Falcon: Scaling Up Hands-Off Crowdsourced Entity Matching to Build Cloud Services

Sanjib Das[1], Paul Suganthan G. C.[1], AnHai Doan[1], Jeffrey F. Naughton[1],
Ganesh Krishnan[2], Rohit Deep[2], Esteban Arcaute[2], Vijay Raghavendra[2], Youngchoon Park[3]
[1]University of Wisconsin-Madison, [2]@WalmartLabs, [3]Johnson Controls

ABSTRACT

Many works have applied crowdsourcing to entity matching (EM). While promising, these approaches are limited in that they often require a developer to be in the loop. As such, it is difficult for an organization to deploy multiple crowdsourced EM solutions, because there are simply not enough developers. To address this problem, a recent work has proposed Corleone, a solution that crowdsources the *entire* EM workflow, requiring no developers. While promising, Corleone is severely limited in that it does not scale to large tables. We propose Falcon, a solution that scales up the hands-off crowdsourced EM approach of Corleone, using RDBMS-style query execution and optimization over a Hadoop cluster. Specifically, we define a set of operators and develop efficient implementations. We translate a hands-off crowdsourced EM workflow into a plan consisting of these operators, optimize, then execute the plan. These plans involve both machine and crowd activities, giving rise to novel optimization techniques such as using crowd time to mask machine time. Extensive experiments show that Falcon can scale up to tables of millions of tuples, thus providing a practical solution for hands-off crowdsourced EM, to build cloud-based EM services.

Keywords

Entity Matching; Crowdsourcing; Hands-Off; Cloud Services

1. INTRODUCTION

Entity matching (EM) finds data items that refer to the same real-world entity, such as (David Wood, UMich) and (Dave K. Wood, UM). In the past few years crowdsourcing has become quite popular, and many crowdsourced EM solutions have been proposed (e.g., [44, 46, 3, 30, 42, 27]).

While promising, these solutions 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 (see Section 2). 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).

SIGMOD'17, May 14-19, 2017, Raleigh, NC, USA

DOI: http://dx.doi.org/10.1145/3035918.3035960

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 (see Example 1 below). To address this problem, we have recently introduced Corleone [17], 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. 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.

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

Example 1. *We are in the process of setting up a center to provide data science services to UW-Madison. Among others, we seek to provide EM services to hundreds of domain scientists. Such users often do not know how to, or are reluctant to, deploy EM systems locally (such systems often require a Hadoop cluster, as we will see). So we want to provide such EM services on the cloud, supported in the backend by a cluster of machines at our center.*

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

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

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. Our users often need to match tables of 50-200K tuples each (and some have tables of millions*

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

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

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

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

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

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

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

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

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

2. RELATED WORK

Parallel Execution of DAGs of Operators: Several pioneering works have developed platforms for the specification, optimization, and parallel execution of directed acyclic graphs (DAGs) of operators (e.g., [22, 37, 21, 16]).

While highly scalable for many applications, these platforms are not applicable to our context for two reasons. First, it is difficult to encode our workflows, which are specific to *learning-based EM*, in their DAG languages. For example, some platforms consider only key-based blockers, i.e., grouping tuples with the same key into blocks [22]. Falcon however learns a more general kind of blockers called rule-based blockers, which cannot be easily encoded using the current operators of these platforms.

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

RDBMS-Style Solutions for Data Cleaning: Several such solutions have been developed, e.g., Ajax, BigDansing, and Wisteria [15, 22, 19]. Compared to these works, Falcon is novel in four aspects. First, Falcon focuses on learning-based EM (which uses active learning to learn blockers/matchers). It provides eight "atomic" operators that we believe are appropriate for (a) modeling such EM processes, (b) facilitating efficient operator implementation, and (c) providing opportunities for inter-operator optimization. In contrast, current works either do not consider learning-based EM [22], or define operators at granularity levels that are too coarse for the above purposes [15, 19]. For example, feature vector generation, a very common step in learning-based EM, is not modeled as an atomic operation. As another (extreme) example, Ajax uses just a single operator called Match to model the entire EM process.

Second, current works consider only certain types of blocking, such as key-based ones [22]. However, such blocking types are not accurate for many real-world data sets, due to dirty/missing data (see Section 3.2). As a result, Falcon considers a far more general type of blocking called rule-based blocking and develops efficient MapReduce solutions.

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

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

Blocking: Key-based blocking (KBB) partitions tuples into blocks based on associated keys (the subsequent matching step then considers only tuples within each block). As such, KBB is highly scalable and is employed in many recent works [22, 11, 49, 8, 5]. Our experience however suggests that it is not always accurate on real-world data, in that it can "kill off" too many true matches (see Section 3.2). As a result, we elect to use rule-based blocking (RBB), as used in Corleone. RBB subsumes KBB, i.e., each KBB method can be expressed as an RBB rule. RBB proves highly accurate in our experiments (Appendix C), but is challenging to scale. As far as we can tell, Falcon provides the first MapReduce solution to scale such rules (each being a Boolean expression of predicates). Recent work has also examined scaling up sorted neighborhood blocking [24] and meta-blocking [11, 49], which combines multiple blocking methods in a scalable fashion. Such methods are complementary to our work here, and can potentially be used in future versions of Falcon.

Similarity Joins: Falcon is also related to scaling up similarity joins (SJs) [41, 48, 38, 47, 43] and theta joins [31]. To avoid examining all tuple pairs in the Cartesian product, work on SJs uses inverted indexes [39], prefix filtering [4], partition-based filtering [10], and other pruning techniques [48] (see [50] for a recent survey). Some have considered special similarity functions such as Euclidean distance [40] and edit distance [47, 43]. Most works however consider join conditions of just a single predicate [41, 48] or a conjunction of predicates [25], and develop specialized solutions for these. In contrast, Falcon develops general solutions to handle far more powerful join conditions in our blocking rules, which are Boolean expressions of predicates.

Active Learning and Optimizing: Like Falcon, [30] also proposes using active learning to reduce the number of tuple pairs to be labeled by the crowd. However, it applies learning to the Cartesian product, and thus does not scale to large tables. The idea of combining and optimizing crowd- and relational operators is also discussed in [35]. But as far as we know, Falcon is the first work to do so for crowdsourced EM. Further, some works on optimizing crowd operators have focused on minimizing cost [29, 36], minimizing crowd latency [20], or studying the trade-offs between the two [12]. These works are complementary to ours, which focuses on minimizing the machine time. As far as we know, no other work has proposed the "masking machine time" optimizations in Section 6.2.

Crowdsourced RDBMSs: Finally, works have proposed crowdsourced RDBMSs [13, 29, 34, 35] and have addressed crowdsourcing enumeration, select, max, count, and top-k queries, among others (e.g., [14, 33, 18, 26, 9, 32, 1]). Crowdsourced joins (CSJs) which at the heart solve the EM problem, have been addressed in [13, 28, 29, 12, 45]. Initial CSJ works [13, 28] however crowdsource all tuple pairs in the Cartesian product of the two tables and hence do not scale. Recent CSJ works [12, 29] ask users to write filters to reduce the number of tuple pairs to be crowdsourced. Such hand-crafted filters can be difficult to write and using them severely limits the applicability of crowdsourced RDBMSs. Falcon can automatically learn such filters (i.e., blocking rules) using crowdsourcing, and thus can potentially be used to perform CSJ over large tables.

3. PROBLEM DEFINITION

We now briefly describe Corleone [17], analyze its limitations, and define the problem considered in this paper.

3.1 The EM Workflows of Corleone

Many types of EM tasks exist, e.g., matching across two tables, within a single table, matching into a knowledge base, etc. Corleone (and Falcon) consider one such kind of tasks: matching across two tables. Specifically, given two tables A and B, Corleone applies the EM workflow in Figure 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.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 [2], 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

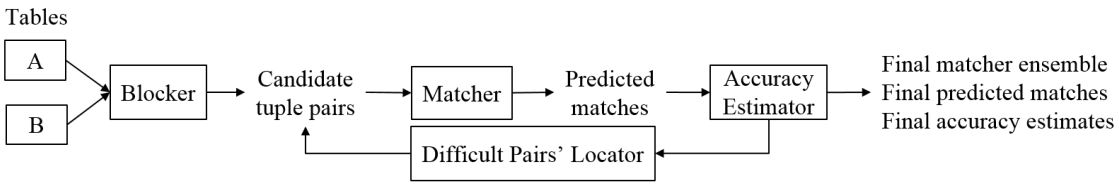

Figure 1: The EM workflow of Corleone.

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 EM Workflows Considered by Falcon

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

We now describe the Blocker and the Matcher, focusing only on the aspects necessary to understand Falcon (see [17] for a complete description).

The Blocker: The key idea underlying this module is to use crowdsourced active learning to learn a random forest based matcher (i.e., binary classifier) M [2], then extract certain paths of M as blocking rules.

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

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

At this point the module returns a final matcher M, which is a random forest classifier consisting of a set of decision trees. Each tree when applied to a tuple pair will predict if it matches, e.g., the tree in Figure 2.a predicts that two book tuples match only if their ISBNs match and the number of pages match. Given a tuple pair p, matcher M applies all of its decision trees to p, then combines their predictions to obtain a final prediction for p.

Next, the module extracts all tree branches that lead from the root of a decision tree to a "No" leaf as candidate blocking rules. Figure 2.b shows two such rules extracted from the tree in Figure 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 [17]).

Finally, the Blocker applies a subset of high-precision blocking rules to $A \times B$ to remove obviously non-matched pairs.

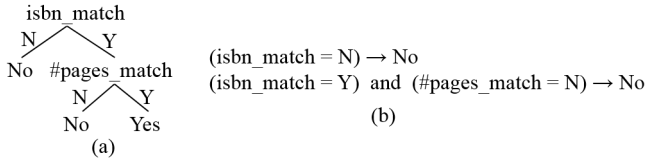

Figure 2: (a) A decision tree learned by Corleone and (b) blocking rules extracted from the tree.

The output is a set of candidate tuple pairs C to be passed to the Matcher.

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

Reasons for Not Using Key-Based Blocking: Recall that we plan to learn blocking rules such as those in Figure 2.b. As we will see in Section 5, it is a major challenge to execute such rules over two tables A and B efficiently, without enumerating the entire Cartesian product as Corleone does.

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

It turns out that KBB does not work well for many data sets, due to dirty data, variations in data values, and missing values. For example, on the Products, Songs, and Citations data sets in Section 7, our extensive effort at KBB produces recalls of 72.6%, 98.6%, and 38.8% (recall measures the fraction of true matches that survive the blocking step; ideally we want 100% recall). In contrast, rule-based blocking produces recalls of 98.09%, 99.99%, and 99.67%.

Thus, we decide to use rule-based blocking. This does not mean we execute blocking rules on the *materialized* Cartesian product, like Corleone. Instead, we analyze the rules, build indexes over the tables, then use them to quickly identify only a small fraction of tuple pairs to which the rules should be applied (see Section 5). In particular, it can be shown that when a blocking rule performs key-based blocking (e.g., the first rule in Figure 2.b, "(isbn_match = N) → No", considers only tuples that share the same ISBN), our solution in Section 5 reduces to current key-based blocking solutions on MapReduce.

3.3 Limitations of Corleone

Corleone is highly promising because it uses only crowdsourcing to achieve high EM accuracy at a relatively low cost [17]. However it suffers from a major limitation: it does not scale to large tables. The largest table pair in [17] is 2.6K tuples × 64K tuples.

Several real-world applications that we have been working with, however, must match tables ranging from 100K to sev-

eral millions of tuples. On two tables of 100K tuples each, Corleone had to be stopped after more than a week, with no result. Clearly, we must drastically scale up Corleone to make it practical.

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

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

Finally, Corleone performs crowdsourcing and machine activities sequentially. For example, in each iteration of active learning in the Blocker, the machine is idle while Corleone waits for the crowd to finish labeling a set of tuple pairs. Thus, we should consider *masking the machine time*, by scheduling as many machine activities as possible during crowdsourcing. As we will see in Section 6.2, this raises very interesting optimization opportunities, and can significantly reduce the total execution time.

3.4 Goals of Falcon

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

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

We observe that active learning in the Blocker and Matcher can take up to 100 iterations. Yet after 30 iterations or so the accuracy of the learned matcher stays the same or increases only minimally. As a result, in Falcon we stop active learning when the stopping criterion is met or when the number of iterations has reached a pre-specified threshold k (currently set to 30). This caps the crowd times t_{ab} and t_{am}.

As for t_{er}, we can show that the procedure of evaluating blocking rules described in [17] is guaranteed to execute at most 20 iterations per rule (see technical report [7] for a proof). As a result, we can estimate an upper bound on the total crowd time (regardless of the table sizes):

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

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

Imposing a threshold on the maximal number of iterations also caps the crowd cost, e.g., at \$349.6 in the current Falcon (see technical report [7]).

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

We will seek to preserve the EM accuracy of Corleone, which are shown to be already quite high in a variety of experiments [17]. Yet we will also explore optimization techniques that may reduce this accuracy slightly, if they can significantly reduce $(t_c + t_u)$.

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

Second, Section 7 shows that we have achieved this goal, reducing machine time from weeks to 52m - 2h 32m on several data sets. Since crowd time on Mechanical Turk was 11h 25m - 13h 33m, it may appear that the next goal should be to minimize crowd time because it makes up a large portion of total time. This however is not quite correct. As we discuss in Section 7.1, crowd time can vary widely depending on the platform. In fact, we describe an application on drug matching that uses in-house crowds, where crowd time was only 1h 37m, but machine time was 2h 10m, constituting a large portion (57%) of the total run time. For such applications further optimizing machine time is important.

Finally, once we have made major progress on reducing machine time, we fully intend to focus on crowd time, potentially using the techniques in [20].

4. THE FALCON SOLUTION

4.1 Adopting an RDBMS Approach

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

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

trade-offs among accuracy, cost, and time. It would be difficult to extend an "opaque" solution focusing on monolithic EM workflows to such scenarios. Finally, the Blocker and Matcher share common operations, e.g., crowdsourced active learning. An opaque solution makes it hard to factor out and optimize such commonalities.

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

We now define a set of operators and show how to compose them to form EM workflows, henceforth called *EM plans*. Sections 5-6 describe efficient implementations of operators, then plan generation, execution, and optimization.

4.2 Operators

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

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

gen_fvs: takes a set S of tuple pairs and a set F of m features, then converts each pair $(a, b) \in S$ into a feature vector $\langle f_1(a, b), \ldots, f_m(a, b)\rangle$, where each feature $f_i \in F$ is a function that maps (a, b) into a numeric score. For example, a feature may compute the edit distance between the values of the attributes *title* of a and b. This operation is important because we want to learn blocking rules (during the blocking stage) and a matcher (during the matching stage), and we need feature vectors to do the learning. (Section 6.1 discusses how Falcon generates features.)

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

get_blocking_rules: extracts a set of blocking rules from a matcher M (typically output by operator *al_matcher*). This operator assumes that M is such that we can extract rules from it. To be concrete, in this paper we will assume that M is a random forest, from which we can extract a set of blocking rules $\{R_1, \ldots, R_n\}$ such as those shown in Figure 2.b. Each rule R_i is of the form

$$p_1^i(a, b) \wedge \ldots \wedge p_{m_i}^i(a, b) \rightarrow \ drop\ (a, b), \qquad (1)$$

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

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

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

apply_blocking_rules: applies a sequence of blocking rules $\bar{R}$ to two tables A and B, producing a set of tuple pairs $C \subseteq A \times B$ to be matched in the matching stage. Applying $\bar{R}$ in a naive way to all pairs in $A \times B$ is clearly impractical. So this operator uses indexes to apply $\bar{R}$ only to certain tuple pairs, on a Hadoop cluster (see Section 5.3).

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

4.3 Composing Operators to Form Plans

The above eight operators (together with relational operators such as selection, join, and projection) can be combined in many ways to form EM plans. As a first step, in this paper we will consider the two common plan templates in Figure 3, which correspond to the EM workflows that use both the Blocker and Matcher, and just the Matcher, respectively.

The first plan template (Figure 3.a, where operators with crowd symbol use crowdsourcing) performs both blocking and matching. Specifically, we apply *sample_pairs* to Tables A and B to obtain a sample S, then convert S into a set of feature vectors S'. Next, we do crowdsourced active learning on S' to obtain a matcher M. Next we extract blocking rules R from M, then use crowdsourcing to evaluate and retain only the best rules E. Next, we select the best rule sequence F from E, then apply F on Tables A and B to obtain a set of tuple pairs C. Finally, we convert C into a set of feature vectors C', do crowdsourced active learning on C' to learn a matcher N, then apply N to match pairs in C'.

The second plan template (Figure 3.b) performs only matching. It computes the Cartesian product C of A and B, converts C into a set of feature vectors C', does crowdsourced active learning on C' to learn a matcher N, then applies N to match pairs in C'.

Falcon selects the first plan template if it deems Tables A and B sufficiently large, necessitating blocking, otherwise it selects the second plan template (see Section 6.1).

5. EFFICIENT IMPLEMENTATIONS OF OPERATORS

We have developed efficient Hadoop solutions for five operators: *sample_pairs*, *gen_fvs*, *al_matcher*, *apply_matcher* and *apply_blocking_rules* (the remaining three operators can

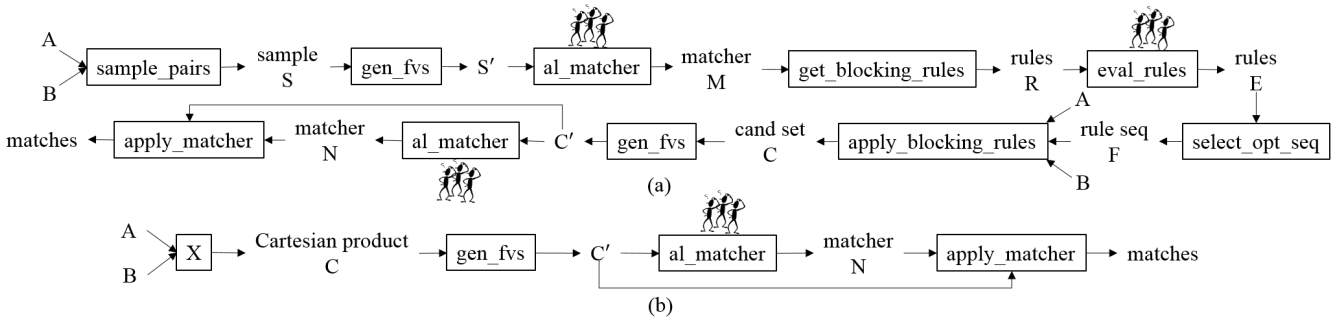

Figure 3: The two plan templates used in Falcon.

be implemented efficiently on a single machine). Among these, *apply_blocking_rules* consumes by far the most of machine time, and is also the most difficult to implement. Hence, for space reasons, in this section we will focus on this operator (Appendix A discusses the other operators and the technical report [7] provides additional details).

Recall that *apply_blocking_rules* takes two tables A and B, and a sequence of rules $\bar{R} = [R_1, \ldots, R_n]$, where each rule R_i is of the form shown in Formula 1, then outputs all tuple pairs $(a, b) \in A \times B$ that satisfy at least one rule in $\bar{R}$.

EXAMPLE 3. *Consider the sequence of two rules $[R_1, R_2]$ in Figure 4.a. Rule R_1 states that two books do not match if their titles are not sufficiently similar (using a Jaccard similarity function over the two titles tokenized as two sets of words). Rule R_2 states that two books do not match if they disagree on years and their prices differ by at least \$10 (here $exact\_match(a.year, b.year)$ returns 1 if the years match and 0 otherwise, and $abs\_diff(a.price, b.price)$ returns the absolute difference in prices).*

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

5.1 Limitations of Current Solutions

We discussed earlier that Corleone applies the rules to all tuple pairs in $A \times B$, using a single-machine implementation. Clearly this does not scale. Hence it is natural to consider using a Hadoop cluster, and in fact, two such solutions have been proposed: *MapSide* and *ReduceSplit* [23].

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

As far as we can tell, these are state-of-the-art solutions that can be applied to our setting. (The works [41, 48, 25] 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 more general type of rules.)

While more advanced than Corleone's solution, *MapSide* and *ReduceSplit* both 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).

5.2 Key Ideas Underlying Our Solution

Both *MapSide* and *ReduceSplit* assume the rules are "black-boxes", necessitating the enumeration of $A \times B$. This is not true in Falcon, where the rules use the features automatically generated by Falcon (see Section 6.1), and these features in turn often use well-known similarity functions, e.g., edit distance, Jaccard, exact match, etc. (see Example 3). Thus, we can exploit certain properties of these functions to build index-based filters, then use them to avoid enumerating $A \times B$.

EXAMPLE 4. *Suppose we want to find all tuple pairs in $A \times B$ that satisfy the predicate $jaccard\_word(a.title, b.title) > 0.6$. It is well known that for a pair of string (x, y), $jaccard(x, y) \geq t$ implies $|y|/t \geq |x| \geq |y| \cdot t$ [50]. This property can be exploited to build a length filter for the above predicate. Specifically, we build a B-tree index I_l over the lengths of attribute a.title (counted in words). Given a tuple $b \in B$ the filter uses I_l to find all tuples a in A where the length of a.title falls in the range $[|b.title| \cdot 0.6, |b.title|/0.6]$, then returns only these (a, b) pairs. We can then evaluate $jaccard\_word(a.title, b.title) > 0.6$ only on these pairs.*

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

5.3 The End-to-End Solution

We now build on the above ideas to describe the end-to-end solution for *apply_blocking_rules*.

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

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

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

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

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

EXAMPLE 5. *The rule sequence $[R_1, R_2]$ in Figure 4.a is converted into the "positive" rule Q in CNF in Figure 4.b.*

2. Analyze CNF Rule to Infer Index-Based Filters: Next, we analyze the CNF rule to infer index-based filters.

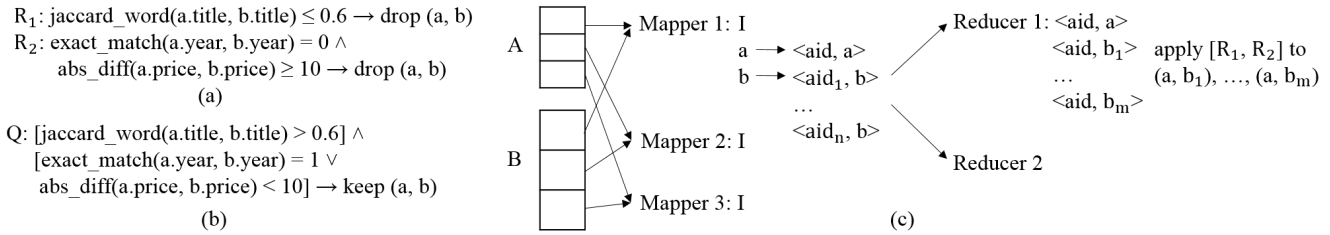

Figure 4: (a) A rule sequence, (b) the same rule sequence converted into a single "positive" rule, and (c) an illustration of how *apply_all* works.

Work on string matching has studied several such filters for similarity functions (e.g., [39, 4]). Falcon builds on this work. It currently uses eight similarity functions (e.g., edit distance, Jaccard, overlap, cosine, exact match, etc.), and five filters. For example, the equivalence filter requires that "a.x" and "b.y" are equivalent for the predicate $f(a.x, b.y)$ *op* v to be satisfied on pair (a, b). It is implemented using a hash index on "a.x", and is used for predicates that use *exact_match* similarity function. (Appendix A.1 describes the other four filters.)

EXAMPLE 6. *Consider again rule Q in Figure 4.b.* Falcon *assigns three filters to predicate $jaccard\_word(a.title, b.title) > 0.6$: length filter, prefix filter, and position filter [50].* Falcon *assigns an equivalence filter to $exact\_match(a.year, b.year) = 1$. Given a tuple $b \in B$, this filter uses a hash index to find all tuples in A that have the same year as $b.year$. Finally,* Falcon *assigns a range filter to $abs\_diff(a.price, b.price) < 10$. Given a tuple $b \in B$, this filter uses a B-tree index to find all tuples in A whose prices fall into the range $(b.price - 10, b.price + 10)$.*

Once we have inferred all filters for rule Q, we execute several MR jobs to build the indexes for these filters (see [7]).

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

A challenge however is that $\mathcal{I}$ (which can be as large as 3G in our experiments) may not fit into the memory of each mapper. So we propose four solutions that balance between the amount of memory available for the indexes at the mappers and the amount of work done at the reducers. Section 6.1 discusses how to select among these four solutions.

(a) apply-all: This solution loads the entire set of indexes $\mathcal{I}$ into the memory of each mapper, which uses $\mathcal{I}$ to locate pairs (a, b) that may match. The reducers then apply rule Q to these pairs (see the pseudo code in Algorithm 1).

EXAMPLE 7. *Consider three mappers into whose memory we already load indexes $\mathcal{I}$ (Figure 4.c). We first partition table A three ways and send 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 $\langle aid_1, b\rangle, \ldots, \langle aid_n, b\rangle$ (we discuss below optimizations to avoid emitting multiple copies of a tuple). The other mappers proceed similarly.*

Each emitted key-value pair is sent to one of the two reducers. For example, for a particular key aid, Reducer 1 receives all key-value pairs with that key: $\langle aid, a\rangle, \langle aid, b_1\rangle, \ldots, \langle aid, b_m\rangle$ (see Figure 4.c). Then this reducer can apply rule Q to the pairs $(a, b_1), \ldots, (a, b_m)$.

(b) apply-greedy: loads only the indexes of the most selective conjunct of rule Q into the mappers' memory. The mappers apply only the filters of this conjunct. The reducers then apply Q to all surviving pairs. The selectivity of each conjunct in Q can be computed from the selectivity of the corresponding rule in $\bar{R}$, which in turn is computed when we evaluate the rule on sample S (see [7]).

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

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

Optimizations: We have extensively optimized the above solutions. First, in the default mode some mappers process only tuples from A and some process only tuples from B. This incurs highly unbalanced loads. We have optimized so that each mapper processes both A's and B's tuples in a way that evens out the loads. Second, we have minimized the intermediate output size, e.g., by passing only the IDs of the tuples from B to the reducers, instead of passing the whole tuples, whenever possible (this is in addition to compressing the intermediate output.) Finally, we extensively optimized processing rule sequences. For example, we cache and reuse computations such as $jaccard\_word(a.title, b.title)$ (as the same rule or two different rules may refer to this), and we simplify predicate expressions such as $p < 0.5$ AND $p < 0.2$ into $p < 0.2$ (see [7] for more details).

6. PLAN GENERATION, EXECUTION, AND OPTIMIZATION

6.1 Plan Generation and Execution

Given two tables A and B (to be matched), we generate a plan p as follows. First, we analyze A and B to automatically generate a set of features F (see Appendix B). Later,

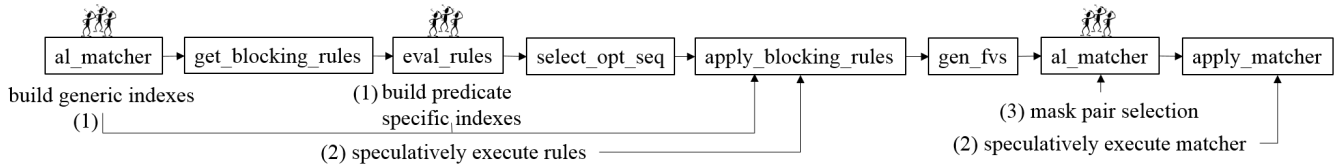

Figure 5: Three types of optimization solutions that use crowd time to mask machine time.

operators such as gen_fvs (Section 4.2) will need these features (e.g., to convert tuple pairs into feature vectors).

Next, we estimate the size of $A \times B$, where each pair is encoded as a feature vector (using the features in F). If this size does not fit in the memory of machine nodes, then blocking is likely to be necessary, so we generate the plan in Figure 3.a. Otherwise we generate the plan in Figure 3.b. (Since we are currently using a rule-based optimization approach, this is just a heuristic rule encoding the intuition that in such cases the plan in Figure 3.b can do everything solely in memory, and hence will be faster. In the future we will consider a cost-based approach that selects the plan with the estimated lower run time.)

Next, we replace each logical operator in p except operator *apply_blocking_rules* with a physical operator. Currently each such logical operator has just a single physical operator, so these replacements are straightforward. We cannot yet replace *apply_blocking_rules* because this logical operator has six physical operators: four provided by us (e.g., *apply_all*, *apply_greedy*, etc.) and two from prior work: *MapSide* and *ReduceSplit* (Section 5). Selecting the appropriate physical operator requires knowing the index sizes and the rule sequence $\bar{R}$, which are unknown at this point.

So in the next step we execute all operators in p from the start up to (and including) the operator right before *apply_blocking_rules*. This produces the rule sequence $\bar{R}$. Next, we convert it into a single positive rule Q, then infer filters and build indexes for Q, as described in Section 5.

Once index building is done, we select a physical operator for *apply_blocking_rules*, i.e., select among the six methods *apply_all*, *apply_greedy*, etc. as follows.

First, let c be the most selective conjunct in rule Q. Let $sel(c)$ and $sel(Q)$ be the selectivities of c and Q, respectively (see [7] on computing such selectivities). Clearly $sel(c) \geq sel(Q)$. If $sel(Q)/sel(c)$ exceeds a threshold (currently set to 0.8), then intuitively c is almost as selective as the entire rule Q. In this case, we will select *apply_greedy*.

Otherwise we proceed in this order (a) if the indexes for all conjuncts fit in memory (of a mapper) then select *apply_all*; (b) if the indexes of at least one conjunct fit in memory then select *apply_conjunct*; (c) if the indexes of each predicate fit in memory then select *apply_predicate*; (d) if the smaller table fits in memory then select *MapSide*, else select *ReduceSplit*.

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

6.2 Plan Optimization

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

To explain, observe that plan p currently executes machine and crowd activities *sequentially*, with no overlap. For example, *eval_rules* uses the crowd to evaluate blocking rules. Only after this has been done would *select_opt_seq* and *apply_blocking_rules* start, which execute machine activities on a Hadoop cluster. Thus this cluster is idle during *eval_rules*. This clearly raises an opportunity: while *eval_rules* is performing crowdsourcing, if we can do some useful machine activities on the idle cluster, we may be able to reduce the total run time. To mask machine time, we have developed three solutions, marked with (1), (2), and (3) respectively in Figure 5.

- Solution (1) uses the crowd time in *al_matcher* and *eval_rules* to build indexes for *apply_blocking_rules*.
- Solution (2) speculatively execute rules and matchers for *apply_blocking_rules* and *apply_matcher*.
- The above solutions are inter-operator optimizations. Solution (3) in contrast is an intra-operator optimization for *al_matcher*. It interleaves "selecting pairs for labeling" with "crowdsourcing to label the pairs". As such, it learns an approximate matcher but drastically cuts down on pair selection time.

We now describe these solutions.

1. Building Indexes for apply_blocking_rules: Recall that *apply_blocking_rules* must build indexes for filters. There are two earlier operators in the plan pipeline, *al_matcher* and *eval_rules*, where crowdsourcing is done and the Hadoop cluster is idle. So we will move as much index building activities to these two operators as possible.

In particular, while *al_matcher* crowdsources, we still do not know the rules that *apply_blocking_rules* will ultimately apply. So we use the Hadoop cluster to build only generic indexes that do not depend on knowing these rules, e.g., hash and range indexes for numeric and categorical attributes, and global token orderings for string attributes. This ordering will be required if later we decide to build indexes for prefix and position filters [50].

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

2. Speculative Execution of Future Operations: Recall that *eval_rules* uses crowdsourcing to evaluate 20 rules and retain only the best ones. Then *select_opt_seq* examines these rules to output an optimal rule sequence $\bar{R}$, which *apply_blocking_rules* will execute.

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

Specifically, we execute the 20 rules individually, in the order that *eval_rules* crowdsources (i.e., executing the most promising rules first). When *eval_rules* finishes, *select_opt_seq* takes over and outputs an optimal rule sequence $\bar{R}$, say $[R_2, R_1, R_3]$. At this point we start *apply_blocking_rules* as usual, but modify it to use the speculative execution results as follows. Suppose the output of one or more rules in $\bar{R}$ has been generated. Then we pick the smallest output then apply the remaining rules to it in a map-only job. For example, suppose that the outputs $O(R_1), O(R_3)$ of rules R_1, R_3 have been generated, and that $O(R_3)$ is the smallest output. Then we apply the sequence $[R_2, R_1]$ to $O(R_3)$.

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

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

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

Finally, if none of the outputs of rules in $\bar{R}$ has been generated, and none of these rules is currently being executed, then we simply start *apply_blocking_rules* as usual. See the technical report [7] for details. That report also describes how we speculatively execute matchers (in the matching phase), in addition to speculatively executing blocking rules, as described above.

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

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

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

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

Thus the above solution masks the pair selection time using the pair labeling time. It approximates the original physical implementation of *al_matcher* since it may not learn the same matcher (because it selects 40 pairs in the first iteration, instead of 20). Our experiments however show that this loss is negligible, e.g., both matcher versions achieve 99.61% F_1 accuracy on the Songs data set, yet the optimized version drastically reduces pair selection time, from 58m 32s to 2m 5s (see Section 7).

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

7. EMPIRICAL EVALUATION

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

We used Mechanical Turk and ran Falcon on each data set three times, paying 2 cents per answer. In each run we used common turker qualifications to avoid spammers, such as allowing only turkers with at least 100 approved HITs and 95% approval rate. We ran Hadoop on a 10-node cluster, where each node has an 8-core Intel Xeon E5-2450 2.1GHz processor and 8GB of RAM.

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

7.1 Overall Performance

We begin by examining the overall performance of Falcon. The first few columns of Table 2 show that Falcon achieves high accuracy, 81.9% F_1 on Products and 95.2-97.6% F_1 on Songs and Citations. Products is a difficult data set used in Corleone, and the accuracy 81.9% here is comparable to the accuracy of Corleone (86% F_1 after the first iteration, see [17]). Note that each row of Table 2 is averaged over three runs. Appendix C reports the results of all runs.

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

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

Table 1: Data sets for our experiment.

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

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

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

Drug Matching: Recently we have successfully deployed Falcon to match drug descriptions across two tables for a major medical research center. The tables have 453K and 451K tuples. For privacy reasons we could not use Mechanical Turk. So an in-house scientist labeled the data, effectively forming a crowd of 1 person.

The scientist labeled 830 tuple pairs, incurring a crowd time of 1h 37m. Machine time was 2h 10m, constituting a significant portion (57%) of the total run time. Our optimizations reduced this machine time by 49%, to 1h 6m, resulting in a total Falcon time of 2h 42m. The end result is 4.3M matches, with 99.18% precision and 95.29% recall on a set-aside sample.

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

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

7.2 Performance of the Components

We now "zoom in" to examine the major components of Falcon. Recall that we run Falcon three times on each data set. Table 3 shows the time of the first run on each data set, broken down by operator.

This table shows that the five "machine" operators not discussed in detail in this paper, *sample_pairs*, *gen_fvs*, *get_block_rules*, *sel_opt_seq*, and *apply_matcher*, finish in seconds or minutes, suggesting that they have been successfully optimized. In what follows we will zoom in on the remaining three operators: the two "crowd" operators, *al_matcher* and *eval_rules*, and the time-consuming "machine" operator *apply_block_rules*.

Operators al_matcher & eval_rules: In Table 3, the first "al_matcher" column shows that the time we learn a matcher via active learning in the blocking step is quite significant, 2h 23m - 8h 14m, due mainly to crowdsourcing. Similarly, column "eval_rules" shows a high rule evaluation time of 46m - 1h 48m, also due to crowdsourcing. This raises an opportunity for masking machine time, which we successfully exploit. For example, column "apply_block_rules" shows the unoptimized time of *apply_blocking_rules* in parentheses: 1m 53s - 1h 13m 20s, which in certain cases is significant. Masking optimization however successfully reduced these times to just 0-7m (the numbers outside parentheses).

The second "al_matcher" column of Table 3 shows that the time we learn a matcher in the matching step is also quite significant, due partly to crowdsourcing and partly to pair selection (see Section 6.2). Pair selection however was successfully optimized. For example, for Songs the unoptimized "al_matcher" time is 6h 40m 34s (the number in parentheses). Pair selection optimization reduced this to 5h 12m 9s (almost all of which is crowdsourcing time).

Operator apply_blocking_rules: The numbers in parentheses in column "apply_block_rules" of Table 3 show that this operator takes 1m 53s - 1h 13m 20s on three data sets, suggesting that our Hadoop-based solution was able to scale up to large tables. Masking optimization successfully reduced this time further, to just 0 - 7m, as shown in the same column (outside the parentheses).

For this operator, recall that we provided four solutions, *apply_all* (AA), *apply_greedy* (AG), *apply_conjunct* (AC), and *apply_predicate* (AP), as well as rules on when to select which solution. In addition, we supplied two Hadoop-based solutions from prior work: *MapSide* and *ReduceSplit* [23]. We now examine the performance of these six solutions. Recall that we ran Falcon three times on each data set, resulting in nine runs. In all runs except two Falcon correctly selected the best solution (i.e., the one with lowest run time). For example, on a run of Songs, the times for AA, AG, AC, and AP are 10m 19s, 1h 3m, 1h 40m, and 1h 45m, respectively, and Falcon correctly picked AA to run. (*MapSide* and *ReduceSplit* did not complete on this data set.)

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

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

Overall, the results here suggest that (a) the solutions can vary drastically in their run times, (b) Falcon often selected the best solution, which is AA, AG, or AP depending on the

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

Table 2: Overall performance of Falcon on the data sets. Each row is averaged over three runs.

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

Table 3: Falcon's run times per operator on the data sets. Each row refers to the first run of each data set.

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

Table 4: Effect of optimizations on machine time.

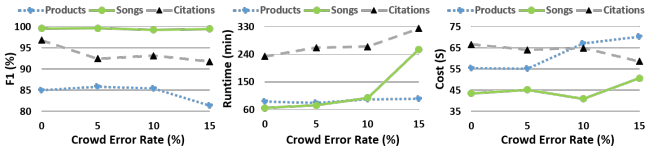

Figure 6: Effect of crowd error rate on F_1, run time, and cost.

amount of available memory, and (c) prior solutions do not scale as they enumerate $A \times B$.

7.3 Effectiveness of Optimization

Recall that our goal is to minimize the machine time beyond the crowdsourcing time (i.e., the machine time that cannot be masked). Column "U" of Table 4 shows this unoptimized time, 18m - 2h 12m, for the first run of each data set (recall that we run Falcon three times with real crowd for each data set; the results are similar for other runs, and we only show the first runs for space reasons). Column "O" shows the optimized time 16m - 40m, a significant reduction, ranging from 11% to 70% (see Column "Reduction"). This result suggests that the current optimization techniques of Falcon are highly effective.

The next three columns show the run time when we turned off each type of optimization: index building (O_1), speculative execution (O_2), and masking pair selection (O_3). The result shows that all three optimization types are useful, and that the effects of some are quite significant (e.g., O_2 on Citations and O_3 on Songs).

7.4 Sensitivity Analysis

We now examine the main factors affecting Falcon's performance (see Appendix C for additional experiments).

Error Rate of the Crowd: First we examine how varying crowd error rates affect Falcon. To do this, we use the random worker model in Corleone to simulate a crowd of random workers with a fixed error rate (i.e., the probability of incorrectly labeling a pair) [17]. Figure 6 shows F_1, run time, and cost vs. the error rate (the results are averaged over three runs). We can see that as error rate increases from 0 to 15%, F_1 decreases and run time increases, but either minimally or gracefully. Interestingly there is no clear

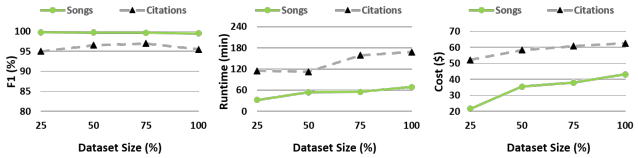

Figure 7: Performance of Falcon across varying sizes of Songs and Citations data.

trend on cost. This is because in some cases (e.g., when the error rate is high), active learning converged early, thereby saving crowdsourcing costs. In any case, recall that there is a cap of $349.6 on crowdsourcing cost (Section 3.4) and the costs in Figure 6 remain well below that cap.

Size of the Tables: So far we have shown that Falcon achieved good performance on tables of size 1-2.5M tuples. We now examine how this performance changes as we vary the table size. Figure 7 shows F_1, run time, cost as we run Falcon on 25%, 50%, 75%, and 100% of Songs and Citations (using simulated crowd with 5% error rate and 1.5m latency per a 10-question HIT; each data point is averaged over 3 runs). The results show that as table size increases, (a) F_1 remains stable or fluctuates in a small range. (b) run time increases sublinearly, and (c) cost increases sublinearly (recall that cost will not exceed the cap of $349.6).

8. CONCLUSIONS

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

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

Falcon also provides a framework for many interesting future research directions. These include minimizing crowd latency / monetary cost, examining more optimization techniques (including cost-based optimization), extending Falcon with more operators (e.g., the Accuracy Estimator [17]), and applying Falcon to other problem settings, e.g., crowdsourced joins in crowdsourced RDBMSs.

Acknowledgment: This work is supported by gifts from WalmartLabs, Google, Johnson Control; by NIH BD2K grant U54 AI117924 and NSF grant 1564282; and by generous support from UW Vilas Foundation and UW 2020 Initiative.

9. REFERENCES

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

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

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

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

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

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

[7] S. Das et al. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services http://pages.cs.wisc.edu/~anhai/papers/falcon-tr.pdf. Technical Report.

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

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

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

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

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

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

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

[15] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, 2001.

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

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

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

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

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

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

[22] Z. Khayyat et al. BigDansing: A system for big data cleansing. In *SIGMOD*, 2015.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[38] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. Tung. Efficient and scalable processing of string similarity join. *TKDE*, 25(10):2217–2230, 2013.

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

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

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

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

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

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

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

[46] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. In *VLDB*, 2013.

[47] C. Xiao, W. Wang, and X. Lin. Ed-Join: An efficient algorithm for similarity joins with edit distance constraints. In *VLDB*, 2008.

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

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

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

APPENDIX

A. IMPLEMENTING THE OPERATORS

Recall that Section 4.2 defines eight operators. We have described *apply_blocking_rules* in Section 5. In what follows we provide more details on this operator, describe operator *sample_pairs*, then the rest of the operators.

A.1 More Details on apply_blocking_rules

Index-Based Filters: Operator *apply_blocking_rules* (Section 5.2) uses five types of filters. Equivalence filter is described in Section 5.2. The remaining four types of filters are:

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

2. *Length Filter:* requires that a constraint on the lengths of "a.x" and "b.y" be satisfied for the predicate to be satisfied. It is implemented using a length index on

Algorithm 1 apply-all

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

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

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

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

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

$length(a.x)$ (probed using $length(b.y)$), and is used for predicates involving *Jaccard*, *overlap*, *Dice*, *cosine* and *Levenshtein*.

3. *Prefix Filter:* requires that there must be at least one shared token in the *prefixes* of "a.x" and "b.y" for the predicate to be satisfied. Note that the tokens of "a.x" and "b.y" are first re-ordered based on a global token ordering and then prefixes of the re-ordered tokens are considered. This filter is implemented using an inverted index over the prefixes of re-ordered tokens of "a.x", and used for predicates involving *Jaccard*, *overlap*, *Dice*, *cosine* and *Levenshtein*.
4. *Position Filter:* requires that at least a certain number of tokens be shared between the *prefixes* of "a.x" and "b.y". It is implemented using an inverted index on the prefixes of "a.x" (the same index constructed for prefix filters) and a length index (constructed for length filters). It is used for predicates involving *Jaccard*, *overlap*, *Dice*, *cosine* and *Levenshtein*.

Pseudo Code for apply_all: Algorithm 1 shows pseudo code for the *apply_all* solution described in Section 5.3.

A.2 Operator sample_pairs

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

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

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

It turns out this solution does not work for large A and B. First, if A is larger than n (as in some of our experiments), the solution is not applicable. Second, even if A is smaller than n, it may not be much smaller, in which case we sample very few tuples from B. For example, if $|A| = 500K$ and $n = 1M$, then we sample only 2 tuples from B. This can produce very few matches in S, especially if we unluckily pick two tuples from B that have no matches in A.

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

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

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

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

A.3 Implementing Other Operators

Operator gen_fvs: Given a set S of tuple pairs and a set F of m features (see Appendix B on how to create F), this

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

* Not used for blocking.

Figure 8: Rules for feature generation.

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

Figure 9: The schemas of the data sets.

operator converts each pair $(a, b) \in S$ into a feature vector $\langle f_1(a, b), \ldots, f_m(a, b)\rangle$. This step is highly parallelizable and is implemented using a Map-only job on the Hadoop cluster. In this job, each mapper reads a pair (a, b) from S (residing on HDFS); computes a feature value $f_i(a, b)$ for each feature $f_i \in F$; and outputs a key-value pair where a composition of tuple IDs: $\langle a.id, b.id\rangle$ is the key, and the feature vector: $\langle f_1(a, b), ..., f_m(a, b)\rangle$ is the value.

Operator al_matcher: This operator performs crowdsourced active learning on a set of pairs V. It trains an initial matcher M, uses M to select a small set of controversial pairs from V (currently set to 20), asks the crowd to label these pairs, uses them to improve M, and so on. This operator was described in [17] and is straightforward to implement. We made only two small changes. First, we execute pair selection on Hadoop, as it can be time consuming (Section 6.2 shows further optimizations of this step). Second, Corleone performs active learning until a convergence criterion T is met. Falcon however stops active learning when either T is met *or* the number of iterations reaches a prespecified threshold (currently set to 30). Our experiments show that this limit has negligible effects on the accuracy yet can significantly reduce the crowd time and cost.

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

Operator eval_rules: This operator uses crowdsourcing to evaluate and retain only the most precise rules. It is also described in [17] and is straightforward to implement. Similar to *al_matcher*, it also operates in iterations. Thus we also impose a threshold on the maximal number of iterations, capping the crowd time and cost. Note that we use the same crowdsourcing strategies as Corleone uses for *al_matcher* and *eval_rules* (see [17]).

Operator select_opt_seq: This operator takes the blocking rules output by *eval_rules* and produces an optimal rule sequence (to be executed later by *apply_blocking_rules*). We have developed an efficient single-machine implementation for this operator, but the development has raised difficult challenges. For space reasons, we do not describe this operator further, referring the reader to the technical report [7] for a detailed description.

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

B. GENERATING FEATURES

For blocking and matching purposes, Falcon automatically generates a set of features F based on the types (e.g., string, numeric) and characteristics (e.g., short string, long string) of the attributes of the two tables. The heuristic rules guiding the generation process are shown in Figure 8. We next describe the implementation in detail.

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

Next, we scan through the tables to determine the characteristics (e.g., single-word string, multi-word long string, etc.) of every attribute. Next, for each attribute correspondence (x, y), we include in F a set of features, each of the form $sim(a.x, b.y)$ where sim is a similarity function chosen based on the rules in Figure 8. If x and y have different attribute characteristics, we choose the characteristic that is at a lower row in Figure 8.

C. EMPIRICAL EVALUATION

We now provide additional experimental results.

Data Sets: Songs describes songs within a single table and was obtained from labrosa.ee.columbia.edu/millionsong. Products describes electronics products and was used in Corleone, and Citations describes citations in Citeseer and DBLP (see Figure 9 for the schemas). We have made all three data sets publicly available at [6].

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

Table 5: All runs of Falcon on the data sets.

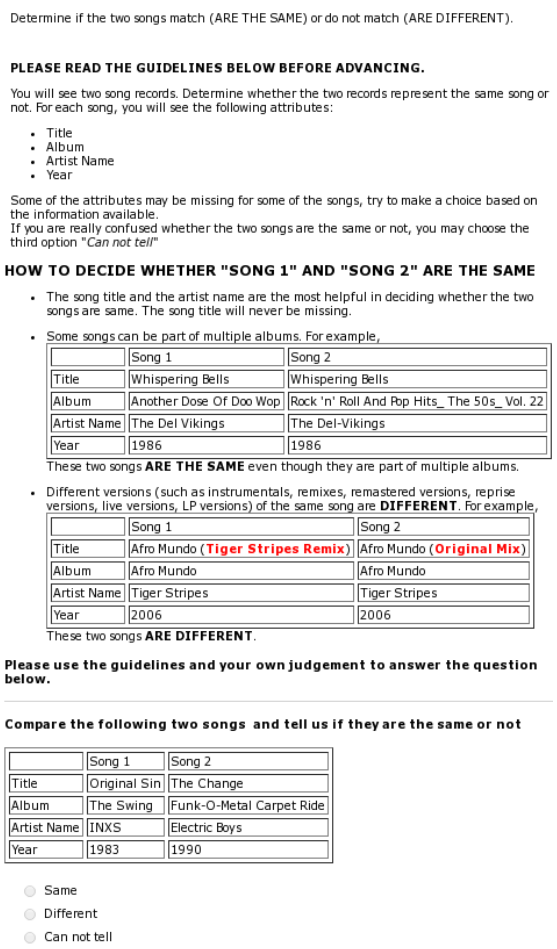

Figure 10: Screenshot of a task to the crowd.

Falcon applied the procedure described in Appendix B to generate features for these data sets. Overall, it generated 50/83 features for Products, 20/47 features for Songs, and 22/30 features for Citations. "50/83" for example means that 50 and 83 features were generated for the blocking and matching steps, respectively. Note that only features involving relatively fast string similarity measures were generated for the blocking step (see Figure 8 for the list of string similarity measures).

Crowdsourcing: Figure 10 shows a screenshot of a task presented to the crowd on Songs. The first half of the screen shows instructions to the crowd and the bottom half asks the crowd if two given song tuples match. A task contains 10 such tuple pairs. On Mechanical Turk workers prefer tasks that contain multiple pairs since it reduces the overhead. Thus, we present 10 pairs per task, following the exact same crowdsourcing procedure of Corleone.

All Runs over the Data Sets: Recall that Table 2 shows the Falcon performance over the three data sets, where each row shows the *average* of three runs (on the same data set). Table 5 shows all nine runs. The results show that while the candidate set size can vary across runs, affecting the machine and crowd time, the cost and the F_1 accuracy stay relatively stable.

Evaluating Operator sample_pairs: Recall that we run Falcon three times on each data set. Table 3 shows the time of the first run on each data set, broken down by operator. Column "sample_pairs" of this table shows that sampling is very fast, taking just 1m 15s - 2m 23s. The candidate sets in the last column of Table 2 contain tuple pairs surviving blocking. These sets are just 0.01-0.95% of the size of $A \times B$, and retain 98.09-99.99% of matching pairs. These results suggest that our sampling solution is fast and effective, in that it helps Falcon learn very good blocking rules.

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

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

Regarding memory size, its largest effect would be on *apply_blocking_rules*, and we have discussed this earlier in Section 7.2. Finally, we have experimented with varying the maximal number of iterations for active learning. As this number goes from 30 to 100, we found that (a) all active learning in our experiments terminated before 100, (b) the run time (including crowdsourcing time) increased significantly, (c) yet F_1 accuracy fluctuates in a very small range. This suggests that capping the number of iterations at some value, say 30 as we have done, is a reasonable solution to avoid high run time and cost yet achieve good accuracy.