CloudMatcher: A Cloud/Crowd Service for Entity Matching

Yash Govind[1], Erik Paulson[1,2], Mukilan Ashok[1], Paul Suganthan G.C.[1], Ali Hitawala[1]
AnHai Doan[1], Youngchoon Park[2], Peggy L. Peissig[3], Eric LaRose[3], Jonathan C. Badger[3]
[1]University of Wisconsin-Madison, [2]Johnson Controls, [3]Marshfield Clinic

ABSTRACT

Entity matching (EM) finds disparate data instances that refer to the same real-world entity. EM is critical in health informatics, and will become even more so in the age of Big Data and data science. Many EM systems have been developed. In this paper, we first discuss why it is still very difficult for domain scientists to use such EM systems. We then describe CloudMatcher, a cloud/crowd service for EM that we have been building. CloudMatcher aims to be a fast, easy-to-use, scalable, and highly available EM service on the Web. We motivate CloudMatcher then describe its design and implementation. Next, we describe its deployment in the past six months, providing a detailed analysis of its performance over four representative datasets. Finally, we discuss lessons learned.

Table A

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

Table B

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

Matches
(a_1, b_1)
(a_3, b_2)

(a)

Table A

	Name	Generic name	Strength
a_1	Dolorex	Acetaminophen/ Phenyltolx	500MG/ 30MG
a_2	Wart Remover	Salicylic Acid	17%
a_3	Maki-DM	Guaifenesin/ Dextromethorphan	500MG/ 300MG

Table B

	Name	Generic name	Strength
b_1	Q-Tussin DM	Guaifenesin/ Dextromethorphan	100-10M G/5
b_2	Dolorex	Acetaminophen/ Phenyltolx	500MG-3 0MG

Matches
(a_1, b_2)
(a_3, b_1)

(b)

Figure 1: Examples of EM across two tables.

1 INTRODUCTION

Entity matching (EM) finds disparate data instances that refer to the same real-world entity. For example, given the two tables in Figure 1.a, where each tuple describes a person, we want to find all tuples across the tables that refer to the same real-world person, such as tuples a_1 and b_1 in the figure. Figure 1.b shows another example of matching drugs across two tables [33]. 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. (see the related work section).

EM is critical in numerous data management applications, and will become even more so in the age of Big Data and data science. In particular, many health informatics applications must match entities such as drugs (see Figure 1.b), genes, proteins, patients in electronic health records, etc. [33].

EM is also 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. Many EM algorithms have been proposed. Building on these algorithms, many EM systems have been developed (see [30] for a discussion of 33 recent open-source and proprietary EM systems).

Today, however, it is still very difficult for domain scientists to use such EM systems. First, it is often non-trivial and time-consuming to install and learn to use such systems. Second, many such systems do not scale to large tables (e.g., those with several hundreds of thousands of tuples). Third, systems that scale often do so by using a cluster of machines (running Hadoop or Spark). However, many domain scientists do not know how to, or do not want to, install and use a machine cluster. Finally, and most seriously, to use such systems effectively, domain scientists often need to know quite a bit about EM, e.g., knowing about string similarity measures (e.g., edit distance, Jaccard, TF/IDF, etc.) and when to use which measure, about machine learning models and when to use which model, etc. (See Sections 2-3). Obtaining such knowledge is difficult even for EM experts, let alone for most domain scientists.

The CloudMatcher Service: To address these problems, in the past few years we have been building CloudMatcher, a cloud/crowd service for EM. 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 will use crowd workers on Amazon's Mechanical Turk (or some other crowdsourcing platform) to label tuple pairs (as matched / no-matched). The user just has to pay for the labeling. Alternatively, instead of using crowdsourcing, the user can just label these tuple pairs. At the end, CloudMatcher will return the desired matches. In the backend, CloudMatcher performs EM using a machine cluster that our group will maintain.

As described, when using CloudMatcher, the user does not need to install or learn how to use any complicated system (using CloudMatcher should be very straightforward). The user does not have to know EM (e.g., knowing string similarity measures). He or she will only perform simple actions such as labeling a tuple pair as 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.

Our initial motivation for building CloudMatcher is to serve the EM needs of domain scientists at the University of Wisconsin,

bigdas at KDD 2017, Hailifax, Nova Scotia, Canada
© 2016 Copyright held by the owner/author(s). 123-4567-24-567/08/06...\$15.00
DOI: 10.475/123_4

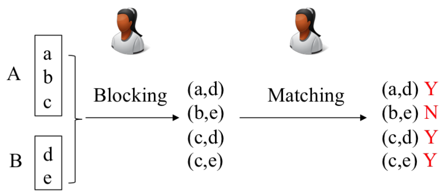

Figure 2: Most current EM solutions consist of a blocking step and a matching step.

Madison, and affiliated research institutions (e.g., Marshfield Clinic), and in fact, we have deployed such a service at UW-Madison, with highly promising results (see Section 5). In the near future, we will open up the service to the broader public, and make the service open source, so that it can also be deployed at other places, as appropriate.

Outline and Contributions: In the rest of this paper, we will describe our ongoing work developing CloudMatcher, our initial experience using it, and lessons learned. Specifically, we first describe the EM problem and typical EM solutions (Section 2). Next, we describe Corleone, which performs EM end to end, using only crowdsourcing. We then describe Falcon, which uses a cluster of machines to scale up Corleone to tables of millions of tuples (Section 3). Both the Corleone and Falcon works have been recently published [14, 21].

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 paper we will describe these challenges in detail, then describe our initial solutions (Section 4).

Using this initial solution, we have implemented a first version of CloudMatcher, and used it for a variety of real-world EM tasks. We describe our experience and lessons learned, regarding debugging and explaining, understanding data/problem/solution, and interaction with the user, among others (Section 5). As far as we can tell, this is the first work that publicly describes how to develop a cloud-based EM service and discusses initial experience and lessons learned (see also the related work section).

CloudMatcher is a part of a major project at UW-Madison on developing data cleaning and integration tools for data scientists. Another major part of this project develops Magellan, a Python package that helps the user perform entity matching end to end [4, 30].

2 PRELIMINARIES

In this section we describe the EM problem considered in this paper, the blocking and matching steps of typical EM solutions, and recent crowdsourcing solutions. Subsequent sections will build on these to discuss the Corleone, Falcon, and CloudMatcher solutions.

Entity Matching: Many EM variations exist, such as matching across two tables, matching within a single table, matching mentions in text documents into a knowledge base, etc. (see the related work section). In this paper, we will 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.a-b). 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.

Blocking and Matching Steps: Most current EM solutions perform a blocking step then a matching step. The blocking step applies a heuristic to remove tuple pairs judged obviously not matched (i.e., "blocking" these tuple pairs from further consideration). The matching step then predicts Y/N, i.e., matched/not-matched, for each remaining tuple pair. Figure 2 illustrates these two steps (here the blocking step removes two tuple pairs out of six possible pairs).

Blocking heuristics are typically specified by the user. For example, when matching the two tables of persons in Figure 1.a, a user may specify that "two persons whose states disagree do not match". Using this heuristic, the blocking step would remove the tuples (a_2, b_1) and (a_2, b_2) because their states (CA and WI) are not the same.

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.

Of course, it would not make sense to do blocking by *enumerating* all tuple pairs in $A \times B$ then removing those judged obviously non-matched, because the enumeration step alone is already very time-consuming. Instead, blocking is typically done by using the blocking heuristic to enumerate only those tuple pairs judged possibly matched. For example, given the above heuristic about disagreeing states, we can build an inverted index over Table B, such that given a state, the index will return all tuples in B with that state 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 state (e.g., WI), then enumerate only those tuple pairs.

Numerous solutions have been proposed for the blocking and matching steps, focusing on accuracy and scalability (see the related work section).

Crowdsourcing: In the past few years, crowdsourcing has been increasingly applied to EM. In crowdsourcing, certain parts of a problem are "farmed out" to a crowd of workers to solve. As such, crowdsourcing is well suited for EM, and indeed many crowdsourced EM solutions have been proposed (see Section 6).

To illustrate such crowdsourcing solutions, consider again the EM workflow in Figure 2. In this workflow, recall that after the blocking step, we apply a matcher to predict Y/N for the surviving four tuple pairs. We can now send the three pairs with the Y prediction to crowd workers to help verify these predictions. Suppose that the crowd verifies that (a, d) and (c, e) are indeed matches, but (c, d) is not. Then we would output Y predictions for only the two pairs (a, d) and (c, e), thereby improving the precision of the matching process. To increase the reliability of the answers obtained from the crowd, a typical solution is to obtain three answers from three crowd workers for each question, then take the majority answer to be the final answer from the crowd. Current works use the crowd to verify predicted matches (as illustrated above), find the best questions to ask the crowd, and find the best UI to pose

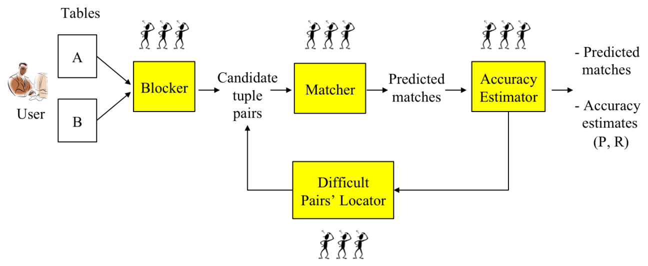

Figure 3: The EM workflow of Corleone.

such questions, among others (Section 6). Many crowdsourcing platforms can be used for the above purpose. The most popular one is Amazon's Mechanical Turk. Others include CrowdFlower, Samasource, oDesk, Elance, etc.

3 THE CORLEONE & FALCON SYSTEMS

We now describe Corleone and Falcon, our prior work, which form the basis for the CloudMatcher cloud/crowd EM service.

3.1 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).

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, we developed Corleone [21], a solution that 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 3 to find all tuple pairs ($a \in A, b \in B$) that match. This workflow consists of four main modules: Blocker, Matcher, Accuracy Estimator, and Difficult Pairs' Locator.

The Blocker generates and applies blocking rules to $A \times B$ to remove obviously non-matched pairs (Figure 4.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.

3.2 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, we developed Falcon, a solution that scales up Corleone to tables of millions of tuples. To do so, we introduced three key ideas. First, we define basic operators and use them to model the EM workflow of Corleone as a directed acyclic graph (DAG). Next, we scale up the operators, using MapReduce if necessary. Finally, we optimize within and across operators.

In what follows we discuss these ideas, but only to the extent necessary for the purpose of this paper (see [14] for a complete description of Falcon).

3.2.1 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,

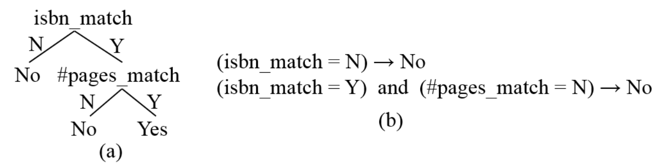

Figure 4: (a) A decision tree learned by Corleone and (b) blocking rules extracted from the tree.

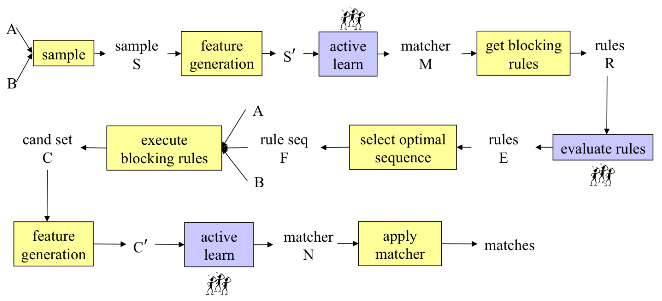

Figure 5: The EM workflow of Falcon as a DAG of basic operators.

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 4.a predicts that two book tuples match only if their ISBNs match and the number of pages match. Given a tuple pair p, matcher M applies all of its decision trees to p, then combines their predictions to obtain a final prediction for p.

Next, the module extracts all tree branches that lead from the root of a decision tree to a "No" leaf as candidate blocking rules. Figure 4.b shows two such rules extracted from the tree in Figure 4.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 [21]).

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.

3.2.2 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 5. In this DAG, given two tables A and B to be matched, we first take a sample S of tuple pairs. Next, we 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, we perform active learning with the crowd[1] over S' to learn a matcher M, then extract a set of blocking rules R from M. We then use the crowd to evaluate these rules and select a sequence of rules F judged to be optimal (see [21]). Next, we 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. We first convert each tuple pair in C into a feature vector, thereby converting C into a set of feature vectors C'. Then we 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 [14]).

It is important to note that we have developed efficient implementations for these operators (using MapReduce where necessary), and have also developed techniques to optimize within and across operators. We omit further details here for space reasons (see [14]).

4 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 our ongoing work on CloudMatcher.

4.1 Motivations and Goals

As mentioned in the introduction, we want 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 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

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

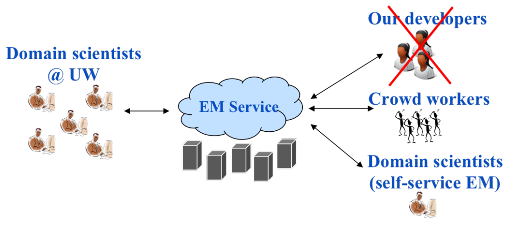

Figure 6: CloudMatcher as a cloud/crowd EM service.

scales to tables of millions of tuples. In particular, we will execute the EM workflow of Falcon described in Figure 5.

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 6 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.

4.2 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 5, 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.

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 5, 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

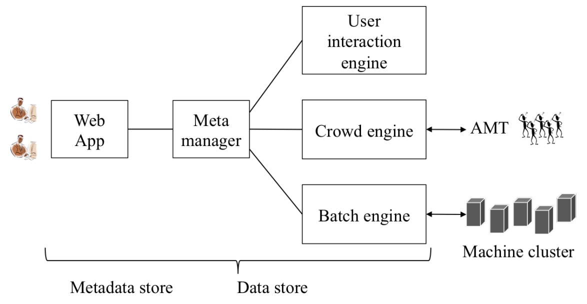

Figure 7: The CloudMatcher architecture

back from lunch, say) and can negatively affect many other DAGs that are being concurrently executed.

4.3 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 7. 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.

4.4 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 8 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 8.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 8.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 8.a ends, we continue with the workflow fragment on Figure 8.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,

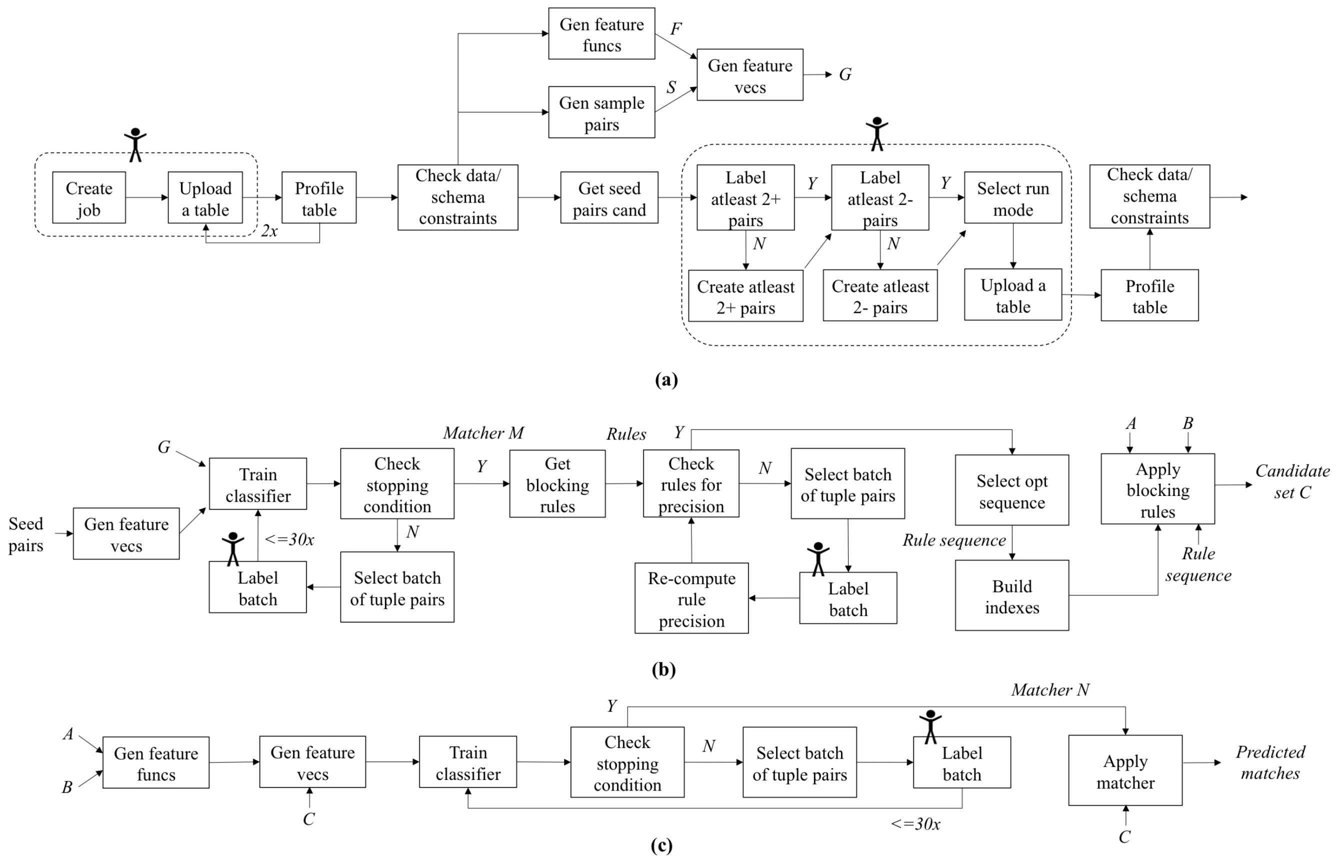

Figure 8: A refinement of the Falcon DAG, to create the workflow for CloudMatcher. The resulting workflow consists of three parts, as shown in (a)-(c).

then evaluate the rules and so on. We omit further description of the rest of the tasks in Figure 8.b, as well as the tasks in Figure 8.c (as they are similar to those in Figure 8.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.

4.5 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 9 shows how the first part of the CloudMatcher workflow (which is the part shown in Figure 8.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".

4.6 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.

4.7 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.

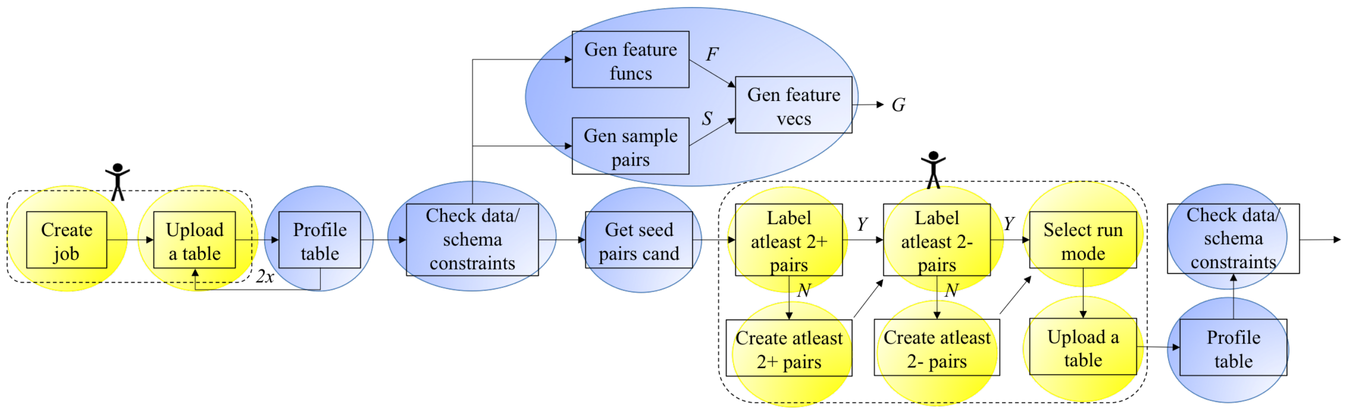

Figure 9: A partitioning of the first part of the CloudMatcher workflow into interactive and batch fragments.

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.

5 DEPLOYMENT & LESSONS LEARNED

We now describe the initial deployment of CloudMatcher, EM results for several representative datasets, and lessons learned. We have deployed CloudMatcher on a 4-node Amazon EC2 cluster, where each node has a 16-core Intel Xeon E5-2676 2.4GHz processor and 64GB of RAM. For crowdsourcing, we use Mechanical Turk and assign each question to three crowd workers, paying 2 cents per answer. In each run we used common turker qualifications to avoid spammers, such as allowing only turkers with at least 100 approved HITs and 95% approval rate.

In this paper (the first on the topic), we focus on discussing results regarding the accuracy and ease of performing EM with CloudMatcher. Subsequent papers will discuss other aspects of CloudMatcher, such as scalability, crash recovery, fault tolerance, etc.

5.1 Deployment

CloudMatcher has been developed for over 1.5 years, in a combination of Python and Java, at cloudmatcher.io:8000. It is not yet available to the general public (we still need to work out issues such as how to let a "public" user pay easily and how to securely store his/her data).

CloudMatcher however has been applied to many datasets at UW-Madison, Johnson Controls Inc. (JCI), and WalmartLabs, and has been opened to several other users, including biomedical researchers in a joint project between Marshfield Clinic and UW-Madison, and users at a non-profit organization (NPO) tracking Wisconsin politics.

The EM results have been good to very good on some datasets, and not so good on some others. A close examination of these results reveal several interesting issues. To discuss them, we now describe applying CloudMatcher to four representative datasets. The first three columns of Table 1 describes these datasets. 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 political activities of local elected officials in Wisconsin. This dataset comes from a user at a non-profit organization (which prefers not to disclose it identity). 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.) Finally, Drugs matches drug descriptions in the Marshfield-UW research team. (We do not yet have the permission to disclose EM results on matching products at WalmartLabs.) Note for these real world datasets we don't have gold matches.

The rest of Table 1 describes the results of applying CloudMatcher to these datasets. These columns are self-explanatory, except for the last one, which lists the size of the set of tuple pairs obtained after blocking. 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 [21] to estimate precision and recall (see Columns "Precision"

Dataset	Table A	Table B	Precision (in %)	Recall (in %)	Cost (# of Questions)	Machine Time	Crowd Time	Total Time	Candidate Set |C|
People	9751	706,878	93.75 – 96.32	95.50 – 97.76	$57.6 (960)	23 m	23h 14m	23h 37m	3,013
Addresses	90,673	231,081	93.22 – 95.72	76.93 – 81.01	$72.0 (1200)	38m	36h 48m	37h 26m	1,095,414
Vendors	50,295	50,292	29.95 – 38.04	91.89 – 98.10	$69.6 (1160)	58m	30h 31m	31h 29m	6,754,287
Vendors (no Brazil)	28,152	28,149	95.44 – 97.75	88.82 – 92.41	$72.0 (1200)	22m	22h 19m	22h 41m	177,337
Drugs	446,048	440,048	99.14 – 99.63	98.45 – 99.14	User labeling (1162)	8h 40m	1h 10m	9h 50m	143,479,768

Table 1: The results of applying CloudMatcher **to several representative datasets.**

and "Recall"). A value such as "93.75-96.32" in Column "Precision" means the precision is in that range with 0.95 confidence.

From the table, we can see that CloudMatcher achieves high accuracy on People, with precision in 93.75-96.32 range and recall in 95.5-97.76. This demonstrates CloudMatcher "at its finest". It shows how an "ordinary" user at a non-profit organization could simply just upload two tables, wait 24 hours, and pay under $60 (to the crowd), to obtain good matching results.

Unfortunately the results for the remaining datasets are less stellar. For Addresses, the precision is high (93-96%), but the recall is somewhat low (77-81%). 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.

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.

Thus, the Brazil data is simply incorrect. So the JCI users removed all Brazil tuples from the two tables and applied CloudMatcher again. The results now improved significantly (see the row starting with Vendors (no Brazil)), achieving precision of 95-98% and recall of 89-92%.

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".

The last dataset, Drugs, 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 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.

5.2 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 [21]. 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.

6 RELATED WORK

Entity Matching: Entity Matching (EM) has received enormous attention in the past few decades [8, 9, 16, 18, 31, 35]. Prior works have addressed various EM scenarios such as matching tuples across two tables [21], matching tuples within a single table [18], matching into a knowledge base [20], matching XML data [40], etc. CloudMatcher considers matching tuples across two tables which is a very common setting in practice.

Crowdsourcing for EM and Hands-Off EM: Crowdsourced EM has received increasing attention in academia [15, 34, 38, 39, 41] and industry (e.g., CrowdFlower [2], CrowdComputing, SamaSource [5] etc.). Most of the works employ crowd to verify the predicted matches [15, 38, 39, 41]. Recent works Corleone [21] and Falcon [14] 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 [23, 25]. However, very limited work has addressed building an EM service in the cloud. A recent effort, Dedupe [3], 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 focussed on developing EM *algorithms* (e.g., for blocking and matching steps), trying to improve their accuracy and minimizing their runtime [8, 18]. In the past few years, however, there has been effort towards building EM systems in academia [7, 12, 30] and industry [37]. Magellan [30] 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 [1], 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 [13, 20, 27, 37]. [27] matches unstructured product offers to structured product records using a probabilistic approach, and [20] 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 [29], key-based blocking [10], meta blocking [17] and so on. A recent work, Falcon [14], 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 [26, 38, 42]. For example, [38] 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 [6, 24, 36]. 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 [11, 19, 22, 24, 28, 32]. Many works address EM as a part of the data cleaning workflow [19, 22]. Based on our experiences with CloudMatcher we believe that there needs to be more effort towards data cleaning solutions specific for EM tasks.

7 CONCLUSIONS

We have described CloudMatcher, a cloud/crowd service for EM. We have motivated CloudMatcher then described its design and implementation. Finally, we have described its deployment and lessons learned. These lessons point to challenges in understanding the EM problem, crowdsourcing, and general human interaction (e.g., with in-house users). We conclude that these challenges must be addressed to develop truly successful EM services, both for health informatics and for other general domains. Much more work remains to be done on CloudMatcher, but the initial results suggest the high promise of this EM-as-a-service approach.

Acknowledgments: This project is being funded by NSF III-1564282 Medium grant, NIH BD2K U54 AI117924 grant, a UW-Madison Vilas Associate Award, UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation, and generous support from Google, WalmartLabs, and Johnson Controls Corp.

REFERENCES

[1] [n. d.]. BigGorilla http://www.biggorilla.org/. ([n. d.]). http://www.biggorilla.org/
[2] [n. d.]. CrowdFlower https://www.crowdflower.com/. ([n. d.]). https://www.crowdflower.com/
[3] [n. d.]. Dedupe https://dedupe.io/. ([n. d.]). https://dedupe.io/
[4] [n. d.]. Magellan project https://sites.google.com/site/anhaidgroup/projects/magellan. ([n. d.]). https://sites.google.com/site/anhaidgroup/projects/magellan
[5] [n. d.]. Samasource https://www.samasource.org/. ([n. d.]). https://www.samasource.org/
[6] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling Relational Data: A Survey. *PVLDB* 24, 4 (2015), 557–581.
[7] Peter Christen. 2008. Febrl -: An Open Source Data Cleaning, Deduplication and Record Linkage System with a Graphical User Interface. In *SIGKDD*.
[8] Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Publishing Company, Incorporated.
[9] Peter Christen. 2012. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE TKDE* 24, 9 (2012), 1537–1555.
[10] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. 2016. Distributed Data Deduplication. *PVLDB* 9, 11 (2016), 864–875.
[11] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *SIGMOD*.
[12] Michele Dallachiesa et al. 2013. NADEEF: A Commodity Data Cleaning System. In *SIGMOD*.
[13] Nilesh Dalvi, Ravi Kumar, Bo Pang, and Andrew Tomkins. 2009. Matching Reviews to Objects Using a Language Model. In *EMNLP*.
[14] Sanjib Das et al. 2017. Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services. In *SIGMOD*.
[15] Gianluca Demartini, Djellel Eddine Difallah, and Philippe Cudré-Mauroux. 2012. ZenCrowd: Leveraging Probabilistic Reasoning and Crowdsourcing Techniques for Large-scale Entity Linking. In *WWW*.
[16] AnHai Doan, Alon Halevy, and Zachary Ives. 2012. *Principles of Data Integration* (1st ed.). Morgan Kaufmann.
[17] Vasilis Efthymiou et al. 2015. Parallel Meta-blocking: Realizing Scalable Entity Resolution over Large, Heterogeneous Data. In *BIG DATA*.
[18] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE TKDE* 19, 1 (2007), 1–16.
[19] Helena Galhardas et al. 2000. AJAX: An Extensible Data Cleaning Tool. In *SIGMOD*.
[20] Abhishek Gattani et al. 2013. Entity Extraction, Linking, Classification, and Tagging for Social Media: A Wikipedia-based Approach. *PVLDB* 6, 11 (2013), 1126–1137.
[21] Chaitanya Gokhale et al. 2014. Corleone: Hands-Off Crowdsourcing for Entity Matching. In *SIGMOD*.
[22] Daniel Haas, Sanjay Krishnan, Jiannan Wang, Michael J. Franklin, and Eugene Wu. 2015. Wisteria: Nurturing Scalable Data Cleaning Infrastructure. *PVLDB* 8, 12 (2015), 2004–2007.

[23] Daniel Halperin et al. 2014. Demonstration of the Myria Big Data Management Service. In *SIGMOD*.

[24] Jeffrey Heer, Joseph M. Hellerstein, and Sean Kandel. 2015. 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*.

[25] Botong Huang, Shivnath Babu, and Jun Yang. 2013. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*.

[26] Ayush Jain, Akash Das Sarma, Aditya G. Parameswaran, and Jennifer Widom. 2017. Understanding Workers, Developing Effective Tasks, and Enhancing Marketplace Dynamics: A Study of a Large Crowdsourcing Marketplace. *CoRR* abs/1701.06207 (2017). http://arxiv.org/abs/1701.06207

[27] Anitha Kannan, Inmar E. Givoni, Rakesh Agrawal, and Ariel Fuxman. 2011. Matching Unstructured Product Offers to Structured Product Specifications. In *SIGKDD*.

[28] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A System for Big Data Cleansing. In *SIGMOD*.

[29] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Multi-pass Sorted Neighborhood Blocking with MapReduce. *Comput. Sci.* 27, 1 (2012), 45–63.

[30] Pradap Konda et al. 2016. Magellan: Toward Building Entity Matching Management Systems. *PVLDB* 9, 12 (2016), 1197–1208.

[31] Hanna Köpcke and Erhard Rahm. 2010. Frameworks for Entity Matching: A Comparison. *Data Knowl. Eng.* 69, 2 (2010), 197–210.

[32] Sanjay Krishnan et al. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *PVLDB* 9, 12 (2016), 948–959.

[33] Eric LaRose et al. 2017. Entity Matching Using Magellan: Mapping Drug Reference Tables. In *AIMA Joint Summit*.

[34] Adam Marcus and Aditya Parameswaran. 2015. Crowdsourced Data Management: Industry and Academic Perspectives. *Found. Trends databases* 6, 1-2 (2015), 1–161.

[35] Felix Naumann and Melanie Herschel. 2010. *An Introduction to Duplicate Detection*. Morgan and Claypool Publishers.

[36] Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB*.

[37] Michael Stonebraker et al. 2013. Data Curation at Scale: The Data Tamer System. In *CIDR*.

[38] Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. 2012. CrowdER: Crowdsourcing Entity Resolution. *PVLDB* 5, 11 (2012), 1483–1494.

[39] Jiannan Wang, Guoliang Li, Tim Kraska, Michael J. Franklin, and Jianhua Feng. 2013. Leveraging Transitive Relations for Crowdsourced Joins. In *SIGMOD*.

[40] Melanie Weis and Felix Naumann. 2004. Detecting Duplicate Objects in XML Documents. In *IQIS*.

[41] Steven Euijong Whang, Peter Lofgren, and Hector Garcia-Molina. 2013. Question Selection for Crowd Entity Resolution. *PVLDB* 6, 6 (2013), 349–360.

[42] Steven Euijong Whang, Julian McAuley, and Hector Garcia-Molina. [n. d.]. *Compare Me Maybe: Crowd Entity Resolution Interfaces*. Technical Report. Stanford University. http://ilpubs.stanford.edu:8090/1061/