CLOUDMATCHER: TOWARD A CLOUD SERVICE FOR ENTITY MATCHING

by

Yash Govind

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2020

Date of final oral examination: 02/20/2020

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

AnHai Doan, Professor, Computer Sciences, UW-Madison
Paraschos Koutris, Assistant Professor, Computer Sciences, UW-Madison
Theodoros Rekatsinas, Assistant Professor, Computer Sciences, UW-Madison
Sadhana Puntambekar, Professor, Education Psychology, UW-Madison

To my dear family

ACKNOWLEDGMENTS

First, I would like to express my deepest and sincere gratitude to my advisor, AnHai Doan. His guidance, encouragement, inspiration, patience, and support with me during the four years I have been a Ph.D. student with him have been an amazing gift. As an academic advisor, he taught me how to look at the bigger picture, how to systematically attack the problem, communicate, and present the problem. His honest and occasionally tough feedback over the years, high research standards helped me to constantly improve myself and conduct rigorous research. It has been a wonderful experience working with AnHai and I am very fortunate to have him as my advisor.

Next, I would like to thank my thesis committee members, Professors Paraschos Koutris, Theodoros Rekatsinas, and Sadhana Puntambekar for their invaluable comments and advice. I wish I had more time to collaborate and work with Paris and Theo as they do such cool work in the database research. I want to thank Paris for guiding and helping me with database theory material when I was preparing for my Ph.D. qualifying examination. Beyond the committee, I have been very fortunate to work with Sadhana Puntambekar on the VidyaMap project from 2015-16. I learned how to build concept graphs, knowledge representation for elementary school science subjects.

Many other faculty members have guided me during graduate school. In particular, I would like to thank Professors Aditya Akella, Remzi Arpaci-Dusseau, and Jeff Naughton for their tremendous support, guidance, and writing recommendation letters to graduate school for my Ph.D. application at UW-Madison. I want to take this opportunity and express my sincere gratitude to Professor Remzi Arpaci-Dusseau who motivated me to apply to UW-Madison, supported my interest in graduate school, mentored me in the first year and helped me think through what I want from the graduate school.

Many thanks must also go to my collaborators, friends, and many wonderful people in the database group throughout my graduate study: Adel Ardalan, Mukilan Ashok, Jeff Ballard, Kaushik Chandrasekhar, Sanjib Das, Shaleen Deep, Harshad Deshmukh, Ali Hitawala, Pradap Konda, Han Li, Sidharth Mudgal, Palani Nagarajan, Derek Paulsen, Erik Paulson, Paul Suganthan, Amanpreet Singh, Aravind Soundarajan, and Haojun Zhang. I have benefited tremendously from the discussions, assistance, and feedback from them. Thank you, Paul, for your guidance on the project and helping us understand the Falcon and Corleone work which forms the basis of CloudMatcher. Thank you, Erik, for being a great friend and a collaborator during the initial stages of CloudMatcher. A big thanks to Palani, Aravind, Ali, and Mukilan for contributing to the development of the CloudMatcher project. Finally, a big thank you to Derek Paulsen for being a collaborator in the last project of my thesis and having several discussions with me on the project.

From industry, I want to thank Glenn Fung, Marshal Carter, and Mingju Sun from American Family Insurance Inc. for allowing me to deploy and evaluate CloudMatcher on real-world data. My experience under the supervision of Mingju Sun and Glenn Fung was amazing and a perfect balance of machine learning research and data science engineering. I would also like to extend my thanks to the Informatica team, especially Rajive Dhar, Darshan Joshi, Syed Awez, Venkat Amirisetty, Jerry Raj, Dave Borean, and Joseph Bracken for being flexible and patient and providing a lot of support to help me finish my Ph.D.

I want to thank all the staff and administrative members of the Computer Sciences department who have helped me in some ways throughout the graduate program. Special thanks to Angela Thorp was being a great friend and an awesome graduate coordinator to work with. Others include Hilary Heffley and SueM.

A special thanks to my Madison friends and family: Aashrith, Nidhi-Raunak, Sachin, Swapnil-Sukriti, and many others who added the fun-factor to graduate school and made the last 6 years interesting.

I would like to thank my parents Dinesh Govind and Dr. Manju Mathur (Ph.D.) from the bottom of my heart for their patience, encouragement, support and unconditional love throughout my life and especially during the degree program. The thought of doing a Ph.D. after working for

7 years was a bit daunting but then I looked at my mom who did her Ph.D. when I was a naughty teenager and this was enough to get going. Thank you for being there - always!

I would next like to thank my brother Harsh Govind for being part of my foundation. For all of the advice and wise words you have provided me over the last several years - thank you for being my rock when I needed it the most and for supporting me through every decision I made. A big thanks to my sister-in-law Tanushree Govind and my lovely niece Vanshika Govind for their love and support.

Next, I would like to thank my wife's parents Ashok Godwal and Poonam for their love, blessings, prayers, and support towards my education. My thanks to my wife's sister Megha Godwal and brother Mohit Godwal for their love, encouragement, and support during my studies.

My wife Neha Godwal deserves special recognition for her love and constant support, for countless late nights and early mornings, and for being there for me at all times. Thank you for being my go-to person, my muse, personal editor - proofreader, sounding board, my favorite chef, an awesome mother to our son Ayaansh Govind and whatnot. But most of all, thank you for being my best friend. I also want to thank my one-year-old son Ayaansh Govind who has made his best effort to not disturb me while I am working on my thesis and have loved me for whatever time I have spent with him in the last one year.

Finally, I would like to thank God ($guru-maharaj-ji$) for giving me the strength to complete the task I started a couple of years ago.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

This dissertation studies entity matching (EM), which finds data instances that refer to the same real-world entity. This problem has been a long-standing challenge in data management and will become even more important as data-driven applications proliferate. As a result, it has been studied intensively over the past several decades, by the database, AI, KDD, and WWW communities, among others. However, the vast majority of work on EM has focused on developing algorithmic solutions. Few if any current works have focused on developing end-to-end EM systems and evaluating them in real-world settings.

To address the above problems, in the past few years I have been building CloudMatcher a cloud service for EM. I envision CloudMatcher to be a fast, easy-to-use, scalable, and highly available service on the Web. Specifically, to use this service, a user simply needs to go to CloudMatcher's Web site, uploads two tables to be matched, perform some basic pre-processing, then push a button. CloudMatcher will perform EM end to end. To do so, it can either use a set of users to label tuple pairs or use crowd workers on Amazon's Mechanical Turk (or some other crowdsourcing platform such as CrowdFlower, etc.) for labeling (as matched / no-matched). The user just has to pay for the labeling if he or she decides to use Mechanical Turk. In the end, CloudMatcher will return the desired matches.

In this dissertation, I make the following contributions. First, I design scalable solution architecture for CloudMatcher such that it can do EM tasks at large scale, can run multiple EM workflows simultaneously, and can support multi-tenants. Over the past 3 years, I have contributed as a lead developer to developing an industrial-strength cloud-based EM service. The CloudMatcher

codebase is now 47K LOC involving 7 contributors. It has been used at UW-Madison in domain sciences, in a data science class at UW-Madison by 70 students, and at several organizations to do EM.

Second, to evaluate the system, I have worked with real users and have used the system to do EM tasks for real-world scenarios. I provide an extensive set of experiments that demonstrate that CloudMatcher can effectively perform EM on a broad variety of datasets from real-world users. Specifically, I have evaluated CloudMatcher for end-to-end matching tasks on two domain science projects at UW-Madison and 11 other projects at enterprise customers. In the summer of 2018, I also deployed the system at American Family Insurance.

Third, I show that breaking the CloudMatcher workflow into a set of basic services then implementing those benefits many domain scientists and EM users. I develop a solution where the users can mix and match the basic services and compose them to flexibly build more EM workflows.

Finally, I focus on scaling up the blocking rule execution for the entity matching task, designing a Spark-based solution and show how the execution of blocking rules scale to tables with 10M tuples, among others. As far as I know, this is the first academic work that has designed, implemented, and evaluated an end-to-end industrial-strength EM system. Parts of this work have been published at academic conferences and commercialized.

Chapter 1

Introduction

This dissertation studies entity matching (EM), which finds data instances that refer to the same real-world entity. This chapter begins by defining the entity matching problem in Section 1.1. It then discusses the prior work on EM and its limitations in Section 1.2. Next, it describes the contributions of my thesis work, as embodied by the CloudMatcher system, in Section 1.3. The chapter concludes with a roadmap to the rest of the dissertation in Section 1.5

1.1 Entity Matching

Entity matching (EM) finds data instances that refer to the same real-world entity. For example, the two tables in Figure 1.1 shows an example of matching two tables A and B that contain "person" information. The EM task here is to find tuple pairs across A and B that refer to the same real-world person. In this example, there are three matching tuple pairs connected with arrows. Matching tuple pairs are often referred to as *matches*, and variations of this problem are known as record linkage, entity resolution, reference reconciliation, deduplication, etc. This problem has been a

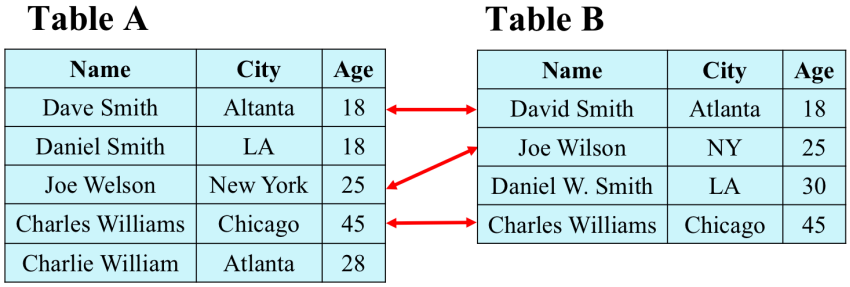


Figure 1.1: An EM example.

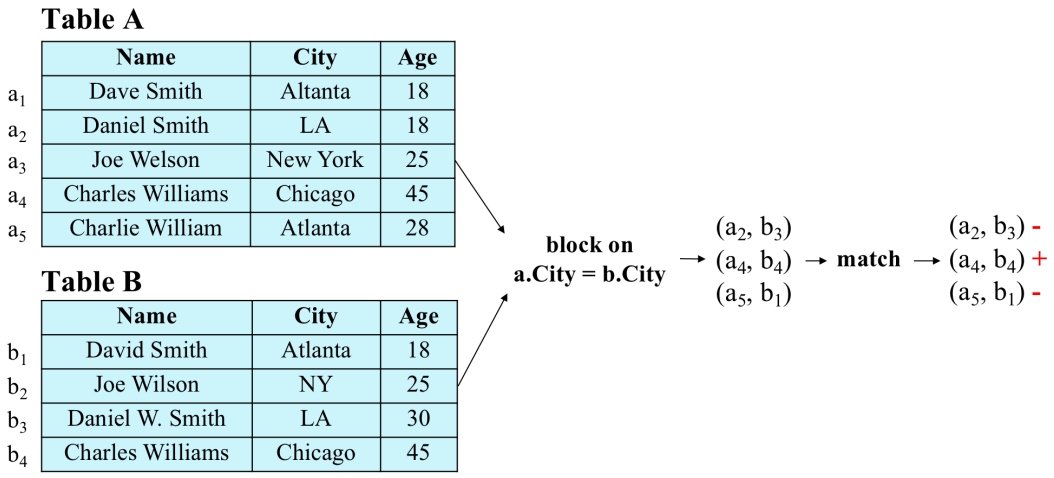


Figure 1.2: An entity matching workflow.

long-standing challenge in data management [29, 47] and will become even more important as data-driven applications proliferate.

Many EM variations exist, such as matching across two tables [50], matching within a single table [47], matching mentions in text documents into a knowledge base [49], matching XML data [93], etc. CloudMatcher consider the problem of matching across two tables, specifically, given two tables A and B, find all tuple pairs $(a \in A, b \in B)$ that refer to the same real-world entity (see Figures 1.1). This problem setting is very common in practice. Our solution, however, can also be applied to two other common settings: matching tuples within a single table (known as *deduplication*), and matching a set of tuple pairs.

Most current EM solutions consist of two important steps: blocking and matching. Figure 1.2 shows an example of the typical EM workflow with blocking and matching.

1.1.1 Blocking

The blocking step applies a heuristic to remove tuple pairs judged obviously not matched (i.e., "blocking" these tuple pairs from further consideration). In the Figure 1.2, by observing the data we choose to block on the attribute "City" to consider only pairs that have the same city value for matching. This gives us three pairs of potential matches, which form a *candidate set*.

Blocking is necessary because matching all tuple pairs in the Cartesian product of the two input tables A and B would be too expensive, e.g., if each table has 100K tuples, $A \times B$ would have 10B tuple pairs. Hence, we need a way to quickly remove as many obviously non-matched tuple pairs as possible, before applying the time-consuming matching step to the remaining tuple pairs.

Blocking is typically done by using the blocking heuristics to enumerate only those tuple pairs judged possibly matched. For example, given the above heuristic about disagreeing city, we can build an inverted index over Table B, such that given a city, the index will return all tuples in B with that city value. Next, given a tuple in Table A, we can consult the index to find only those tuples in Table B that share the same city (e.g., LA), then enumerate only those tuple pairs. The pairs that survive the blocking step constitute the set of potential matches, and we refer to it as the *candidate set*.

Over the past few decades blocking has received much attention and numerous solutions have been proposed focusing on accuracy and scaling up blockers [96, 63, 46, 76, 50, 29, 37, 79, 80, 76] (see [30, 81] for a survey and [29, 47] for an extensive discussion).

1.1.2 Matching

Once the blocking step is completed, we get a candidate set C of potential matching tuple pairs. The matching step then predicts Y/N, i.e., matched/not-matched, for each remaining tuple pair. The output of the matching step is then returned as the EM result to the application or the end-user. There are many proposed ways to do matching, such as rules, active learning, supervised, crowd-sourcing, etc. (see [29] for a detailed description). Recent work [50, 37] applies crowd-sourced active learning on C to learn a matcher N. This matcher is then applied to C to obtain predicted matches.

1.2 Prior Entity Matching Work and Its Limitations

EM has received significant attention as it has many practical usecases. Moreover, it is well-known to be very difficult, raising both accuracy and scalability challenges. As a result, it has

been studied intensively over the past several decades, by the database, AI, KDD, and WWW communities, among others. See the related work in Section 1.4 for more details.

However, the vast majority of work on EM has focused on developing algorithmic solutions (such as for the blocking and matching steps, as I explain earlier). *Few if any current works have focused on developing end-to-end EM systems and evaluating them in real-world settings.*

1.3 Contribution of this Dissertation

To address the above problems, in the past few years I have been building CloudMatcher, a cloud service for EM in collaboration with several other colleagues at UW-Madison. We envision CloudMatcher to be a fast, easy-to-use, scalable, and highly available service on the Web. Specifically, to use this service, a user simply needs to go to CloudMatcher's Web site, uploads two tables to be matched, performs some basic pre-processing, then pushes a button. CloudMatcher will perform EM end to end. To do so, it can either use a set of users to label tuple pairs or use crowd workers on Amazon's Mechanical Turk [1] (or some other crowdsourcing platform such as CrowdFlower [6], etc.) for labeling (as matched / no-matched). The user just has to pay for the labeling if he or she decides to use Mechanical Turk. At the end, CloudMatcher will return the desired matches. In the backend, CloudMatcher performs EM using a machine cluster.

As described, when using CloudMatcher, the user does not need to install or learn how to use any complicated system. The user does not have to know EM (e.g., knowing string similarity measures). He or she will only perform simple actions such as labeling a tuple pair as matched/no-matched. Alternatively, if the user is not even willing to label the tuple pairs, then he or she can pay to "outsource" that work to a crowd of workers (assuming that the data is not sensitive and that crowd workers can be quickly trained to label tuple pairs). Finally, the system can scale to tables of millions of tuples and can automatically add more machine resources as necessary.

In this dissertation, I make the following contributions:

1. **Development of CloudMatcher - an End-to-End EM System:** In this work, I design scalable solution architecture for CloudMatcher such that it can do EM tasks at large scale, can

run multiple EM workflows simultaneously, and can support multi-tenants. In developing the solution, I also make sure that the system is able to handle crash recovery smoothly, is fault-tolerant, is efficiently utilizing the available resources, and is ensuring high availability at all times. To make the system available to a wide spectrum of users, I also focus on user experience i.e., building a system that is very easy to use.

The initial work on CloudMatcher has appeared at BIGDAS - KDD'17 [51]. Over the past 3 years, I have contributed as a lead developer to developing an industrial-strength cloud-based EM service. The CloudMatcher codebase is now 47K LOC involving 7 contributors. It has been used at UW-Madison in domain sciences, in a data science class at UW-Madison by 70 students, and at several organizations to do EM.

2. **Real-World Evaluation of CloudMatcher:** To evaluate the system, I have worked with real users and have used the system to do EM tasks for real-world scenarios. I provide an extensive set of experiments that demonstrate that CloudMatcher can effectively perform EM on a broad variety of datasets from real-world users. Specifically, I have evaluated CloudMatcher for end-to-end matching tasks on two domain science projects at UW-Madison and 11 other projects at enterprise customers. Next, I provide the evaluation results of CloudMatcher when applied to Web data by data science students at UW-Madison. Then, I provide a comparison of CloudMatcher with two existing EM systems. Finally, I discuss the lessons learned.

   The evaluation of CloudMatcher system has been published at the SIGMOD'19 industrial track [52]. In the summer of 2018, I also deployed the system at American Family Insurance and worked with real-world users on performing various tasks in the EM space [13].

3. **Developing Atomic Services for EM:** The initial solution of CloudMatcher deployed a single end-to-end EM workflow. As we interacted with real users, we observed that many users want to flexibly customize and experiment with different EM workflows. For example, a user may already know a blocking rule, so he or she wants to skip the step of learning such

rules. Yet another user may want to use CloudMatcher just to label tuple pairs, etc. In this part of my dissertation work, I show that breaking the workflow into a set of basic services then implementing those benefits many domain scientists and EM users.

I developed a solution where the users can mix and match the basic services and compose them to flexibly build more EM workflows. The implementation was highly challenging as one of the design decisions was to make sure that the system is not only modular and extensible but at the same time allows easy plug-in and plug-out of different components, can add/remove basic services, etc. This feature of CloudMatcher was published at the VLDB'18 demonstration track [53].

For this part of my dissertation work, I also worked with the Columbus team [21] to design and develop a solution where these individual services can be hosted on a system like Columbus as an application and can be used to mix and match with other applications or packages outside CloudMatcher.

4. **Scaling Up Blocking Rule Execution:** In the final part of my dissertation work, I focus on scaling up the blocking rule execution for the entity matching task. Given two tables A and B and a sequence of blocking rules R, the goal is to apply the rule sequence R to two tables A and B, producing a set of tuple pairs $C \subseteq A$ x B to be matched in the matching stage. In practice, the two tables A and B can be large and applying R in a naive way to all pairs in A x B is clearly impractical. This process consumes by far the most of machine time and raises scalability challenges.

   Some of the previous work such as Falcon addresses the scaling problem by providing a Hadoop-based solution. However, these solutions have the following limitations. First, they have worked with data up to 2-3M tuples and haven't gone beyond that to study scalability challenges. Second, these systems provide Hadoop-based implementations which are not the state of the art solutions as many enterprise customers use Spark and have their data stored in some NoSQL databases such as MongoDB. Finally, the previous solutions don't

provide much control (knobs) to the end-user that they can trade-off between job runtime and accuracy.

In this work, I design a Spark-based solution and show how the execution of blocking rules scale to tables with 10M tuples or more. Then, I develop a solution called $rule\_executor$ using Spark and MongoDB. Next, I highlight the challenges and issues seen in the $rule\_executor$ after the initial evaluation and propose solutions to the issues. Finally, I examine the scaling behavior of the solution and the effectiveness of the solution.

1.4 Related Work

Numerous EM solutions have been developed (see [47, 29, 43] for surveys and books). These solutions are limited in that they often require a developer in the loop. Two recent works [50, 37] propose hands-off EM, which a lay user can perform without involving a developer. CloudMatcher leverages these works to build an end-to-end industrial-strength cloud/crowd EM service. This raises many novel challenges. CloudMatcher is the first academic work to build such a service, as far as we know. In industry, we know of only one other similar work, Dedupe [7], which is a cloud-based EM service to match tuples within a single table. Dedupe, however, uses only simple types of blockers and requires the user to label tuple pairs using active learning. In contrast, CloudMatcher can employ crowdsourcing to label the pairs (CloudMatcher also supports the user mode). As far as we can tell, CloudMatcher is the first cloud-based EM service that provides support for crowdsourcing. It is also not clear from the public documentation whether Dedupe can scale to many concurrent users and large tables.

Entity Matching: Entity Matching (EM) has received enormous attention in the past few decades [29, 47, 43, 66, 74, 30]. Prior works have addressed various EM scenarios such as matching tuples across two tables [50], matching tuples within a single table [47], matching into a knowledge base [49], matching XML data [93], etc. In literature the vast body of work in EM falls roughly into three groups: algorithmic, human-centric, and system. Most EM works develop *algorithmic solutions* for blocking and matching, exploiting rules, learning, clustering, crowdsourcing, external

data, etc. [29, 47, 43]. The focus is on improving accuracy, minimizing runtime, and minimizing cost (e.g., crowdsourcing fee), among others [78, 47].

A smaller but growing body of EM work (e.g., those at HILDA workshops) studies *human-centric* challenges, such as crowdsourcing, effective user interaction, and user behavior during the EM process [24].

The third group of EM work develops *EM systems*. Recent work Magellan talks about 18 non-commercial systems (e.g., D-Dupe, DuDe, Febrl, Dedoop, Nadeef) and 15 commercial ones (e.g., Tamr [88], Informatica, Data Ladder, IBM InfoSphere) [65, 77] but most of them are stand-alone monoliths. Magellan discusses their limitations and builds an ecosystem of EM tools targeting power users. CloudMatcher considers matching tuples across two tables which is a very common setting in practice and builds a cloud service to do EM. The system targets a wide spectrum of users and is envisioned to be fast, easy-to-use and a scalable platform to do EM on cloud.

Related Work in Data Integration: The field of data integration (DI), which subsumes EM, has had a long history [43, 12, 69, 44]. It developed a range of DI system architectures in the 90s and 00s: global-as-view (GAV), local-as-view (LAV), GLAV, peer-to-peer, best-effort, data space, etc. [43]. Prominent DI system projects at that time include Garlic, Tsimmis, and Information Manifold [85, 27, 70], and prominent recent commercial efforts include Tamr and Trifacta [88, 56].

Crowdsourcing for EM: Crowdsourced EM has received increasing attention in academia [72, 91, 92, 94, 42] and industry (e.g., CrowdFlower [6], CrowdComputing, SamaSource [19] etc.). Most of the works employ crowd to verify the predicted matches [91, 92, 42, 94]. Recent works Corleone [50] and Falcon [37] consider learning blockers and matchers using crowdsourcing. CloudMatcher builds on Falcon to deploy it as a service in the cloud. As far as we can tell, CloudMatcher is the first hands-off EM service on the cloud.

Cloud-Based EM: In the recent years, cloud-based analytics has become popular [55, 58]. However, very limited work has addressed building an EM service in the cloud. A recent effort, Dedupe [7], is a cloud-based EM service to match tuples within a single table. Specifically, it learns a matcher using active learning, then using the labeled data it learns a blocker. However, Dedupe

uses only simple types of blockers and requires the user to label the tuple pairs selected using active learning. In contrast, CloudMatcher can employ crowdsourcing to label the pairs (CloudMatcher also supports user mode). As far as we can tell, CloudMatcher is the first cloud-based EM service providing support for crowdsourcing.

Building EM Systems and EM in Industry: The vast majority of work on EM has focused on developing EM *algorithms* (e.g., for blocking and matching steps), trying to improve their accuracy and minimizing their runtime [29, 47]. In the past few years, however, there has been effort towards building EM systems in academia [65, 34, 28] and industry [87]. Magellan [65] develops an end-to-end EM system built on the top of Python data ecosystem. It clearly distinguishes between the development and production stages, and provides tools to help users perform EM end to end. CloudMatcher leverages these tools to build a cloud-based service.

Recently, there has also been efforts towards building open-source ecosystems for data integration. A recent effort, BigGorilla [4], is an open-source data integration and data preparation ecosystem built on the top of Python data ecosystem, to enable data scientists to perform integration and analysis of data. We believe that such ecosystems can benefit from the services developed by CloudMatcher.

There has been relatively little published about EM in industry [60, 35, 49, 87]. [60] matches unstructured product offers to structured product records using a probabilistic approach, and [49] links tweets into a knowledge base.

Scaling EM: Most works of scaling EM focus on efficiently executing the blocking step. Prior works have addressed scaling specific blocking approaches such as sorted neighborhood blocking [64], key-based blocking [32], meta blocking [45], and so on.

Some of the previous work such as [32] discusses speeding up the data deduplication step by leveraging parallelism in a shared-nothing computing environment. A more recent survey [31] provides an end-to-end view of EM workflows for big data entities and discusses the pros and cons of existing EM methods.

Recent work in database theory community has also explored how to obtain speedups in entity matching related problems. [40, 39] studied how to build efficient data structures that allow set intersection queries, a fundamental primitive for entity matching, and how to execute them efficiently in practice. [41] extended the problem setting to enumerating set pairs in ranked fashion where the ranking function has a special structure. This is useful in scenarios where the user wishes to enumerate k-top ranked set pairs but the value of k is not known apriori (i.e the algorithm is output sensitive).

A recent work, Falcon [37], considers more general blocking rules (each being a Boolean expression of predicates) and develops a MapReduce solution to efficiently execute such rules over two tables. CloudMatcher deploys the Falcon solution in the cloud.

Novel User Interfaces for Crowd Workers: Prior works have addressed designing novel user interfaces for crowd workers [95, 91, 59]. For example, [91] clusters the tuples into groups, shows the workers a group of tuples and asks them to find all duplicate tuples in the group, rather than asking the workers to label tuple pairs as match/non-match. These works are complementary to ours and CloudMatcher can benefit from them.

Data Profiling, Exploration, and Cleaning: Numerous works have addressed data profiling and exploration [23, 56, 83]. Based on our experiences with CloudMatcher, we observe that users often need to profile and explore the data during an EM task. For example, a user may profile the input tables to identify the various matching definitions, so that he/she can provide better instructions to the crowd workers. However, most current works on data profiling and exploration do not provide tools specific for EM tasks. Hence, we believe that there needs to be more effort towards developing profiling and data exploration capabilities specific for EM tasks.

Data cleaning has received enormous attention [33, 48, 68, 61, 56, 54, 84]. Many works address EM as a part of the data cleaning workflow [54, 48]. Based on our experiences with CloudMatcher we believe that there needs to be more effort towards data cleaning solutions specific for EM tasks.

1.5 Roadmap

The rest of this document is as follows. I describe the design and development of CloudMatcher, our cloud-based multi-tenant EM solution, in Chapter 2. After that, I provide the real-world evaluation of CloudMatcher, comparison with existing EM systems, and discussion on the lessons learned in Chapter 3. In Chapter 4, I discuss how the EM workflow can be broken into a set of atomic services and its use cases. Then, in Chapter 5, I discuss the problem of scaling up the execution of blocking rules and propose a Spark-based solution. Finally, in Chapter 6, I conclude and discuss future work.

Chapter 2

Developing Cloud Based Multi-Tenant EM Solution

In this chapter, we first describe the systems Corleone and Falcon. Corleone performs EM end to end, using only crowdsourcing (Section 2.1.1). We then describe Falcon, which uses a cluster of machines to scale up Corleone to tables of millions of tuples (Section 2.1.2).

We then describe CloudMatcher, which implements Falcon as a cloud service. It turns out that doing so raises challenges in terms of effective user interaction, fault tolerance, crash recovery, and scalability (to hundreds or thousands of EM tasks that users may submit at any time). In this chapter we will describe these challenges in detail, then describe our solutions (Section 2.2).

2.1 Background: The Corleone & Falcon Systems

We now describe previous works Corleone and Falcon, which form the basis for the CloudMatcher cloud/crowd EM service.

2.1.1 The Corleone System

Corleone was motivated by the fact that while recent crowdsourced EM works are promising, they are limited in that they crowdsource only parts of the EM workflow, *requiring a developer* who knows how to code and match to execute the remaining parts. For example, several recent solutions require a developer to write heuristic rules, called *blocking rules*, to reduce the number of candidate pairs to be matched, then train and apply a matcher to the remaining pairs to predict matches. The developer must know how to code (e.g., to write rules in Python) and match entities (e.g., to select learning models and features).

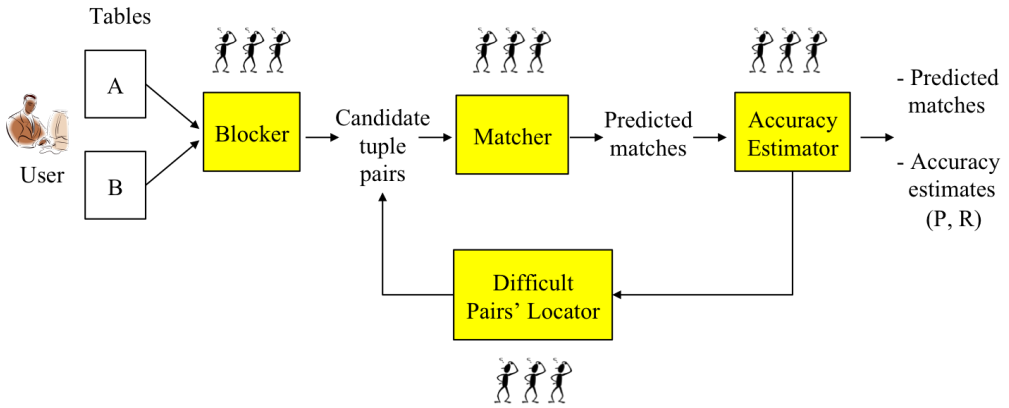


Figure 2.1: The EM workflow of Corleone.

As such, it is very difficult for an organization to concurrently deploy multiple crowdsourced EM solutions, because crowdsourcing each still requires a developer and there are simply not enough developers. To address this problem, Corleone [50] crowdsources the *entire* EM workflow, thus requiring no developers. For example, in the blocking step, instead of asking a developer to come up with blocking rules, Corleone asks a crowd to label certain tuple pairs as matched/no-matched, uses these pairs to learn a classifier, then extracts blocking rules from the classifier (as we will explain soon). Other steps in the EM workflow also heavily use crowdsourcing, but no developers. Thus, Corleone is said to perform *hands-off crowdsourcing* for entity matching.

Specifically, given two tables A and B, Corleone applies the EM workflow in Figure 2.1 to find all tuple pairs $(a \in A, b \in B)$ that match. This workflow consists of four main modules: Blocker, Matcher, Accuracy Estimator, and Difficult Pairs' Locator.

The Blocker generates and applies blocking rules to $A \times B$ to remove obviously non-matched pairs (Figure 2.2.b shows two such rules). Since $A \times B$ is often very large, considering all tuple pairs in it is impractical. So blocking is used to drastically reduce the number of pairs that subsequent modules must consider. The Matcher uses active learning to train a random forest classifier, then applies it to the surviving pairs to predict matches. The Accuracy Estimator computes the accuracy of the Matcher. The Difficult Pairs' Locator finds pairs that most likely the current Matcher has matched incorrectly. The Matcher then learns a better random forest to match these pairs, and so on, until the estimated matching accuracy no longer improves.

Corleone is distinguished in that the above four modules use no developers, only crowdsourcing. For example, to perform blocking, most current works would require a developer to examine Tables A and B to come up with heuristic blocking rules (e.g., "If prices differ by at least \$20, then two products do not match"), code the rules (e.g., in Python), then execute them over A and B. In contrast, the Blocker in Corleone uses crowdsourcing to learn such blocking rules (in a machine-readable format), then automatically executes those rules. Similarly, the remaining three modules also heavily use crowdsourcing but no developers.

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

2.1.2 The Falcon System

As described, Corleone is highly promising. But it suffers from a major limitation: it executes mostly a single-machine in-memory EM workflow, and thus does not scale at all to tables of moderate and large sizes. For example, using Corleone to match tables of 50K-200K tuples would take weeks, rendering the system impractical.

To address this problem, Falcon scales up Corleone to tables of millions of tuples. To do so, it introduces three key ideas. First, it defines basic operators and uses them to model the EM workflow of Corleone as a directed acyclic graph (DAG). Next, it scales up the operators, using MapReduce if necessary. Finally, it optimizes within and across operators.

In what follows we discuss these ideas, but only to the extent necessary for the purpose of this chapter (see [37] for a complete description of Falcon).

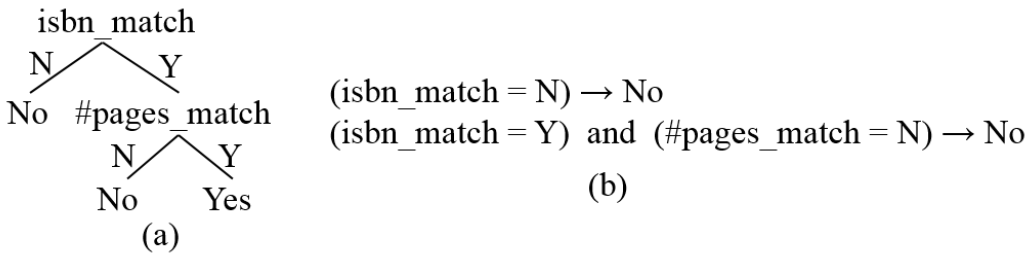


Figure 2.2: (a) A decision tree learned by Corleone and (b) blocking rules extracted from the tree.

2.1.2.1 The EM Workflow Considered by Falcon

Currently, Falcon considers only EM workflows that consist of the Blocker followed by the Matcher, or just the Matcher. We now describe the Blocker and the Matcher, focusing only on the aspects necessary to understand Falcon.

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

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

To learn, the module first asks the user to supply two positive examples (i.e., two tuple pairs labeled matched) and two negative examples (i.e., two tuple pairs labeled non-matched). Next, it uses these "seed" examples to train an initial random forest matcher M, uses M to select a set of controversial tuple pairs from sample S, then asks the crowd to label these pairs as matched / no-matched. In the second iteration, the module uses these labeled pairs to re-train M, uses M to select a new set of tuple pairs from S, and so on, until a stopping criterion has been reached.

At this point the module returns a final matcher M, which is a random forest classifier consisting of a set of decision trees. Each tree when applied to a tuple pair will predict if it matches, e.g., the tree in Figure 2.2.a predicts that two book tuples match only if their ISBNs match and the number of pages match. Given a tuple pair p, matcher M applies all of its decision trees to p, then combines their predictions to obtain a final prediction for p.

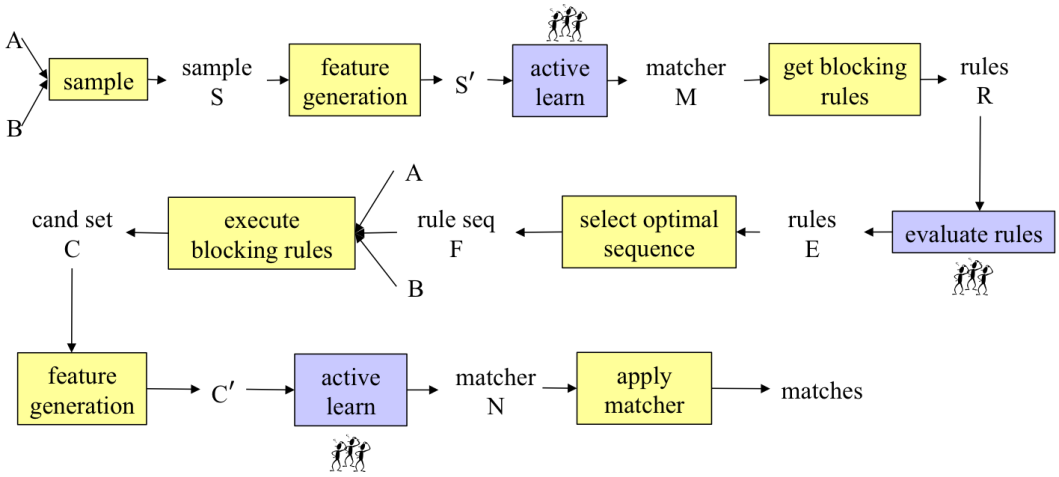


Figure 2.3: The EM workflow of Falcon as a DAG of basic operators.

Next, the module extracts all tree branches that lead from the root of a decision tree to a "No" leaf as candidate blocking rules. Figure 2.2.b shows two such rules extracted from the tree in Figure 2.2.a. The first rule states that if two books do not agree on ISBNs, then they do not match.

Next, for each extracted blocking rule r, the module computes its precision. The basic idea is to take a sample T from S, use the crowd to label pairs in T as matched / no-matched, then use these labeled pairs to estimate the precision of rule r. To minimize crowdsourcing cost and time, T is constructed (and expanded) incrementally in multiple iterations, only as many iterations as necessary to estimate the precision of r with a high confidence (see [50]).

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

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

2.1.2.2 Modeling the EM Workflow as a DAG of Basic Operators

As described, the workflow of Falcon can be modeled as a DAG of basic operators as shown in Figure 2.3. In this DAG, given two tables A and B to be matched, the first step is to take a sample

S of tuple pairs. Next, use the schemas of A and B to automatically generate a set of features (not shown in the figure), then use these features to convert each tuple pair in S into a feature vector, thereby converting S into a set of feature vectors S'.

Next, perform active learning with the crowd[1] over S' to learn a matcher M, then extract a set of blocking rules R from M. Then use the crowd to evaluate these rules and select a sequence of rules F judged to be optimal (see [50]). Next, execute F over the tables A and B. This produces a set of candidate tuple pairs C.

At this point, the blocking step ends, and the matching step begins. First, convert each tuple pair in C into a feature vector, thereby converting C into a set of feature vectors C'. Then perform active learning (again) with the crowd to learn a matcher N. Finally, we apply N to the feature vectors in C' to predict matches.

The above workflow uses eight basic operators. As described, these operators involve complex rules, crowdsourcing, and machine learning, and can be used to compose a variety of EM workflows (see [37]).

It is important to note that Falcon has developed efficient implementations for these operators (using MapReduce where necessary), and has also developed techniques to optimize within and across operators. We omit further details here for space reasons (see [37]).

2.2 The CloudMatcher Service

We are now in a position to discuss CloudMatcher, the cloud/crowd service that we have been building. In what follows we discuss the motivations, goals, and then the CloudMatchersolution.

2.2.1 Motivations and Goals

Our initial goal was to to provide EM services to hundreds of domain scientists at UW-Madison and affiliated institutions. Domain scientists often do not know how to, or are reluctant to, deploy EM systems locally (such systems often require a Hadoop cluster, as discussed earlier). So we

[1] We omit the step of asking the user to supply "seed" examples to avoid making the figure too cluttered.

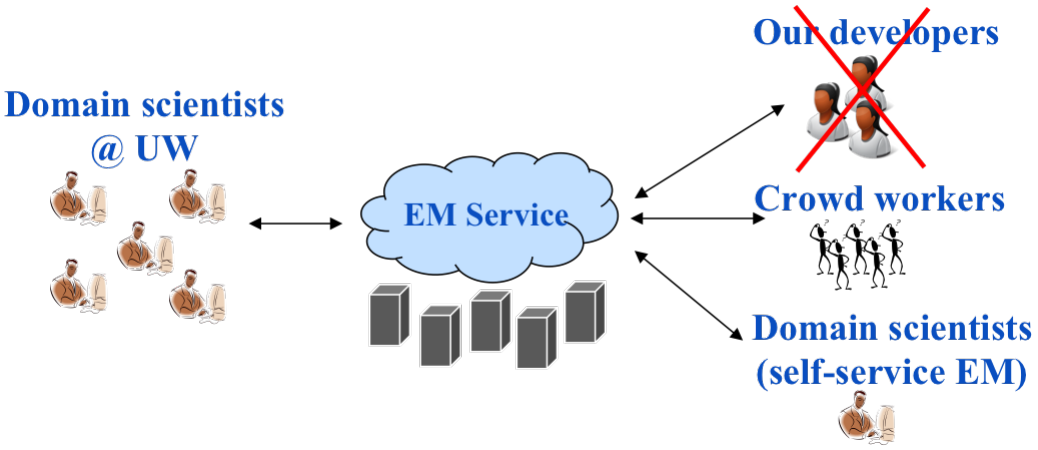


Figure 2.4: CloudMatcher as a cloud/crowd EM service.

want to provide such EM services on the cloud, supported in the backend by a cluster of machines maintained by our group.

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

Thus, we planned to deploy the hands-off solution of Corleone. A user can just submit the two tables to be matched on a Web page and specify the crowdsourcing budget. We will run Corleone internally, which uses the crowd to match. As described, Corleone seems perfect for our situation. Unfortunately, it executes mostly a single-machine in-memory EM workflow, and does not scale at all to tables of moderate and large sizes. So we will use Falcon, which scales to tables of millions of tuples. In particular, we will execute the EM workflow of Falcon described in Figure 2.3.

It is important to note that if users do not want to engage the crowd, they can label the tuple pairs themselves. This in effect provides a self-service EM for the users. Most users we have talked to, however, prefer if possible (e.g., if the data is not sensitive or too difficult for the crowd to match) to just pay a few hundred crowdsourcing dollars to obtain the result in 1-2 days. Figure 2.4 illustrates both the crowdsourcing and the self-service options discussed above.

Our goals for CloudMatcher are as follows:

- **Efficient resource consumption:** Use minimal machine and crowd resources to perform EM tasks.
- **Fault tolerance:** If a machine or process crashes, can recover and continue gracefully.
- **Crash recovery:** Executing an EM task can take hours (or days if crowdsourcing is involved). As a result, crash recovery is critical. Specifically, if CloudMatcher crashes in the middle of an EM task, when resumed, it should continue where it crashed, instead of restarting the task from scratch.
- **Scaling:** CloudMatcher should scale, both for a single EM task and for multiple EM tasks. That is, a single EM task should execute as fast as possible, and the system should be able to handle hundreds or thousands of EM tasks concurrently, without being slow on anyone of them.
- **Optimization:** In order to scale, ideally the system must be such that there are multiple opportunities for optimization, and the system can make use of these opportunities.
- **Efficient management of heterogeneous execution environments:** When executing an EM workflow, each step in the workflow may require its own execution environment. For example, one step is to be executed in Python on a single machine, whereas another step requires Java over a Hadoop cluster. CloudMatcher should be able to handle a broad range of such heterogeneous environments.
- **Smooth user experience:** The user should have a very smooth experience with the system. The GUI must be intuitive and requires very little guessing to work with. The system latency should be at interactive speed, i.e., it should not take more than a few seconds to respond to the user. If the user works in a browser, then stops the work (say for lunch), or close the browser and open another one in the same or another machine, then the user should be able to seamlessly continue working.

- **Progress report:** The system should tell the user where it is in the EM process and give estimations on how much longer it will take to complete certain tasks.
- **Visualization:** The system should provide as much visual information to the user as possible, especially in terms of its progress.

The above set of goals makes it clear that developing CloudMatcher is not a simple matter of deploying Falcon. For example, Falcon focuses on techniques to scale up a single EM workflow. It does not focus on goals such as fault tolerance, crash recovery, smooth user experience, etc.

2.2.2 Limitations of Current Solutions

To implement CloudMatcher, the simplest solution is to convert each submitted EM task into a Falcon DAG, as shown in Figure 2.3, then execute the DAG using a workflow management system (WMS) such as Luigi, Airflow, or Pinball. Many such WMSs have been developed. In theory, they can guarantee certain kinds of fault tolerance and crash recovery. For example, the WMS can write the output of each node in the DAG to disk, and thus can guarantee that in the case of a crash, DAG execution does not have to restart from scratch. In addition, such WMSs can easily handle multiple Falcon DAGs being run concurrently, as is common in cloud settings.

There are however two major problems with the Falcon DAG. First, the DAG's granularity is too coarse, rendering it not effective for crash recovery, optimization, and best usage of resources. Specifically, some of the steps in this DAG can take a very long time, and currently there is no easy way to save their partial results for crash recovery. Consider for example a step that performs active learning with the crowd to learn a matcher. This step can perform up to 30 iterations of active learning, and can take hours or days (if the crowd is slow). Ideally, we should be able to save the output of each iteration, so that in the case of a crash, we can resume at the crashed iteration. However, since currently the entire step is modeled as a single node in the Falcon DAG, it is difficult for us to perform such partial saving. Also, coarse steps make it difficult to optimize within and among the steps.

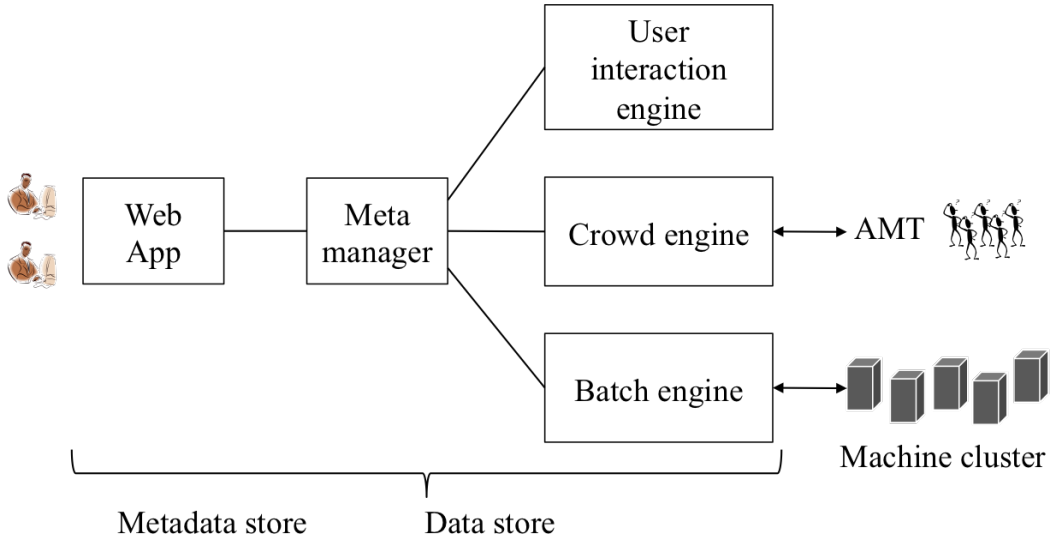


Figure 2.5: The CloudMatcher architecture

Another problem is that some of the steps in the Falcon DAG involve user interaction. For example, before we can start the first active learning process (on sample S'), we need to ask the user to supply at least two positive examples and two negative examples. (This step is not shown in Figure 2.3, to avoid making the figure too cluttered.) Machine-wise this step does not take long to execute. But it involves asking for an input from the user, and this can often cause a problem. The user may stop in the middle, go to lunch, have a phone call, etc., in which case this step will be left "hanging", waiting for the user to get back. If not implemented carefully, this step will continue to "hog" resources until the user responds. The same problem arises for any step involving crowdsourcing.

Note that this second problem is relatively new, because the current workflow management systems (e.g., Luigi, Airflow, etc.) typically are designed to execute DAGs that can be run in batch mode. They are not designed for efficiently executing DAGs that can involve user interaction in the middle. In theory, they can still be used to execute such DAGs, but it will typically result in inefficient use of resources (as the executor waits for the user to get back from lunch, say) and can negatively affect many other DAGs that are being concurrently executed.

2.2.3 Key Ideas of the CloudMatcher Solution

To address the above limitations, in CloudMatcher we employ the following key ideas:

- We convert the Falcon DAG into an EM workflow at a much finer granularity, to maximize the opportunities for crash recovery, optimization, and efficient resource usage. The new EM workflow is not a DAG, as it involves loops (in addition to conditionals).
- For the new CloudMatcher EM workflow, we clearly define tasks (i.e., nodes of the workflow graph) that are *interactive* (i.e., interacting with a user or a crowd to request some input).
- We partition the CloudMatcher workflow into "pieces" such that each piece is either an *interactive* task or a workflow fragment that can be executed entirely in *batch* mode.
- We define three kinds of execution engines: user interaction (UI) engine, crowd engine, and batch engine. The UI engine is designed to execute interactive tasks efficiently, and similarly the crowd engine is designed for executing tasks that require crowdsourcing. Finally, the batch engine is designed for executing batch-mode workflow fragments.
- We use a meta-manager to execute the entire CloudMatcher workflow, by executing each piece of the workflow using the appropriate execution engine.
- Finally, we divide the responsibilities for managing fault tolerance and crash recovery appropriately among the meta-manager and the execution engines.

Putting these ideas together produces the CloudMatcher architecture shown in Figure 2.5. In this figure, the Web app module is responsible for authentication, account creation, and processing GET/POST requests from users. Given a submitted EM task, the meta-manager converts it into an EM workflow, then partitions the workflow into pieces, where each piece is interactive or batch by nature, as described earlier. The meta-manager then executes these pieces using the appropriate execution engines. The meta-manager and the execution engines will coordinate the management of fault tolerance and crash recovery, using the meta-data store (which records for

example which nodes in which graph fragments have been executed) and the data store (which stores the input/output and intermediate data for the workflow nodes).

We now elaborate on the most important aspects of the above solution architecture.

2.2.4 The EM Workflow of CloudMatcher

Recall that we want to break each long-running step in the Falcon DAG into many much smaller steps, whenever possible, and isolate all "points" in the Falcon DAG where we need user interaction, then make those "points" into their own steps.

Figure 2.6 shows the resulting workflow for CloudMatcher. This workflow is quite long and consists of three parts: Part (a) followed by Part (b) followed by Part (c). We now briefly describe this workflow, and contrast it to the Falcon one.

- The very first task in the CloudMatcher workflow is "Create job" (see Figure 2.6.a). In this task, the user goes to the CloudMatcher Web page, creates a job (whose goal is to match two tables), and supplies some job related information (e.g., contact email). Next, the user uploads the first table, Table A. The system then profiles this table, e.g., counting the total number of tuples in the table and showing that to the user (to confirm that the table has indeed been uploaded and everything appears to be in order). See the next two tasks on the figure. These two tasks will be repeated one more time for Table B (the number "2x" on the arrow from "Profile table" back to "Upload table" indicates the maximal number of iterations for this loop).

- The first two tasks, "Create job" and "Upload a table", are *interactive*, in that they must interact with the user to obtain information. As discussed earlier, we "isolate" such interactions into their own tasks. On the figure, we show such tasks either with a small human figure or inside a dotted box with a small human figure. The third task, "Profile table", is not interactive. We refer to such tasks as *batch tasks*, as they can be executed in batch mode.

- The next task, "Check data/schema constraints", verifies certain integrity constraints, e.g., trying to identify a key column, and if none found, then create a key column for the table.

The subsequent tasks on Figure 2.6.a obtain a sample S of tuple pairs, convert S into a set of feature vectors G, and obtain at least two positive examples and two negative examples from the user. We omit further details for space reasons.

- Once the workflow fragment on Figure 2.6.a ends, we continue with the workflow fragment on Figure 2.6.b. Here, we first convert the two positive and two negative examples (called "seeds") into feature vectors, then start the active learning process. Notice that this active learning process was earlier just a single task in the Falcon workflow. Here it has been broken into a loop of four tasks: "Train classifier", "Check stopping condition", "Select batch of tuple pairs", and "Label batch". We repeat this loop up to 30 times. Notice also that the first three tasks of this loop are batch task, whereas the last one ("Label batch") is an interactive task.

- Once the active learning finishes, we obtain a matcher M, from which we obtain a set of candidate blocking rules, then evaluate the rules and so on. We omit further description of the rest of the tasks in Figure 2.6.b, as well as the tasks in Figure 2.6.c (as they are similar to those in Figure 2.6.b).

Compared to the Falcon DAG, the CloudMatcher workflow is different in the following aspects. First, it is at a much finer granularity, with all long-running tasks being broken down into as many smaller tasks as possible. Second, the workflow is no longer a DAG. It now has loops (in addition to conditionals). Finally, the workflow is a combination of interactive tasks and batch tasks.

2.2.5 Partitioning the EM Workflow

Given an EM workflow as described above, the meta-manager of CloudMatcher partitions it into workflow fragments, such that each fragment is strictly interactive or batch by nature. Figure 2.7 shows how the first part of the CloudMatcher workflow (which is the part shown in Figure 2.6.a) is partitioned into eight interactive fragments (each of which is in yellow) and six batch fragments (in blue). The uppermost batch fragment, for example, consists of three tasks: "Gen feature funcs", "Gen sample pairs", and "Gen feature vecs".

2.2.6 Executing the Workflow Fragments

After partitioning, the meta-manager executes the workflow fragments, each in the appropriate execution engine. Specifically, each batch fragment will be executed using the batch engine, which uses a well-known current workflow management system, such as Luigi, Airflow, or Pinball. If an interactive workflow fragment does not require crowdsourcing, then it only needs to interact with a single user to request some input. In this case, we execute the fragment using the user interaction (UI) engine. Otherwise, we execute the fragment using the crowd engine.

The meta-manager uses the metadata store and the data store to coordinate the execution of the various workflow fragments, and to handle fault tolerance and crash recovery. We omit further details for space reasons.

2.2.7 The User Interaction/Crowd Engines

Finally, we describe the working of the UI engine and the crowd engine. Consider executing an UI task E. The UI engine starts by sending a request for the user to do an action (e.g., providing the name of the job to be created and a contact email address) to the user's Web browser (via the Web app). At some point, after the user has filled out the requested information, he or she will click the submit button, which sends a request to the Web app, which in turn contacts the UI engine.

The engine then processes the information from the user. If this information is complete, then the engine indicates to the meta-manager that this UI task E has been completed. Otherwise, if the information is incomplete (e.g., only the job name is provided, the requested email address is still missing), then the UI engine sends another request (for email address) to the user's Web browser, and so on.

As described, the UI engine does not run a process that is dedicatedly trying to interact with the user. Instead, it operates in a "transactional" mode, in which it sends a request to the user, then "goes do something else", and returns only when the user has sent in something. This transactional mode is desired because we simply do not know how long it would take for the user to process the request (he or she may stop for lunch in the middle, etc.), and hence we do not want to run a dedicated process to wait on the user.

If the task is not UI, but instead requires crowdsourcing, then the situation is a bit more involved. Suppose the task is to label 20 examples (for active learning). To execute, the crowd engine will send these 20 examples to a crowdsourcing platform, say Amazon's Mechanical Turk (AMT), for labeling. The problem is that AMT does not get back to the crowd engine (i.e., there is nothing that is equivalent to "a user clicking the submit button" in the AMT case). So the crowd engine needs to "ping" AMT at regular intervals to check on the progress of the labeling task.

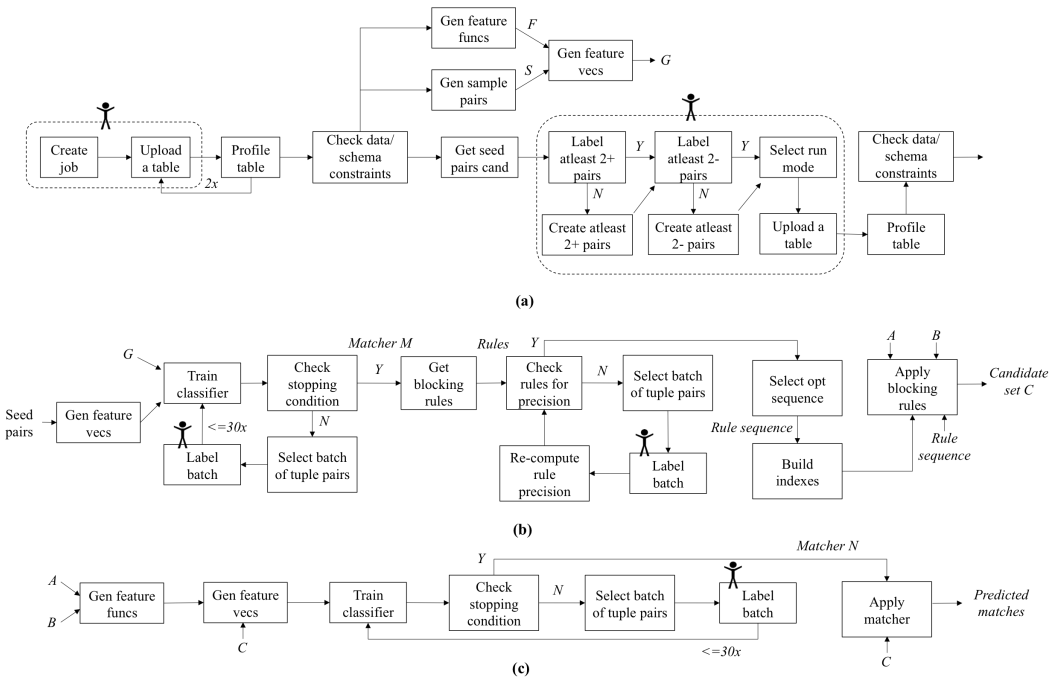


Figure 2.6: A refinement of the Falcon DAG, to create the workflow for CloudMatcher. The resulting workflow consists of three parts, as shown in (a)-(c).

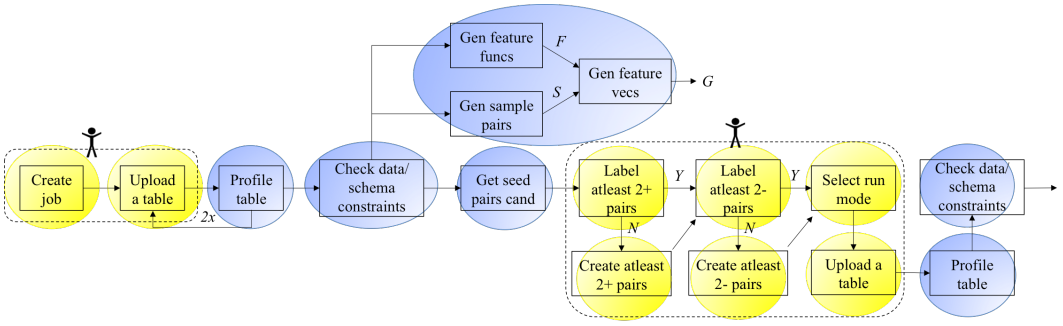


Figure 2.7: A partitioning of the first part of the CloudMatcher workflow into interactive and batch fragments.

Chapter 3

Real-World Evaluation of CloudMatcher & Lessons Learned

In this chapter, we present the evaluation of CloudMatcher. Evaluation of such a system can be difficult with real-world data and users. To address this challenge, we evaluate CloudMatcher in a few different ways. First, we deploy the system for domain scientists at UW-Madison and help them do EM tasks. Then, we extend the system to be used at several organizations such as Johnson Control, American Family Insurance, etc. We report the results obtained by CloudMatcher on real-world EM scenarios and our experience working with real-world data and users. Second, we use the system at a data-science class at UW-Madison and evaluate the system for many real-world EM scenarios on the Web data with the help of 70 students. Third, we compare CloudMatcher with three other existing EM systems and discuss our findings. We then focus on the blocking component of the system as we believe that: (i), it is an important piece that takes a lot of machine time and aims for high recall, (ii) the matching step reuses most of the components we use during the blocking step (for example, the feature generation, extracting feature vectors, and the active learning step). We do the evaluation on the datasets we have gold so that we can compute the recall and make sure the system is working as expected. Finally, we discuss the lessons learned in developing and evaluating such a system. Some of the scaling experiments for CloudMatcher are discussed here [82].

3.1 Deployment

We now describe the typical deployment of CloudMatcher at UW-Madison and several other organizations. Specifically, we deploy CloudMatcher on a 4-node Amazon EMR cluster, where

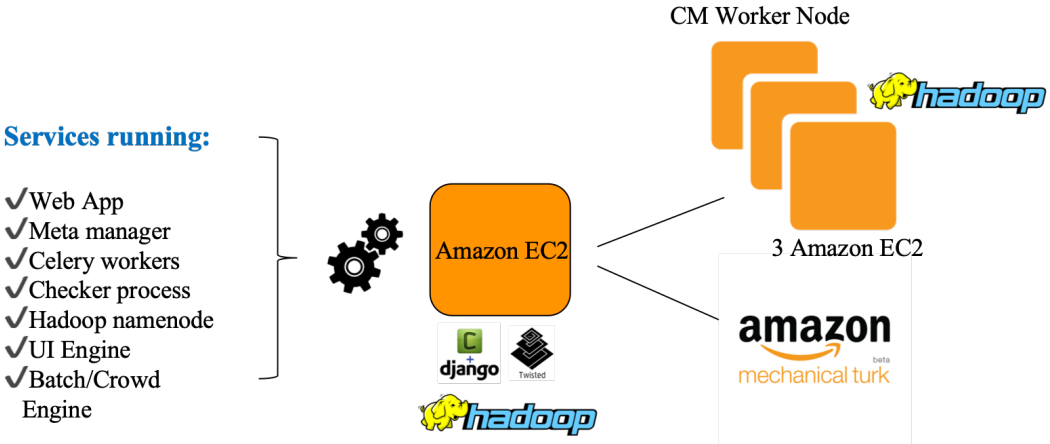


Figure 3.1: Typical Cluster Setup for CloudMatcher

each node has a 4-core Intel Xeon E5-2686 2.30 GHz processor and 16GB of RAM. To ease the deployment step, we make use of docker containers [10], custom bootstrapping script, and terraform [20]. We have completely automated the deployment process and it takes no more than 15-20 minutes to get the end-to-end system running on the cluster. In addition, our deployment is totally infrastructure agnostic and could be easily used with other cloud providers as well such as Azure or GCP.

CloudMatcher solution also provides support for crowdsourcing and we deploy the system with a default crowdsourcing platform i.e., Amazon Mechanical Turk. For crowdsourcing setup, we assign each HIT (Human Intelligence Task) to three crowd workers, paying 2 cents per answer. In each crowdsourcing run, we used common turker qualifications to avoid spammers, such as allowing only turkers with at least 100 approved HITs and 95% approval rate. We provide default crowd configuration for the deployment that the user can easily modify before or after the system has been deployed.

Typical Cluster Setup for CloudMatcher In Figure 3.1, we show a typical cluster set up required to deploy and run CloudMatcher. Recall that in Section 2.2.3, we discuss the architecture for the system. From the architecture, we currently run most of the services on the Hadoop master node. As we can see in Figure 3.1, the following services run on the master node: the web app, the

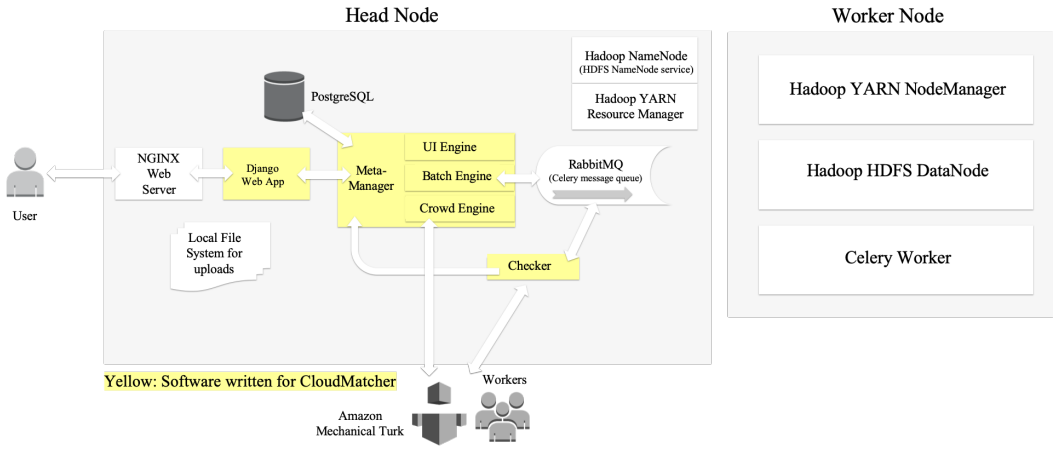


Figure 3.2: The master and worker node in CloudMatcher

meta manager, celery [5] workers, checker process, Hadoop namenode, user interaction engine, and batch/crowd engine.

Now, we will zoom into the master node and worker node of the cluster to show the different components running in the system. We deploy the system in such a way that all the components in the system can be easily scaled out depending on the system load. Figure 3.2 shows the components running on the master and worker nodes respectively.

To serve many users/requests, we use Nginx [15] as the web-server, Gunicorn [11] as the Python HTTP server and our web-app component is written using the popular Django web framework [9]. We store state for the web application and the workflows/tasks in a PostgreSQL database. We use Celery [5] as our task queue with RabbitMQ [18] as the message broker (using binary-only AMQP message format). For portability, we run our Postgres instance and our Task Queue in Docker containers to make our development and production environments identical. The meta-manager is implemented using Twisted (an asynchronous event-driven engine for network application development) [22] and we use Networkx library [14] to handle workflow graphs/DAGs. In addition to these softwares we use a bunch of licensed Python packages from the PyData ecosystem using the Anaconda distribution [2].

3.2 Empirical Evaluation on Real-World Applications

From 2017 to date CloudMatcher has been successfully applied to multiple real-world EM applications and has attracted commercial interest. It has been deployed at American Family Insurance, a Fortune-500 company, since Summer 2018 and is being considered for production at a large data integration company.

Data Sets: At American Family Insurance, we started by doing a proof of concept on a Phoenix customer dataset of 300 records in each table. The idea behind this evaluation was to set up the system with the instream and downstream applications. Then, the second dataset was to match commercial insurance policyholders. This was again a piece of personal information (e.g., name, dob, phone, address, etc.). These two datasets mostly helped in establishing the system and making sure that the accuracy numbers look good to the business users before a large scale matching is done. The third dataset matched the commercial farm/ranch policy members for all members in the state of Phoenix. This experiment was performed as a medium scale verification of the matching process and for all the three datasets we had gold matches from the business. The Vehicles datasets were from NADA (National Automobile Data Association) and Price digest and the goal was to match vehicles between the two sources. This was the hardest of all the datasets we matched as the data was very dirty and even a human was having a hard time to label the tuple pairs. Finally, the last dataset to match contained Drivers information along with their schedule across different roasters.

For Johnsons Control, we used CloudMatcher to do EM tasks for three datasets. Addresses attempts to match addresses of JCI customers. Vendors identifies the same JCI vendors across the tables, given their names and addresses. (We will explain the dataset Vendors (no Brazil) later.)

Next, we received a dataset from UW Health which was matching UW hospital's doctors and staff. This was an example of deduplication where we had to find matches within a single table. Then, we received a Person data from Informatica which was again a single table with personal information. Next, Drugs matches drug descriptions in the Marshfield-UW research team.

Problem Owner	Problem Type	Table A	Table B	Precision (in %)	Recall (in %)	Cost			Time		
						Questions	Crowd	Compute	User/Crowd	Machine	Total
American Family Insurance	Phoenix customers	300	300	96.4	99.03	160	-	$2.33	9m	5m	14m
	Commercial insurance policy holders	1,049	17,572	96.15	97.22	321	-	$2.33	18m	25m	43m
	Commercial farm/ranch policy members	109,974	4,922,505	99.5	95	780	-	$13.96	50m	4h 58m	5h 48m
	Vehicles	18,938	72,898	66.02 – 80.02	81.65 – 93.15	851	-	$7.00	2h	46m	2h 46m
	Drivers	790	634	99.86	94.89	250	-	$2.33	10m	8m	18m
Johnson Controls International	Addresses	90,673	231,081	93.22 – 95.72	76.93 – 81.01	1200	$72	-	36h 48m	38m	37h 26m
	Vendors	50,295	50,292	29.95 – 38.04	91.89 – 98.10	1160	$69.60	-	30h 31m	58m	31h 29m
	Vendors (no Brazil)	28,152	28,149	95.44 – 97.75	88.82 – 92.41	1200	$72	-	22h 19m	22m	22h 41m
UW Health	Doctors & staff	1,786	1,786	99.66	98.18	1200	-	$4.66	50m	15m	1h 5m
Informatica	Persons	48,119	48,119	100 – 100	98.42 – 100	462	-	$7.00	36m	1h 35m	2h 11m
Marshfield Clinic	Drugs	446,048	440,048	99.14 – 99.63	98.45 – 99.14	1162	-	-	1h 10m	8h 40m	9h 50m
Non-profit Org	Elected officials	9,751	706,878	93.75 – 96.32	95.50 – 97.76	960	$57.60	-	23h 14m	23m	23h 37m
Domain Science	UMetrics economics	2,616	21,530	94.5 – 96.5	98.12 – 99.21	680	$61.20	-	23h 12m	12m	23h 24m

Table 3.1: Real-world deployment of CloudMatcher.

People identifies the same persons across two tables, given their names and addresses. These tables capture political activities, e.g., signing up for a recall, donations, etc. The goal is to track the political activities of local elected officials in Wisconsin. This dataset comes from a user at a non-profit organization (which prefers not to disclose its identity). Finally, the last dataset was from a domain science group at UW-Madison where we were matching funding sources between two tables. (We do not yet have the permission to disclose EM results on matching products at WalmartLabs.) Note for some of these real-world datasets we don't have gold matches.

Table 3.1 summarizes CloudMatcher's performance on 13 real-world EM tasks The first two columns show that CloudMatcher has been used in 5 companies, 1 non-profit, and 1 domain science group, for a variety of EM tasks. The next two columns show that CloudMatcher was used to match tables of varying sizes, from 300 to 4.9M tuples.

Ignoring the next two columns on the accuracy, let us zoom in on the three columns under "Cost" in Table 3.1. The first column ("Questions") lists the number of questions CloudMatcher had to ask, i.e., the number of tuple pairs that needed to be labeled. This number ranges from 160 to 1200 (the upper limit for the current CloudMatcher).

In the next column ("Crowd"), a cell value such as "$72" indicates that for the corresponding EM task, CloudMatcher used crowd workers on Mechanical Turk to label tuple pairs, and it cost $72. A cell value "-" indicates that the task did not use crowdsourcing. It used a single user instead, typically the person who submitted the EM task, to label, and thus incurred no monetary cost.

In the third column ("Compute"), a cell value such as "$2.33" indicates that the corresponding EM task used AWS, which charged $2.33. A cell value such as "-" indicates that the EM task used a local machine owned by us, and thus incurred no monetary cost.

Turning our attention to the last three columns under "Time". The first column ("User/Crowd") lists the total labeling time, either by a single user or by the Mechanical Turk crowd. We can see that when a single user labeled, it was typically quite fast, with time from 9m to 2h. When a crowd labeled, time was from 22h to 36h (this does not mean crowd workers labeled non-stop and took that long, it just meant Mechanical Turk took that long to finish the labeling task). These results suggest that CloudMatcher can execute a broad range of EM tasks with very reasonable labeling time from both users and crowd workers. The next two columns under "Time" show the machine time and the total time.

We now zoom in on the accuracy. To compute accuracies, we took a sample of 500-1000 pairs from the output of CloudMatcher, manually labeled them, then followed the accuracy estimation procedure in [50] to estimate precision and recall (see Columns "Precision" and "Recall"). A value such as "93.75-96.32" in Column "Precision" means the precision is in that range with 0.95 confidence. The columns "Precision" and "Recall" show that in all cases except three, CloudMatcher achieves high accuracy, often in the 90 percentage. The three cases of limited accuracy are "Vehicles", "Addresses", and "Vendors". A domain expert at American Family Insurance (AmFam) labeled tuple pairs for "Vehicles". But the data was so incomplete that even he was uncertain in many cases on whether the tuple pair match. At some point he realized that he had incorrectly labeled a set of tuple pairs, but CloudMatcher provided no way for him to "undo" the labeling, hence the low accuracy.

For "Vendors", it turned out that the portion of data that consists of Brazilian vendors is simply incorrect: the vendors entered some generic addresses instead of their real addresses. For Vendors, the recall is good (92-98%), but the precision is very low (30-38%). This dataset contains many vendors in Brazil, and it turned out there are serious problems with their descriptions. Specifically, many Brazilian vendors have the same address and the same last name but different first name, e.g., "LUCIANA DOS SANTOS, RUA JOAO TIBIRICA - 900, VILA ANASTACIO, SAO

PAULO, BRA, 34756800" vs "FERNANDA DOS SANTOS, RUA JOAO TIBIRICA - 900, VILA ANASTACIO, SAO PAULO, BRA, 34756800" (note that a vendor can be a person representing a small business, or a large company; the above two tuples represent two small businesses). Many crowd workers declared such tuples matched. In reality, they do not. An extensive examination reveals some problem with the Brazil data, namely, when two vendors were entered into the JCI system, instead of entering their real addresses, often the address of the JCI office that they were doing business with were entered. Hence two vendors with different names often end up "sharing" the same address. As a result, even human users cannot match such vendors. Once, we removed such vendors from the data, the accuracy significantly improved (see the row for "Vendors (no Brazil)"). Another problem with the above dataset is that some of the tuple pairs are difficult even for domain experts to match, e.g., "Juan Carlos Caldelas, Monterrey, MX" vs "Juan Carlos Espinosa Mari, Monterrey, MX".

It turned out that "Addresses" had similar dirty data problems, which explained the low recall of 76-81%. Specifically for "Addresses", a close examination reveals that this dataset is quite dirty. For example, addresses often contain extra strings, such as "AS AGENT FOR 105 East 17th St Associates 423 W 55th St, New York, NY, 10019". This suggests data cleaning is necessary. Further, the matching instruction for crowd workers was incomplete, so when crowd workers saw addresses that are identical except for the "P.O. Box" numbers, they did not know if those should be matched.

Finally, the Drugs dataset was not hard to match accuracy-wise (achieving precision of 99% and recall of 98-99%), but was hard to manage runtime and space-wise. Specifically, different runs of this dataset produced different blocking results, and these results vary drastically. In some cases the size of the blocking result was quite reasonable, in the millions (of tuple pairs). But we have also seen cases of 1-2 billions of tuple pairs. Table 3.1 shows a case in the middle, where we "only" have 143M tuple pairs after blocking. This raises the question of what CloudMatcher should do if blocking produces very large outputs. If left unmanaged, the blocking process can take a very long time, consume all available memory and disk space, and stall or crash.

3.3 Empirical Evaluation on Web Data

We now present the evaluation of CloudMatcher to do EM tasks on the web data. For this evaluation, CloudMatcher was deployed on a 3 node Amazon EMR cluster and was made available to 23 teams of Computer Sciences students (a total of 70 students) to use in a Spring 2019 data science class at UW-Madison. The idea behind this evaluation was three-fold. First, we would see the performance of CloudMatcher on real-world web data in terms of accuracy and runtime. Secondly, we will have real-world users simultaneously using the system and running concurrent EM workflows. This will test the scalability aspect of the system and the underlying infrastructure. Third, we will collect logging information about user actions that will help us improve the user experience and finally, we will get feedback about the system from the user side. **Data Sets:** For this evaluation, we asked the students to select two web data sources from which structured data can be extracted by using the rule-based wrapper construction method discussed in the coursework. These two data sources must contain information about a set of overlapping entities, such as books, movies, cars, etc. Then each table should have at least 3000 tuples, and they should share at least 100 persons. Then extract data from these two sources to form two tables A and B (one from each source). The two tables should have the same schema, and each tuple in each table must describe a single entity (all of the same type). The students identified two web data sources and extracted the two tables to be matched. From Table 3.2, we see that across 23 teams, 7 different domains were selected. To be specific, 3 books data from three different web sources, 15 different datasets for movies, and then one each for computer vision field, restaurants, Cricket, Cosmetics, and academic papers. Each table had an average of 3.9K tuples, 5-7 attributes.

The students then used CloudMatcher to do the EM task on these datasets. The students were provided a step-by-step instruction on how to use the system, the EM service, and report accuracy numbers. The first four columns of Table 3.2 show the teams, domains, and the sizes of the two tables, respectively. Note, that few domains are covered multiple times e.g., ”Movies”, but the data is extracted from different sites. Each team was asked to follow the step-by-step guide and

Team	Name	Input Tables		Blocking			Matching			Precision	Recall
		|A|	|B|	# Iterations	# Pairs Labeled Total ('match', 'non-match')	Candidate Set	# Iterations	# Pairs Labeled Total ('match', 'non-match')	Predicted Matches		
1	Books1	3056	3000	11	224 (44, 180)	33792	8	160 (28, 127)	245	97.86 - 100	92.45 - 99.06
2	Movies1	3003	3000	11	224 (78, 146)	41270	15	300 (106, 194)	721	95.5 - 99.3	98.5 - 100
3	Movies2	3000	3000	14	284 (59, 225)	2997034	7	140 (21, 119)	752	95.97 - 99.86	100 - 100
4	CompVision	3547	10000	21	424 (109, 313)	5347	18	360 (107, 253)	186	90.12 - 91.59	99.41 - 99.41
5	Movies3	3140	3043	30	604 (270, 331)	675218	19	380 (163, 217)	867	60.21 - 68.08	87.52 - 94.21
6	Movies4	3507	3009	11	224 (68, 156)	26208	14	280 (63, 217)	303	93.37 - 95.39	97 - 98.46
7	Movies5	5000	4000	29	584 (189, 394)	70169	8	160 (44, 114)	2700	95 - 100	100 - 100
8	Movies6	4000	3894	17	344 (113, 231)	3179	17	340 (205, 135)	2121	100 - 100	98.9 - 100
9	Movies7	3250	3100	11	224 (59, 165)	716	19	380 (228, 152)	530	97.2 - 98.7	100 - 100
10	Books2	5934	6740	14	284 (54, 230)	12362734	7	140 (6, 133)	87	68.38 - 93.88	46.38 - 93.38
11	Movies8	3345	3095	10	204 (71, 133)	2012	18	360 (200, 160)	1017	98 - 100	100 - 100
12	Movies9	3475	3060	25	504 (152, 352)	867508	12	240 (71, 168)	1289	94.5 - 97.6	100 - 100
13	Restaurants	3808	5643	11	224 (60, 164)	93712	23	460 (145, 315)	508	92.71 - 97.19	94.93 - 99.81
14	Movies10	5000	5000	16	324 (130, 193)	464455	18	360 (132, 224)	1771	96.65 - 99.45	96.96 - 99.78
15	Movies11	3262	3100	13	264 (98, 162)	92607	26	520 (263, 257)	1543	95.07 - 100	89.97 - 98.70
16	Movies12	5000	5000	10	204 (71, 133)	8506	7	140 (56, 84)	601	98.86 - 100	100 - 100
17	Cosmetics	3212	3089	13	264 (77, 187)	162226	30	600 (234, 358)	954	65.77 - 74.87	92.24 - 97.75
18	Movies13	4319	3000	21	424 (109, 310)	5600	24	480 (184, 290)	1550	96.10 - 100	100 - 100
19	Movies14	3000	3071	11	224 (83, 141)	72055	7	140 (69, 70)	974	91.86 - 97.82	93.71 - 99.10
20	Cricket	4175	5597	30	604 (240, 364)	311722	19	380 (98, 281)	4050	99.10 - 100	98.50 - 100
21	AcadPapers	4000	4397	10	204 (40, 163)	1640824	7	140 (28, 111)	127	96.94 - 100	100 - 100
22	Movies15	4885	5396	10	204 (77, 127)	27414	7	140 (28, 111)	140	98.31 - 99.32	98.28 - 99.25
23	Books3	4753	5469	10	204 (61, 143)	7048	25	500 (185, 315)	595	95.66 - 99.57	96.21 - 100

Table 3.2: Experiments with CloudMatcher on Web data.

do simple steps like uploading the two tables, following the user interface, label tuple pairs as match/non-match and download the results at the end.

20 out of 23 teams had a recall in the range of 90-100%. The EM results as seen have been good to very good on some datasets, and not so good on some others. A close examination of these results reveals several interesting issues. First, in the case of a few datasets, the data was incomplete, i.e., dirty and was missing a lot of values. Second, for the "Books2" dataset run, the students reported to be not converging to a matching definition and this made the labeling non-uniform. As a result, the rules learned were not good and that we could see from the size of the candidate set as well.

Other observations reported by teams were as follows. All the teams reported spending 2-5 hours, for an average of 2.5 hours (including reading the instructions and labeling samples) on the end-to-end matching task. Along with the accuracy numbers, the teams reported non-functionality bugs, UX improvement, and suggested data cleaning steps to be the part of the system which we plan to implement in the near future.

3.4 Comparison With Existing EM Systems

In this section, we compare the CloudMatcher solution with two commercial EM systems X and Y. We do not have the permissions to disclose the name of these systems yet and hence we will use X and Y in the rest of the description. System X uses machine learning, whereas we are not sure of the underlying matching technology in System Y.

3.4.1 Comparison With Systems X and Y

We now compare the CloudMatcher solution with the systems X and Y. The system X is a cloud-based EM solution that uses machine learning technology to find matching records whereas we are not sure of the system Y. We now run the end-to-end matching task on two datasets: (i) UW Health, and (ii) Restaurants using the default configuration for all the three EM systems. For comparison, we currently focus on the accuracy matrix.

On the restaurant's dataset, CM consistently achieved a precision (P) of 100% and recall (R) of 99.10% whereas, on the system X, the numbers reported were P = 97.08% and the R = 89.28%. The accuracy numbers on system Y were P = 100%, R = 84%. Next, we run the three systems to do EM tasks on the UW Health dataset which was a case of deduplication. CloudMatcher recorded a precision P = 99.66% and a recall of 99.10%. On system X the performance numbers were P = 97.4% and recall R = 69.08% whereas on system Y the P = 100% but the R = 7.9%. For both the datasets, we observed that CloudMatcher outperformed the systems X and Y in terms of precision and recall. Specifically, the recall was very low for both the datasets on the system X and Y as compared to CloudMatcher. Note that it is important to mention that all the systems here were running in the default configuration mode and for system X and Y the processing time was much quicker as compared to CloudMatcher. Also, we believe that we could have improved the numbers by updating the default configurations but this would have incurred significant human time. For the same reason, we did not try the systems X and Y on other datasets.

3.5 Empirical Evaluation of the Blocking Component in CloudMatcher

In this section, we present the evaluation of the blocking component of the system. Our initial goal to evaluate the blocking step on a variety of datasets was to make sure that the major components of the system are working correctly and the overall recall is high after the blocking step. From the end-to-end EM workflow described in Figure 2.7, we observed that the matching step in the system reuses most of the blocking step component and if we are able to test the blocking step thoroughly, the components in the matching step will also be tested for correctness.

Toward this goal, we started out by testing the blocking step on a variety of datasets and report the size of the candidate set and recall of the blocking step. Now, we will describe the datasets we used for the evaluation. We considered three different types of real-world data sets in Table 3.3, which describe products (electronics, clothing, etc.), identity (matching person, organization or vendors), and non-identity (for e.g., matching citations in Citeseer and DBLP). For all the datasets, we had gold matches to compute recall.

Data Sets: Walmart-Amazon product describes electronics products and was used in the Corleone paper. Abt-Buy also represents electronic products obtained from Abt.com and Buy.com and was used in the DeepMatcher paper [73, 8]. Clothing_5000 and Electronics_5000 are a sample of 5000 real-world clothing and electronics data we obtained from a large organization. Amazon-Google also represents product data from Amazon and Google and was also used in the DeepMatcher paper.

The survey data set describes hospital and staff data obtained from domain scientists at UW-Madison and the Person data set describes person information (name, dob, phone, address, etc.) which was obtained from a large data integration company.

Next, for the non-identity datasets, we had Songs and three other data sets matching citations. Songs describe songs within a single table, contain 1M tuples, and was obtained from the source labrosa.ee.columbia.edu/millionsong. Then, we had citations in Citeseer and DBLP, DBLP and ACM, DBLP and Google Scholar to match. Most of the datasets used here are publically made available at [36].

Input tables				Run1		Run2	
Dataset	**Domain**	**|A|**	**|B|**	**|C|**	**Recall**	**|C|**	**Recall**
Walmart-Amazon	Products	2554	22074	1786	66.11	95313	93.41
Abt-buy	Products	1081	1092	3398	77.14	8335	91.53
Clothing_5000	Products	5000	5000	4254	91.09	52091	97.83
Electronics_5000	Products	5000	5000	44517	93.53	72287	96.97
Amazon-Google	Products	1363	3226	5251	83.69	13227	90.92
Survey	Identity	1786	1786	4718	97.49	4718	97.49
Person	Identity	48119	48119	18962	99.95	18990	99.98
Songs	Non-identity	1000000	1000000	1982338	97.83	1982348	97.83
Citeseer-DBLP	Non-identity	1823978	2512927	669812	98.79	877558	95.35
DBLP-ACM	Non-identity	2616	2294	2477	99.1	2419	99.4
DBLP-SCHOLAR	Non-identity	2616	64263	6800	92.5	6480	90.04

Table 3.3: Evaluating blocking step on products, identity and non-identity datasets

Table 3.3 summarizes the result of running the blocking step on 11 real-world EM tasks. The first two columns show that the system has been used on 5 product data sets, 2 identity data sets, and 4 non-identity data sets. The next two columns show the number of tuples in the two tables used for matching. Then, we have two columns "Run1" and "Run2", under which we report the size of the candidate set and recall achieved on running the blocking step. We define "Run1" to be the run with the default configuration of the blocking step whereas "Run2" being another configuration that was fine-tuned for the workload. Now, we will "zoom in" to the size of the candidate set and recall obtained for each runs.

In"Run1", we obtained a recall of more than 90% for 8 out of 11 datasets. Specifically, for the identity and non-identity datasets, the recall was very high i.e., more than 95% in most cases except for the DBLP-Scholar dataset. On the other hand, the recall obtained for three product datasets (Walmart-Amazon, Abt-buy, and Amazon-Google) was very low. It turned out that the data for these datasets was very textual, had missing values, and the blocking rules learned in the default configuration were very strict and were pruning away a lot of tuple pairs. We could see that the rules were pruning away a lot of tuple pairs by looking at the size of the candidate set. For example, for the Walmart-Amazon dataset, we had two tables with 2,554 and 22,074 tuples. With "Run1" we obtained a candidate set of 1,786 tuple pairs and a recall of 66.11%.

To improve the recall, in "Run2", we fine-tuned the default configuration to use less strict rules and then we ran the blocking pipeline again for all the datasets. We were able to achieve higher recall for all the datasets including products (i.e. recall greater than 90%) but we saw an increase in

the size of the candidate sets. For the same Walmart-Amazon dataset, running through the "Run2" configuration, we got a candidate set of 95,313 tuple pairs and a recall of 93.41%. As we can see the recall was increased by 27% but the candidate set size was increased by more than a factor of 50. We observed a similar pattern for other datasets as well.

3.6 Lessons Learned

From our experience with CloudMatcher so far, we have learned a set of interesting lessons. Some of the lessons are not so surprising. For example, users would like to have a way to estimate accuracy (e.g., precision and recall) at the end of the EM process. This is understandable and we plan to implement the crowdsourced accuracy estimation procedure in Corleone [50]. Other requests include ways to store EM models, data, and results, and a dashboard to monitor the EM process in real time. Other lessons that we have learned are more significant, as we describe below.

Debugging and Explaining: If there is some problem, such as low recall on Addresses, low precision on Vendors, or the blocking process taking too long on Drugs, the user is often at a loss as to why. They highly desire tools that they can use to debug or find explanations, so that they know what they can do next to improve the EM process.

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

Second, the user may also think he/she knows the problem, i.e., the "match" definition, e.g., what it means for two tuples to match. But we have found that this is rarely the case in practice, and it can have serious consequences. For example, the user thinks he/she knows the match definition. So he/she will write an instruction to crowd workers based on that knowledge. Then crowd workers run into ambiguous cases not covered by the instruction (e.g., addresses that are the same except

P.O. Box numbers in Addresses). They do not know what to do. So some will say yes, and some will say no. As a result, CloudMatcher learns incorrect blocking rules and matching models, which in turn seriously degrades the quality of the EM process. Some crowd workers may ask about such ambiguous cases in their emails. Often that is when the user realizes that the current match definition is not complete. He/she may need to revise it, then run the EM process again, incurring unnecessary time and expenses.

Finally, the user may have no idea what a tool such as CloudMatcher is capable of. Recall that CloudMatcher produces precision of 94-96% and recall of 95-98% on People dataset. Suppose the user wants to increase precision to 99%. Can he/she still use CloudMatcher? Or would it already reach its limit and a new tool needs to be explored?

A New Way to Do EM? As a result of these observations, we do not believe practical EM can be done in "one shot". Instead, it appears to require multiple iterations. Each iteration is used to gain increasingly more knowledge about the data, the problem, and the capabilities of the tools. To do so, in each iteration we must have an arsenal of tools to help users profile, explore, and understand these three targets. We must also have a clear methodology to guide users on how to use these tools.

How to execute such multiple-iteration processes is an interesting question. Perhaps at the start, a system (such as CloudMatcher) can help the user execute EM on small data samples (selecting a variety of data samples so that the user can be exposed to all the diversity in the data, the problem, and the tool capabilities). Subsequent iterations can operate on large samples, and then eventually when the user has been satisfied, then a final EM process over the entirety of the data is launched. This is an open research question. But we believe solving it is critical for successful EM in practice.

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 (e.g., as in the Addresses dataset), incorrect data (e.g., Brazil data in Vendors), and incomplete data that even domain experts cannot match (e.g., as in the Vendors dataset). 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 (as discussed earlier), then ways to help the user "split" the data into different parts and develop different EM strategies for different parts.

User Needs Iteration with Crowd Workers: For crowd workers, we found that the most serious problem is giving them clear instructions of what it means to be a match. As discussed earlier, at the start the user often does not have a complete knowledge yet of what it means to be a match. So he/she will give incomplete instructions to the crowd. This can have serious consequences, as discussed in that section.

As a result, we believe the problem of how to work with the crowd to arrive at the clearest instructions possible (as quickly as possible) is critical to enable practical crowdsourcing. Ultimately, the larger challenge here is that working with a crowd is not an "one-way" street. The user needs to be interacting with the crowd, possibly in multiple iterations, in order to perform EM efficiently.

More Expressive UIs: For in-house users (who label the tuple pairs), we found that they do not like the labeling UI (of seeing tuple pairs and labeling them). They find this wasteful and inefficient. To explain, observe that most current crowdsourced EM works do not consider the time it takes for a crowd worker to *understand* a tuple pair. They often focus instead on minimizing the total number of pairs that the crowd must label (as we also do in this work).

In practice, however, there is a real cost of trying to understand a tuple. For example, in the Drugs dataset, each drug description is a complex tuple that describes many ingredients. A domain expert needs 5-6 seconds to understand such a drug. Suppose this expert has just labeled drug pair (a, b), then suppose 3 minutes later he/she needs to label another pair (a, c). In this case the expert needs to spend time trying to understand a again, i.e., recognizing that this new tuple a is the same as the old tuple a. This wastes the expert's time and cause resentments. A suggestion that we have heard is to have a more expressive and efficient UI, e.g., one that shows a cluster of drugs, so that an expert needs only to try to understand a drug once, yet can still efficiently match it with many

other drugs. It is interesting to explore whether more expressive UIs like this can work with crowd workers as well (e.g., on AMT), not just with in-house workers.

The Promise of Self-Service EM: The idea of just having to label a set of tuple pairs sounded very appealing, and our customers were enthusiastic about using CloudMatcher. CloudMatcher also seemed to perform well on a broad range of EM tasks. We concluded that self-service EM is a great way to start the "EM journey". If a user has only one-shot or short-term EM needs, then perhaps it is best to start out trying CloudMatcher. If CloudMatcher already achieves the desired accuracy, then great. Otherwise, the user can look into using PyMatcher, the more powerful system [52].

Challenges in Data Cleaning and Labeling: Our experience also made clear that data cleaning is critical for EM (e.g., see the "Vendors" and "Addresses" cases). It is important that we can detect dirty data, isolate it, and then clean it, to maximize EM accuracy. But how to do so in a self-service fashion is still unclear.

We also want to reduce the number of pairs that users must label. In the case of matching two sets of strings (a special case of EM), we have developed Smurf, which removes the need to label to learn blocking rules. It only needs labeling to learn a random forest-based matcher [25]. This drastically reduces the labeling effort by 43-76%, yet achieving the same accuracy. We are currently extending Smurf to match tuples, and incorporating it into CloudMatcher.

Challenges in Developing a Monolithic System: Our biggest challenge, however, is that CloudMatcher *has been slowly growing into a big stand-alone monolithic EM system, exactly the kind of system we would like to avoid building.* Its code base is large (47K LOC, as discussed in Appendix D), and its many modules are highly interdependent. We found that it is increasingly hard to understand, maintain, and extend CloudMatcher. This is exacerbated by developing it in academia, where most students do not stay for long.

Another problem with CloudMatcher being a monolithic system is that we cannot easily use "pieces" of it. For example, most basic services of CloudMatcher (e.g., learning blocking rules, executing blocking rules, labeling tuple pairs, etc.) would be very helpful for both the development

and production stages of PyMatcher. But to use any one of them, a PyMatcher user would need to install the entire CloudMatcher, too much overhead. It is also cumbersome to quickly get data into and out of CloudMatcher for some of these services (e.g., to move data between CloudMatcher and PyMatcher).

The above problems are not new. They commonly arise in stand-alone monolithic systems. But as we built and worked with CloudMatcher, we experienced them first hand. This experience further motivated the Magellan approach of developing ecosystems of interoperable tools.

Toward Cloud-Based Interoperable Microservices: To address the above "monolithic system" challenges, we are leveraging ideas from a recent trend in building data science systems. Specifically, many DS projects have started to adopt a microservice software approach, where the code is decomposed into a set of *microservices*, which are self-contained but interoperable services, each doing just one task [75]. Tools have been developed to easily deploy such services (e.g., Docker) and to coordinate their execution on the cloud (e.g., Kubernetes) [57].

In short, many DS projects are moving to the cloud, and a support infrastructure is emerging to help build "cloud native" applications, which are composed of interoperable microservices that can be smoothly deployed, executed, and scaled out on the cloud.

Chapter 4

Developing Atomic Services for EM

In this chapter, we present the rationale behind developing atomic services for EM and then present scenarios where the users benefit from these atomic services. We then briefly discuss how these individual services can be hosted on a third-party platform such as Columbus and can be used as a component in a variety of data science workflows.

4.1 Developing an EM System for Multiple Users

Version 1.0 of CloudMatcher implemented the end-to-end Falcon EM workflow. As we interacted with real users, however, we observed that many users want to flexibly customize and experiment with different EM workflows. As a result, in Version 2.0, we solved this problem by (a) extracting a set of basic services from the Falcon EM workflow and making them available on CloudMatcher, and (b) allowing users to flexibly combine them to form different EM workflows, including the original Falcon one.Figure 4.1 shows examples of services that we currently provide. *Basic services* include uploading a dataset, profiling a dataset, edit the metadata of a dataset, sampling, generating features, training a classifier, etc. We have combined these basic services to provide *composite services*, such as active learning, obtaining blocking rules, and Falcon (see the bottom of the figure). For example, the user can invoke the "Get blocking rules" service to ask CloudMatcher to suggest a set of blocking rules that he/she can use. As another example, the user can invoke the "Falcon" service to execute the end-to-end Falcon EM workflow. These services are listed in more detail in Table 4.1. In the later sections, we will discuss how we use some of these individual services to do tasks in the EM space.

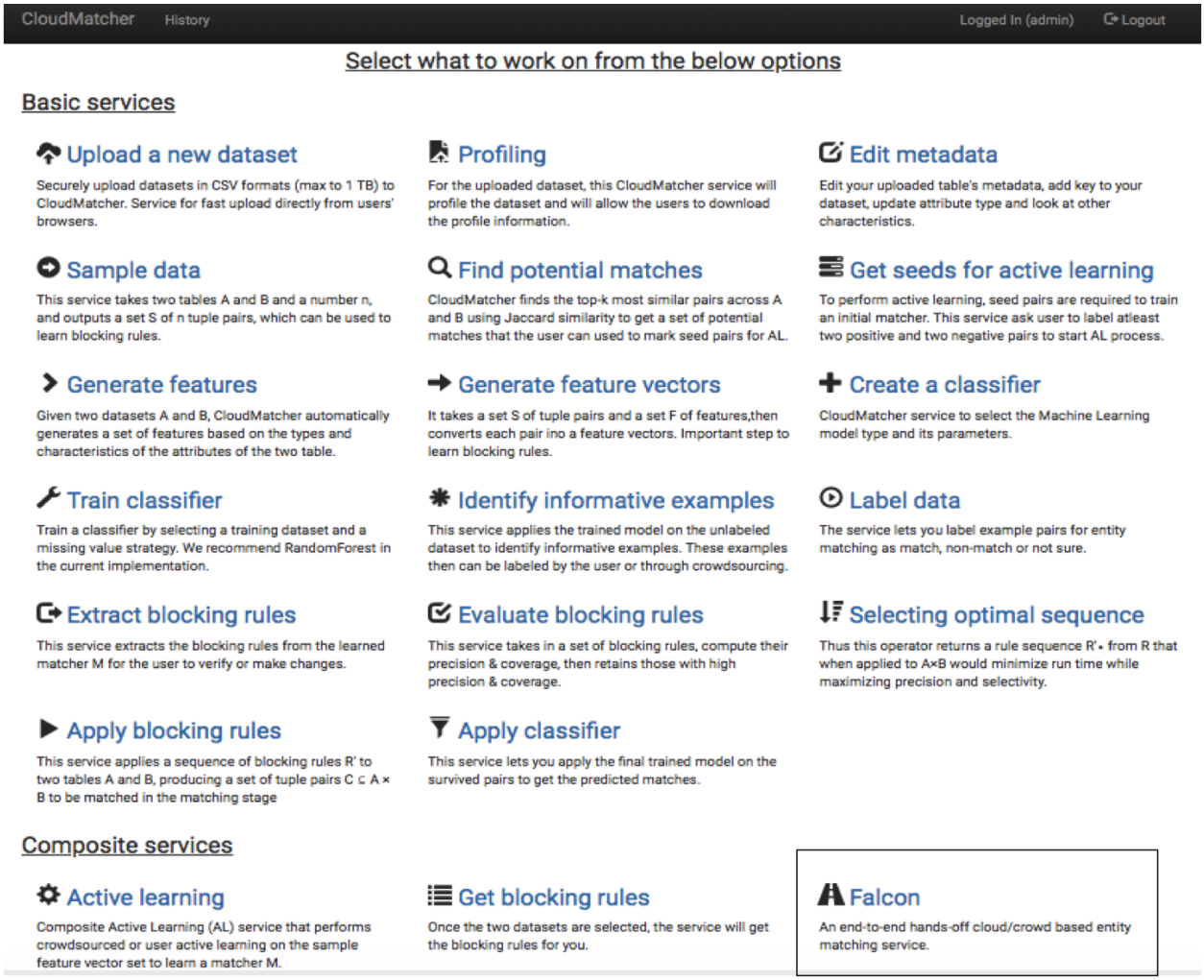


Figure 4.1: The services of CloudMatcher.

List of Services in CloudMatcher Table 4.1 shows the list of services that we currently have in the system. From the table, we see that we have extracted a bunch of basic services from the end-to-end EM Falcon service. These services could be run individually or could be combined into a workflow in the EM space. Currently, we combine these services using an Adhoc script but this could be improved in the future by providing a rich Web user interface. The system is designed in a way that allows users to add/remove services without much manual effort. In addition, the system maintains metadata, logging, and other interesting properties for each of these individual

Services	Description
Upload dataset	Uploads dataset into the system for matching, labeling, etc.
Profiling	Profiles the input dataset to provide more insights on the data.
Edit metadata	Verify/add primary keys, attribute types, etc.
Sample data	Get sample pairs or down-sample two tables for matching purposes.
Find potential matches	Finds the top-k most similar pairs across A and B using Jaccard similarity to be used to select seed pairs for active learning (AL).
Get seed pairs for active learning	Recommends matching/non-matching pairs to the user to select seed for AL.
Generate features	Generates a set of features based on the types and characteristics of the attributes of the two table
Generate feature vectors	Take a set S of tuple pairs and a set F of features, then converts each pair into a set of feature vectors.
Create a classifier	Allows the user to select scikit-learn based machine learning model and its parameters.
Train classifier	Train a classifier by selecting a training dataset and a missing value strategy.
Identify informative ex.	Applies the trained model on the unlabeled dataset to identify informative (controversial) examples.
Labeling service	This service provides two modes of labeling. One is where the user labels the example pairs and other is using crowdsourcing platforms.
Extract blocking	Extracts the blocking rules from the learned matcher M. These rules can be evaluated later or used as it is for blocking purposes.
Evaluating blocking rules	This service takes in a set of blocking rules, compute their precision & coverage, then retains those with high precision & coverage.
Selecting optimal sequence	This service returns a rule sequence $R'*$ from R that when applied to $A \times B$ would minimize run time while maximizing precision and selectivity.
Apply blocking rules	This service applies a sequence of blocking rules R' to two tables A and B, producing a set of tuple pairs $C \subseteq A \times B$ to be matched in the matching stage.
Apply classifier	This service lets you apply the final trained model on the survived pairs to get the predicted matches.
Cross validation	Performs cross-validation and learning of a best model over a sample of dataset.
Active learning	Composite Active Learning (AL) service that performs crowdsourced or user active learning on the sample feature vector set to learn a matcher M.
Falcon	Falcon service executes the end-to-end entity matching workflow on the two datasets to identify matching records.

Table 4.1: List of services in CloudMatcher.

services in the metadata store. We believe that this information could be used to further improve the system, services, and user experience.

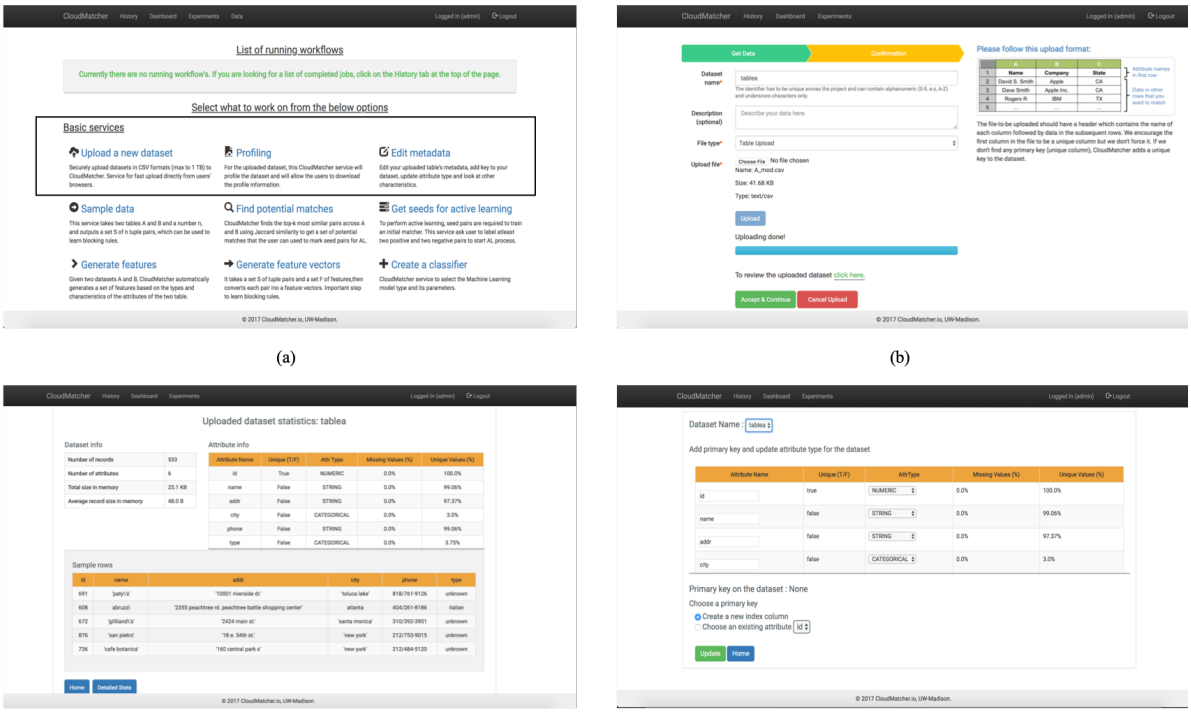


Figure 4.2: (a) The initial three services to use before starting the matching process (b) Upload service (c) Profiling service (d) Edit metadata service.

In the rest of the section we will show that (a) it is easy for a lay user to perform EM on CloudMatcher, end-to-end, via interactive labeling, or by using a crowd of workers; and (b) the user can easily customize and experiment with a wide range of EM workflows, by combining CloudMatcher services. We will focus on the scenario of matching two tables.

4.1.1 How Lay Users Can Easily Perform EM with CloudMatcher

In this section, we show how easy it is for a lay user to perform EM using CloudMatcher. First, the user goes to the CloudMatcher Web site. Then, as a first step, we ask the user to upload two tables, profile them, and then edit the metadata if needed. This is shown in Figure 4.2.(a) - (d). Once these steps are completed, we then ask the user to use the Falcon service to perform the end-to-end matching task. This is shown in Figure 4.3.(a) - (d). In this service, the user starts by selecting the seed pairs first as they are required to start the learning process. Next, the user

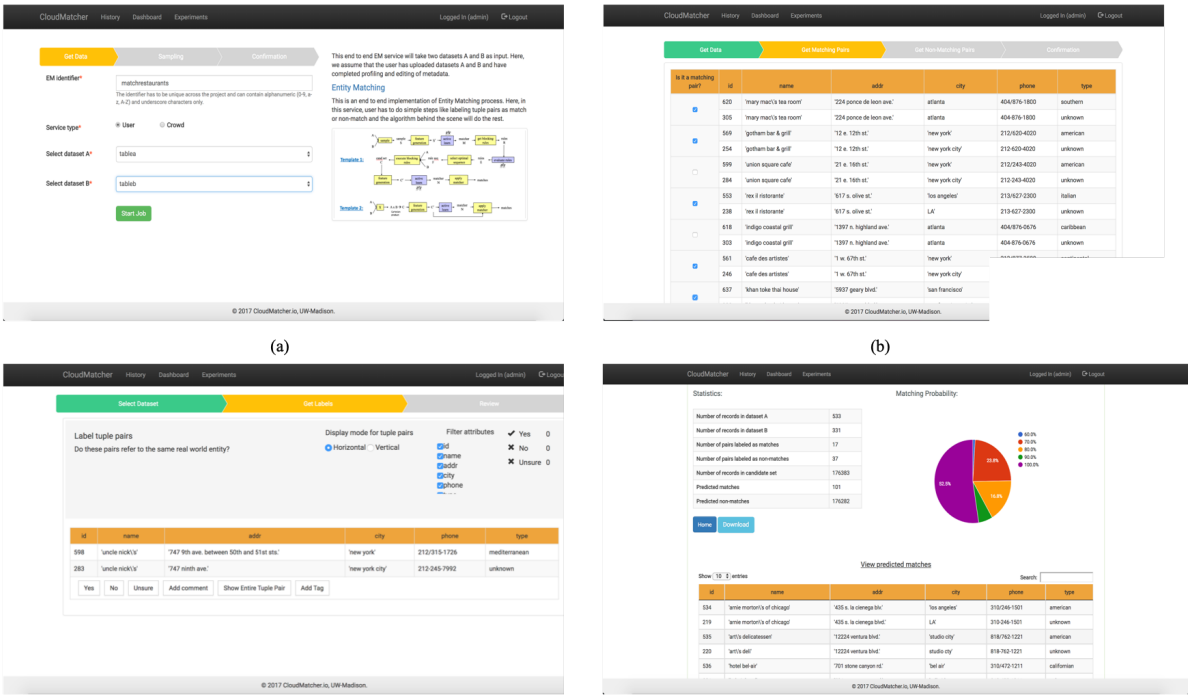


Figure 4.3: (a) Falcon service (b) Seed selection (c) Labeling for active learning (d) Matching results page.

interactively labels tuple pairs to perform EM. Figure 4.3.(c) shows the current labeling interface of CloudMatcher. In particular, it shows one tuple pair that the user can label as yes/no/unsure, and so on. Finally, after completing a few rounds of labeling, the user will get the predicted matches. As you can notice, the user has to do simple steps to perform EM using CloudMatcher.

The user can even decide to outsource the labeling task to a crowd of workers. In that case, the scenario will demonstrate that with crowdsourcing, performing EM on CloudMatcher is as easy as uploading the tables, doing some pre-processing, then providing a credit card to pay for the crowd.

4.1.2 How Lay Users Can Create and Experiment with Many EM Workflows

The above scenarios demonstrate how CloudMatcher executes the Falcon EM workflow (see Section 2.1.2). In practice, as mentioned earlier, while many users are satisfied with executing

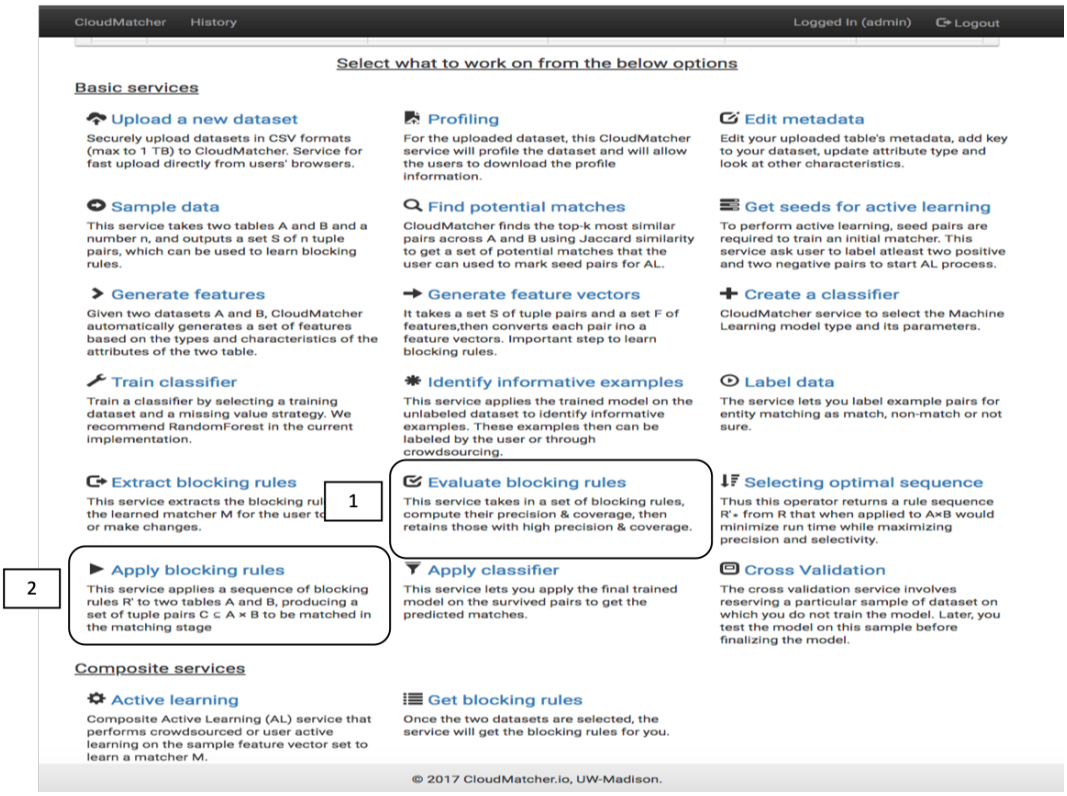


Figure 4.4: CloudMatcherservices to run for scenario 1

just this workflow, many other users want to create and experiment with different EM workflows. In this section, we will show two scenarios to demonstrate that lay users can indeed do so, by combining the basic services of CloudMatcher in different ways.

Scenario 1: We will demonstrate a scenario in which a user wants to match two tables of restaurant descriptions. Earlier, to do so, CloudMatcher would have started by asking the user to label tuple pairs so that it can learn a set of blocking rules. Here, however, suppose that the user already knows a good blocking rule, namely, R_1 = "if two restaurants disagree on the value of the city attribute, then they will not match and hence should be blocked". Thus, the user wants to skip the step of learning blocking rules, and use the blocking rule R_1. For this scenario, the user would run the following two individual services shown in the Figure 4.6 one after the other.

However, just because the user thinks R_1 is a good rule does not mean it is indeed a good rule. It can be a bad rule for multiple reasons, such as many values of city are missing or misspelled.

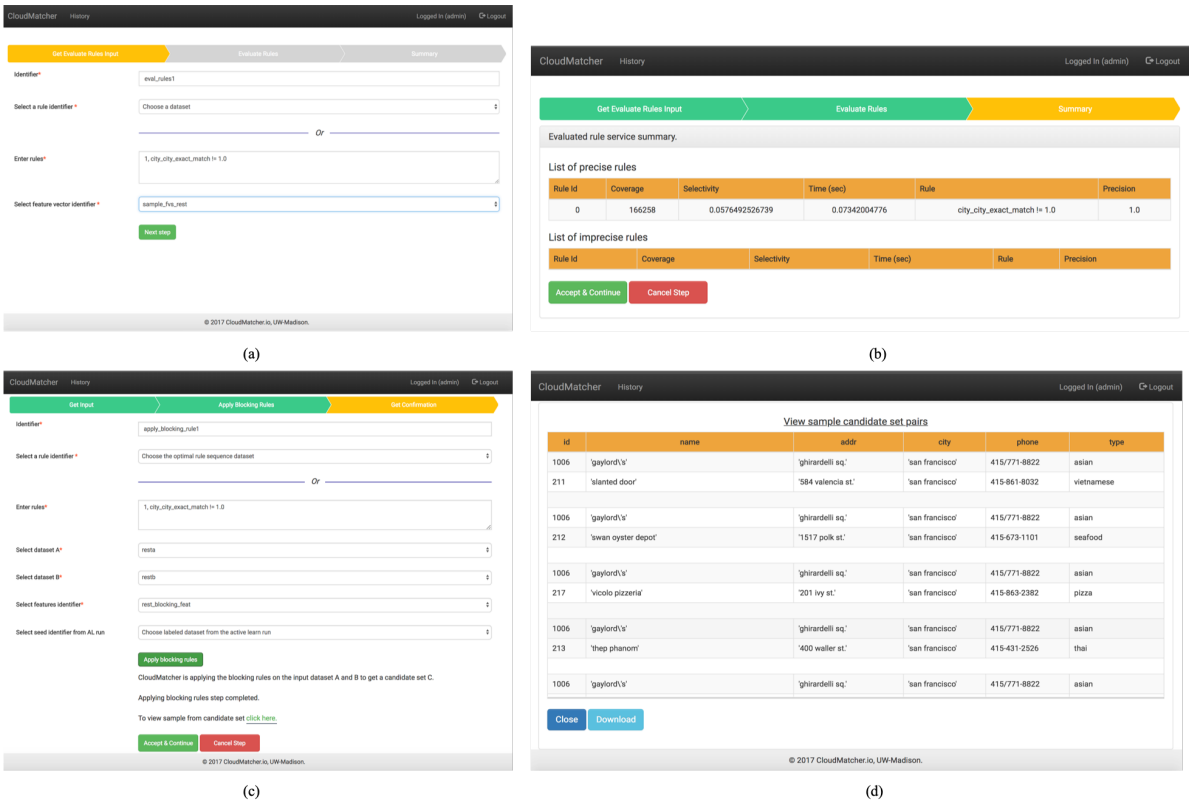


Figure 4.5: (a) How a user enters a blocking rule to be evaluated. (b) Evaluation result for the rule. (c) The service to apply the blocking rules (d) Viewing the candidate set.

As a result, the user first invokes the basic service "Evaluate Blocking Rules", and enters rule R_1 (see the screen shot in Figure 4.5).(a). CloudMatcher then asks the user to interactively label a set of tuple pairs, and uses these labeled pairs to evaluate the quality of R_1. The results of evaluating the rule are shown in Figure 4.5.(b). If R_1 turns out to be a good blocking rule, then the user can invoke the basic service "Apply blocking rules" 4.5.(c). to perform blocking using R_1, view the candidate set 4.5.(d), then invoke other basic services to learn a matcher and apply the matcher. Thus, this EM workflow differs from the Falcon workflow in that here the user directly supplies a blocking rule (rather than learning it, as in Falcon).

Scenario 2: To continue with the above scenario, let C be the set of tuple pairs obtained after applying the blocking rule R_1 to the two input tables. Earlier, executing the Falcon EM workflow, CloudMatcher would have interacted with the user in an active learning fashion on C to learn

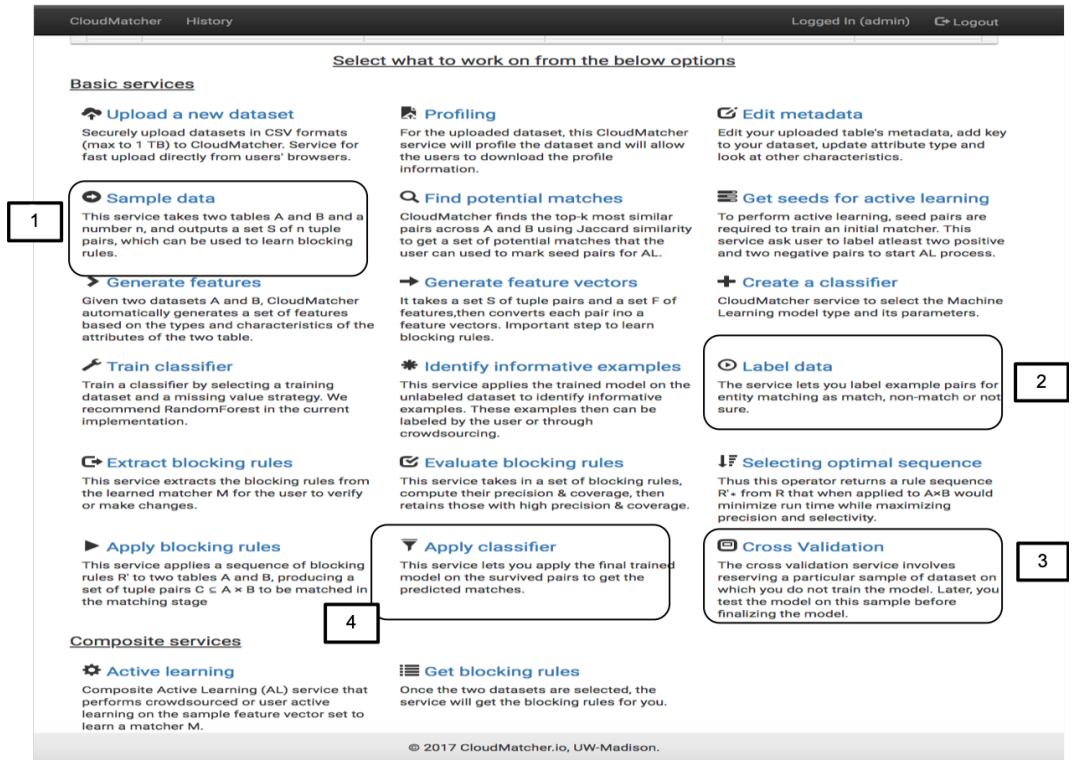


Figure 4.6: CloudMatcherservices to run for scenario 2

a matcher M, then apply M to C. However, suppose the user does not want to perform active learning. Rather, the user wants to take a random sample S from C, labels S using crowdsourcing, splits S into two sets I and J, uses I to train a matcher M, applies M to J to compute precision and recall on J, then uses these numbers to estimate precision and recall on C. (The paper [50] shows how to perform this estimation, and the Magellan EM system [65] supports such EM workflows.) In the Figure 4.7(a) - (d), we will show how the user can invoke the basic services of CloudMatcher to execute the above workflow in CloudMatcher.

4.2 EM Services Hosted on Columbus

While working with citizen data scientists it was learned that they typically use multiple data environments to do EM work. For example, they start working on their laptops, desktop's local Python environment using the PyData ecosystem to generate or test the EM workflow. Next, they move to AWS or similar to run the workflow at scale, and finally download the result and save it

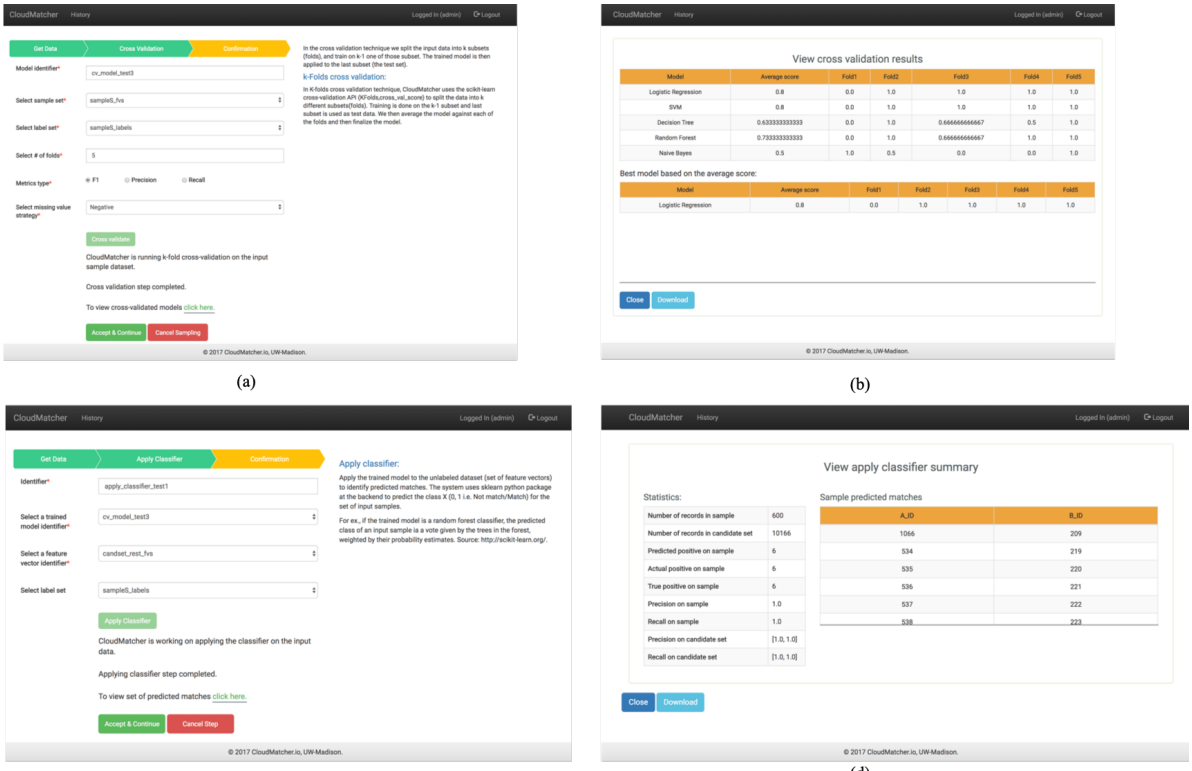


Figure 4.7: (a) Use of cross validation service (b) Results from the cross-validation service (c) The service to apply the model (d) Viewing the predicted matches summary.

on S3 or Google Drive to share it with others. This is a very common practice used by many data scientists and will continue, but this raises many important challenges such as, the scientists should be able to move data seamlessly among the different environments, should be able to use multiple tools from different ecosystems, should be able to work in a collaborative environment, and many others.

In this direction, Columbus [21]: a Platform as a Service solution for data exploration, cleaning, and integration is being developed at the University of Wisconsin - Madison in collaboration with many others. This initiative provides a scalable, collaborative, cloud-native and programming language agnostic extension of the PyData ecosystem which is a very popular and growing ecosystem of Python tools to do data science.

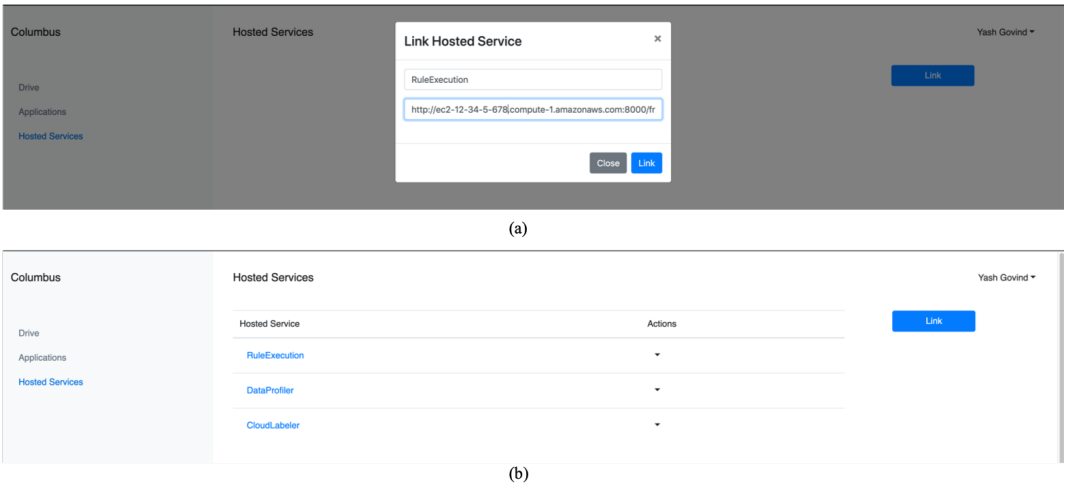


Figure 4.8: (a) Linking a service to CDrive account (b) List of hosted services

The major component of the project is the CDrive application (which is very similar to Google Drive) that is deployed on AWS and is private to each user. The CDrive application lets the user download and install other apps on DockerHub to his/her account which can be invoked to perform data science tasks. In practice, some of the apps could be private and not available for download. In these cases, CDrive provides instructions on how you could make your app CDrive compliant and provide it as a hosted service.

Although CloudMatcher version 2.0 extracted a set of basic services from the FalconEM workflow and gave the flexibility to the users to combine them to form different EM workflows, it was still limited by the use of single data environment and didn't provided the flexibility to use these services with other apps or packages outside CloudMatcher. In addition, we foresee that the code base for CloudMatcher would eventually be very big making it unmanageable and difficult to deploy/debug.

To address the above limitations, I have extracted a few important services from CloudMatcher that can be run as individual services, can be called by using simple APIs, and are available as hosted service on the CDrive application to do different tasks in the EM space. In addition, I have created a template that could be used in the future to extract other individual services from CloudMatcher or develop new services that could be hosted on CDrive. This work was done in collaboration with another Ph.D. student Kaushik Chandrasekhar.

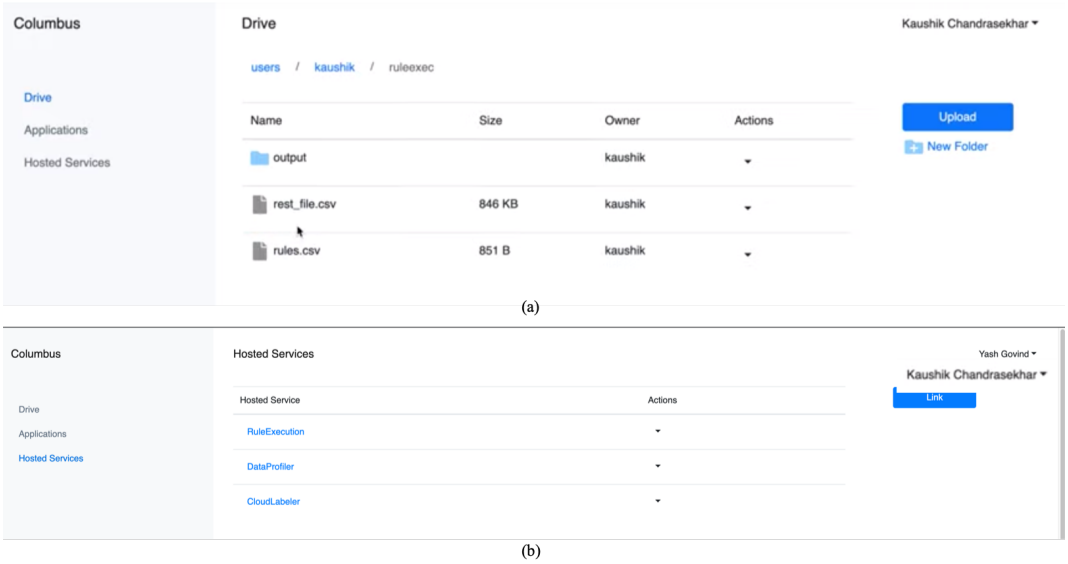


Figure 4.9: (a) Uploading the data to CDrive (b) Using the RuleExecution hosted service

As a CDrive user, the bare minimum thing that you need to do is to link the individual service as hosted service on the CDrive account as shown in Figure 4.8 (a). If the services are CDrive compliant you will see the linking process to be successful and the list of services will appear in the hosted services tab on your account going forward as shown in Figure 4.8 (b). The list of services that are available as hosted apps on CDrive account currently is: data profiling, labeling, and execution of blocking rules.

In the rest of the section, I will demonstrate how easy it is for a user to use these hosted services and execute their EM workflow with a few demo scenarios.

Scenario 1: Consider a scenario where the data scientist wants to match two tables A and B. The scientists use the PyMatcher system [52] to start working on the EM workflow and come up with blocking rules. Now, when he or she starts executing the workflow it runs very slow due to the package behavior or the system is not powerful. Then, the user may want to push the computation on a bigger machine or to a machine cluster. This could be a time-consuming process as the user may have to write code to achieve the desired parallelism on a cluster of machines. In addition, the user may even have to write code to push the data to a distributed file system such as HDFS or S3.

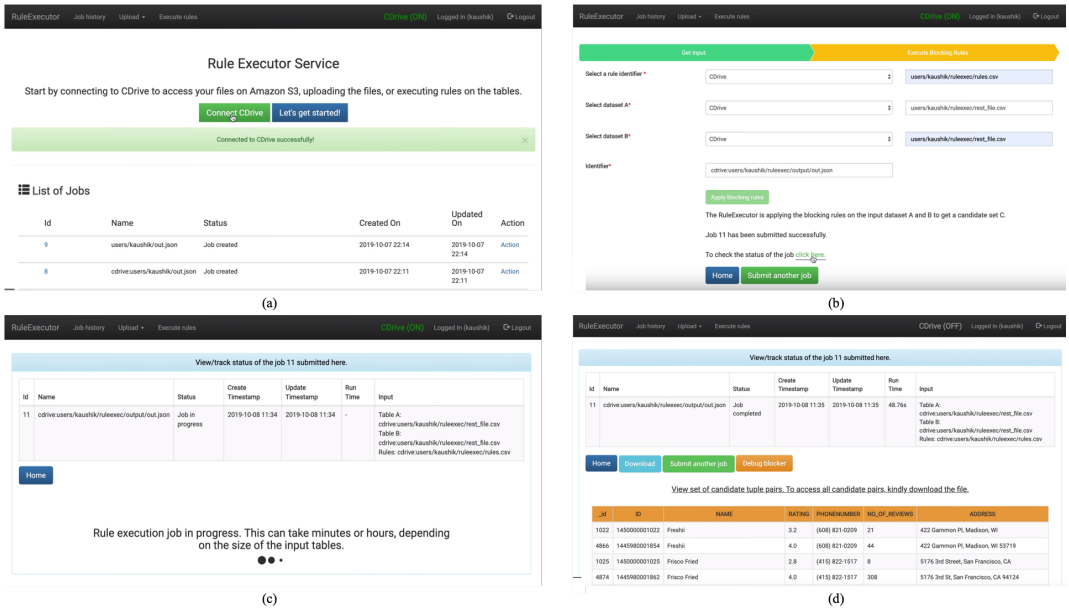


Figure 4.10: (a) The service (b) Input data to run the service (c) Checking the status of job (d) View or download the results

Instead, the user can get into the CDrive account, upload the two tables A and B, and the rules and could invoke the "RuleExecution" service as in Figure 4.9 (a) and (b). Next, the user sees the user interface of the "RuleExecution" service. The user then clicks on the Rule Execution tab, give the required input to run the job and click on the "Apply blocking rules" button. This submits the job to a cluster of machines and the interface allows the user to check the status of the job. Once the job is completed the user can view, download or debug the output. Figure 4.10 (a) - (d) shows how this is done in practice. As you can see, the user has to do the minimal steps in order to make progress to his EM workflow.

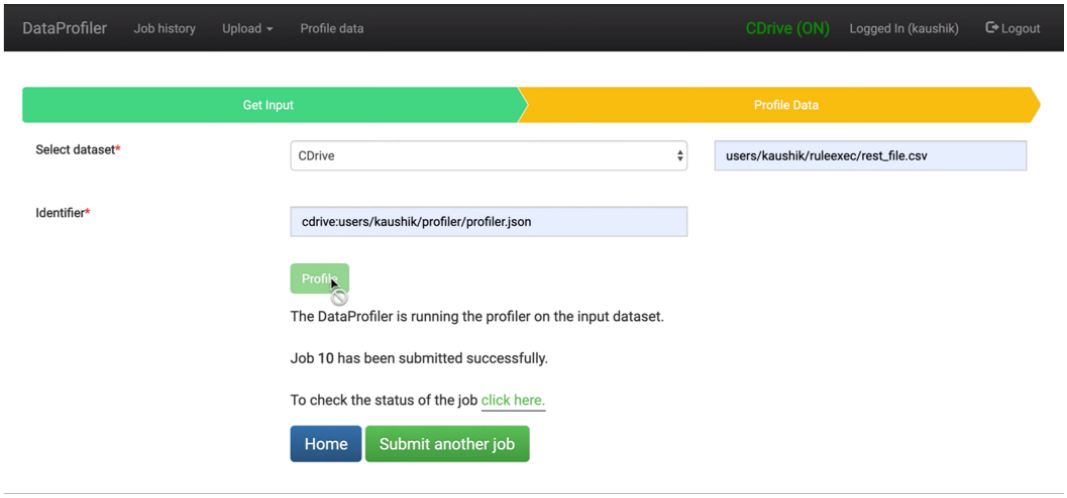
In another setting, the user could call the service APIs to execute the rules on a cluster of machines.

Scenario 2: Another common scenario that we have seen is when the user wants to profile and get an understanding of the data. The profiling steps can highlight missing values, uniqueness, get attribute type, etc. In practice, the tables can be large and profiling could be time consuming. Moreover, the users want to share the profiling information with other members of the team so this

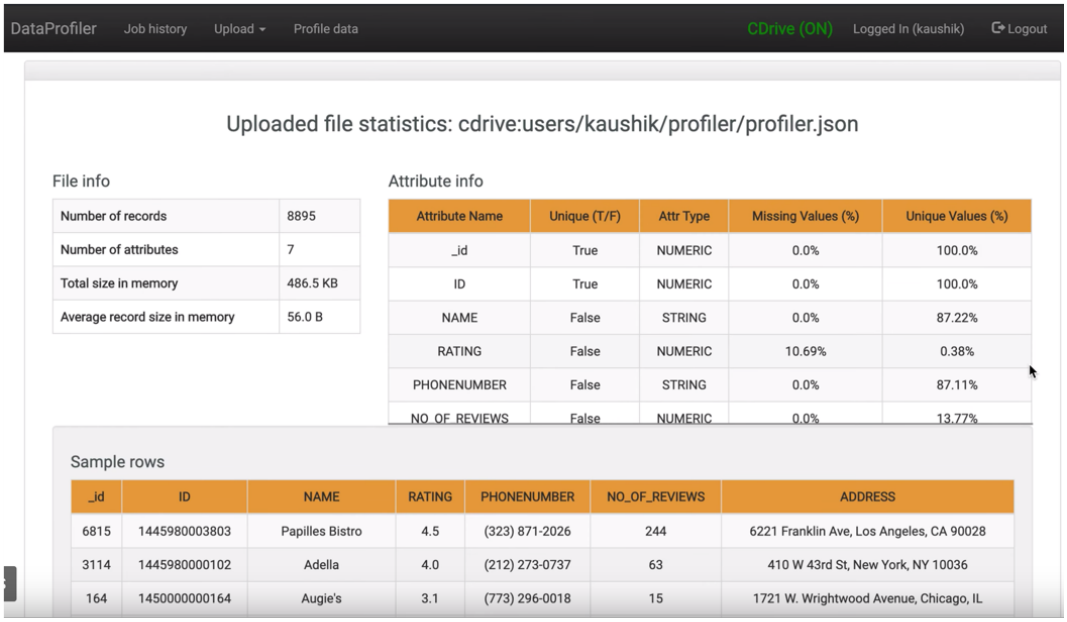
could easily take up a lot of time as they have to first put the big file on a shared file system, then run the profiling (which could take hours), and then share the results again.

This being a common practice led us to develop the “Profiling” service that could be used with CDrive with minimal user effort. Similar to scenario 1, the user has to upload the files to CDrive once. Then, the user could invoke the “DataProfiler” service to profile the data and view/download the result as you can see in Figure 4.11.

The service also allows users to get detailed statistics using the pandas-profiling package [16].



(a)



(b)

Figure 4.11: (a) Input data to run the service (b) View or download the results

Chapter 5

Scaling up Blocking Rule Execution

In this chapter, we will focus on scaling up the execution of blocking rules on two tables A and B for the entity matching task. This problem is time consuming and raises major scalability challenges.

5.1 Problem Definition

In the entity matching workflow, once the blocking design phase has been completed (as explained in Section 1.1.1), we have a blocking rule sequence $R = \{R_1, R_2, \ldots, R_n\}$. Then, we apply the sequence R to two tables A and B, producing a set of tuple pairs $C \subseteq A$ x B to be matched in the matching stage. In practice, the two tables A and B can be large and applying R in a naive way to all pairs in A x B is clearly impractical.

Previous work such as Falcon [37] has proposed solutions to scale up the execution of blocking rules but has examined table size of up to only 2-3M tuples. It has not looked into data with 10M tuples or beyond. In this work, we propose a solution that can execute rules on tables with 10M tuples each and even beyond. Another issue with Falcon is that it provides a Hadoop-based solution that is no longer the state of the art. Besides, the solution doesn't provide much control to the end-user to have a trade-off between job run-time and accuracy. In this work, we propose a solution that provides more control on a job level to the user by providing knobs that the user could control and choose between runtime and accuracy.

Now, we are going to look at this problem of scaling up blocking rule execution and to be concrete, we will assume that all the data (input tables A, B, and the rules R, etc.) resides in

MongoDB and we are going to run this using Spark. MongoDB is a cross-platform document-oriented NoSQL database and uses JSON-like documents with a schema. The reason we are using this setting is that many enterprise customers we interacted with are using this setting where the input tables and the blocking rules are stored in MongoDB as three separate collections. Then, they have a Spark cluster of N nodes for executing the blocking rules over the input tables A and B.

In the rest of the chapter, we will start by discussing the blocking rules, the key ideas underlying our solution, an end-to-end solution using MapReduce in Falcon and then some related work. Next, we will propose a solution to execute blocking rules at large scale using MongoDB and Spark. Then, we will discuss challenges and proposed solutions to overcome the challenges. Finally, we will examine the scaling properties and effectiveness of the proposed solution and then conclude the chapter.

5.2 Preliminaries & Related Work

In this section, we will define blocking rules, discuss the key ideas behind the solution, how Falcon executes blocking rules using MR and then will briefly discuss related work.

5.2.1 Blocking Rules

Recall that at the end of the blocking phase we get a sequence of blocking rules. Each rule in the sequence R_i is of the form:

$$p_i^1(a,b) \wedge \ldots \wedge p_{m_i}{}^i(a,b) \rightarrow drop(a,b), \tag{5.1}$$

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

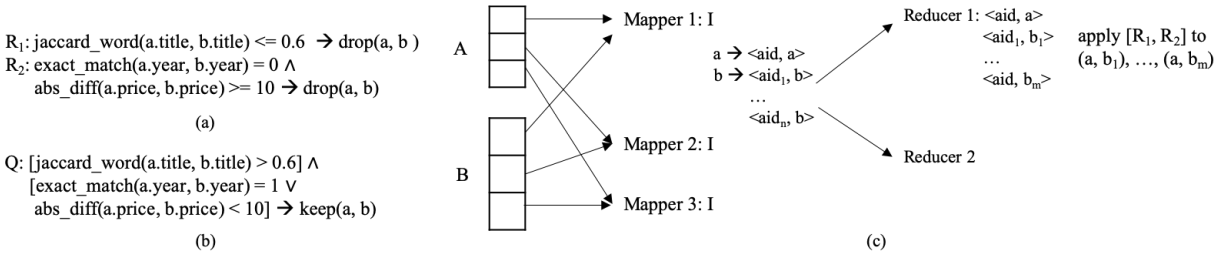


Figure 5.1: (a) A rule sequence, (b) the same rule sequence converted into a single "positive" rule, and (c) an illustration of how the baseline solution works.

Example 5.2.1. *Consider the sequence of two rules [R_1, R_2] in Figure 5.1.a. Rule R_1 states that two books do not match if their titles are not sufficiently similar (using a Jaccard similarity function over the two titles tokenized as two sets of words). Rule R_2 states that two books do not match if they disagree on years and their prices differ by at least $10 (here exact match(a.year, b.year) returns 1 if the years match and 0 otherwise, and abs diff(a.price,b.price) returns the absolute difference in prices).*

5.2.2 Key Ideas Underlying Our Solution

As seen in the example, the blocking rules in the current solution uses features that often use well-known similarity functions (e.g., edit distance, Jaccard, etc.). Thus the idea is to exploit certain properties of these functions to build index-based filters, then use them to avoid enumerating A x B.

Example 5.2.2. *Suppose we need to find all pairs in A and B for the movie dataset that satisfy the predicate jaccard_word(a.title, b.title) $\geq$ 0.7. From [100], we know that for a pair of string (x, y), jaccard(x, y) $\geq t \implies |y|/t \geq |x| \geq |y| \cdot t$. Knowing this property, a length filter (i.e. length of tokens) can be built for the above predicate. To be concrete, a B-tree index I_1 is built over the lengths of tokens of the attribute a.title. Now, given a tuple b in B, the filter uses I_1 to find all tuples a in A where the length of a.title falls in the range [[$|b.title| \cdot 0.7$, $|b.title|/0.7$], then return only these (a, b) pairs. Then the actual rule jaccard_word(a.title, b.title) $\geq$ 0.7 is applied only to these pairs.*

5.2.3 Falcon Solution Using MapReduce

The CloudMatcher service to execute blocking rules uses "apply blocking rules" operator from Falcon [37]. We will next descibe how Falcon builds an end-to-end solution to execute blocking rules using the above key ideas.

1. Convert the Rule Sequence into a CNF Rule: The first step in the Falcon solution is to rewrite the rule sequence $R = \{R_1, R_2, \ldots, R_n\}$ into a form that is amenable to distributed processing in subsequent steps. Specifically, it rewrites R as a single "negative" rule P in disjunctive normal form (DNF):

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

Next, it converts this negative rule into a "positive" rule Q in conjunctive normal form (CNF):

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

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

Example 5.2.3. *The rule sequence [R_1, R_2] in Figure 5.1.a is converted into the "positive" rule Q in CNF in Figure 5.1.b.*

2. Analyze CNF Rule to Infer Index-Based Filters: In this step, the CNF rule is analyzed to infer index-based filters. There has been a lot of work in the literature (e.g., [86, 26]) that has studied several such filters for similarity functions. Falcon builds on this work. It currently uses eight similarity functions (e.g., edit distance, Jaccard, overlap, cosine, exact match, etc.), and five filters.

Example 5.2.4. *Consider again rule Q in Figure 5.1.b.* Falcon *assigns three filters to predicate $jaccard\_word(a.title, b.title) > 0.6$: length filter, prefix filter, and position filter [100].* Falcon

assigns an equivalence filter to $exact\_match(a.year, b.year) = 1$. *Given a tuple* $b \in B$, *this filter uses a hash index to find all tuples in* A *that have the same year as* $b.year$. *Finally,* Falcon *assigns a range filter to* $abs\_diff(a.price, b.price) < 10$. *Given a tuple* $b \in B$, *this filter uses a B-tree index to find all tuples in* A *whose prices fall into the range* $(b.price - 10, b.price + 10)$.

Once the solution has inferred all filters for rule Q, it then execute several MapReduce (MR) jobs to build the indexes for these filters.

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

Implementing the above three steps in MapReduce raises the challenge that the indexes may not fit into the memory. Falcon [38] proposes four solutions (apply-all, apply-greedy, apply-conjunct, and apply-predicate) that balance between the amount of available memory and amount of work done at the mappers and reducers, then develop rules for when to select which solution. In Cloud-Matcher we use the "apply-all" solution. This solution loads the entire set of indexes $\mathcal{I}$ into the memory of each mapper, which uses $\mathcal{I}$ to locate pairs (a, b) that may match. The reducers then apply rule Q to these pairs.

Example 5.2.5. *Consider three mappers into whose memory we already load indexes* $\mathcal{I}$ *(Figure 5.1.c). We first partition table* A *three ways and sending each partition to a mapper. We do the same for table* B. *Now consider Mapper 1. For each arriving tuple* $a \in A$, *it emits a key-value pair* $\langle aid, a \rangle$, *where* aid *is the ID of* a. *For each arriving tuple* $b \in B$, *Mapper 1 applies the filters by using* $\mathcal{I}$ *to find a set of IDs of tuples in* A *that may match with* b. *Let these IDs be* $aid_1, \ldots, aid_n$. *Then Mapper 1 emits key-value pairs* $\langle aid_1, b \rangle, \ldots, \langle aid_n, b \rangle$. *The other mappers proceed similarly.*

Each emitted key-value pair is sent to one of the two reducers. For example, for a particular key aid, *Reducer 1 receives all key-value pairs with that key:* $\langle aid, a \rangle, \langle aid, b_1 \rangle, \ldots, \langle aid, b_m \rangle$ *(see Figure 5.1.c). Then this reducer can apply rule* Q *to the pairs* $(a, b_1), \ldots, (a, b_m)$.

Falcon then empirically evaluates the end-to-end solution on datasets with up to 2.5M tuples and extensively optimizes the solution. While promising, we don't build on top of Falcon as it uses Hadoop which is not the state of the art solution and moreover don't fit the needs of many enterprise customers. We do take the best practices and lessons learned from this solution while building our own solution using Spark and MongoDB.

5.2.4 Related Work

Falcon was not the first solution to try to scale up blocking rule execution. Previously, two MapReduce solutions to apply rules to tuple pairs in $A \times B$ have been proposed: $MapSide$ and $ReduceSplit$ [62], but both were significantly less effective than Falcon.

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

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

As far as we can tell, these are state-of-the-art solutions that can be applied to our setting. (The works [89, 97, 71] are related, but consider specialized types of rules and develop specialized solutions for these. Hence they do not apply to our setting that uses a far more general type of rules.)

5.3 Proposed Solution

In this section, we propose a solution to scale up the execution of blocking rules using Spark and MongoDB. In this setting, we assume that the two tables A and B and the blocking rules Q are in MongoDB and we have a Spark cluster of N nodes to execute the blocking rules. We will name this operator "rule_executor" and will use this name in the rest of the chapter. In our proposed solution, the execution of blocking rules can be viewed as a workflow with three key fragments:

1. Infer and build indexes: The first fragment analyzes the rules (assuming the rule sequence has been converted into a CNF rule) and infers index-based filters. Then, it builds the required indexes I on table A and stores them in MongoDB. This is done by running a Spark job. Recall that one of the problems with the "apply-all" solution in Falcon is that the indexes may not fit into the memory. In our solution, we store the indexes into MongoDB and do not run into this problem. Based on our experience with MongoDB, if the indexes are built carefully, the latency for the index lookup query is negligible in the total runtime.

2. Index probing: In this fragment, we run another Spark job where we probe the indexes built on table A for each record in table B. This probing helps us get a list of table A tuples that are candidate match for a tuple in table B. We realize that the output of index probing can be huge and hence we don't store this in MongoDB. Instead, we simply stream the output to the next fragment. Doing an individual index lookup query for each record in table B is inefficient and so in our solution, we send a bulk query to MongoDB for index lookup. This raises certain memory challenges that we will describe later.

3. Evaluation of rules: We apply the sequence of blocking rules to all the pairs that have survived the index probing step to get the candidate set C. As mentioned the output of index probing is stream through this fragment and the output of this is written to MongoDB.

We now describe the end-to-end solution of the $rule\_executor$ operator with the help of Figure 5.2. To start with, we have two tables A and B in MongoDB. We also have a rule sequence Q stored in MongoDB. As described above, the first fragment would infer the filters F by going through the rule sequence and will build the required indexes using Spark jobs as shown in Figure 5.2.a. Let's say we build an inverted index I_1 on tokens and a size index I_2 on the table A. The indexes will also be stored as multiple collections in MongoDB.

Next, in the index probing fragment, we will first partition table B into smaller pieces (current partition size is 500) and will read that partition from MongoDB into the memory of the Spark node. For simplicity, let's say we have partitioned table B into three pieces B_1, B_2, B_3 and each Spark node is running one executor and one thread. Then, the fragment would generate a bulk

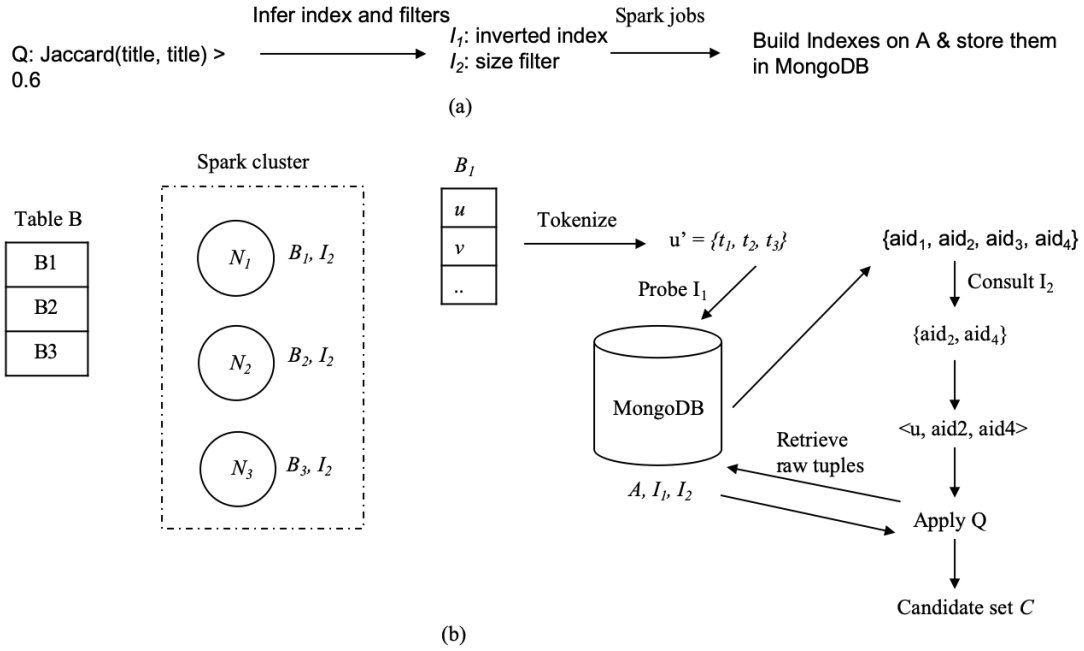


Figure 5.2: (a) Index creation (b) Execution of blocking rules (probing and applying rules)

index lookup query for all the tuples in table B and will read the portion of the index into the memory of the Spark node. Specifically, for each tuple $u, v, ..$ in the partition, it will first apply tokenization and then will send a bulk inverted index I_1 lookup query for all the tokens in the table B partition and will bring back the slice of index into the memory of the Spark node. Similarly, we will bring the slice of the size index I_2 into the memory of the Spark node.

Next, it will do the index probing in memory for I_1 followed by I_2 and will generate a set of possible candidate tuple pairs. For example, the table B record u is seen to be paired with $aid2, aid4$. Finally, for all the surviving tuple pairs, the third fragment will apply the rule and output the candidate tuple pairs to MongoDB. This is shown in Figure 5.2.b.

The above baseline solutions work well for cases where the size of input data is small. As the size of data increases, the baseline solution raises major challenges that we will discuss next.

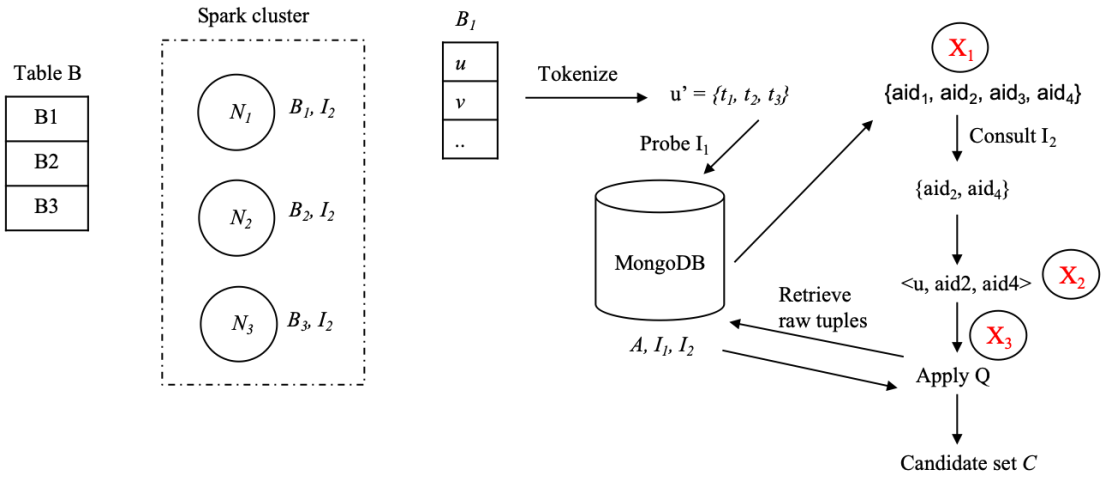


Figure 5.3: Challenges in the rule execution workflow

5.3.1 Challenges

The baseline solution raises the following two important challenges. First, we envision the $rule\_executor$ operator to not fail with "out of memory" exceptions as the input data scales. In the baseline solution, as shown in Figure 5.3 we observe two areas marked as X_1 and X_2 where memory issues are observed. First, the solution could run into memory issues during the index probing step i.e., X_1 for a predicate in the rule Q. This could happen when for certain tokens the bulk index lookup query returns a huge list of $aids$ back to the Spark job from MongoDB. Secondly, the system could throw an "out of memory" exception when we are collecting the probing results for each predicate in rule Q in a Spark dataframe. This is marked as X_2 in the Figure.

The second challenge is to control the size of intermediate results in the workflow. In practice, the solution could get a sequence of rules which have very poor pruning power. In those cases, the output of index probing could be as big as the Cartesian product in the worst case. This is marked as X_3 in the Figure 5.3. Now to all the pairs that have survived the index probing step, the evaluation of blocking rule Q could take hours to run. In addition, if not designed correctly, this could also run into memory issues and cause the Spark job to fail. Hence, we need a way to control the size of tuple pairs that go through the rule evaluation process without dropping the accuracy.

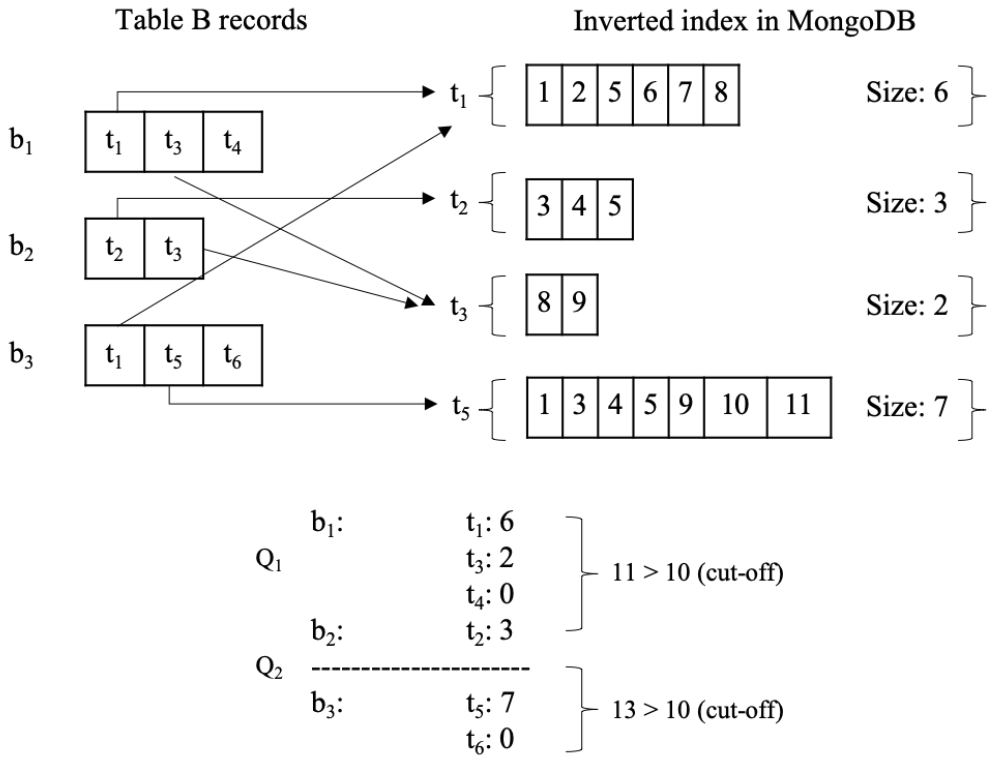


Figure 5.4: Example of how dynamic partitioning of query is done

5.3.2 Proposed Solutions

In this section, we propose solutions to address the two important challenges: (i) memory issues, and (ii) controlling the size of intermediate results.

A. Handling "out of memory" issues: In this part, we will show how we address the memory issues seen during the index probing step. Recall that we have marked these memory issues as X_1 and X_2 in Figure 5.3. Now, let us zoom into how we propose to resolve these two issues.

1. X_1 **Memory exception:** Recall, that we see X_1 when we do a bulk index lookup query for records in table B (number of records per partition is set to 500 currently) and the query result set returned from MongoDB is huge and exceeds Spark's executor memory limit. To be concrete, this issue happens when most of the table B records in the partition are being paired with a lot of table A records. Clearly, we cannot query for one table B record at a time as it would be very inefficient and time-consuming. Instead, we solve this problem by implementing a "dynamic

query partitioning". This approach gets triggered during index lookup for each partition of table B records.

We implement "dynamic query partitioning" approach as follows. First, we introduce a knob i.e. a cut-off that will control the number of records for which we will send a bulk lookup query to MongoDB. Next, for each record in the partition, we will apply tokenization and then will query MongoDB to collect the size of table A records associated with each token. We will do this until we have met the cut-off. Once we have met the cut-off, we will send a bulk index lookup query for those table B records. We will repeat this process until all the table B records are processed in a particular partition.

We now demonstrate how "dynamic query partitioning" works. To be concrete, we will use the example in Figure 5.4.

Example 5.3.1. *Let's assume that we have built an inverted index of tokens on table A and have stored it in MongoDB as a collection. For simplicity, we have an inverted index of 4 records $t_1, t_2, t3, t_5$. In addition, we also store the size of the list in the MongoDB collection as shown in Figure 5.4. For example, the token t_1 is mapped to $1, 2, 5, 6, 7, 8$ records and has a size of 6. Now, let's assume that we have three tables B records in a partition that have been tokenized. The cut-off is set to 10.*

Now, for each token, we will send a query to get the sizes from MongoDB until the cut-off is met. As shown in the figure, for the first record t_1, t_3, t_4 would return sizes 6, 2, and 0 respectively. This adds up to 8 which is less than the cut-off. So, we will move to the next record and get sizes for t_2. Now the total size is 11 which is more than the cut-off. Next, we will send a bulk query Q_1 to get the table A records. We will repeat this until all the records are processed in the partition. In this example, we will send two bulk queries Q_1 and Q_2.

To evaluate this approach, we ran multiple datasets [36] through the $rule\_executor$ and did not get any "out-of-memory" exceptions. In addition, our logs also reported the Spark executor memory consumption to be under control.

While promising, this approach can lead to an increase in job runtime if the data has skew and we are hitting the cut-off for each record in the partition. In this case, we will be sending multiple

queries to MongoDB within each partitioning which can render the approach to be ineffective. Another approach to this problem is to cache high-frequency tokens on each Spark nodes but this needs some careful work as we don't want to increase memory footprint by caching a lot of tokens either.

2. X_2 **Memory exception:** We also observe "out-of-memory" exception thrown by Spark job when we are collecting index probing results for all the predicates in the blocking rule Q. This problem arises when a lot of records from table A are being paired with a record from table B for each index lookup.

To solve this problem, we force the Spark optimizer to flush the results of one index probing to disk before moving on to probe the second one. This approach reduces the Spark executor memory footprint by flushing the results of a single index probing to disk. While this approach is promising, it does come with a cost. With this approach, you might end up writing a lot of data to the disk when running the $rule\_executor$ for datasets in the range of 20-30M tuples.

B. Controlling the size of intermediate results: In this part, we solve the problem of exploding intermediate results. Recall that if the system has learned a bad blocking rule (i.e., a rule with poor pruning power), the index probing could be ineffective and we could end up getting a candidate set of tuple pairs equal to the size of the Cartesian product of the two tables.

From Figure 5.3, this challenge has been marked as X_3. If the index probing is ineffective, we could end up evaluating a huge number of tuple pairs against the blocking rules. This can incur a lot of machine time as for each tuple pair, we would first compute the required features (complex string similarity measures) and will then check to see if it satisfies the blocking rules or not. Again, if the rules learned have a bad pruning power, we could get a large candidate set.

Let's "zoom in" to this problem. This situation happens when a tuple in table B is paired with a long list of tuples from table A. Ideally, we should be able to put a limit to pairing a tuple in table B with no more than $k = 25$ tuples from table A. Intuitively, these 25 table A tuples should be the ones who have the highest possible Jaccard score with the tuple in table B. Computing the exact

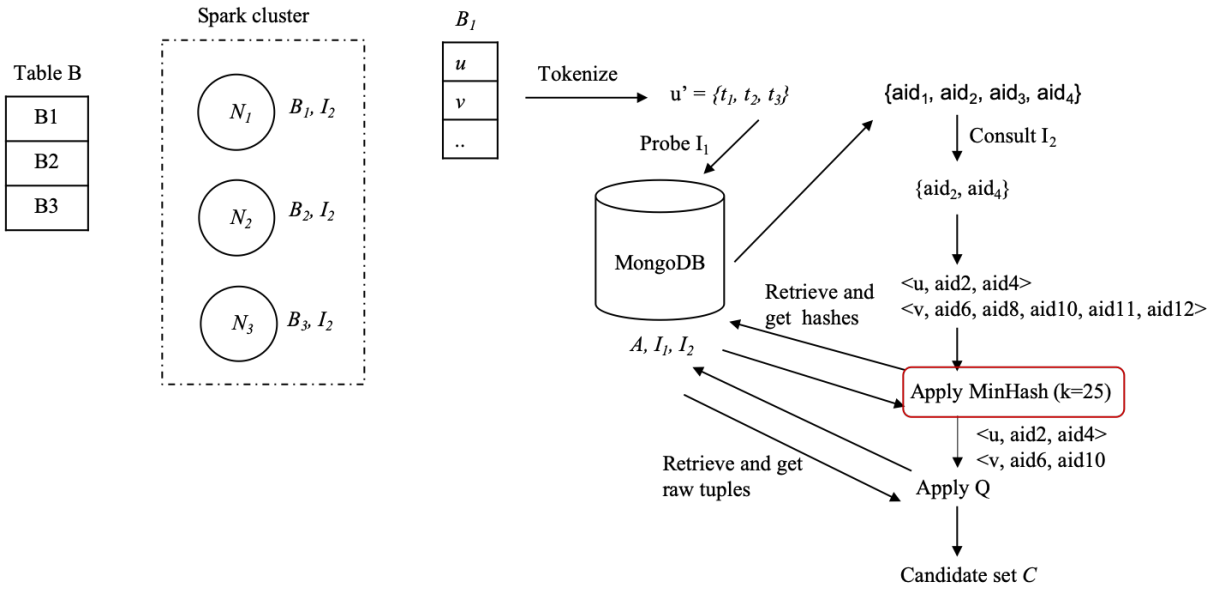


Figure 5.5: Applying MinHash technique after index probing

Jaccard score is going to very expensive. Fortunately, there is a way to approximate the Jaccard score using the MinHash technique.

Use of MinHash technique: In Figure 5.5, we show where the MinHash technique is used in the $rule\_executor$ workflow. We use the technique after the index probing step is completed. We believe that this is where we will have the maximum benefit of using MinHash and making sure that in the worst case, the size of the candidate set is no more than $|B| \cdot$ k. We implement the MinHash technique using a two-step process. First, we build hashes for all the tuples in table A and B. Specifically, use h hash functions to generate a set of h hash values for each tuple. Currently h is set to 10. A MinHash function converts tokenized data into a set of hash integers, then selects the minimum value. The function then does the same thing repeatedly with different hashing functions. We store the set of hash values for each tuple in table A and B into MongoDB.

The second step is to retrieve hash values for each tuple pair that has survived the index probing step and compute the approximate Jaccard score. This is done by doing a pairwise comparison of hash values for a particular tuple pair. We then select $k = 25$ tuples in A that have the highest

Dataset	|A|	|B|	Blocking Rules
persons	48,119	48,119	1. DICE(name, name) <= 0.36 2. COSINE(dob, dob) <= 0.37 3. EXACT_MATCH(phone, phone) <= 0.5
songs	1,000,000	1,000,000	1. COSINE(title, title) <= 0.98 & EXACT_MATCH(title, title) <= 0.5 2. DICE(artist_name, artist_name) <= 0.976 & EXACT_MATCH(title, title) <= 0.5
songs_bad_rule	1,000,000	1,000,000	1. DICE(title, title) < 0.7 & JACCARD(title, title) < 0.7
big_citations	1,823,978	2,512,927	1. COSINE(title, title) <= 0.73 & JACCARD(title, title) <= 0.69 2. DICE(authors, authors) <= 0.24

Table 5.1: Datasets for our experiments

approximate Jaccard score for tuple in B. The pairs that have survived the MinHash technique with $k = 25$ are then evaluated through the actual blocking rule.

We evaluate this strategy in detail in the evaluation section but this clearly is a trade-off between runtime, recall and the size of the candidate set. Although this technique solves the problem of bounding the size of the candidate set to no more than $|B| \cdot k$ and potentially lower job runtime, it runs the risk of achieving lower recall. This knob can be adjusted to achieve higher recall but then it could result in higher job runtime and a larger candidate set size.

5.4 Empirical Evaluation

5.4.1 Scaling Behavior of Rule Execution by Varying Cluster Size

In this part, we will examine the scaling behavior of the $rule\_executor$ operator as we increase the Spark cluster size horizontally. We now describe the experiment and the setup.

Table 5.1 describes three real-world datasets we will use for this experiment. The "Person" dataset describes identity information within a single table. "Songs" describes songs within a single table and was obtained from the freely available Million Song Dataset[1]. "Citations" match citations between DBLP and Google Scholar [67]. "Songs_bad_rule" is another run on song dataset but with a bad rule. This was done to examine the scaling behavior if the rule learned is a bad one,

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

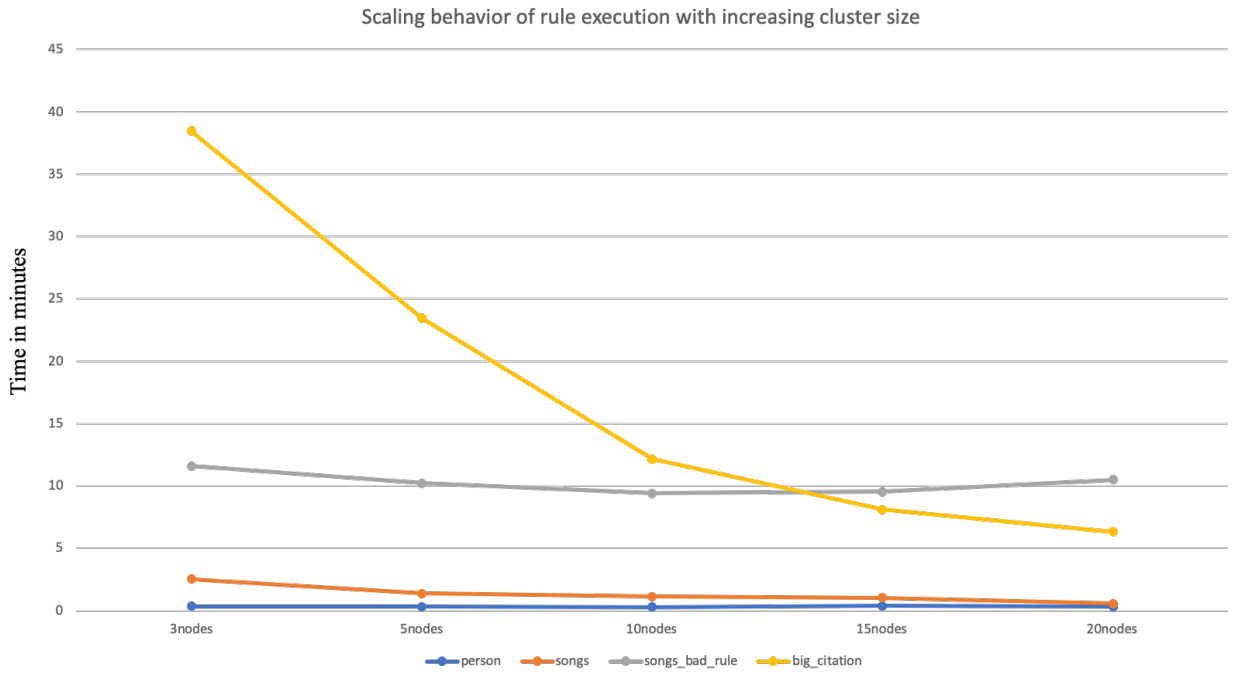


Figure 5.6: Scaling rule execution by varying cluster size

i.e., having very low pruning power. Along with the dataset, the table 5.1 also contains the blocking rules that were learned during the blocking phase of CloudMatcher.

In this experiment, we will run the code to execute blocking rules on the above three datasets on a cluster of 3, 5, 10, 15, and 20 nodes respectively and will measure runtime. For every single cluster size, every single dataset, we will run the $rule\_executor$ three times and will report the average runtime. Each node in the cluster has a 16 threads Intel Xeon E5-2666 v3 processor and 32GB of RAM.

In Figure 5.6, if we look at the runtime for "big_citation", as we vary the cluster size the runtime decreases. To be specific, the runtime for big_citation reduces from 38 minutes on a 3 node cluster to 6 minutes on a 20 node cluster.

Key observations: Based on the experiments, here are the key observations.

Dataset	Blowup factor	|A|	|B|	Blocking Rules
persons	1x	48,119	48,119	1. DICE(name, name) <= 0.36 2. COSINE(dob, dob) <= 0.37 3. EXACT_MATCH(phone, phone) <= 0.5
	3x	144,357	144,357	
	5x	240,595	240,595	
songs	1x	1,000,000	1,000,000	1. COSINE(title, title) <= 0.98 & EXACT_MATCH(title, title) <= 0.5 2. DICE(artist_name, artist_name) <= 0.976 & EXACT_MATCH(title, title) <= 0.5
	3x	3,000,000	3,000,000	
	5x	5,000,000	5,000,000	
big_citations	1x	1,823,978	2,512,927	1. JACCARD(title, title) <= 0.7
	3x	5,471,934	7,538,781	
	5x	9,119,890	12,564,635	

Table 5.2: Datasets for our experiment

1. On these three real-world datasets, as we increase the number of nodes, the runtime goes down. The benefit is seen clearly for the bigger dataset because you have more reduction in runtime.

2. We see that if the runtime is fairly low relatively on a smaller cluster, increasing the number of nodes will give you lesser benefits. At some point, increasing the number of nodes is probably not going to reduce your runtime anymore.

3. The runtime also depends on the complexity of the blocking rule. In Figure 5.6, the run for "songs" which has 1M tuples runs in about 2.5 minutes on a 3 node cluster whereas "songs_bad_rule" which is the same dataset but is running a bad rule takes about 12 minutes to execute.

5.4.2 Scaling Behavior of Rule Execution by Varying Data Size

In this part, we will examine the scaling behavior of the $rule\_executor$ as we increase the data size keeping the cluster size fixed. We now describe the experiment and the setup.

Table 5.2 describes the datasets we will use for this experiment. For this experiment, we will blow up the datasets that we discussed in Section 5.4.1 by a factor of 3x and 5x respectively. We will fix the Spark cluster size to be 10 nodes where each node in the cluster has a 16 threads Intel

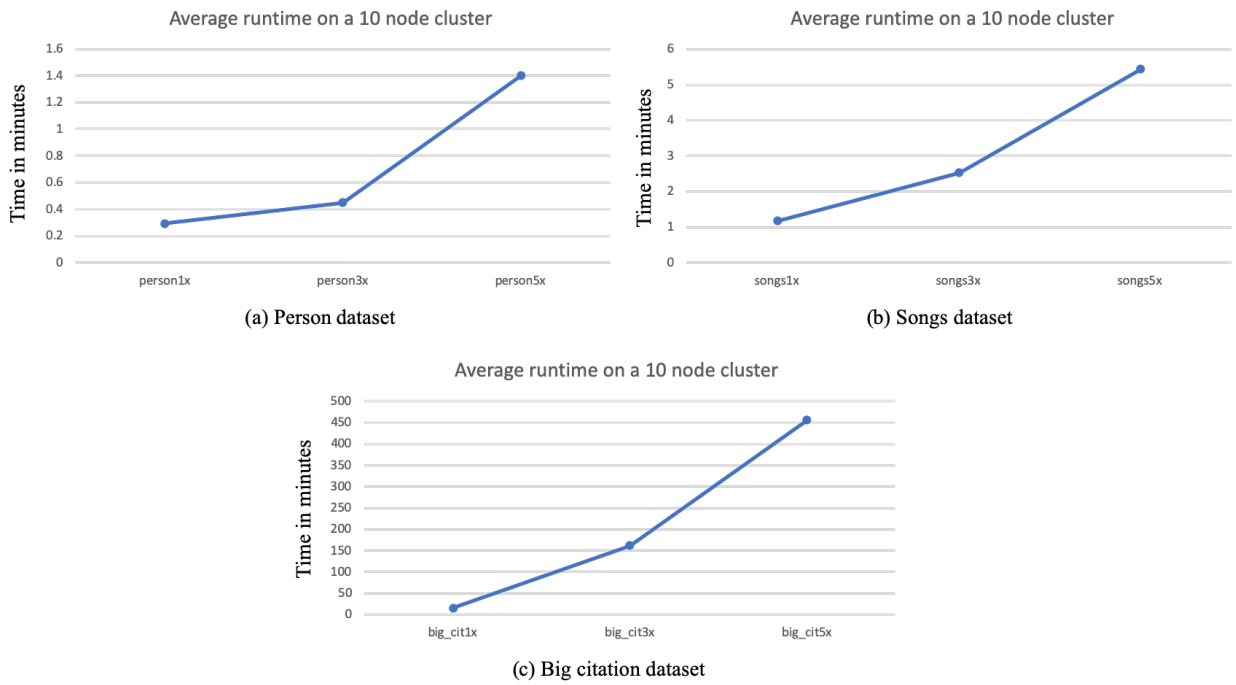


Figure 5.7: Scaling rule execution by varying data size

Xeon E5-2666 v3 processor and 32GB of RAM. Next, we will run the $rule\_executor$ code on three versions 1x, 3x, and 5x for each dataset, three times and will report the average runtime.

In Figure 5.7, if we look at the runtime for "songs", as we vary the data size, the runtime increases.

Key observations: Based on the experiments, here are the key observations.

1. As we vary the dataset size, the runtime increases and we could see that in all the three cases as shown in Figure 5.7. In the case of songs, we are able to get the $rule\_executor$ to run on 5x i.e. 5M tuples under 6 minutes and in the case of big_citations 5x i.e. (9.1M x 12.5M) in about 7.5 hours.

2. For the three datasets, as we vary the dataset size by increasing it 3x and 5x factor, we observe data skew to be the factor for the large increase in the runtime. This means that for certain datasets like big_citation if the skew is heavy, we may not be able to run 10x unless we put in more control over the trade-off between the runtime and recall.

						No MinHash			MinHash k = 25				
Dataset	|B|	Avg # aids	Min # aids	Max # aids	# of pairs after index probing	|C|	Recall	# dropped	# of times minhash hits	# of pairs after minhash	|C|	Recall	# dropped
person1x	48,119	0.4	0	5	18,962	18,962	99.95	0	0	18962	18962	99.95	0
person3x	144,357	2	0	29	315,009	315,009	99.95	0	3	315002	315002	99.95	0
person5x	240,595	4	0	53	955,220	955,220	99.95	0	24	955028	955028	99.95	0
songs1x	1,000,000	4	0	1509	3,968,002	3,926,345	99.27	41,657	26,824	2,023,181	1,982,338	97.83	40,843
songs3x	3,000,000	13	0	4529	38,701,641	38,337,195	99.27	364,446	235,784	14,180,690	13,846,719	94.93	333,971
songs5x	5,000,000	22	0	7549	109,164,775	108,158,875	99.27	1,005,900	580,724	32,854,767	32,002,523	90.09	852,244
big_cit1x	2,512,927	358	0	11741	899,310,117	4,126,102	95.46	895,184,015	305,359	58,980,019	877,558	95.35	58,102,461
big_cit3x	7,538,781	1074	0	35223	8,093,503,746	37,245,600	95.46	8,056,258,146	7,119,105	180,742,488	6,447,942	94.93	174,294,546
big_cit5x	12,564,635	1790	0	58705	22,481,414,000	103,460,000	95.46	22,377,954,000	11,964,850	303,092,050	16,667,270	94.06	286,424,780

Table 5.3: Examining MinHash strategy on the datasets

5.4.3 Examining MinHash Strategy

We now examine the MinHash strategy by running the $rule\_executor$ over the datasets described above. We run the $rule\_executor$ over the datasets twice. First, we turn off the MinHash knob and record the numbers (for e.g., size of the candidate set, recall) and then later we turn on the MinHash knob and record the numbers.

The first two columns of the Table 5.3 represents the dataset and the size of the dataset that we will use to probe the indexes. The next three columns labeled as "Avg # aids", "Min # aids", and "Max # aids" tell us about the average, minimum, and the maximum number of records from table A that are a candidate for a record in table B after the index probing is completed. The next column "# of pairs after index probing" tell us the number of candidate tuple pairs generated after the indexes are probed.

Next, we report the size of the candidate set "$|C|$", "Recall", and "# dropped" when the MinHash knob is turned off. Here "# dropped" represents the number of pairs that survived the index probing step but were killed off by the blocking rules. In rest of the columns, we report the numbers times MinHash strategy was hit when the knob was turned on, the number of pairs that survived the MinHash with k set to 25, and then the size of candidate set, recall and number of pairs that were killed of by the blocking rule post MinHash.

Now, let us zoom into the run for one dataset. For example, if we look at the "big_cit3x" run where the number of records in table B is 7.5M, we see that a single tuple from B was paired

with an average of 1074 records from table A. The maximum number of tuples paired with a single record from table B was 35,223. Then, the number of pairs that survived the index probing step is 8.09B. Now, first when we ran the $rule\_executor$ with the MinHash knob turned off, the size of the candidate set was 37.24M and the recall recorded was 95.46. The number of pairs that were killed off by the blocking rule was 8.05B. On the other hand, when we ran again with the MinHash knob turned on, the MinHash strategy was triggered for 7.1M records out of 7.5M table B records. The MinHash strategy was able to cut down the number of tuple pairs for rule evaluation to 180.7M instead of 8.09B when the MinHash knob was turned off. The size of the candidate set was recorded as 6.4M and the recall was 94.93. The number of tuple pairs killed off by the blocking rule was 174.2M. As you can notice, the MinHash strategy helped in cutting down the size of the candidate set with a minimal drop in the recall.

Key observations: Based on the experiments, here are the key observations.

1. For the dataset, as we blowup to 3x and 5x, more tuples from A are paired with a record in B and the output of index probing could be a huge set of tuple pairs. This also means that the MinHash strategy would get triggered very often. For the big_cit3x and big_cit5x the MinHash strategy gets triggered more than 90% times.

2. The MinHash strategy helps bring down the set of tuple pairs that will go through the rule evaluation process tremendously as seen for the songs3x, big_cit1x, big_cit3x, and big_cit5x runs. In addition, we don't see a huge drop in recall for most of the runs except for the songs5x where the recall drops from 99.27 to 90.09. We believe that the recall could be improved by changing the k value from 25 to a higher number but that may also increase your runtime. Clearly there is a trade-off here between the runtime, recall, and the size of the candidate set.

5.5 Conclusion

In this part of the work, we have proposed a solution to scale up blocking rule execution on two tables A and B using MongoDB and Spark. We have presented the key ideas underlying our

solution, how prior work such as Falcon have built a solution using MapReduce and some related work. Then, we have discussed the proposed solution, challenges (memory issues and exploding intermediate results), and proposed solutions to these challenges. Finally, we examine the scaling behavior of the solution by varying the cluster size and the data size and also evaluate how well the challenges are addressed. Our work demonstrates that the solution can scale to large tables without running into memory issues or large intermediate results.

Chapter 6

Conclusion and Future Work

This dissertation studies entity matching (EM), which finds data instances that refer to the same real-world entity. This problem has been a long-standing challenge in data management and will become even more important as data-driven applications proliferate. As a result, it has been studied intensively over the past several decades, by the database, AI, KDD, and WWW communities, among others. However, the vast majority of work on EM has focused on developing algorithmic solutions. Few if any current works have focused on developing end-to-end EM systems and evaluating them in real-world settings.

To address the above problems, in the past few years I have been building CloudMatcher a cloud service for EM. I envision CloudMatcher to be a fast, easy-to-use, scalable, and highly available service on the Web. Specifically, to use this service, a user simply needs to go to CloudMatcher's Web site, uploads two tables to be matched, perform some basic pre-processing, then push a button. CloudMatcher will perform EM end to end. To do so, it can either use a set of users to label tuple pairs or use crowd workers on Amazon's Mechanical Turk (or some other crowdsourcing platform such as CrowdFlower, etc.) for labeling (as matched / no-matched). The user just has to pay for the labeling if he or she decides to use Mechanical Turk. In the end, CloudMatcher will return the desired matches.

In this dissertation, I make the following contributions. First, I design scalable solution architecture for CloudMatcher such that it can do EM tasks at large scale, supports hands-off cloud service, can run multiple EM workflows simultaneously, can support multi-tenants, and is very easy-to-use for any lay user. In developing the system, I ensure that the system can recover from

crashes, is fault-tolerant, efficiently utilizing underlying cluster resources, and supports high availability at all times. The CloudMatcher codebase is now 47K LOC involving 7 contributors. It has been used at UW-Madison in domain sciences, in a data science class at UW-Madison by 70 students, and at several organizations to do EM. This work has been published at BIGDAS-KDD'17.

Second, I evaluate the CloudMatcher system on a variety of data sets across domain sciences group at UW, enterprise customers, non-profit organizations, and with 70 data-science students at UW-Madison. The results from CloudMatcher are very promising. I describe the results to do EM, challenges, and lessons learned in detail. Then, I also compare CloudMatcher with two other existing EM solutions and discuss the findings in detail. The details about this work have been published on an industrial track at SIGMOD'19.

Third, in this part, I motivate the need for developing atomic services in the EM space. Version v1.0 of CloudMatcher system only supported a single EM workflow. As we interacted with domain science users and enterprise customers, we realized that we need to enhance the system and allow the users to run different EM workflow. Toward this goal, I show that breaking the CloudMatcher workflow into a set of basic services then implementing those benefits many domain scientists and EM users. In developing this architecture and solution, the goal was to let the user mix and match a bunch of these services to compose different EM workflows. Besides, I also worked with the Columbus team to design and develop a solution where these individual services can be hosted on a system like Columbus as an application and can be used to mix and match with other apps or packages outside CloudMatcher.

Finally, I zoomed into the scaling of the blocking rule execution component of the EM workflow. This problem is time consuming and raises major scalability challenges. Previous work such as Falcon [37] has proposed solutions to scale up the execution of blocking rules but has examined table size of up to only 2-3M tuples. It has not looked into data with 10M tuples or beyond. In this work, I propose a solution that can execute rules on tables with 10M tuples each and even beyond. Another issue with Falcon is that it provides a Hadoop-based solution that is no longer the state of the art. Besides, the solution doesn't provide much control to the end-user to have a trade-off between job run-time and accuracy. In this work, we propose a solution that provides

more control on a job level to the user by providing knobs that the user could control and choose between runtime and accuracy.

There are many interesting challenges and research directions that could follow this dissertation. In what follows I discuss these directions and possible ways of improving the system.

Optimization Opportunities in CloudMatcher I believe that there are many areas in CloudMatcher where we could do some optimization and improve the system performance on different tasks in the EM space. Let me describe them briefly.

- In CloudMatcher, the data is currently ingested through CSV files. We also write the intermediate data to HDFS in CSV format. Parsing a CSV file can be inefficient in terms of time and space. This situation is worse as we scale to large data. Converting intermediate data or the output data to Parquet [17] or equivalent format would improve the read/write performance. These formats (Parquet) are space-efficient and also have a low disk footprint. Besides, we should also look into caching and optimizing file transfer to improve performance.

- The system should aim to use the native cloud infrastructure. For example, we store the data into HDFS though we run in AWS. We should aim to use S3 as it is the state of the art solution now and can be much more efficient. Similarly, in order to handle crash recoveries, fault tolerance, infrastructure, and system health we should be using AWS CloudWatch [3] instead of ad-hoc scripts.

- Auto-scaling and power save mode of operation. We should always deploy a cluster using the auto-scaling feature such that the system could automatically scale up/down based on the load on the cluster. This immensely improves the system performance on the runtime than a fixed cluster size deployment. In addition, the deployment cluster should be able to go in power save mode when not in use for a brief period of time.

- There are several operators in the EM space that could be further optimized for runtime. For example, we could improve the sampling operator by improving the performance of the

index lookup step. We have identified that creating an inverted index is not time-consuming, whereas index probing can be very expensive and could take the sampling step to take longer. This indexing could be improved by using ElasticSearch.

- Masking pair selection using the pair labeling time. The Falcon work talks about this optimization and discuss how this could cut down on the user wait time. We should incorporate this optimization as it would tremendously improve the user experience if the user wait time is minimized or completely eradicated. This situation arises mostly in the matching stage when we have a huge candidate set and we are learning a matcher using active learning. Each iteration of active learning to identify controversial examples could take more than 1-2 minutes (depending on the cluster size) and this leads to the user wait time. The idea here is to cache a higher number of pairs to be labeled in the first iteration say 40 examples. Now, let's say if the user has labeled 20 pairs, the system could show the user the next 20 examples to label but internally has retrained the classifier and has scheduled the example selection procedure to run thus masking the pair selection using the pair labeling time. This could even be done on a streaming basis if carefully implemented.

Data Profiling and Cleaning We believe that for running end-to-end EM workflow to achieve high accuracy we need the following two services as a part of pre-processing steps: a data profiler and a data cleaning service. In CloudMatcher, we already have a basic profiling step that could judge the type of attributes, missing value %, and uniqueness of each attribute. This is the bare minimum requirement as we generate features based on the attribute types. What would be interesting is if we are also able to figure out the identity of the attribute if it's an address, date, or name and then apply appropriate features. Another thing the profiler could do is identify if the characteristics of data during the "delta" load are the same as the "initial load" or not. This information could be useful in judging if we should apply the blocker and matcher from the initial load to delta load or not.

During our evaluation of CloudMatcher (see Section 3.2), we encountered cases where the accuracy numbers were not good and the main reason behind that was that the data was very dirty

(had a lot of missing values, sprinkled attributes, etc.). We believe a data-cleaning service at the beginning of the EM workflow could help to improve the accuracy numbers.

Enhancing Crowdsourcing Component In CloudMatcher, we have a baseline crowdsourcing solution where one could run the EM task in a completely hands-off manner by using crowd workers as labelers. In this case, the user just uploads the two tables, specify a crowdsourcing platform, and some crowd configuration and click a single button to start the matching process. Once the matching is completed, the predicted matches are returned back to the user.

We believe that the crowdsourcing work in CloudMatcher could be significantly improved by doing the following enhancements. First, better management of crowd workers and processing crowd answers. For example, we can identify the best way to infer answers by estimating worker accuracy and labels. Second, optimizing the entire crowdsourcing operation and masking the heavy machine time operations when crowd work is in progress. Third, assessing the expertise of crowd workers and assigning questions based on their expertise. Some of the previous work has studied this problem [98, 99]. Finally, incorporating crowd feedback into the question-answering process to improve the labeling process. There are many other things that we could do with the crowdsourcing solution(for example, what is an efficient user interface to chose for asking crowd workers to label a particular type of data, etc.).

Cost Models Providing CloudMatcher as an EM cloud service make it imperative to have a cost-model associated with it in the near future. The model could be formed by keeping the following points into consideration such as the number of active learning iterations, the cost of labeling tuple pairs per iteration, time to label cost, model retrieval/update cost, etc. This could help solve optimization problems such as given a budget constraints by the user, how to run the EM workflow and yet achieve higher accuracy? Also, given time constraints, how to select the resources to complete the EM job achieving higher accuracy?

Collaborative EM From our experience with real-world users, collaborative labeling for an EM task is a popular use case for CloudMatcher. In other words, many users do not want to take their data to any crowdsourcing platform for labeling because of data sensitivity issues. The solution

they are looking for is an "in-house" crowdsourcing platform where the workers are none other than their own employees (possible experts in the data domain). This will not only keep the data within the company security boundaries but will also leverage the expertise of an individual on the data, hence targeting higher quality labels. In CloudMatcher we provide a labeling service that can be technically used by multiple users to label the same set of training data but there is a lot that can be done here. For example, we could let the user decide on the labeling policies, strategy to gather answers, assignment strategy, deal with cases where the workers are not sure about the labels (need of supervisor), etc.The same idea could be extended to collaborative data cleaning for EM tasks.

Real-time EM Scenario With the CloudMatcher implementation, we believe that we have a solution for the two EM needs that many real-world users encounter: the EM need at the initial load, and for the delta load. However, there is a requirement for real-time matching from many enterprise customers as well. We believe that the blocker and matcher could be used to solve this problem but it raises many interesting challenges. First, how do we address the latency issue? Typically, in the case of real-time matching the user is looking for a sub-second latency. The approach that we follow for initial and delta load would not fit into the sub-second latency model. Secondly, in the case of real-time matching the user could enter a full, partial or very limited search query. Our blocker and matcher from CloudMatcher may not render good results. Third, even if the solution is able to return results to the user, how should we rank those? Finally, we need a way to calculate the accuracy of the solution.

Toward this requirement, we believe that the solution could be a two-fold process. First, a blocking step as in CloudMatcher but probably a simple TF-IDF than complex string similarity measures. We could try to leverage ElasticSearch technology but at this moment we are not sure what would be the pros and cons. The second step would be ranking the results in a way that the pair that the user is looking for appears in the top 5 results. In the second step, we could refine the ranking using the blocker and matchers we have learned during the initial and delta load. This could be an interesting research topic for EM.

Resource Management and Scheduling We believe that for a large system like CloudMatcher we need to worry about resource management and resource sharing when we are running multiple EM workflows simultaneously for multiple users. This is a challenge currently as we don't have a good understanding of the resource requirements of a particular task in the EM workflow. For example, currently, we don't know the CPU, memory, and disk requirements for a particular task. For now, we run the system with a default configuration that is to have set conservative limits on what we expect worst-case output would be and assume concurrent execution would require worst-case resource usage from all tasks. This could be improved significantly if we have a task profiler which could say run on a small sample of data and gather system requirements in an efficient manner.

Even though we have scaled up the blocking rule execution part, we still have to deal with the Spark configuration issue (i.e. the number of executors to run, number of cores per executor, the default batch size, executor memory, etc.). These issues come up more frequently in production as we scale the data. The idea is to have a task profiler or an automatic approach to assign systems resources to each task. Besides, we also need to record this information to do the scheduling of tasks in an efficient manner.

Other enhancements Finally, there are a few more enhancements that could be done in the future to further improve the CloudMatcher system.

- Allow matching to be done on data sources with different schema. Currently, in CloudMatcher, we match two data sources that have the same schema. For the sources, that don't have the same schema, we do an ad-hoc process to build attribute correspondence mapping. This should not be an ad-hoc process and should be part of the EM workflow. We should develop the user interface and let the user do the mapping of attributes between the two data sources.

- The system should allow users to upload data in different formats (such as JSON, etc.) and not just CSV. Also, we should update the system to be able to point to databases on which the users want to do deduplication or entity matching task.

- Improving user interface and adding more expressive UI is a key thing for a system like this. Adding visualization at various stages could make the system user-friendly. For example, for long-running tasks, the user should be able to see the runtime statistics and progress. This could help them plan the work accordingly.

- Verification of predicted matches: Need to have a verification component that could verify the correctness of the predicted matches. Some of the techniques proposed in recent crowd-sourced EM solutions [42, 91, 90] can be used to implement such a verification component.

Above is an incomplete list of enhancements that could be done to improve the performance of the system.

Bibliography

[1] *Amazon Mechanical Turk*. http://www.mturk.com.

[2] Anaconda https://www.anaconda.com/.

[3] AWS cloudwatch https://aws.amazon.com/cloudwatch/.

[4] BigGorilla http://www.biggorilla.org/.

[5] Celery http://www.celeryproject.org/.

[6] CrowdFlower https://www.crowdflower.com/.

[7] Dedupe https://dedupe.io/.

[8] DeepMatcher datasets https://github.com/anhaidgroup/deepmatcher/blob/master/Datasets.md.

[9] Django https://www.djangoproject.com/.

[10] Docker https://www.docker.com/.

[11] Gunicorn https://gunicorn.org/.

[12] IEEE Data Engineering Bulletin, Special Issue on Large-Scale Data Integration, 2018, http://sites.computer.org/debull/A18june/issue1.htm.

[13] In data science, the insurance industry reaches for the sky, https://link.medium.com/CZzTdh9zY3.

[14] Networkx https://networkx.github.io/.

[15] NGINX https://www.nginx.com/.

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

[17] Parquet https://pypi.org/project/parquet/.

[18] RabbitMQ https://www.rabbitmq.com/.

[19] Samasource https://www.samasource.org/.

[20] Terraform https://www.terraform.io/.

[21] The Columbus Ecosystem of Data Solutions https://columbustech.io/.

[22] Twisted https://twistedmatrix.com/.

[23] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: A survey. *PVLDB*, 24(4):557–581, 2015.

[24] C. Binnig, A. Fekete, and A. Nandi, editors. *Proc. of the Workshop on Human-In-the-Loop Data Analytics (HILDA)*, 2016.

[25] P. S. G. C. et al. Smurf: Self-Service String Matching Using Random Forests. *PVLDB*, 12(3), 2019.

[26] S. Chaudhuri et al. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[27] S. S. Chawathe et al. The TSIMMIS Project: Integration of heterogeneous information sources. In *IPSJ*, pages 7–18, 1994.

[28] P. Christen. Febrl -: An open source data cleaning, deduplication and record linkage system with a graphical user interface. In *SIGKDD*, 2008.

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

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

[31] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. End-to-end entity resolution for big data: A survey. *CoRR*, abs/1905.06397, 2019.

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

[33] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. In *SIGMOD*, 2016.

[34] M. Dallachiesa et al. Nadeef: A commodity data cleaning system. In *SIGMOD*, 2013.

[35] N. Dalvi, R. Kumar, B. Pang, and A. Tomkins. Matching reviews to objects using a language model. In *EMNLP*, 2009.

[36] S. Das, A. Doan, P. S. G. C., C. Gokhale, P. Konda, Y. Govind, and D. Paulsen. The magellan data repository. https://sites.google.com/site/anhaidgroup/useful-stuff/data.

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

[38] S. Das, P. S. G.C., A. Doan, J. Naughton, G. Krishnan, R. Deep, and V. Raghavendra. Scaling up crowdsourced RDBMS joins to large tables. 2016. UW-Madison Technical Report, available at `www.cs.wisc.edu/~sanjibkd/falcon-tr.pdf`.

[39] S. Deep, X. Hu, and P. Koutris. Join project query evaluation using matrix multiplication. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2020.

[40] S. Deep and P. Koutris. Compressed representations of conjunctive query results. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 307–322, 2018.

[41] S. Deep and P. Koutris. Ranked enumeration of conjunctive query results. *arXiv preprint arXiv:1902.02698*, 2019.

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

[43] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Morgan Kaufmann, 1st edition, 2012.

[44] X. L. Dong and T. Rekatsinas. Data integration and machine learning: A natural synergy. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1645–1650, 2018.

[45] V. Efthymiou et al. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *BIG DATA*, 2015.

[46] V. Efthymiou, G. Papadakis, G. Papastefanatos, et al. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *IEEE Big Data*. IEEE, 2015.

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

[48] H. Galhardas et al. Ajax: An extensible data cleaning tool. In *SIGMOD*, 2000.

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

[50] C. Gokhale et al. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

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

[52] Y. Govind, P. Konda, P. S. GC, P. Martinkus, P. Nagarajan, H. Li, A. Soundararajan, S. Mudgal, J. R. Ballard, H. Zhang, et al. Entity matching meets data science: A progress report from the magellan project. *integration*, 1:3, 2019.

[53] Y. Govind, E. Paulson, P. Nagarajan, A. Doan, Y. Park, G. M. Fung, D. Conathan, M. Carter, M. Sun, et al. Cloudmatcher: a hands-off cloud/crowd service for entity matching. *Proceedings of the VLDB Endowment*, 11(12):2042–2045, 2018.

[54] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. *PVLDB*, 8(12):2004–2007, 2015.

[55] D. Halperin et al. Demonstration of the myria big data management service. In *SIGMOD*, 2014.

[56] J. Heer, J. M. Hellerstein, and S. Kandel. Predictive interaction for data transformation. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[57] K. Hightower. *Kubernetes : up and running: dive into the future of infrastructure*. O'Reilly Media, Sebastopol, CA, 2017.

[58] B. Huang, S. Babu, and J. Yang. Cumulon: Optimizing statistical data analysis in the cloud. In *SIGMOD*, 2013.

[59] A. Jain, A. D. Sarma, A. G. Parameswaran, and J. Widom. Understanding workers, developing effective tasks, and enhancing marketplace dynamics: A study of a large crowdsourcing marketplace. *CoRR*, abs/1701.06207, 2017.

[60] A. Kannan, I. E. Givoni, R. Agrawal, and A. Fuxman. Matching unstructured product offers to structured product specifications. In *SIGKDD*, 2011.

[61] 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.

[62] L. Kolb et al. Learning-based entity resolution with MapReduce. In *CloudDb*, 2011.

[63] L. Kolb, A. Thor, and E. Rahm. Parallel sorted neighborhood blocking with mapreduce. BTW, 2011.

[64] L. Kolb, A. Thor, and E. Rahm. Multi-pass sorted neighborhood blocking with mapreduce. *Comput. Sci.*, 27(1):45–63, 2012.

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

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

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

[68] S. Krishnan et al. Activeclean: Interactive data cleaning for statistical modeling. *PVLDB*, 9(12):948–959, 2016.

[69] F. Kruse. *Towards a Record Linkage Layer to Support Big Data Integration*, pages 625–636. 12 2019.

[70] A. Y. Levy et al. The world wide web as a collection of views: Query processing in the information manifold. In *VIEWS*, pages 43–55, 1996.

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

[72] A. Marcus and A. Parameswaran. Crowdsourced data management: Industry and academic perspectives. *Found. Trends databases*, 6(1-2):1–161, 2015.

[73] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 19–34, 2018.

[74] F. Naumann and M. Herschel. *An Introduction to Duplicate Detection*. Morgan and Claypool Publishers, 2010.

[75] S. Newman. *Building microservices : designing fine-grained systems*. O'Reilly Media, Sebastopol, CA, 2015.

[76] G. Papadakis, G. Alexiou, G. Papastefanatos, and G. Koutrika. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. VLDB, 2015.

[77] G. Papadakis et al. The return of JedAI: End-to-End entity resolution for structured and semi-structured data. *PVLDB*, 11(12):1950–1953, 2018.

[78] G. Papadakis et al. Web-scale, Schema-Agnostic, End-to-End Entity Resolution. In *The Web Conference (WWW), Lyon, France, April*, 2018.

[79] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederee, and W. Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE TKDE*, 25(12):2665–2682, 2013.

[80] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. Meta-blocking: Taking entity resolutionto the next level. *IEEE TKDE*, 26(8):1946–1960, 2014.

[81] G. Papadakis, D. Skoutas, E. Thanos, and T. Palpanas. A survey of blocking and filtering techniques for entity resolution. *CoRR*, abs/1905.06167, 2019.

[82] E. S. Paulson. *Big Data Analytics: Methods and Applications*. PhD thesis, University of Wisconsin–Madison, 2018.

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

[84] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: holistic data repairs with probabilistic inference. *Proceedings of the VLDB Endowment*, 10(11):1190–1201, 2017.

[85] M. T. Roth et al. The Garlic Project. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, page 557, 1996.

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

[87] M. Stonebraker et al. Data curation at scale: The data tamer system. In *CIDR*, 2013.

[88] M. Stonebraker et al. Data Integration: The current status and the way forward. *IEEE Data Eng. Bull.*, 41(2):3–9, 2018.

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

[90] J. Wang et al. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.

[91] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

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

[93] M. Weis and F. Naumann. Detecting duplicate objects in xml documents. In *IQIS*, 2004.

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

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

[96] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. SIGMOD, 2009.

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

[98] Y. Yan, R. Rosales, G. Fung, and J. G. Dy. Active learning from crowds. 2011.

[99] Y. Yan, R. Rosales, G. Fung, M. Schmidt, G. Hermosillo, L. Bogoni, L. Moy, and J. Dy. Modeling annotator expertise: Learning when everybody knows a bit of something. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 932–939, 2010.

[100] M. Yu et al. String similarity search and join: A survey. *Frontiers of Computer Science*, 10(3):399–417, 2016.