USDA do not have values for "Award Number." In such cases, the records may be matched by checking whether the "AwardTitle" (in UMETRICS) and the "Project Title" in USDA are similar. An example of such a match is shown in Figure 6.8.

- M_3. A record pair from UMETRICS and USDA can also be matched by comparing the individuals involved in the project.

From the above matching definition, we could infer one positive matching rule based on M_1. Specifically, for a record pair from UMETRICS and USDA tables, if the second part of "UniqueAwardNumber" in the UMETRICS record matches exactly the "Award Number" in the USDA record, then the record pair could be declared a match. One possibility at this stage was to use this positive rule and filter out all the positive matches and proceed with a smaller set for matching. We did not do that because we were not sure whether the match definition had been stabilized yet. So we decided to incorporate this rule as a part of the blocking step. Apart from M_1, the other two instructions (M_2 and M_3) in the above matching definition are not precise. For example, what does it mean to say "similar," or what if the award and project titles are an exact match, but the titles were very generic (for example, "Lab Supplies"). Clearly we cannot write them as rules, apply them over the input tables, and get the match results.

As we see here, matching definitions are written in English, which can be verbose and imprecise. Business owners often train analysts to perform matching. These analysts often gain experience over time and tune their understanding of a "match." Specifically, they do this by exploring a wide variety of examples and constantly checking with the business owners when in doubt. *This suggests that understanding a matching definition is an iterative process that involves continuous interaction with the business owners.*

UMETRICSSubAwardMatching

Unique Award Number	Address	BldgNum	City	Country	DUNS	Domestic ZipCode	EIN	Foreign Zip Code	Object Code
93.395 MSN1006 83	8701 Watertown Plank Road	NaN	Milwaukee	US	937639060	53226-05 09	390806261	NaN	3845
47.078 MSN1021 20	110 TECHNOLOGY CENTER BLDG	NaN	UNIVERSI TY PARK	US	3403953	16802-70 00	246000376	NaN	3845

Org Name	Organization ID	POBox	Period End Date	Period Start Date	Recipient Account Number	Srt Name	Srt Number	State	Str Name	Str Number	SubAward Payment Amount	Data File Year
MEDICAL COLLEGE OF WISCONSIN	007H055	NaN	2008-02-21	2008-02-21	MSN100683	NaN	NaN	WI	NaN	NaN	4114	2008
PENNSYLVANIA STATE UNIVERSITY	G067944	NaN	2007-11-05	2007-11-05	MSN102120	NaN	NaN	PA	NaN	NaN	2843.97	2008

Figure 6.4: Example rows from UMETRICS tables (part 3).

6.2.4 Subsetting and Transforming Input Tables

Recall that we were given seven tables that included information about UMETRICS and USDA awards. Specifically, UMETRICS tables included information about how the funding from different agencies was used for the research projects at UW-Madison, and the USDA table included information about funding from the USDA to research projects at UW-Madison. Even with this moderate number of tables, the matching process will be difficult if we have to include all UMETRICS and USDA tables. So we decided to subset the tables (i.e., selecting only a portion of information from the tables, perhaps even using just a subset of the tables) and apply transformations to get two tables that can be used for matching.

We used the matching document provided by the UW-UMetrics team to subset the tables and apply transformations. Specifically, we first selected three tables (as judged relevant by the domain experts, i.e., the UW-UMetrics team), we did a basic sanity check for these tables, then we checked the remaining tables to see if they contained any relevant information for matching, and finally we applied transformations to get two tables that can be used for matching purposes.

First, we selected three tables judged most relevant for matching by the UW-UMetrics team. Specifically, the document mentioned that only three tables were relevant for matching: (1) “UMETRICSAwardAggMatching,” (2) “UMETRICSEmployeeMatching,” and (3) “USDAAwardMatching.” Next, we double checked that “UniqueAwardNumber” and “Accession Number” were indeed

UMETRICSObjectCodesMatching

ObjectCode	ObjectCode Text	DatFile Year
1000	Salary Default	2008
4880	Domest Govt's Documents	2008

UMETRICSOrgUnitMatching

CampusID	SubOrgUnit	CampusName	SubOrg Unit Name	DatFile Year
UWMSN	37	University of Wisconsin - Madison	Wisconsin Collaboratory for Enhanced Learning	2011
UWMSN	85	University of Wisconsin - Madison	University Housing	2008

Figure 6.5: Example rows from UMETRICS tables (part 4).

Accession Number	Project Title	Sponsoring Agency	Funding Mechanism	Award Number	Initial Award Fiscal Year	Recipient Organization	Recipient DUNS	Project Director	Multistate Project Number	Project Number	Project Start Date	Project End Date	Project Start Fiscal Year	…	…	Financial: USDA Contracts, Grants, Coop Agmt
175763	GENETIC ORGANIZATION AND EPIGENETIC SILENCING OF MAIZE R GENES	State Agricultural Experiment Station	State Funding	NaN	NaN	SAES – UNIVERSITY OF WISCONSIN	NaN	Kermicle, J.L	NaN	WIS04059	1997-07-01	2010-09-30	1997	…	…	NaN
190977	The Changing Location and Extent of the Wildland-Urban Interface During the 1990's	State Agricultural Experiment Station	State Funding	NaN	NaN	SAES – UNIVERSITY OF WISCONSIN	NaN	Hammer, R	NaN	WIS04593	2001-10-01	2011-09-30	2002	…	…	NaN

Figure 6.6: Example rows from the USDA table.

the key columns in the "UMETRICSAwardAggMatching" and "USDAAwardMatching" tables and the "UniqueAwardNumber" was indeed a foreign key in the "UMETRICSEmployeesMatching" table with valid values and could be joined with the "UMETRICSAggAwardMatching" table. We used Magellan tools and pandas to perform these validations.

Then we checked the remaining UMETRICS tables to see if they contained any relevant information for matching. To do this, first we listed the column names for each table. Next, we found similar column names between the USDA and UMETRICS tables. Specifically, we found similar column names by manual inspection. For example, we observed that "Recipient Organization" and "Recipient DUNS" from the USDA table were similar to "OrgName" and "DUNS" in the "UMETRICSVendorMatching" table. Next, we checked to determine whether the columns with similar names included similar values. Specifically, we checked for any overlap of values and checked the distribution of values using general statistics such as mean, median, etc. Based on this check, we observed that "OrgName"and "DUNS" from the "UMETRICSVendorMatching" table did not include overlapping values with "Recipient Organization" and "Recipient DUNS," so we decided to ignore them. In this step, we primarily used pandas and custom Python scripts to check

UMETRICS		USDA	
field	*value*	*field*	*value*
UniqueAwardNumber	10.200 2008-34103-19449	Accession Number	214335
AwardTitle	DEVELOPMENT OF IPM-BASED CORN FUNGICIDE GUIDELINES FOR THE NORTH CENTRAL STATES	Project Title	Development of IPM-Based Corn Fungicide Guidelines for the North Central States
FirstTransDate	10/1/08	Award Number	2008-34103-19449
FirstTransDate	10/1/08	Project Start Date	8/15/08
		Project End Date	8/14/11
		Project Director	ESKER, PAUL

Figure 6.7: Example of matching pairs based on Award Number.

the overlap and the distribution of values. Finally, we applied transformations to the subsetted tables. Specifically, we performed three transformations: (1) projecting out UMETRICS and USDA tables to create two tables with relevant columns for matching, (2) matching the columns between the tables and renaming them with the same names, and (3) adding a key column with a unique sequence of numbers. In the following, we explain the above steps in detail.

First, as mentioned in the matching definition document, we selected the relevant tables ("UMETRICSAwardAggMatching," "USDAAwardMatching") for matching and created two new tables that included only the relevant columns in them. The schemas of the newly created tables are shown below.

```
UMETRICSProjected(UniqueAwardNumber, AwardTitle, FirstTransDate,
 LastTransDate)
USDAProjected(Award Number, Project Title, Project Start Date,
 Project End Date, Accession Number, Project Director)
```

Next, we matched the column names between the two tables and renamed them with the same name. Specifically, we matched "UniqueAwardNumber," "AwardTitle," "FirstTransDate," "LastTransDate" from the "UMETRICSProjected" table to "Award Number," "Project Title," "Project

UMETRICS		USDA	
field	*value*	*field*	*value*
UniqueAwardNumber	10.203 WIS01040	Accession Number	206746
AwardTitle	SWAMP DODDER (CUSCUTA GRONOVII) APPLIED ECOLOGY AND MANAGEMENT IN CARROT PRODUCTION	Project Title	Swamp Dodder (Cuscuta gronovii) Applied Ecology and Management in Carrot Production
FirstTransDate	10/1/07	Award Number	-
LastTransDate	12/31/08	Project Start Date	10/1/06
		Project End Date	9/30/08
		Project Director	Colquhoun, J.

Figure 6.8: Example of matching pairs based on Award Title.

Start Date," "Project End Date," respectively. We named them "AwardNumber," "AwardTitle," "FirstTransDate," "LastTransDate." We renamed "Project Director" in the "USDAProjected" table as "EmployeeName." The updated schemas are shown below.

```
UMETRICSProjected(AwardNumber, AwardTitle, FirstTransDate, LastTransDate)
USDAProjected(AwardNumber, AwardTitle, FirstTransDate, LastTransDate,
AccessionNumber, EmployeeName)
```

Then we added a new column, "EmployeeName," to the "UMETRICSProjected" table. To add this column, we joined the "UMETRICSProjected" table with the "UMETRICSEmployeesMatching" table on the "AwardNumber" and "UniqueAwardNumber" columns. There were multiple employee names for the same award in the "UMETRICSEmployeesMatching" table. Therefore, for each award, these employee names were concatenated, and each employee name was separated by the | character. Finally, we added a key column ("RecordId") to the "UMETRICSProjected" and "USDAProjected" tables. This key column included just a sequence of numbers to uniquely identify the records. The final schema of "UMETRICSProjected" and "USDAProjected" is given below.

UMETRICSProjected

RecordId	AwardNumber	AwardTitle	FirstTrans Date	LastTrans Date	Employee Name
78	00.070 58-3655-8-123	FY08 RSA TASK ORDER AGREEMENT 123	2007-10-01	2009-05-31	NaN
198	10.000 14-JV-11242309-078	Fire Risk and Fire Mitigation Zones and the WUI	2014-09-01	2016-02-29	HELMERS| DAVID P| STEWART| SUSAN I

USDAProjected

RecordId	Award Number	AwardTitle	FirstTransDate	LastTrans Date	Employee Name	Accession Number
63	NaN	Food Research	2000-07-01	2011-09-30	Yu J	186338
172	NaN	Potato Research	1998-10-01	2011-09-30	Kelling, Keith	181962

Figure 6.9: Example rows from the UMETRICSProjected and USDAProjected tables after preprocessing.

```
UMETRICSProjected(RecordId, AwardNumber, AwardTitle, FirstTransDate,
LastTransDate, EmployeeName)
USDAProjected(RecordId, AwardNumber, AwardTitle, FirstTransDate,
LastTransDate, AccessionNumber, EmployeeName)
```

The “AccessionNumber” was included in the USDAProjected table because the UW-UMetrics team required the output matches to be listed as pairs of “UniqueAwardNumber” and “AccessionNumber.” We mainly used pandas, Magellan tools, and custom Python scripts to perform the joining and transformations. A few example rows from preprocessed the “UMETRICSProjected” and “USDAProjected” tables are shown in Figure 6.9.

6.2.5 Blocking

After applying the transformations, the input tables included 1336 and 1915 records respectively. Because the individual tables were relatively small, an obvious choice would be to take a Cartesian product (of the input tables) and match the record pairs in the Cartesian product. However, to perform the matching and to evaluate the match results, we must first take a sample from this set (of record pairs in the Cartesian product) and label them. In our case, the Cartesian product of the input tables would result in 2.5M record pairs, and most of them would be non-matches. Random sampling from this set will result in very few matches. Therefore, we applied blocking to the projected input tables to remove obvious non-matching record pairs. We used the matching definition provided by the UW-UMetrics team to guide the blocking step.

First, we applied a blocking scheme to include all the record pairs that satisfy M_1. We did this because, if M_1 is indeed a positive matching rule, then all the record pairs satisfying M_1 must be included in the candidate set. Recall that M_1 specified that a record pair is a match if the second half of the string in the “AwardNumber” column of the “UMETRICSProjected” table matches exactly the “AwardNumber” column in the “USDAProjected” table. To include all the record pairs that satisfy M_1, we decided to apply an attribute equivalence blocker to these tables. An attribute equivalence blocker will include a record pair only if the value of the blocking attribute is exactly the same between the two input tables. In our case, the attribute equivalence blocker cannot be applied directly because the “AwardNumber” from “UMETRICSProjected” and “USDAProjected” cannot be compared for an exact match. Therefore, we first used a regular expression to extract the suffix of the column values in the “AwardNumber” of the “UMETRICSProjected” table and stored the result as a temporary column, “TempAwardNumber,” in the same table. Then we applied the attribute equivalence blocker using “TempAwardNumber” from the “UMETRICSProjected” table and “AwardNumber” from the “USDAProjected” table as blocking attributes to obtain a candidate set C_1. This blocking scheme resulted in a candidate set C_1. After blocking, we removed the temporary column (“TempAwardNumber”) from the “UMETRICSProjected” table.

Next, based on the matching definition M_2, we decided to include the record pairs that have similar award titles. We explored the award titles between the “USDAProjected” and “UMETRICSProjected” tables and observed that the titles were not short and that they included multiple tokens. Intuitively, two similar award titles would include at least a few overlapping tokens between them. Therefore, we decided to apply an overlap blocker over the input tables using “AwardTitle” as the blocking attribute. An overlap blocker will discard a record pair if the number of overlapping tokens is less than an overlap threshold K. To apply the overlap blocker, first, we normalized all the strings in the “AwardTitle” column by lower casing and removing special characters such as single quotation marks, double quotation marks, hash symbols, exclamation marks, round braces, curly braces, etc. We did this normalization because, when we explored the award titles, we observed that the use of letter cases and special characters in the “AwardTitle” column was inconsistent. For example, some titles were in lower case and some others were in upper case, some titles included double quotation marks and some titles did not, and so on. Next, we performed overlap blocking using a word-level tokenizer, and we set the overlap threshold to 3 after trying a few other thresholds (for example the threshold of 1 resulted in 200K record pairs, and a threshold of 7 resulted in a few hundred record pairs). Finally, after applying the overlap blocker, we obtained a candidate set C_2.

Next, from the semantics of the overlap blocker, we knew that this blocker would drop the record pairs if the number of tokens in the blocking attribute was less than the overlap threshold K (in our case, K was 3), so we explored the award titles between the two tables to see if similar titles with fewer than 3 tokens existed, and we found quite a few such title pairs. To include these record pairs, we decided to apply an overlap-coefficient-based blocking over the input tables using “AwardTitle” as the blocking attribute. For two strings X and Y, the overlap coefficient is given by the following equation:

$$overlap\_coefficient(X, Y) = \frac{|X \cap Y|}{\min(|X|, |Y|)}$$

The semantics of the overlap coefficient is similar to the overlap measure, except that in the overlap coefficient, the number of overlapping tokens is normalized by the minimum of the number of tokens in the input strings. For example, if two sets of tokens each with cardinality 2 exist and if they include same tokens, then the overlap coefficient will be 1. To apply the overlap-coefficient blocker, we performed the steps similar to those for the overlap blocker. Specifically, first we lower cased all the strings in the “AwardTitle” column, then we removed special characters such as single quotation marks, double quotation marks, hash symbols, exclamation marks, round braces, curly braces, etc., and finally, we performed overlap-coefficient blocking using a word-level tokenizer and a threshold of 0.7 (after trying a few other thresholds) to obtain a candidate set C_3.

Then we took a union of C_1, $C2$, and C_3 to obtain a consolidated candidate set C. The candidate set C contained 3177 record pairs. Finally, we checked for any potentially missing matches in C using a blocking debugger for one iteration. We observed that the top pairs returned by the blocking debugger had a low similarity, and hence we decided to stop modifying the blocking pipeline further. We primarily used Magellan tools for blocking. We also used custom Python scripts and pandas commands to preprocess the columns before applying blocking.

6.2.6 Sampling and Labeling

After the blocking step, we decided to obtain labeled record pairs from the UW-UMetrics team. Labeled record pairs are essential for selecting the best learning-based matcher and then training this matcher to predict matches in the candidate set. Sampling and labeling is not straightforward because the candidate set is skewed with relatively few matches and the match definition is still evolving. Also, because we are collaborating with an external team, we had to manage the logistics for labeling.

We began by having an email conversation with the UW-UMetrics team and then followed up with a face-to-face meeting to discuss the logistics of labeling the record pairs. Initially, we proposed to email the record pairs as a CSV file that they could download to their local machine and use tools such as MS Excel to mark the record pairs as a match or a no-match. The UW-UMetrics team stated that a mechanism that would allow multiple users to label the record pairs would be good because different members of their team could label the record pairs, thus speeding up the labeling process. Following their feedback, we decided to reuse a cloud-based labeling tool that we developed for another project (drug matching) to label the record pairs. The cloud-based tool would allow multiple users to label the record pairs with a good UI, but the tool had the constraint that only one person could label at a particular time. The UW-UMetrics team accepted that restriction and mentioned that they would discuss the matter internally and schedule the labeling.

Next, we performed the sampling and labeling steps iteratively. Our goal for this step was to obtain a sufficient number of positive matches in the labeled set. We first took a random sample of 100 record pairs from the candidate set; then we uploaded the sampled pairs into the cloud-based labeling tool and asked the UW-UMetrics team to label the record pairs. Specifically, we asked them to label each record pair with “Yes” if they matched, “No” if they did not match, and “Unsure” if they were not sure about the match.

The UW-UMetrics team trained a student to label using the tool. The student, with guidance from the senior members of the UW-UMetrics team, labeled the record pairs for matching. Meanwhile, we labeled the same set of record pairs internally based on our understanding of the match definition.

After the first set of labeled pairs, we cross-checked their labels against our labels. We observed 22 mismatched labels. Specifically, one record pair was marked as a no-match despite satisfying the matching rule, M_1, and the other 21 record pairs had similar award titles, but they were marked as a mix of match, no-match, and primarily unsures. We had a face-to-face discussion with the UW-UMetrics team about the mismatches. During our meeting, the UW-UMetrics team confirmed that the record pair satisfying M_1 must be declared as match (they also confirmed that M_1 is indeed a positive matching rule), and for others, they mentioned that, though the award titles were similar, some of them were not unique enough to be declared matches. They said that they would have a closer look at the mismatches and update the labels. After the discussion, we shared the record pairs with mismatched labels using Google Sheets, and the UW-UMetrics team updated 4 labels to “Yes” and the rest retained the original labels. After the updated labels were submitted, 15 pairs were labeled as “Yes,”, 63 labeled “No,” and 21 labeled “Unsure.”

Next, because we had only 15 positive matches, we decided to obtain more labeled pairs. We had an email conversation with the UW-UMetrics team and conveyed that they would need to label at least 100-200 more record pairs to obtain a sufficient number of positive matches. Following this, we internally decided to obtain record pairs labeled in two iterations, with 100 record pairs per iteration. As per the plan, we sampled 100 record pairs, uploaded the sampled pairs into the cloud-based labeling tool, and asked the UW-UMetrics team to label them. After they finished labeling, the labeled set included 29 pairs labeled “Yes,” 63 labeled “No,” and 8 labeled “Unsure.” Then we sampled another 100 record pairs and followed the above procedure. In this set, we obtained 24 pairs labeled "Yes," 71 labeled "No," and 5 labeled "Unsure."

Finally, we consolidated the 300 labeled pairs, and in total, 68 pairs labeled "Yes," 195 pairs "No," and 37 pairs "Unsure." Because we obtained a sufficient number of positive matches, we decided to stop seeking more labeled pairs.

After we obtained the consolidated set of labeled pairs, we decided to debug the labels because these labeled examples were to be used to train a matcher, and any errors in the labeled data would impact the predicted matches. To debug the labels, we used the leave-one-out cross-validation method. Specifically, first, we trained an ML matcher over all the labeled record pairs except one, next we predicted the label for the left-out record pair, and finally we compared the label predicted by the ML model and the label given by the UW-UMetrics team. We used random forest as the ML matcher, and we removed the unsure and the sure matches (record pairs that satisfy M_1) from the labeled data before debugging the labels. We applied the leave-one-out cross-validation method using the labeled record pairs and observed the following discrepancies.

- D_1. Record pairs were predicted matches, but labeled as a no-matches if the award titles were very similar except for the fact that the award title in the "USDAProjected" table included the suffix "NC/NRSP."

- D_2. Record pairs were predicted as matches but labeled as no-matches if the award number were different but the award titles were the same or very similar.

- D_3. Record pairs were predicted as matches but labeled a mix of matches and no-matches if the award number was missing from the "USDAProjected" table but the award titles were very similar.

Following this, we had an email conversation and a face-to-face meeting with the UW-UMetrics team to clarify the ambiguities. Specifically, we shared example record pairs (using Google Sheets) for each of the above discrepancies. During the discussion, the UW-UMetrics team (based on their domain knowledge) mentioned that for D_1, the labels should be updated as unsure. For D_2, the original labels must be retained. For D_3, the labels must be updated as matches if the transaction dates for the awards are within a difference of few years (e.g., two years). Based on the above understanding, the UW-UMetrics team updated the labeled set and provided us with it. After all the changes, the labeled set included 300 labeled pairs, 68 labeled "Yes," 200 pairs labeled as "No," and 32 pairs labeled as "Unsure."

6.2.7 Matching

After we received the updated set of labels from the UW-UMetrics team, we decided to use the labeled data to produce matches between the "UMETRICSProjected" and "USDAProjected" tables. We used the labeled set first to select the best learning-based matcher and then used the selected matcher to predict the matches in the candidate set.

6.2.7.1 Selecting a Matcher

To select the best matcher, we used Magellan. First, we removed the record pairs labeled "Unsure" and sure matches (record pairs that satisfy M_1) in the labeled set and then converted the labeled set into a set of feature vectors. We observed some missing values, and we imputed those missing values with the mean value of their respective columns.

Next, we selected the best matcher using five-fold cross-validation. We used decision tree, SVM, random forest, logistic regression, naive Bayes, and linear regression matchers to select the best one. Among the matchers, the random forest matcher performed the best but had low precision, recall, and F_1.

Then we debugged the random forest matcher using Magellan. Specifically, first, we split the labeled data into two sets, training and testing sets, then we trained the random forest matcher on the training set, debugged the matcher using the testing set, and finally, we observed that many mismatches were occurring because of award titles with different letter cases, so we decided to add more features to handle them.

Next, we added award title-related features that handled variance in letter cases and selected the best matcher again. In this iteration, the decision tree performed the best with precision at 97%, recall at 95% and F_1 at 94.7%, on average.

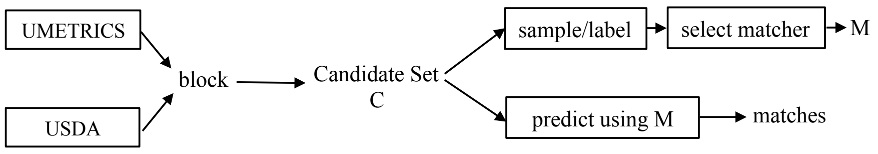

Figure 6.10: The initial EM workflow for matching UMETRICS and USDA data sets.

UMETRICS Unique Award Number	UMETRICS Award Title	USDA Award Number	USDA Award Title
00.070 58-1935-6-655	HOW DO THE SEROTYPES 4B SPECIFIC GENES OF LISTERIAMONOCYTOGENES CONTRIBUTE TO THE PATHOGENESIS OF LISTERIOSIS?		HOW DO THE SEROTYPE 4B SPECIFIC GENES OF LISTERIA MONOCYTOGENES CONTRIBUTE TO THE PATHOGENESIS OF LISTERIOSIS?
00.070 T-7-3655-121	FY07 RSA AGRICARS: POST-HARVEST PHYSIOLOGY OF POTATO TUBERS		Post-harvest physiology of potato tubers

Figure 6.11: One-one matches between UMETRICS and USDA tables.

Finally, we debugged the matcher using the decision tree matcher debugger (in Magellan) to see if the accuracy could be improved further. We tried adding more features, but they did not result in any improvement in accuracy numbers. Therefore, we decided to stop and selected the decision tree as the best matcher.

6.2.7.2 Predicting the Matches

To predict the matches, we first trained the decision tree using the feature vectors from the labeled set. Next, we removed the record pairs satisfying the positive matching rule (210 record pairs) from the candidate set. Then we converted the updated candidate set into feature vectors and imputed the missing values with the mean of their respective columns. Finally, we used the trained matcher to predict the matches and non-matches in the candidate set. The overall EM workflow is shown in Figure 6.10. In the predicted set, we obtained 807 matches. In total, we obtained 1017 matches, including the record pairs that satisfied the positive matching rule. We shared the predicted matches in a CSV file with the UW-UMetrics team and followed up with a face-to-face meeting to discuss the results.

UMETRICS Unique Award Number	UMETRICS Award Title	USDA Award Number	USDA Award Title
10.200 2008-34266-19271	BABCOCK INSTITUTE FOR INTERNATIONAL DAIRY RESEARCH AND DEVELOPMENT	2005-34266-16416	The Babcock Institute for International Dairy Research and Development
10.200 2008-34266-19271	BABCOCK INSTITUTE FOR INTERNATIONAL DAIRY RESEARCH AND DEVELOPMENT	2010-34266-20760	Babcock Institute for International Dairy Research and Development

Figure 6.12: One-many matches between UMETRICS and USDA tables.

UMETRICS Unique Award Number	UMETRICS Award Title	USDA Award Number	USDA Award Title
10.200 2005-34101-15664	THE ORGANIZATION, REGULATION, AND PERFORMANCE OF THE US FOOD SYSTEM	2001-34101-10526	The Organization, Regulation and Performance of the U.S. Food System
10.200 2006-34101-16999	FOOD SYSTEM RESEARCH GROUP: "THE ORGANIZATION, REGULATION AND PERFORMANCE OF THE US FOOD SYSTEM"	2001-34101-10526	The Organization, Regulation and Performance of the U.S. Food System

Figure 6.13: Many-one matches between UMETRICS and USDA tables.

Discussion on One-One, One-Many, and Many-One Matches: During the discussion, we observed a gap between the UW-UMetrics team's definition of a match between UMETRICS and USDA awards and our understanding of the same. Specifically, the UW-UMetrics team intended to cluster the records in the "UMETRICSProjected" and "USDAProjected" tables and then perform matching between them. However, we were matching records from the "UMETRICSProjected" table to records in the "USDAProjected" table.

Following this, we analyzed the one-one, one-many, and many-one match predictions and shared our analysis with the UW-UMetrics team. A few examples of one-one, one-many, many-one matches are shown in Figures 6.11, 6.12, and 6.13. We had another face-to-face discussion with them, and in that meeting, the UW-UMetrics team decided that they would go ahead with the definition that one record from the "UMETRICSProjected" table can match many records in the "USDAProjected" table.

6.2.8 Match Results from the Existing Solution

After we predicted the matches, we had an email conversation with the UW-UMetrics team and followed up with a face-to-face meeting to discuss the next steps. During the meeting, it was decided that the predicted matches produced by us must be compared with the matches produced by IRIS. IRIS had developed a tool that was being used to produce matches between UMETRICS and USDA. The IRIS tool was installed in a virtual desktop infrastructure (VDI), meaning that the tool was deployed on a secure remote server and the UW-UMetrics team uses a remote desktop connection to access the tool.

To compare the match results, we had to either obtain the matched data from the IRIS tool or send our match results to the UW-UMetrics team, and they would use the IRIS tool to compare the matches. After discussing the matter with the UW-UMetrics team, it was decided that the UW-UMetrics team would retrieve the IRIS matches from the VDI and use them to compare our matches.

6.2.9 Updated Match Definition and More Data

After the UW-UMetrics team obtained the matches from the IRIS tool and the predictions from us, they manually went through the match results and observed two things: (1) another positive matching rule existed using the project number and award number, and (2) the file containing the table "UMETRICSAwardAggMatching" was incomplete and it must include more records.

6.2.9.1 Updated Match Definition

While the UW-UMetrics team members went through the matches, they observed another positive rule that could be used to pull more sure matches directly from the "UMETRICSProjected" and "USDAProjected" tables. Following this, we received the following positive matching rule from the UW-UMetrics team.

Positive Matching Rule: If the award number from UMETRICS matches the project number in USDA, then the record pair is considered a match. After they shared the positive rule with us, first we checked to determine whether the ML matcher was already learning the rule from the labeled data. Specifically, we checked how many record pairs in the candidate set that satisfy the above rule were predicted as matches. We observed that 397 out of 411 of such record pairs in the candidate set were predicted as matches, which means that the matcher included most of the record pairs that satisfy the rule. Next, we checked to determine whether the blocker pipeline was discarding any record pairs that satisfy the new rule. Specifically, we checked how many record pairs in the Cartesian product of the "UMETRICSProjected" and "USDAProjected" tables that satisfy the above rule were included in the candidate set. We observed that 473 such record pairs existed in the Cartesian product, and the candidate set included 411 records, which means that the blocking pipeline was discarding a few record pairs. Finally, we decided (internally) to write a Python script capturing the new matching rule and apply it on the input tables directly to obtain the sure matches.

6.2.9.2 More Data

When the UW-UMetrics team was manually inspecting the matches produced by IRIS, they observed that some of the award numbers declared as matches were not present in the matches produced by us. Then they checked to determine whether the original table that was given to us; i.e., "UMETRICSAwardAggMatching" included those matching award numbers. They observed that the original data given to us was incomplete, missing 496 records. We had an email conversation with the UW-UMetrics team to decide whether they had to send a consolidated file including all the UMETRICS awards or just the extra records. Based on the discussion, we decided that the UW-UMetrics team would send us a CSV file with only the missing records, including all the columns as before.

6.2.10 Updated EM Pipeline

The UW-UMetrics team emailed us the extra records in a CSV file format. We had an internal discussion on how should we incorporate the updated match definition and handle the extra records. Our goal was to minimize the changes we had to make to our existing workflow. Based on the discussion, we decided that we would handle the extra records separately from the original tables, and we came up with the following procedure (see Figure 6.14)

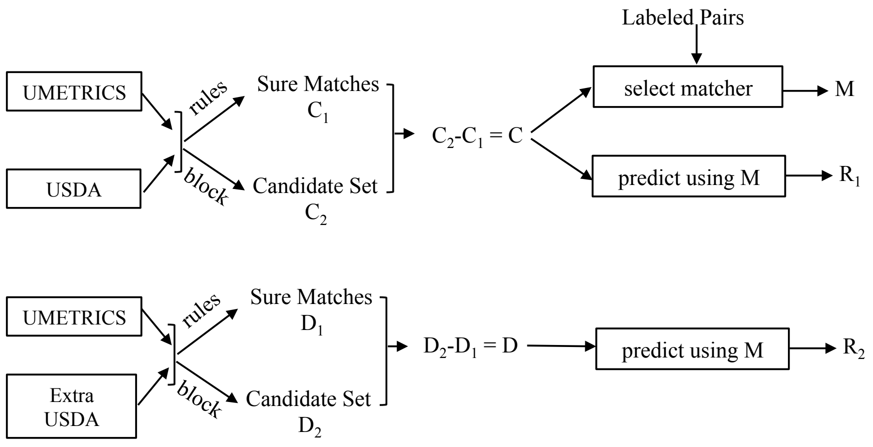

Figure 6.14: The updated EM workflow to accommodate extra data and positive matching rules.

- First, apply the sure-match rules to the original input tables. Specifically, apply the rule M_1 (from the match definition) and the positive matching rule involving award number and project number to obtain a set, and call this set C_1.
- Next, apply the blocking pipeline as before and obtain a candidate set, and call this set $C2$.
- Then remove the sure matches (C_1) from the set $C2$ to obtain a set C; this set C is what will be predicted as matches and non-matches.
- Next, use the labeled set without the sure matches to train the best matcher and predict on C, and call the resulting matches R_1.
- Then repeat the above steps for the extra records in UMETRICS and the whole USDA table until a set of sure matches (D_1) and a candidate set for prediction (D) are obtained (until step 3 in this procedure).
- Then apply the best matcher obtained using labeled data (in step 4) to the candidate set obtained from the extra records to get match predictions. Call this set R_2.
- Finally, the union of C_1, D_1, R_1, and R_2 will include the consolidated set of matches.

We applied the above procedure and obtained a consolidated set of 1137 matches. Specifically, first, we applied the sure matches rule to obtain 683 sure matches from the original input tables and

55 sure matches from the additional records. Then we applied the blocking pipeline and removed the sure matches, and this resulted in a candidate set from the original input tables including 2556 record pairs and a candidate set from the extra records including 1220 record pairs. Next, we removed the sure matches from the labeled set and selected the best matcher. The best matcher was the decision tree. Finally, we applied the decision tree matcher to the candidate sets and obtained 399 matches from the original tables and no matches from the additional records.

6.2.10.1 Accuracy Estimation

Now, with the updated predicted matches and IRIS matches, we needed to estimate the accuracy of our matcher and how well it compared with the IRIS matcher. Here, the challenge was to estimate the accuracy of the matcher over the whole input tables. Ideally, if we have the true labels for the record pairs in the Cartesian product, then we can compute the accuracy directly. However, having the true labels would mean that there was no need to do the matching in the first place. To address this, we had a face-to-face meeting with the UW-UMetrics team and decided to follow Corleone [66] approach to estimate the precision and recall for the two matchers and then use the accuracy numbers to compare them.

First, we observed that, to use the Corleone approach, both the IRIS and our predicted matches must be from the same candidate set of record pairs. Therefore, we checked for any award number-accession number pairs from the IRIS matches that were not included in our consolidated candidate set (from the original and extra records). We observed only one such record pair, and when we checked with the UW-UMetrics team, they mentioned that the award number in question was a terminated award (no longer valid) and could be discarded safely. Next, we needed to obtain a labeled set for estimation purposes. To do this, first, we took a random sample of 200 record pairs from the consolidated candidate set. Then we uploaded the sample to the cloud-based labeling tool and finally asked the UW-UMetrics team to label them as before. The UW-UMetrics team used the tool (as before) and labeled the record pairs. The labeled set included 54 matches.

Then we followed the Corleone approach to estimate precision and recall. Specifically, we used the labeled data and estimated the precision and recall for each matcher. We estimated that our matcher had precision in the range (79.6%, 86.01%) and recall in the range (96.8%, 99.42%). The IRIS matcher had precision in the range (100%, 100%) and recall in the range (52.7%, 62.07%). We observed that the precision for the IRIS matcher was high compared to ours, but its recall was low. The estimated precision of our matcher was low, but the recall was high. In other words, our matcher found more actual matches than the IRIS matcher.

Next, we asked the UW-UMetrics team to label another 200 record pairs because the interval size of the estimated precision and recall was large. As before, we randomly sampled another 200 record pairs, uploaded them to the cloud-based labeling tool, and asked the UW-UMetrics team to label the record pairs. After they finished labeling, we estimated the precision and recall again. Now, with 400 labeled pairs, our matcher had an estimated precision in the range (75.2%, 80.3%) and recall in the range (98.1%, 99.6%). The IRIS matcher had precision in the range (100%, 100%) and recall in the range (65.1%, 71.8%). Finally, we shared the results with UW-UMetrics team and discussed the final estimated precision and recall numbers. The UW-UMetrics team liked the the fact that our matcher was able to find more matches than the IRIS matcher.

6.2.11 Improving Accuracy Using Rules

Though we received a positive feedback from the UW-UMetrics team, we had an internal discussion on how to improve the precision. Specifically, the question was how we could improve the precision without affecting the recall much. Based on our discussion, we decided that if we could obtain some domain-specific rules from the experts (the UW-UMetrics team) to reduce the number of false positives, then we could apply these rules to the predictions from the learning-based matcher and thus improve the precision.

Following this, we had an email conversation with the UW-UMetrics team to understand how we could reduce the number of false positives. The UW-UMetrics team iterated over the predicted matches and then, based on their domain knowledge, they defined a negative rule (i.e., the rule will flip matches to non-matches) that could be applied after the predictions from the learning-based matcher. The rule is defined as follows. A record pair is considered a no-match if one of the following rules is satisfied.

"Unique Award Number" Pattern in UMETRICS	**Matching "Award Number" Pattern in USDA**	**Matching "Project Number" Pattern in USDA**
YYYY-#####-#####	YYYY-#####-#####	WIS#####
WIS#####	##-#####-#####	####-#####-###-##X
##-XX-########-###		WISX-YYYY-#####
##-####-####-XX		
##-####-#-###		
MSN ######		

Figure 6.15: Valid patterns in UMETRICS and USDA for applying rules.

- Award numbers from UMETRICS and USDA are "comparable," (defined below), and they are not the same.
- Award number from UMETRICS and a project number from USDA are "comparable," and they are not the same.

In the above rule, "comparable" means that the award numbers are considered (for this rule) only if they have the same pattern. For example, if the UMETRICS award number is "03-CS-112313000-031" and the USDA award number is "2001-34101-10526," then the award numbers are not comparable because they follow different patterns. Specifically, a UMETRICS award number follows the pattern "##-XX-########-###," and the USDA award number follows the pattern "YYYY-#####-#####" (here, "#" is any number, "X" is any character, and "YYYY" is four-digit year). As another example, if the UMETRICS award number is "W1S01560" and the USDA project number is "WIS04509" then they are comparable because they follow the same pattern (because the values are different this pair will be considered a non-match). Specifically, UMETRICS award number and USDA award number, follow the pattern "WIS#####." The UW-UMetrics team shared us the list of possible patterns for the award numbers from UMETRICS and USDA and project numbers from USDA (see Figure 6.15).

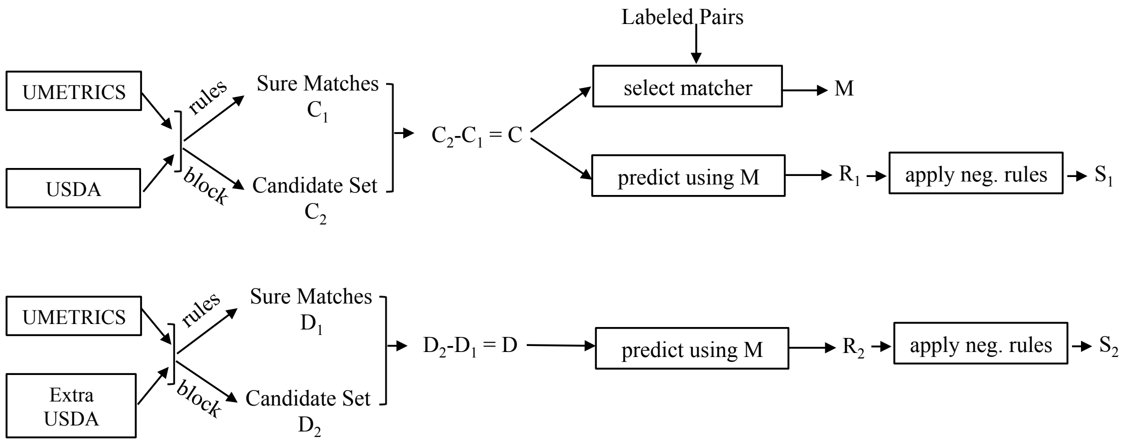

Figure 6.16: Updated EM workflow with negative rules applied to the results from the learning-based matcher.

6.2.11.1 Accuracy Estimation After Applying Rules

We applied the negative rule (provided by the UW-UMetrics team) to the matches and decided to estimate the accuracy again. Conceptually, the learning-based matcher followed by rules could be considered just another matcher like the IRIS matcher. The updated EM procedure is shown in Figure 6.16. Specifically, the matches (R_1, R_2) obtained from the machine learning-based matcher were applied with rules to get S_1 and S_2. The final set of matches was obtained by taking the union of C_1, D_1, S_1, and S_2.

We consider this new workflow a new matcher. Because the candidate set of this new matcher is same the as that of the learning-based matcher from our previous iteration, we can reuse the labeled set. We used the Corleone approach for estimating the new precision and recall. We estimated that our new matcher (decision tree followed by rules) had precision in the range (96.7%, 98.8%) and recall in the range (94.2%, 97.05%). In contrast, the learning-based matcher (without rules) had precision in the range (75.2%, 80.3%) and recall in the range (98.1%, 99.6%). The IRIS matcher had the precision in the range (100%, 100%) and recall in the range (65.1%, 71.8%). The learning-based matcher followed by rules had a significant improvement in precision with a small drop in recall.

We shared our estimated results with the UW-UMetrics team, and they were quite pleased with the results. The director of the team said this in an email conversation after we shared the results:

> *That is really stupendous news! I'm surprised to see how much you were able to raise the precision and recall.*
>
> ...
>
> *Thanks for all your brilliant work on this.*

Following this, we decided to stop improving the accuracy and shared the final set of matches with the UW-UMetrics team. Specifically, the final set contained 845 matches. We shared the results in a CSV file including the "UniqueAwardNumber"and "Accession Number" pairs for the matches.

6.2.12 Next Steps

As the next step, UW-UMetrics wanted us to package the matcher so they could move it into a protected environment (similar to the IRIS matcher) to do matching for other data slices. It is similar to moving the workflow that was developed (in the development stage) into production. This step is not straightforward and involves the following three challenges. First, the workflow is not straightforward. It involves rules at multiple places (to find sure matches and to update the predictions from the learning-based matcher) and a machine learning-based matcher. Second, the new data may be dirty, so we need to monitor the accuracy of the match results. Third, if the accuracy is not good enough, we should have a mechanism to move back to the development stage and update the workflow. Currently, we are working with the UW-UMetrics team to package the matcher and move it into their environment.

6.3 Lessons Learned

We now discuss lessons learned from using Magellan "in the wild". First, we discuss lessons learned from using Magellan in the applied economics project. Then we discuss lessons learned from other projects.

6.3.1 Lessons Learned from the Applied Economics Project

In this project we followed the Magellan how-to guide closely to solve the EM problem (as described earlier). Because Magellan integrates well with the PyData ecosystem, we used tools developed in Magellan and tools in the PyData to perform matching. We will now discuss what worked well with Magellan and what did not.

Exploring and Cleaning the Tables: We primarily used pandas-profiling (integrated with Magellan), pandas commands, and custom Python scripts to explore the input tables. We observed that the PyData ecosystem lacked proper tools for exploring and profiling tables. Specifically, the pandas-profiling tool hung when we tried to profile a table that included a few million records, and the tool included only basic features. We had to write custom Python scripts to profile the data sets. Similar to exploration tools, the PyData ecosystem lacked the right tools for cleaning purposes. We primarily used pandas commands and often Python scripts to clean the data sets.

Blocking: The tools developed as a part of Magellan served well for blocking purposes. It scaled well for the data sets had for the EM problem at hand. Also, the blocking debugger from Magellan helped to find when to stop updating the blocking pipeline. Specifically, we decided to stop modifying the blocking schemes after we observed that the top few record pairs from the debugger had low similarity scores.

Sampling and Labeling: The sampling tool in Magellan worked well for sampling. However, we observed that the standalone labeling tool in Magellan lacked an expressive UI and included only minimal features. We observed that the users wanted a web-based tool (with a good GUI) to do labeling collaboratively. Currently, we are working toward building such a labeling tool.

Matching: We used Magellan to perform supervised learning-based matching. Also, while doing the matching, we had to include triggers to update the results from the machine learning matcher. Magellan included features to support triggers, and they were easy to use. The matching debuggers in Magellan served well for this problem. However, often, we receive feedback from other users to include debuggers for more matchers and to provide expressive UIs. We are working toward developing debuggers for more matchers and improving current debuggers.

Evaluation: We used Magellan to perform the evaluation, and it worked well. The tool implemented the Corleone solution for evaluating the match results.

Overall, the tools developed as a part of Magellan and tools in the PyData ecosystem worked well for EM purposes. However, we found a few gaps. First, no proper tools for exploring and cleaning the data sets existed. Second, no good labeling tools existed for labeling in a collaborative fashion. Third, during the EM process, the existing pipeline could be updated for various reasons (e.g., more data, positive matching rules, etc.). Currently, we handle these updates manually in Magellan. We need to have tools and mechanisms to easily handle updates to existing EM pipelines. We are currently working on developing such tools to include as a part of Magellan.

6.3.2 Lessons Learned from Other Projects

We now discuss lessons learned from other projects.

Understanding the Data/Problem/Solution: This is perhaps the most important lesson we learned from the Magellan experience. It is clear that, in many cases, the user starts with a very limited understanding of the data, the problem, and the capabilities of the solution (which is Magellan in this case). First, the user may have no idea that the data is dirty (e.g., project titles containing extra strings), or that parts of the data are simply incorrect, that parts of the data are so incomplete or ambiguous that even domain experts cannot match them.

Second, the user may also think that he or she knows the problem, i.e., the "match" definition, e.g., what it means for two records to match. However, we have found that this is rarely the case in practice, and it can have serious consequences. For example, a user who thinks he or she knows the match definition will perform blocking and sampling based on that knowledge. When the user performs labeling, he or she will encounter ambiguous cases that will require the user to revise the current match definition and execute the previous steps again, thus incurring unnecessary time. Finally, the users may not be fully aware of the capabilities of the solution. For example, if the current precision is 92% from the learning-based matcher and suppose the user wants to add rules to improve the precision, can he or she use Magellan or new tools to be explored? Most often we get such queries from the users despite the elaborate documentation and examples provided.

Output Knowledge about the Problem/Data/Tools: As discussed earlier, at the start of the development stage, the user often knows very little about the problem definition, the data profile, and the tools? capabilities. he or she often gains far more knowledge about these along the way. We found that at the end of the development process, it is often highly desirable for the user to output, not just a good EM workflow, but also knowledge about the problem/data/tools.

For example, besides a good EM workflow, the user U can produce a report listing possible match definitions, the selected definition, and the reasons for selecting it. U can produce another report listing various problems with the data, how they can affect EM, actions that should be avoided, as well as actions that can be taken to fix the problems. For example, this document can say that there are many missing and dirty values in column "state", hence do not do blocking using "state" (e.g., drop all record pairs whose records disagree on "state"). Finally, the user can produce a report discussing the capabilities of the tools on the current tables and actions taken based on those (e.g., learning-based matchers do not appear to work well here for such and such reasons, and hence rule-based matching is added, to reach the desired EM accuracy).

Having such reports is tremendously useful because EM is an iterative rather than "one-shot" process. Even after an EM work flow has been pushed into production, problems often occur, which necessitates working in the development stage again to repair/fine tune the EM workflow. Having access to such reports makes this process much easier (especially in the case the user has moved on, and a new data scientist is now debugging the EM process). Also, maintaining such report will help to solve similar problems later.

Different Solutions for Different Parts of the Data: Another important observation is that the vast majority of current EM works treat the input data as of uniform quality, but in practice, this is rarely the case. Instead, the data commonly contains dirty data of varying degree, incorrect data, and incomplete data that even domain experts cannot match. It makes no sense trying to debug the system, then spending more time and money to match incorrect and incomplete data. As a result, it is important to have tools that help the user explore and understand the data, then ways to help the user "split" the data into different parts and develop different EM strategies for different parts of the data.

Support for Easy Collaboration: We found that in many EM settings there is actually a team of people wanting to work on the problem. Most often they collaborate to label a data set, debug, clean the data, etc. For example, in the case study discussed in this chapter the UW-UMetrics team collaboratively labeled and helped to debug the labeled data. However, most current tools are rudimentary in helping users to collaborate. They often require tools so that they can efficiently communicate their findings about the dataproblemsolution and converge at end (e.g., to a single match definition) As a result, it is important to develop tools that help user for easy collaboration.

More Expressive UIs: Another feedback that we have received from many users is that current UIs are too limited. More expressive UIs are highly desirable. For example, when the user performs labeling, they are limited to just labeling record pairs a common feedback that we received is provide capabilities to update the record pairs inline. As another example, some indicated that a UI that shows them a cluster of records (that are supposed to match) may help them "label" data faster than showing one record pair at a time. Perhaps more expressive UIs would make users more efficient as well.

Chapter 7

Related Work

Throughout this dissertation we have discussed related work (such as work on building EM systems, discussed in Chapter 2). We now discuss additional related work, from multiple perspectives.

7.1 Data Integration

Data integration (DI), broadly interpreted as covering all major data preparation steps such as data extraction, exploration, profiling, cleaning, matching, and merging [51]. This topic is also known as data wrangling, munging, curation, unification, fusion, preparation, and more. Over the past few decades, DI has received much attention, and tremendous progress has been made (e.g., [125, 106, 108, 85, 115, 110, 41, 78, 131, 91, 92, 38, 47, 120, 68, 119, 35, 95, 130, 96, 109, 84, 29, 59, 52, 74, 29, 32]). Today, as data science grows, DI is receiving even more attention. This is because many data science applications must first perform DI to combine the raw data from multiple sources, before analysis can be carried out to extract insights. Entity matching (EM) is a major problem in DI, and is the focus of this dissertation. The solutions developed in this dissertation can potentially be applied to other problems in DI, as we discuss in [49].

7.2 Entity Matching

Numerous EM algorithms have been proposed [37, 56]. But far fewer EM systems have been developed. We discussed these systems in Chapter 2 (see also [37]). For matching using supervised learning, some of these systems provide only a set of matchers. None provides support for 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 [44], being flexible and open source [36], 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 [58]. 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.

Several works have addressed scaling up blocking (e.g., [42, 76, 123, 21]), learning blockers [30, 46], and using crowdsourcing for blocking [66] (see [39] for a survey). As far as we know, there has been no work on debugging blocking, as we do in Magellan.

On sampling and labeling, several works have studied active sampling [116, 24, 27]. These methods however are not directly applicable in our context, where we need a representative sample in order to estimate the matching accuracy. For this purpose our work is closest to [66], which uses crowdsourcing to sample and label.

Debugging learning models has received relatively little attention, even though it is critical in practice, as this paper has demonstrated. Prior works help users build, inspect and visualize specific ML models (e.g., decision trees [23], Naive Bayes [25], SVM [33], ensemble model [121]). But they do not allow users to examine errors and inspect raw data. In this aspect, the work closest to ours is [22], which addresses iterative building and debugging of supervised learning models. The system proposed in [22] can potentially be implemented as a Magellan's tool for debugging learning-based matchers.

The notion of "open world" has been discussed in [60], 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 [31] 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.

On scaling the EM workflows, there has been several work on developing platforms for the specification, optimization, and parallel execution of directed acyclic graphs (DAGs) of operators [75, 111, 73, 65, 118, 114, 93]. But,they focus on scaling aspects of the workflow but do not address helping users providing how-to guides and tools as we do in Magellan.

On evaluating Dask performance, the work [112] provides benchmark results of Dask for a variety of different workloads under increasing scales of data set size and cluster size. In [99], the authors study the performance and applicability of Dask for scientific imaging workloads in a cluster setting. Our work is related to [99] but we focus of EM workloads on a single machine.

Several works have addressed parameter tuning problem in the context of self-tuning databases [34, 122, 71, 72]. In [87, 88]. the authors develop a system on top of hadoop to set the parameter values in the presence of skew for scientific user-defined functions. In [90] the authors perform parameter tuning for schema matching. Our work uses the technique developed in [90] for parameter tuning.

Finally, there has not been many case studies published that describe solving an end-to-end EM problem in the wild. In [40], the authors provide a technical report summarizing the techniques that they followed to match bank-related information In our work, we focus on the end-to-end steps followed to solve a EM problem and not just on the techniques used.

7.3 The PyData Ecosystem of Open-Source Data Science Tools

In the past decade, using open-source tools to do data science has received significant growing attention. The two most well-known ecosystems of such open-source tools are in Python and R. In this dissertation we focus on the Python data ecosystem, popularly known as PyData. We now discuss PyData in detail, to further motivate our decision to build Magellan into PyData.

What Do They Do? First and foremost, the PyData community has been building a variety of tools (typically released as Python packages). These tools seek to solve data problems (e.g., Web crawling, data acquisition, extraction), implement cross-cutting techniques (e.g., learning, visualization), and help users manage their work (e.g., Jupyter notebook). As of May 2018, there are 138,000+ packages available on $pypi.org$ (compared to "just" 86,000 packages in August 2016). Popular packages include NumPy (49M downloads), pandas (27.7M downloads), matplotlib (13.8M downloads), scikit-learn (20.9M downloads), jupyter (4.9M downloads), etc.

The community has also developed extensive software infrastructure to build tools, and ways to manage/package/distribute tools. Examples include nose, setuptools, $pypi.org$, anaconda, conda-forge, etc. They have also been extensively educating developers and users, using books, tutorials, conferences, etc. Recent conferences include PyData (with many conferences per year, see $pydata.org$), JupyterCon, AnacondaCon, and more. Universities also often hold many annual Data Carpentry Workshops (see $datacarpentry.org$) to train students and scientists in working with PyData. Finally, the PyData community has fostered many players (companies, non-profits, groups at universities) to work on the above issues. Examples include Anaconda Inc (formerly Continuum Analytics, which releases the popular anaconda distribution of selected PyData packages), NumFocus (a non-profit organization that supports many PyData projects), $datacarpentry.org$ (building communities teaching universal data literacy), $softwarecarpentry.org$ (teaching basic lab skills for research computing), and more.

Why Are They Successful? Our experience suggests four main reasons. The first obvious reason is that PyData tools are free and open-source, making it cheap and easy for a wide variety of users to use and customize. Many domain scientists in particular prefer open-source tools, because they are free, easy to install and use, and can better ensure transparency and reproducibility (than "blackbox" commercial software). The second reason, also somewhat obvious, is the extensive community effort to assist developers and users, as detailed earlier. The third, less obvious, reason is that PyData tools are *practical*, i.e., they often are developed to address *creators' pain points*. Other users doing the same task often have the same pain points and thus find these tools useful.

Finally, the most important reason, in our opinion, is *the conscious and extensive effort to develop an ecosystem of interoperable tools and the ease of interoperability of these tools*. As discussed earlier, solving EM problems often requires many capabilities (e.g., exploration, visualization, etc.). No single tool today can offer all such capabilities, so EM (and data science in general) is often done by using a set of tools, each offering some capabilities. For this to work, *tool interoperability is critical*, and PyData appears to do this far better than any other EM platforms, in the following ways. (a) The interactive Python environment makes it easy to interoperate: one just has to import a new tool and the tool can immediately work on existing data structures already in memory. (b) Tool creators understand the importance of interoperability and thus often consciously try to make tools easy to interoperate. (c) Much community effort has also been spent on making popular tools easy to interoperate. For example, for many years Anaconda Inc. has been selecting the most popular PyData packages (536 as of May 2018), curating and making sure that they can interoperate well, then releasing them as the popular anaconda data science platform (with over 4.5M users, as of May 2018).

What Are Their Problems? From data integration (DI) perspectives we observe several problems. First, even though PyData packages cover all major steps of DI, they are very weak in certain steps. For example, there are many outstanding packages for data acquisition, exploration (e.g., using visualization and statistics), and transformation. But until recently there are few good packages for string matching and similarity join, schema matching, and entity matching, among others.

Second, there is very little guidance on how to solve DI problems. For example, there is very little published discussion on the challenges and solutions for missing values, string matching, entity/schema matching, etc. Third, most current PyData packages do not scale to data larger than memory and to multicore/machine cluster settings (though solutions such as Dask have been proposed).

Finally, building data tools that interoperate raises many challenges, e.g., how to manage metadata/missing values/type mismatch across packages. Currently only some of these challenges have been addressed, in an ad-hoc fashion. Clearly, solving all of these problems can significantly benefit from the deep expertise of the DI research community.

Chapter 8

Discussion and Future Work

We have discussed numerous opportunities for future work throughout this dissertation. First, Magellan has unearthed many research problems, such as exploring multiple match definitions, performing collaborative EM, developing expressive UIs to perform EM efficiently, etc. These research problems can be further examined to extend Magellan with more capabilities.

Second, the production stage in Magellan should be explored further. Executing the production stage workflows across different production environments (e.g., cluster of machines, machines with GPUs and CPUs, etc.) and exploring methods to easily transition between the production and development stage are important.

Third, the how-to guides developed as a part of Magellan can be used to help determine which capabilities to add to CloudMatcher [67] to make it useful in performing EM end-to-end.

Fourth, currently we handle only the EM scenario of matching two tables. Exploring other EM scenarios (e.g., matching entity mentions in text, privacy preserving entity matching) is important (see more below).

Finally, applying the Magellan approach of solving EM problems (by providing how-to guides, developing tools for pain points, etc.) to other problem domains such as data science is another important future direction.

In addition to the above future directions, the reader may wonder if Magellan will be able to solve all different types of EM problems out there. Our goal in this dissertation is not to show that we can develop a single EM management system (EMMS) that unifies all existing EM approaches. In fact, given the wide variety of existing EM approaches (that use a wide variety of EM workflows), we suspect it would be extremely difficult to build a single unifying EMMS.

Instead, our goal is to show that (a) it is important to go beyond EM algorithms to develop EM systems, (b) current EM systems have major limitations that prevent their widespread use in practice, (c) we can develop a methodology and architecture, as exemplified by Magellan, to build what we call "EM management systems" that address these limitations, and (d) doing so also raises many novel research challenges.

Our hope is that the methodology and architecture of Magellan, as well as lessons learned building it, can be used as a "unifying template" to develop other EMMSs. We envision that each EMMS will address a set of related EM scenarios using a set of Python packages, but that the systems can seamlessly reuse a large portion of one another's code and commands. (It is important to note that we do not think each EM scenario merits its own EMMS; an EMMS can address multiple EM scenarios, as we discuss at the end of this section.) To make the above discussion more concrete, in what follows we will discuss how the methodology, architecture, and lessons of Magellan, which so far has focused on the EM scenario of matching two tables using learning and rules, can be applied to three additional EM scenarios: matching strings, linking a table into a knowledge base, and EM using iterative blocking.

8.1 Matching Strings

This is the problem of finding strings from a single given set or across two given sets that refer to the same real-world entity, e.g., "David Smith" and "Dave M. Smith". This problem is a special case of EM, but due to its restrictive setting, it has typically been studied apart from EM, and numerous string matching solutions have been developed [103, 50].

Most string matching solutions focus on developing similarity measures (e.g., edit distance, Jaccard, TF/IDF, soft TF/IDF, etc) and scaling up matching a large number of string pairs. The latter is often studied under the topic "string similarity joins" or "set similarity joins" [117, 132]. To scale, many techniques called "filtering" have been developed, such as length filtering, prefix filtering, etc. For example, length filtering states that two strings x and y match only if their lengths satisfy a constraint. Given this property, we can build an index on the length of the strings, then use this index to quickly find string pairs that can possibly match.

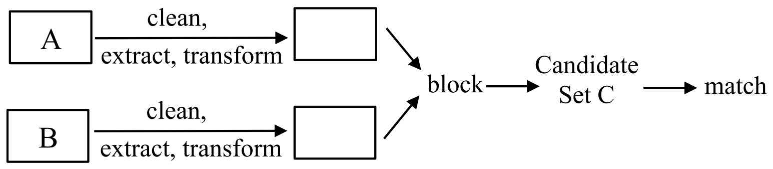

Figure 8.1: The EM workflow for the learning-based matching scenario.

Today string matching suffers from problems similar to those of EM, namely there are numerous matching algorithms but very few effective end-to-end string matching systems. In particular, many software packages exist that implement string similarity measures (e.g., SimMetrics [18], SecondString [17], Jellyfish [11], Abydos [1]), but surprisingly very few open-source packages exist that scale up these measures (Flamingo [9] is one such package). There is also no user guidance, e.g., to select a good string similarity measure and to debug the filtering and matching steps. To address these problems, we advocate building end-to-end string matching systems, and we believe that the methodology/architecture/lessons of Magellan can be applied here. Specifically,

1. First we consider a few common string matching scenarios. One such scenario is to match two large sets of strings A and B.

2. Next, we develop a how-to guide for this scenario. This guide proposes that the user matches A and B in two stages: development and production. In the development state the user tries to come up with an accurate string matching workflow. Similar to the current Magellan's workflow (see Figure 8.1), this workflow consists of cleaning/extracting/transforming, blocking, then matching (where blocking basically implements one or more filtering strategies).

3. To help the user develop this workflow, we can provide tools similar to those in Magellan. For example, we need a tool to sample sets A and B to produce two smaller sets A' and B'; we need tools to help debug the blockers and matchers; and so on.

4. To help the user execute the workflow fast in the production stage, we will develop tools that scale up steps of the workflow, on a single machine or a cluster (using Hadoop or Spark).

Since the workflow for string matching described above is relatively similar to those of the current Magellan system, we can consider extending Magellan to this string matching scenario.

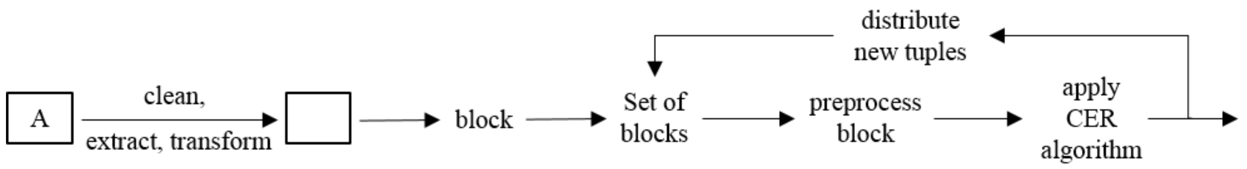

Figure 8.2: The EM workflow for the scenario of matching using iterative blocking.

8.2 Linking a Table into a Knowledge Base

We now examine the problem of linking a table into a knowledge base (KB). A KB captures information about a particular domain. It typically consists of a taxonomy of concepts (that cover the domain), a set of instances for each concept, relationships among the concepts, and domain integrity constraints. Given a table and a KB, we want to find all pairs $x, y)$ such that x is a tuple in the table and y is an instance in the KB and they refer to the same real-world entity. For example, let $A(name, phone, address, affiliation)$ be a table where each tuple describes a person. Let K be a KB that contains a set of person instances (e.g., those of concepts such as $professor$ and $student$). Then we want to link each tuple in A to the instance in K (if any) that describes the same person. A growing body of work (including some of our own [61]) has examined this EM scenario, as it arises in a growing number of applications (e.g., search, data integration, question answering, query interpretation). We believe that the current Magellan solution can be applied to this problem, but it may also need to be extended. Specifically, we can proceed as follows:

1. Each concept in the KB K is typically described using a set of attributes (e.g., "phone", "organization", etc for concept $professor$), so each instance is typically described using a set of attribute-value pairs. As such, we can extract all "person" instances from K and store them in a relational table B.

2. Our linking problem then reduces to matching tuple pairs between tables A and B, and a Magellan-like system can be applied to this problem.

3. If the above approach already produces sufficiently high EM accuracy (e.g., greater than a desired threshold), then we stop. Otherwise, we need to exploit KB-specific information to increase the accuracy. Many solutions to do this have been proposed, and we can consider implementing those solutions as extensions to the current Magellan.

   For example, in a recent work [61] we have developed the following solution. Suppose the EM pipeline so far has predicted that a tuple x matches an instance y. To verify, classify x into a node C in the taxonomy (e.g., "Academic Personnel"), then check if y is an instance of a concept in the subtree rooted at C. If not, then we can conclude that x does not match y. We can implement this solution (as well as others) as extensions to the Magellan's pipeline considered so far.

Building on the above ideas, we propose to develop a table-to-KB EM management system. First, we will develop a how-to guide based on Steps 1-3 described above. This guide will subsume the how-to guide of the current Magellan, but significantly extend it. The new EM workflow will start with the current EM workflow of Magellan (which consists of cleaning/extracting/transforming, blocking, then matching), but extend it with steps that exploit KB-specific information to improve accuracy (as described above). We will still distinguish the development stage and the production stage. In the development stage the user can use all Magellan tools, but we will also develop tools specifically to help exploit KB-specific information.

While it is possible to extend the current Magellan to handle linking a table into a KB, we believe it is better to build this as a separate (though related) table-to-KB EM management system that addresses just this table-to-KB EM scenario. First, this system will already be quite complex. So separating it from the current Magellan makes it simpler to manage conceptually and implementation-wise. Second, and more importantly, we suspect that a generic table-to-KB solution may not work well for all domains. For example, a solution that works well for social media may not work well for biomedicine, and vice versa. Thus, we may need to have a generic table-to-KB system and ways to help users customize this system to each domain of interest. This generic table-to-KB system can be implemented as a set of Python packages (which can rely quite heavily on the current Magellan packages).

8.3 EM Using Iterative Blocking

So far Magellan has considered EM scenarios that cleanly separate the blocking and matching steps. However, some EM scenarios, such as iterative blocking [126], interleave the two. The iterative blocking approach takes as input a table of tuples A and outputs a partition of A into groups such that all tuples within a group match and tuples across groups do not match. Briefly, this approach (see Figure ibworkflow) works as follows.

1. First, we use multiple blocking heuristics to partition A into multiple blocks. For example, one heuristic partitions A based on "zipcode"; another heuristic partitions A based on "affiliation". Note that a tuple from A can end up in multiple blocks.

2. Next, for each block D, we preprocess it, then apply a CER (i.e., "core entity resolution") algorithm to partition D into groups of matching tuples. Each such group forms a "super" tuple.

3. Next, we send the newly created "super" tuples to all the other blocks. The intuition is that if a block B_1 has two tuples s and t, then by comparing them in isolation, we may not be able to decide that they match. However, if we have just applied the CER algorithm to a different block B_2 and determined that s matches r, then we can send the super tuple (s, r) to B_1 and this time with the information from r, we may be able to decide that (s, r) matches t (and thus s matches t).

4. Then we repeat Steps 2-3 again, until no new super tuples are created. At this point we can examine the groups in the blocks to produce the final partition of A.

As described, in principle we can extend the current Magellan solution to incorporate this approach. First, the current Magellan assumes blocking will produce a set of candidate tuple pairs. We can extend blocking to produce a set of blocks (each of which is a set of tuples), to handle Step 1 (described above). Second, we can encapsulate Steps 2-3 in a matcher, which takes as input a set of blocks and outputs a final partition of table A. As such, the workflow in Figure 8.2 reduces to the typical workflow of current Magellan shown in Figure 3.3.

In practice, we do not believe extending the current Magellan is a good idea. The iterative blocking approach is sufficiently different from the current EM approaches considered in the current Magellan system (which clearly separates out the blocking and matching steps) that it is best to place it in a new EM management system. However, we should still be able to apply the same methodology/architecture/lessons in building Magellan to building this new EMMS. For example, we need to start with a concrete how-to guide that gives step-by-step instructions to the user, then consider how to reuse Magellan's tools or build new tools to help the user do these steps.

For example, at the start, how do we know which blocking heuristics to use and how to debug these heuristics? Another important decision (in the development stage) is to select and debug the CER algorithm. The paper [126] describes an elegant iterative blocking framework. But this framework assumes a set of blocking heuristics and a CER algorithm have already been specified. The new EMMS should help the user make these decisions, which can have a great effect on the ultimate accuracy of the EM process. And in helping the user make these decisions, the new EMMS can reuse many tools provided by the current Magellan. For example, the Magellan tool to debug a blocker (described in Section 3.2.2) can also be used here to debug and find out which set of blocking heuristics to use. Finally, we note that the iterative blocking algorithm works in a way that is similar to the way many EM-by-clustering algorithms work. Thus, when we build a clustering-based EMMS, we can also consider whether that EMMS can also naturally cover the iterative blocking algorithm.

8.4 How Many EMMSs Do We Need?

The above discussion may give the impression that each EM scenario merits its own EMMS. We do not believe this should be the case. Instead, if a set of EM scenarios are naturally related, they all should be addressed in a single EMMS. For example, the current Magellan can naturally handle EM scenarios that use supervised learning, rules, and a combination of both. (Note that each of these is actually a "group" of EM scenarios. For example, there are EM scenarios using supervised learning that aim for high precision, high recall, high F-1, etc.) As another example, many clustering-based EM scenarios follow sufficiently similar algorithms that they should be grouped into a single EMMS. And this EMMS may be able to incorporate the iterative blocking scenario described earlier as well.

At the moment we do not yet know how many EMMSs we will ultimately need to cover most common EM scenarios. But we expect that over time, as we attempt to extend Magellan or build new EMMSs to cover new EM scenarios, this situation will become clearer. Further, as discussed earlier, we believe that the methodology, architecture, and lessons of Magellan can be applied to build these EMMSs. Finally, even though this chapter has focused on EM, we believe that this methodology/architecture/lessons may also carry over to building systems that manage other kinds of problems, such as schema matching, IE, and data cleaning.

Chapter 9

Conclusions

In this dissertation we have argued that significantly more attention should be paid to building EM systems. We then proposed Magellan a new kind of EM systems, which is novel in several important aspects: how-to guides, tools to support the entire EM pipeline, tight integration with the PyData eco-system, open world vs. closed world systems, and easy access to an interactive script environment.

We have shown that realizing the above novelties raises major challenges in developing effective how-to guides, developing tools to address the pain points of the guides, and designing the system to be "open world", in that it can easily interoperate with other systems and tools in the Python data science ecosystem.

We have discussed how we addressed these challenges, built, and open sourced Magellan. As far as we can tell, Magellan is the most comprehensive open-source EM system today (August 2018), in terms of the number of features it supports. Magellan has been successfully used in several domain science projects in academia and projects in industry. We have described these "in the wild" experience with Magellan, as well as extensive experiments in controlled settings. Finally, we have discussed lessons learned and many possible future research directions. Beside concrete contributions, this dissertation also introduces a new template of research, system development, and education for EM, with many potential impacts.

Bibliography

[1] Abydos. https://github.com/chrislit/abydos.

[2] American family and uw-madison team up in data science. http://ls.wisc.edu/news/american-family-uw-madison-team-up-in-data-science.

[3] Apache spark ecosystem. https://databricks.com/spark/about/.

[4] BigGorilla: An Open-source Data Integration and Data Preparation Ecosystem: `https://recruit-holdings.com/news_data/release/2017/0630_7890.html`.

[5] CS 838: Data Science: Principles, Algorithms, and Applications `https://sites.google.com/site/anhaidgroup/courses/cs-838-spring-2017/project-description/stage-3`.

[6] Cython implementation of the toolz package. https://github.com/pytoolz/cytoolz.

[7] Emabarrassingly parallel for loops. https://pypi.org/project/joblib/.

[8] Fast non-standard data structures for python. https://kmike.ru/python-data-structures.

[9] Flamingo. http://flamingo.ics.uci.edu/.

[10] Institute for research on innovation & science. https://iris.isr.umich.edu/research-data.

[11] Jellyfish. https://github.com/jamesturk/jellyfish.

[12] Launching parallel tasks. https://docs.python.org/3/library/concurrent.futures.html.

[13] Magellan home page `https://sites.google.com/site/anhaidgroup/projects/magellan`.

[14] Multiprocessing module in python. https://docs.python.org/3/library/multiprocessing.html.

[15] pandas-profiling. https://github.com/pandas-profiling/pandas-profiling.

[16] Python interface to apache spark. https://pypi.org/project/pyspark/.

[17] SecondString. https://github.com/TeamCohen/secondstring.

[18] SimMetrics. https://github.com/Simmetrics/simmetrics.

[19] Universities: Measuring the impacts of research on innovation, competitiveness, and science. https://www.btaa.org/research/umetrics.

[20] A utilty functions for iterators, functions and dictionaries. https://github.com/pytoolz/toolz.

[21] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman. Fuzzy joins using MapReduce. ICDE, 2012.

[22] S. Amershi, M. Chickering, S. M. Drucker, B. Lee, P. Simard, and J. Suh. Modeltracker: Redesigning performance analysis tools for machine learning. CHI, 2015.

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

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

[25] B. Becker, R. Kohavi, and D. Sommerfield. Visualizing the simple Bayesian classifier. In *Information Visualization in Data Mining and Knowledge Discovery*, 2002.

[26] S. Behnel et al. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011.

[27] K. Bellare, S. Iyengar, A. G. Parameswaran, and V. Rastogi. Active sampling for entity matching. KDD, 2012.

[28] M. Bernstein et al. MetaSRA: normalized human sample-specific metadata for the sequence read archive. *Bioinformatics*, 33(18):2914–2923, 2017.

[29] P. A. Bernstein and L. M. Haas. Information integration in the enterprise. *Commun. ACM*, 51(9):72–79, 2008.

[30] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. ICDM, 2006.

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

[32] M. J. Cafarella, A. Y. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *PVLDB*, 1(1):538–549, 2008.

[33] D. Caragea, D. Cook, and V. Honavar. Gaining insights into support vector machine pattern classifiers using projection-based tour methods. KDD, 2001.

[34] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 1–10, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[35] C. Chen et al. Biggorilla: An open-source ecosystem for data preparation and integration. In *IEEE Data Eng. Bulleting. Special Issue on Data Integration*, 2018.

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

[37] P. Christen. *Data Matching*. Springer, 2012.

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

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

[40] F. W. Christopher-Johannes Schild, Simone Schultz. Linking deutsche bundesbank company data using machine-learning-based classification. 2017.

[41] X. Chu et al. Distributed data deduplication. In *PVLDB*, 2016.

[42] X. Chu, I. F. Ilyas, and P. Koutris. Distributed data deduplication. *PVLDB*, 9(11):864–875, 2016.

[43] W. Cohen. A mini-course on record linkage and matching, 2004. `http://www.cs.cmu.edu/~wcohen`.

[44] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: A commodity data cleaning system. SIGMOD, 2013.

[45] S. Das et al. The Magellan data repository. `https://sites.google.com/site/anhaidgroup/projects/data`.

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

[47] D. Deng et al. The data civilizer system. In *CIDR*, 2017.

[48] A. Doan et al. Human-in-the-loop challenges for entity matching: A midterm report. In *HILDA*, 2017.

[49] A. Doan et al. Toward a system building agenda for data integration and cleaning. In *IEEE Data Engineering Bulletin, Special Issue on Data Integration (to appear)*, 2018.

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

[51] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[52] X. L. Dong and D. Srivastava. *Big Data Integration*. Synthesis Lectures on Data Management. Morgan & Claypool, 2015.

[53] M. Ebraheem et al. DeepER–deep entity resolution. *arXiv preprint arXiv:1710.00597*, 2017.

[54] B. H. Elan Segarra and Others. Matching overview and examples. https://goog.gl/6rPboz.

[55] A. Elmagarmid, P. Ipeirotis, and V. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

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

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

[58] M. Fortini, M. Scannapieco, L. Tosco, and T. Tuoto. Towards an open source toolkit for building record linkage workflows. In *In Proc. of the SIGMOD Workshop on Information Quality in Information Systems*, 2006.

[59] M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.

[60] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. SIGMOD, 2011.

[61] A. Gattani et al. Entity extraction, linking, classification, and tagging for social media: A Wikipedia-based approach. *PVLDB*, 6(11):1126–1137, 2013.

[62] C. Ge et al. Private exploration primitives for data cleaning. *arXiv preprint arXiv:1712.10266*, 2017.

[63] L. Getoor and R. Miller. Data and metadata alignment, 2007. Tutorial, the Alberto Mendelzon Workshop on the Foundations of Databases and the Web.

[64] M. Goetz. The forrester wave: Data quality solutions. In *Forrester Quarterly Report*, 2015.

[65] 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.

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

[67] Y. Govind et al. Cloudmatcher: A cloud/crowd service for entity matching. In *BIGDAS*, 2017.

[68] J. Heer et al. Predictive interaction for data transformation. In *CIDR*, 2015.

[69] M. A. Hernandez and S. J. Stolfo. The merge/purge problem for large databases. In *Proc. of the SIGMOD Conf.*, pages 127–138, 1995.

[70] M. A. Hernández and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2:9–37, 1998.

[71] H. Herodotou and S. Babu. Automated sql tuning through trial and (sometimes) error. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pages 12:1–12:6, New York, NY, USA, 2009. ACM.

[72] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *In CIDR*, pages 261–272, 2011.

[73] 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.

[74] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

[75] 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.

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

[77] P. Konda et al. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[78] P. Konda et al. Magellan: Toward building entity matching management systems. In *VLDB*, 2016.

[79] P. Konda et al. Magellan: Toward building entity matching management systems over data science stacks. *PVLDB*, 9(13):1581–1584, 2016.

[80] P. Konda et al. Performing entity matching end to end: A case study. 2016. Technical Report, `http://www.cs.wisc.edu/~anhai/papers/umetrics-tr.pdf`.

[81] P. Konda et al. Toward building end-to-end entity matching management systems. In *SIGMOD RECORD*, 2018.

[82] N. Koudas. Special issue on data quality. *IEEE Data Engineering Bulletin*, 29(2), 2006.

[83] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: Similarity measures and algorithms, 2006. Tutorial, the ACM SIGMOD Conference.

[84] S. Krishnan et al. PALM: machine learning explanations for iterative debugging. In *HILDA*, 2017.

[85] S. Kruse et al. Efficient discovery of approximate dependencies. In *PVLDB*, 2018.

[86] V. Kumar et al. Numpy/scipy with intel mkl and intel compilers. https://software.intel.com/en-us/articles/numpyscipy-with-intel-mkl, 2017.

[87] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 75–86, New York, NY, USA, 2010. ACM.

[88] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[89] E. LaRose et al. Entity matching using Magellan: Mapping drug reference tables. In *AMIA Joint Summit*, 2017.

[90] Y. Lee, M. Sayyadian, A. Doan, and A. S. Rosenthal. etuner: Tuning schema matching software using synthetic scenarios. *The VLDB Journal*, 16(1):97–122, Jan. 2007.

[91] G. Li. Human-in-the-loop data integration. In *PVLDB*, 2017.

[92] G. Li et al. Crowdsourced data management: A survey. In *ICDE*, 2017.

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

[94] H. Li et al. Matchcatcher: A debugger for blocking in entity matching. In *EDBT*, 2018.

[95] C. Lockard et al. CERES: distantly supervised relation extraction from the semi-structured web. In *CoRR*, 2018.

[96] A. Marcus et al. Crowdsourced data management: Industry and academic perspectives. In *Foundations and Trends in Databases*, 2015.

[97] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of the ACM Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 169–178, 2000.

[98] W. McKinney. pandas: a foundational python library for data analysis and statistics. In *PyHPC*, 2011.

[99] P. Mehta, S. Dorkenwald, D. Zhao, T. Kaftan, A. Cheung, M. Balazinska, A. Rokem, A. Connolly, J. Vanderplas, and Y. AlSayyad. Comparative evaluation of big-data systems on scientific image analytics workloads. *Proc. VLDB Endow.*, 10(11):1226–1237, Aug. 2017.

[100] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, Jan. 2016.

[101] S. Mudgal et al. Deep learning for entity matching: A design space exploration. In *SIGMOD*, 2018.

[102] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection (Synthesis Lectures on Data Management)*. Morgan & Claypool, 2010. M. Tamer Ozsu (editor).

[103] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, Mar. 2001.

[104] H. B. Newcombe, J. M. Kennedy, S. Axford, and A. James. Automatic linkage of vital records. *Science*, 130(3381):954–959, 1959.

[105] F. Panahi et al. Towards interactive debugging of rule-based entity matching. In *EDBT*, 2017.

[106] O. Papaemmanouil et al. Interactive data exploration via machine learning models. In *IEEE Data Engineering Bulletin*, 2016.

[107] P. Pessig. Entity matching using Magellan - Matching drug reference tables. In CPCP Retreat 2017. `http://cpcp.wisc.edu/resources/cpcp-2017-retreat-entity-matching`.

[108] R. Pienta et al. Visual graph query construction and refinement. In *SIGMOD*, 2017.

[109] F. Psallidas et al. Smoke: Fine-grained lineage at interactive speed. In *PVLDB*, 2018.

[110] T. Rekatsinas et al. Holoclean: Holistic data repairs with probabilistic inference. In *PVLDB*, 2017.

[111] 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.

[112] M. Rocklin. Dask benchmarks. https://matthewrocklin.com/blog/work/2017/07/03/scaling/.

[113] M. Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, number 130-136. Citeseer, 2015.

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

[115] S. W. Sadiq et al. Data quality: The role of empiricism. In *SIGMOD Record*, 2017.

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

[117] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 743–754, New York, NY, USA, 2004. ACM.

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

[119] R. Singh et al. Synthesizing entity matching rules by examples. In *PVLDB*, 2017.

[120] M. Stonebraker et al. Data curation at scale: The Data Tamer system. In *CIDR*, 2013.

[121] J. Talbot, B. Lee, A. Kapoor, and D. Tan. Ensemblematrix: Interactive visualization to support machine learning with multiple classifiers. CHI, 2009.

[122] R. Thonangi, V. Thummala, and S. Babu. Finding good configurations in high-dimensional spaces: Doing more with less. *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, pages 1–10, 2008.

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

[124] S. v. d. Walt, S. C. Colbert, and G. Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, Mar. 2011.

[125] X. Wang et al. Koko: A system for scalable semantic querying of text. In *VLDB*, 2018.

[126] S. E. Whang et al. Entity resolution with iterative blocking. SIGMOD, 2009.

[127] W. E. Winkler. Improved decision rules in the Fellegi-Sunter model of record linkage, 1993. Technical Report, Statistical Research Report Series RR93/12, U.S. Bureau of the Census.

[128] W. E. Winkler. The state of record linkage and current research problems, 1999. Technical Report, Statistical Research Report Series RR99/04, U.S. Bureau of Census.

[129] W. E. Winkler. Methods for record linkage and Bayesian networks, 2002. Technical Report, Statistical Research Report Series RRS2002/05, U.S. Bureau of the Census.

[130] D. Xin et al. Accelerating human-in-the-loop machine learning: Challenges and opportunities. In *CoRR*, 2018.

[131] M. Yu et al. String similarity search and join: a survey. In *Frontiers Comput. Sci.*, 2016.

[132] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, pages 1–19, 2015.

[133] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.