Crowd-Based Deduplication: An Adaptive Approach

Sibo Wang Xiaokui Xiao Chun-Hee Lee
School of Computer Engineering
Nanyang Technological University
Singapore
{wang0759, xkxiao, lee.ch}@ntu.edu.sg

ABSTRACT

Data deduplication stands as a building block for data integration and data cleaning. The state-of-the-art techniques focus on how to exploit crowdsourcing to improve the accuracy of deduplication. However, they either incur significant overheads on the crowd or offer inferior accuracy.

This paper presents *ACD*, a new crowd-based algorithm for data deduplication. The basic idea of *ACD* is to adopt *correlation clustering* (which is a classic machine-based algorithm for data deduplication) under a crowd-based setting. We propose non-trivial techniques to reduce the time required in performing correlation clustering with the crowd, and devise methods to postprocess the results of correlation clustering for better accuracy of deduplication. With extensive experiments on the Amazon Mechanical Turk, we demonstrate that *ACD* outperforms the states of the art by offering a high precision of deduplication while incurring moderate crowdsourcing overheads.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Crowdsourcing; Data Deduplication; Correlating Clustering

1. INTRODUCTION

Given a set of records from heterogenous sources, the *data deduplication* problem aims to cluster the records into several groups, such that records in the same group correspond to the same entity in the real world. This problem finds important applications in data integration [28], data cleaning [13, 40], and web searching [21], etc. The problem is easy to address when the records to be clustered share some common identifiers; but in many practical scenarios, such common identifiers either do not exist or are rather noisy, which makes it challenging to deduplicate records accurately.

Numerous techniques have been proposed for automatic data deduplication based on pre-defined rules, similarity metrics, supervised learning, etc. (see [17] for a recent survey). Such *machine-based* techniques, however, often face difficulties when processing records that represent different entities but look highly similar. For example, consider three records about commercial brands: Chevrolet, Chevy, and Chevron. All of these records resemble each other, due to which they might be identified as the same entity by machine-based techniques; nevertheless, only Chevrolet and Chevy represent the same brand. To remedy the deficiency of machine-based techniques, recent works [46–48] propose to utilize crowdsourcing to improve the accuracy of data deduplication. The rationale is that humans are often better than machines in tackling complex tasks. In particular, given the three brand records above, humans can easily determine that Chevron is a different brand from Chevrolet and Chevy.

Given a set R of records, a naive approach to deduplicate R via crowdsourcing is to generate all possible pairs of records from R, and then ask the crowd to examine each pair to see if it contains duplicate records. Nevertheless, this approach incurs a prohibitive cost due to the enormous number of record pairs that need to be processed by the crowd. To address this problem, Wang et al. [46] propose a hybrid approach that utilizes both the crowd and machines. Specifically, Wang et al.'s approach runs in three steps. First, it evaluates the similarity between each pair of records in R using a machine-based algorithm. After that, it eliminates the pairs whose similarity scores are below a pre-defined threshold τ. Finally, it invokes the crowd to process the remaining pairs of records, and then derives the final deduplication results accordingly.

The above hybrid approach is shown to be much more accurate than conventional machine-based methods [46]. However, it still entails a large crowdsourcing overhead when there exist a large number of record pairs whose similarity scores are above the threshold τ. To improve, Wang et al. [47] present a more advanced approach (referred to as *TransM*) that reduces crowdsourcing costs by exploring the transitive relationship among records. In particular, for any three records x, y, z, if the crowd has decided that $x = y$ and $y = z$, then *TransM* determines that $x = z$, without issuing the pair (x, z) to the crowd. On the other hand, if the crowd suggests that $x = y$ and $y \neq z$, then *TransM* infers that $x \neq z$. As such, *TransM* considerably reduces the number of record pairs that requires human efforts.

The reduced crowdsourcing overhead of *TransM*, however, comes with a cost of deduplication accuracy, as demonstrated in the experiments in [47]. The reason is that *TransM* implicitly assumes that the crowd always returns correct results, which is not always the case in practice. When human errors occur in the processing of crowdsourced record pairs, *TransM*'s transitivity mechanism tends to amplify errors, leading to less accurate deduplication results. For example, consider the scenario in Figure 1, where we have two groups of records $\{a_i\}$ and $\{b_j\}$, each of which corre-

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2723739 .

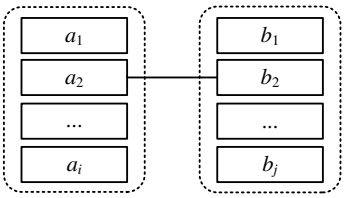


Figure 1: Illustration of the deficiency of *TransM*.

sponds to one entity. Assume that *TransM* correctly clusters all a_i into one group and all b_j into another, and then issues a pair (a_2, b_2) to the crowd. If the crowd mistakenly marks $a_2 = b_2$, then by transitivity, *TransM* would put all a_i and b_j into one group, resulting in significant errors.

Contributions. To address the deficiency of existing solutions, this paper presents *ACD (Adaptive Crowd-based Deduplication)*, a new deduplication technique that utilizes both human and machine power. *ACD* consists of three phases: a pruning phase, a cluster generation phase, and a cluster refinement phase. The pruning phase applies a machine-based algorithm to eliminate record pairs with low similarities. After that, the cluster generation phase collects the remaining record pairs, and produces an initial clustering of the records using a small number of *Human Intelligent Tasks (HITs)*. The initial clustering is then passed over to the cluster refinement phase, which refines the clustering using additional HITs and then returns the final deduplication results. Compared with *TransM*, *ACD* is much less sensitive to human errors, as it does not overly rely on the transitivity relationship among records. Instead, *ACD* adopts a more robust approach that is able to reconcile inconsistent results from the crowd, by borrowing idea from *correlation clustering* [9]. Furthermore, *ACD* incorporates a parallelization method that enables it to issue HITs in a batch manner in both the cluster generation and refinement phases, which significantly improves the efficiency of crowdsourcing. We experimentally evaluate *ACD* with real-world datasets, and we demonstrate the superiority of *ACD* against existing state-of-the-art crowd-based deduplication solutions.

In summary, the main contributions of this paper are as follows:

- We present *ACD*, a crowd-based data deduplication algorithm that overcomes the deficiencies of *TransM* and strikes a better balance between deduplication accuracy and crowdsourcing overheads.
- We devise a novel crowd-based approach for refining the clusters generated by *ACD*, yielding further improved deduplication results.
- We propose mechanisms to process HITs in *ACD* in a batch manner, which significantly improves the efficiency of crowdsourcing.
- We experimentally compare *ACD* with the states of the art on the Amazon Mechanical Turk (AMT). Our results show that *ACD* outperforms competitors by offering a high precision of deduplication while incurring moderate crowdsourcing overheads.

2. PRELIMINARIES

In this section, we formally define the data deduplication problem, and revisit the existing crowd-based deduplication approaches. Table 1 lists the notations that we will frequently use in the paper.

Notation	Description
$R = \{r_1, r_2, \cdots, r_n\}$	a set of records
S	a candidate set of record pairs (see Section 3)
$G = (V_R, E_S)$	an undirected graph where each vertex represents a record in R and each edge represents a record pair in S
$f(r_i, r_j)$	similarity score between r_i and r_j
$f_c(r_i, r_j)$	crowd's confidence that $r_i = r_j$
$\Lambda'(R)$	the optimization goal of *ACD* (see Equation 2)
$b(o)$	benefit of operation o
$b^*(o)$	estimated benefit of operation o
$c(o)$	crowdsourcing cost of performing operation o

Table 1: Table of notations.

2.1 Problem Definition

Let $R = \{r_1, r_2, \ldots, r_n\}$ be a set of records and g be a function that maps each r_i to the real world entity that r_i represents. Given R, the data deduplication problem aims to divide the records in R into a set of disjoint clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$, such that for any two records $r_i, r_j \in R$, if $g(r_i) = g(r_j)$ then r_i and r_j appear in the same cluster, and vice versa.

In practice, however, the function g is either unknown or difficult to obtain in many scenarios. In that case, it is often assumed that there exists a similarity score function $f : R \times R \rightarrow [0, 1]$, such that $f(r_i, r_j)$ equals the likelihood that r_i and r_j represent the same entity. (Such a function can be constructed using either machine-based or crowd-based algorithms.) Under this setting, the objective of data deduplication is to construct the clustering $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ in a manner that minimizes a certain error metric based on f. In this paper, we use a cost metric $\Lambda(R)$ that has been widely adopted in previous work [8, 11, 27, 41] on machine-based deduplication. In particular,

$$\Lambda(R) = \sum_{r_i, r_j \in R, i<j} x_{i,j} \cdot (1 - f(r_i, r_j)) + \sum_{r_i, r_j \in R, i<j} (1 - x_{i,j}) \cdot f(r_i, r_j), \quad (1)$$

where $x_{i,j} = 1$ if r_i and r_j are in the same cluster and $x_{i,j} = 0$ otherwise. In other words, for any record pair (r_i, r_j), if r_i and r_j are put into the same cluster, then $\Lambda(R)$ assigns a penalty of $1 - f(r_i, r_j)$ on the pair; otherwise, the penalty assigned is $f(r_i, r_j)$. As such, the minimization of $\Lambda(R)$ tends to put similar (resp. different) records into the same (resp. different) clusters. Example 1 shows an example of the data deduplication problem with $\Lambda(R)$ as the optimization goal.

EXAMPLE 1. Suppose that we have a set of six records $R = \{a, b, c, d, e, f\}$. Table 2 shows the similarity score of each pair of records, omitting those pairs with low similarity scores. It can be verified that $\Lambda(R)$ is minimized when the records in R are divided into two clusters, $\{a, b, c\}$ and $\{d, e, f\}$. □

In general, minimizing $\Lambda(R)$ is an NP-hard problem [9]. Motivated by this, a number of approximation or heuristic algorithms [5, 9, 14, 36, 41] have been proposed for machine-based data deduplication. All these algorithms formulate similarity score function f (used in $\Lambda(R)$) with machine-based approaches, such as character-based metrics [32], token-based metrics [12], and phonetic similarity metrics [39], etc. However, these algorithms tend to struggle when handling "difficult" pairs of records that have similar

Record Pair	Similarity Score	Record Pair	Similarity Score
(a, b)	0.81	(e, f)	0.69
(b, c)	0.75	(c, d)	0.45
(a, c)	0.73	(a, d)	0.43
(d, e)	0.72	(a, e)	0.37
(d, f)	0.70	$\cdots$	≤ 0.3

Table 2: Similarity scores for the record pairs in Example 1.

representations but differ inherently. This motivates the utilization of crowdsourcing to improve the accuracy of data deduplication, as we explain in Section 2.2.

2.2 Crowd-based Deduplication

Crowdsourcing. There exist a number of crowdsourcing platforms, such as Amazon Mechanical Turk, CrowdFlower, and MobileWorks. In such platforms, we can request human "workers" to complete *micro-tasks*, e.g., we may ask them to answer questions like "do r_i and r_j refer to the same entity?". Each micro-task is referred to as a *human intelligent task (HIT)*. The completion of a HIT by a worker is rewarded with a certain amount of money based on the difficulty of the HIT, i.e., invoking the crowd for data deduplication comes with a monetary cost. In addition, a human worker may not always produce a correct answer for a HIT. A standard practice to mitigate such human errors is to issue a HIT to multiple workers, and then take a majority vote. As we will show in Section 6, however, even when majority votes are used, we may still get incorrect answers from the crowd. As a consequence, it is crucial to take human errors into account when designing a crowd-based algorithm.

Existing Crowd-based Deduplication Algorithms. There exist five crowd-based deduplication algorithms in the literature, namely, *Crowdclustering* [25], *CrowdER* [46], *GCER* [48], *TransM* [47], and a very recent approach by Vesdapunt et al. [44] (referred to as *TransNode*). As we have explained *CrowdER* and *TransM* in Section 1, we focus on the other three approaches in the following.

Given a set R of records, *Crowdclustering* first creates a number of small subsets of R, and then ask the crowd workers to perform clustering on each subset individually. After that, *Crowdclustering* inspects the clustering results produced by the crowd workers on each subset, and applies a machine learning approach to generalize the clustering results to the original dataset R. In other words, *Crowdclustering* assumes that the clustering results on a small subset of records can be representative of the results on R. This, however, does not always hold. In particular, if there do not exist too many records in R corresponding to the same entity (as is often the case in practice), then a small subset of R may not even contain two records that are duplicates. In that case, the clustering results on the subset would not give a machine learning algorithm any useful information about when two records should be regarded as duplicates; as a consequence, the deduplication accuracy of *Crowdclustering* would be poor. In general, *Crowdclustering* is more suitable for *data categorization*, which asks whether different records belong to the same category (i.e., whether *Chevrolet* and *Chevron* are both car brands) instead of whether they are duplicates. The reason is that, in data categorization, it is common that a large number of records belong to the same category, in which case even a small subset of the records could provide representative information about the whole dataset.

GCER shares the same spirit as *Crowdclustering* in that it first asks the crowd to process a subset of the data, and then generalizes the crowd's results to the entire data. In particular, it first constructs a machine-based similarity score function f. After that, it iteratively issues record pairs to the crowd, and utilizes the crowd's answers to refine f for better deduplication accuracy. As with *TransM*, however, *GCER* is also susceptible to mistakes made by the crowd, in the sense if the crowd gives an incorrect answer regarding whether two records are duplicate, then *GCER* would adjust f based on the incorrect answer, which degrades the precision of deduplication.

TransNode performs data deduplication in a way that is similar to *TransM*. In particular, they both discard dissimilar record pairs based on a machine-based similarity score function, and then exploit the crowd to process the remaining record pairs (referred to as *candidate pairs*), using transitive relations to improve efficiency and reduce crowdsourcing costs. The main difference between *TransM* and *TransNode* lies in the order which they inspect candidate pairs. In *TransM*, candidate pairs are issued in decreasing order of their machine-based similarity scores. In contrast, *TransNode* adopts a more advanced approach to order record pairs, which leads to a non-trivial worst-case guarantee on the number of record pairs that need to be crowdsourced to complete the clustering task. However, as *TransNode* also relies on transitive relations, it suffers from the same deficiency as *TransM*.

3. SOLUTION OVERVIEW

At a high level, our *ACD* algorithm for crowd-based deduplication consists of three phases:

1. *Pruning Phase:* This phase invokes a machine-based algorithm to construct the similarity score function f, and then generates a candidate set S from the set R of the input records, using a similarity threshold τ. In particular, S consists of all record pairs (r_i, r_j) such that $f(r_i, r_j) > \tau$.
2. *Cluster Generation Phase:* This phase issues a selected subset of the pairs in S to the crowd. After that, based on the crowd answers, it divides the records in R into a set of disjoint clusters $\mathcal{C} = \{C_1, C_2, \cdots, C_k\}$.
3. *Cluster Refinement Phase:* This phase issues some pairs in S that are previously not processed by the crowd, and then refines $\mathcal{C}$ based on the crowd answers.

Note that the pruning phase utilizes only machine-based algorithms, which have been well studied in the literature. Therefore, we will focus only on the cluster generation and refinement phases of *ACD*, assuming that a machine-based algorithm for the pruning phase is given.

Cluster Generation Phase. Let $f_c : R \times R \rightarrow [0, 1]$ be a crowd-based similarity score function, such that $f_c(r_i, r_j)$ equals the crowd's confidence that $r_i = r_j$. Without loss of generality, we assume that $f_c(r_i, r_j)$ equals the fraction of crowd workers (among those who inspect the record pair (r_i, r_j)) that mark r_i and r_j as duplicates. In addition, we set $f_c(r_i, r_j) = 0$ if (r_i, r_j) is eliminated in the pruning phase. The cluster generation phase of *ACD* aims to minimize the following metric $\Lambda'(R)$:

$$\Lambda'(R) = \sum_{r_i, r_j \in R, i<j} x_{i,j} \cdot (1 - f_c(r_i, r_j)) + \sum_{r_i, r_j \in R, i<j} (1 - x_{i,j}) \cdot f_c(r_i, r_j), \quad (2)$$

where $x_{i,j} = 1$ if r_i and r_j are in the same cluster and $x_{i,j} = 0$ otherwise. The minimization of $\Lambda'(R)$ is an instance of the *correlation clustering* problem [9], which is NP-hard. The classic approximate

solution for the problem is *Pivot* [5], which is a randomized algorithm that yields a 5-approximation in expectation. Accordingly, we adopt *Pivot* in the cluster generation phase of *ACD*.

The adoption of *Pivot* under the crowd-based setting is non-trivial, since *Pivot* is a *sequential* algorithm. In particular, *Pivot* constructs clusters one by one, such that the formation of the i-th ($i > 1$) cluster depends on those of the first $i - 1$ clusters. Therefore, if we are to translate *Pivot* directly into a crowd-based algorithm, we need to crowdsource the generation of clusters one at a time, i.e., we cannot construct the i-th cluster until we get the crowd's answer to the HITs pertinent to the first $i - 1$ clusters. Such sequential processing of clusters leads to a large total overhead in terms of processing time. In contrast, if we can simultaneously crowdsource the construction of multiple clusters, then the total time required to complete the clustering procedure could be significantly reduced, due to the concurrent generation of clusters by the crowd.

To parallelize the construction of clusters, one may attempt to post a large number of HITs in one batch, and *dynamically* adjust the content of HITs according to the crowd workers' answers. For example, we may issue a number n of HITs to the crowd, such that the first n' HITs are pertinent to the generation of the first cluster, while the remaining HITs are left as blank initially. Then, whenever a crowd worker (say, Alice) finishes answering the first n' HITs, we inspect her answers to decide the formation of the first cluster; based on that, we dynamically generate the HITs required for the construction of the second cluster, and show them to Alice by utilizing some of the $n - n'$ blank HITs previously "reserved". (Such dynamic adjustment can be implemented on Amazon Mechanic Turk using the *iFrame* technique.) This approach, albeit intuitive, assumes that the HITs required for the i-th cluster can be chosen based only on Alice's answers to the HITs for the first $i - 1$ clusters. In practice, however, we usually need to take answers from multiple workers on each HIT, as a means to mitigate human errors. Therefore, Alice's answers for the first $i - 1$ clusters are insufficient to decide the HITs for the i-th cluster; instead, we need to incorporate multiple users' responses to HITs for the first $i - 1$ clusters, before we can decide HITs to be shown next. This requires us to synchronize the efforts of multiple workers, which is highly challenging and not supported by the standard interface of the existing crowdsourcing platforms.

To remedy the drawbacks of the above approach, we propose a method to parallelize the *Pivot* algorithm under the crowd-based setting, while retaining its approximation guarantee. The basic idea of our method is to predict the HITs required by *Pivot* in constructing clusters, and then issue them to the crowd in one batch, at the minor cost of posting some HITs that are unnecessary in the sequential version of *Pivot*. In Section 4, we provide the details of our parallel method.

Cluster Refinement Phase. Although *Pivot* returns a 5-approximation in the expected case, its performance in a single run may be rather inferior, due to the fact that it is a randomized algorithm. To circumvent this drawback of *Pivot*, a standard under the machine-based setting is to repeat *Pivot* a large number of times, and then choose the result from the run that minimizes the cost metric. This approach, however, does not work under the crowd-based setting. First, repeating *Pivot* for a large number of times would incur a significant crowdsourcing overhead. Second, if we are to evaluate the output of a run of *Pivot* according to the cost metric $\Lambda'(R)$ in Equation 2, we would have to first obtain the crowd-based similarity score function f_c, in which case we need to issue all record pairs in the candidate set S to the crowd. This also leads to a tremendous crowdsourcing cost.

Algorithm 1: *Crowd-Pivot*

input : The set R of records and the candidate set S generated by the pruning phase
output: A set $\mathcal{C}$ of clusters }

```
1  initialize C as an empty set;
2  construct an undirected graph G = (V_R, E_S), where each vertex in
   V_R represents a record in R, and each edge (r_i, r_j) ∈ E_S
   corresponds to a record pair in S;
3  while V_R ≠ ∅ do
4      randomly pick a vertex r from V_R as a pivot;
5      initialize an empty set P;
6      for each neighbor r' of r in G do
7          add the record pair (r, r') into P;
8      issue all record pairs in P to the crowd;
9      initialize a set C = {r};
10     for each record pair (r, r') in P do
11         if the crowd decides that f_c(r, r') > 0.5 then
12             add r' into C;
13     add C into C;
14     remove from G all vertices that appear in C;
15 return C;
```

To tackle the above problem, we propose a crowd-based algorithm to refine the initial clustering result returned by the cluster generation phase. The basic idea of the algorithm is to issue selected record pairs to the crowd, and check if the crowd answers are consistent with the initial clustering; if many of crowd answers contradict the initial clustering, then we adjust the clustering to alleviate its contradiction with the crowd. As such, we mitigate the deficiency of *Pivot* and obtain higher accuracy in deduplication. As with the case of cluster generation phase, we also devise a parallel version of our cluster refinement algorithm, so as to reduce the total time required in crowdsourcing. We provide details of our refinement algorithms in Section 5.

4. CLUSTER GENERATION

This section details the algorithms used in the cluster generation phase of *ACD*. We first present a crowd-based version of the *Pivot* algorithm in Section 4.1, and then describe its parallelization in Sections 4.2 and 4.3.

4.1 Crowd-Pivot Algorithm

Algorithm 1 shows the pseudo-code of *Crowd-Pivot*, a simple adoption of the *Pivot* algorithm [9] under the crowd-based setting. The input of *Crowd-Pivot* includes the set R of records to be deduplicated, and the candidate set S of record pairs generated by the pruning phase of our solution (see Section 3). Its output is a set of disjoint clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ whose union equals R.

Crowd-Pivot first initializes $\mathcal{C} = \emptyset$, and constructs an undirected graph $G = (V_R, E_S)$, such that each vertex in V_R is a record in R, and each edge $(r_i, r_j) \in E_S$ corresponds to a record pair in S (Lines 1-2). The subsequent part of *Crowd-Pivot* runs in several iterations (Lines 3-14). In the i-th iteration, *Crowd-Pivot* randomly selects a vertex r from V_R as a *pivot*, and creates a set P that contains all record pairs (r, r') that correspond to the edges incident to r in G (Lines 4-7). After that, it posts all record pairs in P to the crowd (Line 8). Each record pair (r, r') will be evaluated by multiple crowd workers, and we set the crowd-based similarity score between r and r' (denoted as $f_c(r, r')$) to the percentage of crowd workers who believe that r and r' correspond to the same entity.

Once all record pairs in P are processed by the crowd, *Crowd-Pivot* forms a cluster C that includes r, as well as any record r' such that $(r, r') \in W$ and $f_c(r, r') > 0.5$ (Lines 9-12). Then, *Crowd-Pivot* removes from G all vertices that appear in C, and proceeds to the next iteration (Line 13). This iterative process terminates when all vertices in G are removed.

Crowd-Pivot is almost identical to the standard *Pivot* algorithm [9]. The only difference is that *Pivot* assumes that the similarity score for each edge in G is predefined, whereas *Crowd-Pivot* decides the similarity scores on the fly with the crowd. By the existing results on *Pivot* [5], we have the following lemma:

LEMMA 1. *Algorithm 1 returns a 5-approximation for the minimization of $\Lambda'(R)$ (in Equation 2), in expectation to its random choices made.* □

We omit the proof of Lemma 1 and refer interested readers to [5].

4.2 Parallel Crowd-Pivot: Rationale

Recall that, in each iteration of *Crowd-Pivot*, we issue a set of candidate pairs to the crowd and wait for the crowd to answer, before we can proceed to the next iteration. As a consequence, the running time of *Crowd-Pivot* mainly depends on the number of iterations. To reduce the running time of *Crowd-Pivot*, we aim to reduce the number of iterations in *Crowd-Pivot*, by selecting multiple pivots in each iteration and constructing multiple clusters simultaneously. Although similar ideas have been studied under a machine-based setting [34], the algorithm thus proposed does not retain the approximation guarantee of *Pivot*. In contrast, we propose a parallel version of *Crowd-Pivot* that constructs multiple clusters at the same time, without compromising the worst-case guarantee on the clustering results.

Let $\mathcal{M}$ be a random permutation of the records in R. For any record $r \in R$, if r is the i-th record in $\mathcal{M}$, then we say that the *permutation rank* of r in $\mathcal{M}$ equals i. Recall that, in each iteration of *Crowd-Pivot*, we randomly pick a record r from the un-clustered ones as a *pivot*, and then construct a cluster centering at r. Equivalently, we can first select a random permutation $\mathcal{M}$ before *Crowd-Pivot* starts, and then in each iteration of *Crowd-Pivot*, we can choose r as the record that has the smallest permutation rank among the un-clustered ones.

Our idea to parallelize *Crowd-Pivot* is that, in each iteration of the algorithm, we choose multiple pivots with the smallest permutation ranks among the un-clustered vertices, instead of selecting only one pivot. Given the chosen pivots, we issue to the crowd all record pairs pertinent to the pivots, and we construct clusters based on the crowd workers' answers, in such a way that the clustering results produced are identical to the case when we run the sequential version of *Crowd-Pivot* given the same permutation order $\mathcal{M}$. This ensures that the parallelization of *Crowd-Pivot* retains its approximation guarantee. In the following, we first consider a simple setting where we choose, in each iteration, 2 pivots with the smallest permutation ranks among the un-clustered vertices. (We will soon extend our discussions to the case when k pivots are chosen simultaneously.)

Consider the graph G constructed in Line 2 of *Crowd-Pivot*. Let G_i be the version of G at the beginning of *Crowd-Pivot*'s i-th iteration, i.e., each vertex in G_i represents a record that is not clustered in the first $i-1$ iterations. Assume that, in the i-th iteration of *Crowd-Pivot*, we choose two pivots r_1 and r_2 that have the smallest and second smallest permutation ranks, respectively, among the vertices in G_i. Let $d_i(r_1, r_2)$ be the number of hops between r_1 and r_2 in G_i, e.g., $d_i(r_1, r_2) = 1$ if they are neighbors in G_i. We differentiate three cases: (i) $d_i(r_1, r_2) > 2$, (ii) $d_i(r_1, r_2) = 2$, and (iii) $d_i(r_1, r_2) = 1$. We will demonstrate each case, assuming that G_i is as illustrated in Figure 2a, where the number on each edge (x, y) indicates the similarity score $f_c(x, y)$ that we would obtain if we issue (x, y) to the crowd.

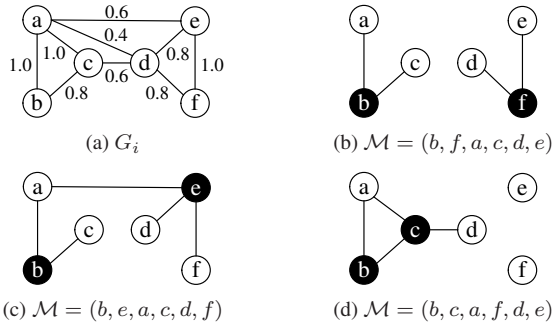


Figure 2: Different cases for the pivot choice.

Case 1: $d_i(r_1, r_2) > 2$. Consider a permutation $\mathcal{M} = (b, f, a, c, d, e)$ on the vertices in G_i in Figure 2a. Given $\mathcal{M}$, we would choose $r_1 = b$ and $r_2 = f$ as pivots, and then issue to the crowd all edges incident to r_1 and r_2, namely, (b, a), (b, c), (f, d), and (f, e), as shown in Figure 2b. Observe that, for each of these four edges, the crowd-based similarity score for the two records in the edge is above 0.5. Accordingly, we construct a cluster C_1 that contains $r_1 = b$ and its two neighbors a and c, as well as a cluster C_2 consisting of $r_2 = f$ and all of its neighbors d and e. Notice that this parallel clustering result is identical to the case when we run the sequential version *Crowd-Pivot* on G_i, using $r_1 = b$ as a pivot, followed by $r_2 = b$. In addition, the number of edges crowdsourced are also identical in both cases. □

Case 2: $d_i(r_1, r_2) = 2$. Consider a permutation $\mathcal{M} = (b, e, a, c, d, f)$ on the vertices in G_i in Figure 2a. In that case, we set $r_1 = b$ and $r_2 = e$, and crowdsource all edges incident to b and e, i.e., (b, a), (b, c), (e, a), (e, d), and (e, f), as shown in Figure 2c. Each of these edges has a crowd-based similarity score above 0.5. Given the crowd answer on each edge, we first generate a cluster C_1 that includes $r_1 = b$ and its neighbors a and c. After that, we construct a cluster C_2 that consists of $r_2 = e$ and two of its neighbors, namely, d and f. Notice that C_2 does not include a, as it has been contained in C_1. Observe that this result is the same as the case when we apply *Crowd-Pivot* on G_i with $r_1 = b$ as a pivot, followed by $r_2 = e$. However, *Crowd-Pivot* would not issue the edge (e, a) to the crowd, because once C_1 is formed, it removes all records in C_1 from G_i, after which (e, a) is no longer under consideration. In other words, our parallel construction of C_1 and C_2 incurs a small additional overhead, in that it crowdsources one "useless" edge (e, a) that has no effect on the clustering result. □

Case 3: $d_i(r_1, r_2) = 1$. Consider a permutation $\mathcal{M} = (b, c, a, f, d, e)$ on the vertices in G_i in Figure 2a. With this permutation, we select $r_1 = b$ and $r_2 = c$ and post all edges of b and c to the crowd. Figure 2d shows all edges posted, i.e., (b, a), (b, c), (c, a), and (c, d), all of which have crowd-based similarity scores above 0.5. In that case, we first form a cluster C_1 with $r_1 = b$ and its neighbors a and c. Then, we are unable to form a second cluster using c as a pivot, since c is already included in C_1. This result is identical to the case when we run *Crowd-Pivot* on G_i with $r_1 = b$ as a pivot. In other words, although we crowdsource all edges incident to $r_1 = b$ and $r_2 = c$, we end up with one cluster C_1 only. Furthermore, compared with *Crowd-Pivot*, we pay extra cost in crowdsourcing the edge (c, d) (which is not considered in *Crowd-Pivot*). □

As we can observe from the above discussions, it is possible to parallelize the construction of clusters in *Crowd-Pivot*, at the potential cost of having some *wasted pairs*, e.g., the edge (e, a) in Case 2 and the edge (c, d) in Case 3. In particular, we define a candidate pair as a wasted pair, if it is crowdsourced by our parallelized version of *Crowd-Pivot* but not in the sequential version. We now extend our discussions to the case when we choose k pivots simultaneously.

Given G_i, we first identify the k records $r_1, r_2, \ldots, r_k$ with the smallest permutation ranks in G_i. After that, we crowdsource all edges in G_i that are adjacent to r_j ($j \in [1, k]$). Let P' be the subset of those edges whose crowd-based similarity scores are above 0.5, and H_i be the subgraph of G_i induced by the edges in P'. Then, we generate a cluster that consists of r_1 and all of its neighbors in H_i, and remove all records in the cluster from H_i. Subsequently, we inspect each r_j ($j \in [2, k]$) in ascending order of j. For each r_j, if r_j has not been removed from H_i, we create a cluster that contains r_j and its remaining neighbors in H_i (i.e., we exclude the neighbors of r_j that have been eliminated in the processing of $r_1, \ldots, r_{j-1}$). In other words, we simultaneously crowdsource all record pairs pertinent to $r_1, \ldots, r_k$, and we construct multiple clusters from $r_1, \ldots, r_k$ in one batch based on the crowd workers' answers. Algorithm 2 shows the pseudo-code of this parallel method (referred to as *Partial-Pivot*). Lemma 2 establishes the correctness of the algorithm.

LEMMA 2. *Let G_i be the version of G in the beginning of Crowd-Pivot's i-iteration, and $\mathcal{M}$ be the permutation of records implicitly used by Crowd-Pivot. Let r_k be the record with the k-th smallest permutation rank in G_i, and $\mathcal{C}'_i$ be the set of clusters generated by Crowd-Pivot from its i-th iteration until it chooses a pivot with a larger permutation rank than r_k. Then, given G_i, k, and $\mathcal{M}$, Partial-Pivot outputs $\mathcal{C}_i = \mathcal{C}'_i$.* □

In Section 4.3, we will use *Partial-Pivot* to develop a parallelized version of *Crowd-Pivot*. Before that, we first introduce a method for deriving the maximum number of wasted pairs incurred by *Partial-Pivot*. Let $r_1, r_2, \ldots, r_k$ be as defined in Line 2 of *Partial-Pivot*, and w_j be defined as follows:

$$w_j = \begin{cases} \text{the number of vertices (in } G_i\text{) adjacent to } r_j\text{, except } r_1, \ldots, r_{j-1}, \\ \qquad \text{if } d(r_j, r_x) = 1 \text{ for some } x < j; \\ \text{the number of vertices (in } G_i\text{) adjacent to both } r_j \text{ and some } r_l\ (l < j), \\ \qquad \text{otherwise.} \end{cases} \tag{3}$$

We have the following lemma:

LEMMA 3. *The number of wasted pairs incurred by Partial-Pivot equals $\sum_{j=1}^{k} w_j$ in the worst case.* □

4.3 Parallel Crowd-Pivot: Algorithm

Observe that, if we set $G_i = G$ and $k = |R|$ in the input to *Partial-Pivot*, then it would cluster all records in R with a single-batch of HITs, and its output would be identical with the output of *Crowd-Pivot* given G and the same permutation $\mathcal{M}$. This leads to a maximum degree of parallelization, but at a tremendous cost: It requires issuing all edges in G to the crowd, which incurs a significant amount of wasted pairs. On the other hand, if we set $k = 1$, then *Partial-Pivot* can only generate one cluster, in which case it avoids wasted pairs but fails to achieve any parallelism. A natural idea here is to choose k in a way that strike a balance between the number of wasted pairs and the degree of parallelism. Towards that end,

Algorithm 2: *Partial-Pivot*

input : G_i, k, and $\mathcal{M}$
output: A set $\mathcal{C}_i$ of clusters and G_{i+1} (a subgraph of G_i)

1 initialize $\mathcal{C}_i$ as an empty set;
2 identify the record r_j ($j \in [1, k]$) with the j-th smallest permutation rank in G_i;
3 let P be a set that contains all edges in G_i that are adjacent to $r_1, r_2, \ldots, r_k$;
4 issue all edges in P to the crowd;
5 let P' be the set of edges $(r, r') \in W$ with $f_c(r, r') > 0.5$;
6 let H_i be the subgraph of G_i induced by the edges in P';
7 **for** $j = 1$ *to* k **do**
8 **if** r_j *remains in* H_i **then**
9 create a cluster C that contains r_j and all of its neighbors in H_i;
10 insert C into $\mathcal{C}_i$;
11 remove all vertices in C from both H_i and G_i;
12 **return** $\mathcal{C}_i$ and $G_{i+1} = G_i$;

Algorithm 3: *PC-Pivot*

input : The set R of records and the candidate set S generated by the pruning phase
output: A set $\mathcal{C}$ of clusters

1 initialize $\mathcal{C}$ as an empty set;
2 construct an undirected graph $G = (V_R, E_S)$, where each vertex in V_R represents a record in R, and each edge $(r_i, r_j) \in E_S$ corresponds to a record pair in S;
3 select a random permutation $\mathcal{M}$ of the records in R;
4 **while** $V_R \neq \emptyset$ **do**
5 derive the maximum k that satisfies Equation 4;
6 $(\mathcal{C}', G)$ = *Partial-Pivot*(G, k, $\mathcal{M}$);
7 $\mathcal{C} = \mathcal{C} \cup \mathcal{C}'$;
8 **return** $\mathcal{C}$;

we introduce a constant ε, and consider the following optimization problem: Given G_i and $\mathcal{M}$, we aim to select the maximum k under the constraint that

$$\sum_{j=1}^{k} w_j \quad \leq \quad \varepsilon \cdot |P_j|, \tag{4}$$

where P_j is the set of edges in G_i that are incident to $r_1, r_2, \ldots, r_j$. In other words, we maximize k while ensuring that, among the record pairs issued by *Partial-Pivot* to the crowd, at most a ε fraction is wasted. Observe that such k can be easily derived using a linear scan of all w_j and P_j.

Based on the above method for deciding k, we propose a parallel version of *Crowd-Pivot* (referred to as *PC-Pivot*), as shown in Algorithm 3. The input to *PC-Pivot* is identical to that of *Crowd-Pivot*, which includes a set S of records to be clustered and a candidate set S of record pairs produced by the pruning phase of our solution. As with *Crowd-Pivot*, *PC-Pivot* also begins by initializing an empty set $\mathcal{C}$ and constructing a graph G that represents the records in R as vertices and the record pairs in S as edges (Lines 1-2). After that, it selects a random permutation $\mathcal{M}$ of the records in R, and generates clusters in an iterative manner (Lines 3-7). In particular, in the first iteration, *PC-Pivot* inspects G, and derives the maximum k that satisfies Equation 4. Then, it invokes *Partial-Pivot* with $G_i = G$, k, and $\mathcal{M}$ as input, and obtains as output a set of clusters $\mathcal{C}'$ and a modified version of G (where some vertices are removed). After that, *PC-Pivot* inserts the clusters in $\mathcal{C}'$ into $\mathcal{C}$. Each subsequent iteration of *PC-Pivot* is performed in the same manner: *PC-Pivot* first decides k based on the current version of G, and applies *Partial-Pivot* to construct clusters. This iterative proce-

dure terminates when all vertices in G are removed, in which case *PC-Pivot* returns $\mathcal{C}$ as output.

The following lemma shows the performance guarantees of *PC-Pivot*.

LEMMA 4. *PC-Pivot returns a 5-approximation for the minimization of $\Lambda'(R)$ (in Equation 2), in expectation to its random choices made. In addition, among the record pairs crowdsourced by PC-Pivot, at most a fraction ε is wasted.* □

5. CLUSTER REFINEMENT

Although our cluster generation phase returns a 5-approximation in expectation (for the minimization of $\Lambda'(R)$), its performance in a particular run could be rather inferior, because of its randomized nature. To address this issue, the cluster refinement phase of our solution *post-processes* the results produced by the cluster generation phase, by using additional HITs to fine-tune the clusters, so as to reduce $\Lambda'(R)$. In what follows, we will introduce postprocessing operations in Section 5.1, analyze the cost and benefit of postprocessing operations in Section 5.2, present crowd-based postprocessing algorithm in Section 5.3, and demonstrate parallelized postprocessing algorithms in Section 5.4.

5.1 Rationale and Basic Operations

There exist several machine-based algorithms (e.g., [5, 23]) for postprocessing the results generated by a randomized correlation clustering algorithm. These algorithms, however, require as input the similarity scores of all record pairs, which incur significant monetary costs under a crowd-based setting. For example, consider the *BOEM* algorithm [5]. It runs in several iterations, each of which moves one record from one cluster to another. To determine the record to be moved in an iteration, *BOEM* first identifies a set R of candidate records, and then examines the similarity scores of all record pairs involving at least one record in R, based on which it selects a record whose move leads to the largest reduction in the cost function of correlation clustering (see Equation 1). If we directly adopt *BOEM* in a crowd-based setting, then in each iteration, we need to crowdsource a large number of record pairs to obtain their crowd-based similarity scores, which results in significant overheads.

To address the above problem, we propose to adjust clusters with a sequence of operations that are economical in terms of the number of additional record pairs that need to be crowdsourced. In particular, we consider two basic operations, *split* and *merger*. A split operation removes a vertex v from a cluster, and uses v to constructs a singleton cluster. A merger operation combines two clusters C_1 and C_2 into a new cluster $C_1 \cup C_2$. Our objective is to develop a crowd-based algorithm that takes as input a set of clusters, and identifies a split or merger operation that will reduce $\Lambda'(R)$ without substantial crowsourcing costs. Given such an algorithm, we can then recursively apply it on the clustering results produced by our cluster generation phase, so as to improve the quality of clustering. As a first step, we will analyze the cost and benefit of performing a split or merger operation on a set of clusters.

5.2 Cost-Benefit Analysis

Suppose that we split a record r from a cluster C. Let o_s denote this split operation, and $C' = C \setminus \{r\}$. According to Equation 2, after the split, $\Lambda'(R)$ decreases by:

$$b(o_s) = \sum_{r' \in C \setminus \{r\}} \big(1 - 2f_c(r, r')\big). \quad (5)$$

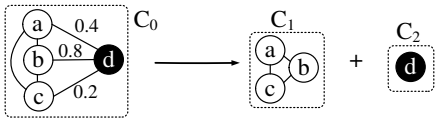


Figure 3: illustration of a split operation.

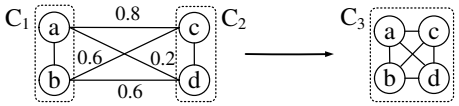


Figure 4: illustration of a merger operation.

We define $b(o_s)$ as the *benefit* of o_s. On the other hand, if we merge two clusters C_1 and C_2, then the merger operation (denoted as o_m) would reduce $\Lambda'(R)$ by:

$$b(o_m) = \sum_{r_1 \in C_1, r_2 \in C_2} (2f_c(r_1, r_2) - 1). \quad (6)$$

We refer to $b(o_m)$ as the benefit of o_m. We illustrate $b(o_s)$ and $b(o_m)$ with an example.

EXAMPLE 2. Figure 3 demonstrates a split operation o_s to split record d from a cluster $C_0 = \{a, b, c, d\}$. The number on each edge (x, y) indicates the similarity score $f_c(x, y)$ that we would obtain if we issue (x, y) to the crowd. We omit the similarity score if the edge is irrelevant to the operation. Based on Equation 5, the benefit of this operation is 0.2. That is, by applying this split operation, $\Lambda'(R)$ can be reduced by 0.2. After the split operation, we obtain two clusters $C_1 = \{a, b, c\}$ and $C_2 = \{d\}$. Figure 4 illustrates a merger operation o_m to merge cluster $C_1 = \{a, b\}$ and $C_2 = \{c, d\}$. With Equation 6, it can be calculated that the benefit of this merger operation is 0.4. After the merger operation, we obtain a new cluster $C_3 = \{a, b, c, d\}$. □

Intuitively, if we are to fine-tune a set $\mathcal{C}$ of clusters, we should identify the operation o with the largest positive benefit $b(o)$, and then apply o on $\mathcal{C}$. However, the computation of $b(o)$ requires the crowd-based similarity scores of various record pairs, which may not be readily available. In particular, if a record pair (r, r') has not been crowdsourced during the cluster generation phase, then $f_c(r, r')$ remains unknown in the cluster refinement phase, unless we post (r, r') to the crowd. That is, the derivation of $b(o)$ incurs crowdsourcing overheads. Therefore, we cannot solely rely on $b(o)$ to identify the best operation for cluster refinement. Instead, we adopt a more economical approach as follows:

1. Let O be the set of all possible (split and merger) operations on a given set $\mathcal{C}$ of clusters, and O^+ be the set of operations in O whose benefits are known and are larger than zero. If $O^+ \neq \emptyset$, then we choose the operation in O^+ with the largest benefit, and apply it on $\mathcal{C}$.
2. If $O^+ = \emptyset$, then we estimate the benefit of each operation in O, and choose an operation o. After that, we compute the exact benefit $b(o)$ of o (by crowdsourcing the relevant record pairs with unknown similarity scores), to check if $b(o) > 0$. If $b(o) > 0$, then we perform o on $\mathcal{C}$; otherwise, we ignore o.

We now clarify how we estimate $b(o)$ when it is unknown. For simplicity, we assume that o is a split operation that removes a record r from a cluster C; our discussions can be easily extended to the case of merger operations. By Equation 5, the computation of $b(o)$ requires the crowd-based similarity scores $f_c(r, r')$ of

all record pairs (r, r') with $r' \in C \setminus \{r\}$. Let P be the set of record pairs (among those required by the computation of $b(o)$) that are unknown. Our idea is to (i) estimate the crowd-based similarity score $f_c(r, r')$ of each record pair $(r, r') \in P$ based on its machine-based similarity score $f(r, r')$, and then (ii) compute an estimated value of $b(o)$ accordingly. Towards this end, a straightforward solution is to set directly use $f(r, r')$ as an estimation of $f_c(r, r')$ [46, 47]. However, this solution is often inaccurate due to the limitations of machine-based algorithms. As an improved solution, previous work [48] proposes to (i) crowdsource a number of record pairs and then (ii) utilize the crowd workers' answers to construct a histogram, which maps each $f(r, r')$ into a more accurate estimation of $f_c(r, r')$. We adopt a similar solution as follows.

First, we collect the set A of record pairs whose crowd-based similarity scores are known from the cluster generation phase. Then, we inspect the machine-based similarity scores of those record pairs, and construct an equi-depth histogram H with m buckets on the machine-based scores. (Following [48], we set $m = 20$.) After that, for each bucket B in H, we examine the record pairs whose machine-based similarity scores fall in B, and compute the average crowd-based similarity score of those record pairs. Subsequently, if we have any record pair (r, r') whose crowd-based similarity score $f_c(r, r')$ is unknown, we first identify the bucket that covers $f(r, r')$, and then we estimate $f_c(r, r')$ as the average crowd-based similarity score associated with the bucket. In addition, whenever we crowdsource additional record pairs in the cluster refinement phase, we also insert the record pairs into A and reconstruct the histogram H, for more accurate estimation.

Next, we explain how we choose the operation to be performed on $\mathcal{C}$, given a benefit estimation of each operation of interest. As we aim to minimize $\Lambda'(R)$, an intuitive choice is to select the operation o with the maximum estimated benefit. However, this choice may incur a large crowdsourcing cost, since we need to first compute the exact value of $b(o)$ (to confirm o's benefit) before we apply o on $\mathcal{C}$. That is, if the computation of $b(o)$ requires crowsourcing a large number of record pairs, then o may not be an ideal choice of operation. Motivated by this, we select operations based on a metric that takes into account both the estimated benefit of an operation and the crowdsourcing overhead incurred. In particular, we choose the operation o that maximizes $b^*(o)/c(o)$, where $b^*(o)$ is the estimated value of $b(o)$, and $c(o)$ is the *cost* of o defined as follows. If o splits a vertex v from a cluster C,

$$c(o) = \left| \left\{ r' \;\middle|\; r' \in C \setminus \{r\} \wedge f_c(r, r') \notin A \right\} \right|, \qquad (7)$$

where A denotes the set of record pairs that have been crowdsourced. In other words, $c(o)$ equals the number of record pairs that need to be posted to the crowd, if we are to compute $b(o)$ exactly. Similarly, if o merges two clusters C_1 and C_2, then

$$c(o) = \left| \left\{ (r_1, r_2) \;\middle|\; r_1 \in C_1 \wedge r_2 \in C_2 \wedge f_c(r_1, r_2) \notin A \right\} \right|. \quad (8)$$

Intuitively, by selecting an operation o with the maximum $b^*(o)/c(o)$, we strike a better balance between the minimization of $\Lambda'(R)$ and the overhead of crowdsourcing. For convenience, we refer to $b^*(o)/c(o)$ as the *benefit-cost ratio* of o.

5.3 Crowd-Based Postprocessing

Based on the discussions in Sections 5.1 and 5.2, Algorithm 4 illustrates a (sequential) crowd-based method for cluster refinement, referred to as *Crowd-Refine*. The algorithm takes as input the set $\mathcal{C}$ of clusters produced by the cluster generation phase, as well as the set A of record pairs that have been crowdsourced. It first constructs, based on A, a histogram H that maps machine-based similarity scores to estimated crowd-based similarity scores (Line 1), using the method described in Section 5.2. The subsequent execution of *Crowd-Refine* consists of a number of iterations (Lines 2-16). In each iteration, it first identifies the set O of all (split and merger) operations on $\mathcal{C}$, and the set O^+ of operations in O whose benefits are known and positive (Lines 3-4). If $O^+ \neq \emptyset$, then there exists at least one operation that can reduce $\Lambda'(R)$ without incurring any crowdsourcing overhead; accordingly, *Crowd-Refine* pinpoints the operation in O^+ with the maximum benefit, and applies it on $\mathcal{C}$ (Lines 5-7). On the other hand, if $O^+ = \emptyset$, then *Crowd-Refine* proceeds to derive the operation $o' \in O$ with the largest benefit-cost ratio $b^*(o')/c(o')$ (Line 9). If this ratio is larger than 0, then *Crowd-Refine* computes the exact value of $b(o)$ to check if the benefit of o' is indeed positive (Line 12), and the record pairs crowdsourced during the derivation of $b(o)$ are added into A and used to update the histogram H (Lines 15-16). In case that $b(o') > 0$ holds, *Crowd-Refine* performs o' on $\mathcal{C}$ (Line 13-14). However, if the benefit-cost ratio of o' is non-positive in the first place, then all operations in O should have non-positive estimated benefits. In that case, *Crowd-Refine* terminates by returning $\mathcal{C}$ (Lines 10-11).

Algorithm 4: *Crowd-Refine*

```
   input : A set C of clusters and a set A of crowdsourced record pairs,
           both obtained from the cluster generation phase
   output: A set C of improved clusters
1  construct a histogram H that maps machine-based similarity scores to
   estimated crowd-based similarity scores;
2  while true do
3      let O be the set of all operations on C;
4      let O+ be the set of operations (in O) whose benefits are known
       and larger than zero;
5      if O+ ≠ ∅ then
6          let o be the operation in O+ with the largest benefit;
7          apply o on C;
8      else
9          let o' be the operation in O with the largest benefit-cost ratio
           b*(o')/c(o');
10         if b*(o')/c(o') ≤ 0 then
11             return C;
12         compute b(o') by crowdsourcing relevant record pairs with
           unknown crowd-based similarity scores;
13         if b(o') > 0 then
14             apply o' on C;
15         insert all newly crowdsourced record pairs into A;
16         update H;
```

Let n be the total number of records in R. Observe that, on any set $\mathcal{C}$ of clusters on R, there are $O(n)$ split operations and $O(n^2)$ merger operations. Therefore, in each iteration of *Crowd-Refine*, we consider $O(n^2)$ operations, which is a reasonable number given that we need to process $O(n^2)$ record pairs in our clustering tasks. We refer the reader to Appendix B for an example that demosntrates the execution of *Crowd-Refine*.

5.4 Parallel Postprocessing

Notice that *Crowd-Refine* is a sequential algorithm, in that it adjusts clusters using one operation at a time, and each operation may require posting some record pairs to the crowd and wait for the answers. Such sequential processing leads to significant processing time, as we point out in Section 3. To remedy this deficiency, we aim to parallelize *Crowd-Refine*, in a manner similar to the paral-

lelization of *Crowd-Pivot*. To that end, we first introduce the concept of *independence* between different operations on $\mathcal{C}$.

Specifically, we say that two operations o_1 and o_2 are *independent*, if the clusters adjusted by o_1 are completely different from those adjusted by o_2. For instance, if o_1 splits a vertex from a cluster C_0, and o_2 merges two other clusters C_1 and C_2, then o_1 and o_2 are independent of each other. In contrast, o_1 is not independent of an operation that merges C_0 with another cluster, or one that splits another vertex from C_0. Observe that if two operations are independent, then we can apply them simultaneously on $\mathcal{C}$ without any side effect. Therefore, if we are to parallelize *Crowd-Refine*, a natural idea is to first identify a large number of independent operations, and then process them concurrently by crowdsourcing all relevant record pairs in one batch.

Based on the above idea, we propose to refine a given set $\mathcal{C}$ of clusters with a two-step approach as follows:

1. Let O^+ be the set of operations on $\mathcal{C}$ whose benefits are known and are larger than zero. We recursively choose the operation in O^+ with the largest benefit and apply it on $\mathcal{C}$, until O^+ becomes empty. Note that this step can be conducted without the crowd.

2. Whenever $O^+ = \emptyset$, we identify a set O^i of independent operations on $\mathcal{C}$, and process the operations in O^i concurrently with the crowd. After that, if O^+ becomes non-empty, we go back to Step 1; otherwise, we repeat this step.

The key issue in the above approach is the choice of O^i. Intuitively, O^i should contain operations o with large estimated benefits $b^*(o)$ but small crowdsourcing costs $c(o)$. Motivated by this, we select O^i as the set with the maximum *overall benefit-cost ratio*, which is defined as:

$$\Psi(O^i) = \frac{\sum_{o \in O^i} b^*(o)}{\sum_{o \in O^i} c(o)}. \tag{9}$$

Unfortunately, deriving such a set of independent operations is NP-hard, as shown in the following lemma.

LEMMA 5. *Given a set $\mathcal{C}$ of clusters, it is NP-hard to identify a set O^i of independent operations that maximizes $\Psi(O^i)$.* □

Given Lemma 5, we resort to a greedy heuristic for constructing O^i. In particular, we first set $O^i = \emptyset$, and then linearly scan all operations in O in descending order of their benefit-cost ratios. For each operation inspected, if its benefit-cost ratio is positive and it is independent of all operations in O^i, then we insert it into O^i; otherwise, we ignore the operation. This linear scan terminates when $\sum_{o \in O^i} c(o)$ (i.e., the total crowdsourcing cost of the operations in O^i) reaches a predefined threshold T. After that, the construction of O^i is completed. The reason that we impose a threshold τ on the total crowdsourcing cost of O^i is to avoid packing O^i with an excessive number of operations, since the operations examined in the later stage of the linear scan tend to have small benefits.

Based on the above heuristic, Algorithm 5 shows our parallel method for crowd-based cluster refinement, referred to as *PC-Refine*. *PC-Refine* is similar to *Crowd-Refine* (Algorithm 4), but differs in the handling of the case when $O^+ = \emptyset$, i.e., when there does not exist any operation whose benefit is known and positive. Specifically, when $O^+ = \emptyset$, *PC-Refine* adopts the heuristic approach previously discussed to construct a set O^i of independent operations (Lines 9-14), and then computes the exact benefits of all operations in O^i by crowdsourcing all relevant record pairs in one batch (Line 15), instead of processing one operation at a time with the crowd. After that, for each operation in O^i with positive benefit,

Algorithm 5: *PC-Refine*

```
   input : A set C of clusters and a set A of crowdsourced record pairs,
           both obtained from the cluster generation phase
   output: A set C of improved clusters
1  construct a histogram H that maps machine-based similarity scores to
   estimated crowd-based similarity scores;
2  while true do
3      let O be the set of all operations on C;
4      let O+ be the set of operations (in O) whose benefits are known
       and larger than zero;
5      if O+ ≠ ∅ then
6          let o be the operation in O+ with the largest benefit;
7          apply o on C;
8      else
9          set Oi = ∅;
10         for each o' ∈ O in descending order of b*(o')/c(o') do
11             if b*(o')/c(o') <= 0 or Σ_{o∈Oi} c(o) ≥ T then
12                 break;
13             if o' is independent of all operations in Oi then
14                 insert o' into Oi;
15         compute b(o) for all o ∈ Oi by crowdsourcing all relevant
           record pairs in one batch;
16         for each o ∈ Oi do
17             if b(o) > 0 then
18                 apply o on C;
19         if none of the operation in Oi is applied on C then
20             return C;
21         insert all newly crowdsourced record pairs into A;
22         update H;
```

PC-Refine applies the operation on $\mathcal{C}$ to reduce $\Lambda'(R)$. However, if none of the operation in O^i has a positive benefit, then it is unlikely that we can further decrease $\Lambda'(R)$, in which case *PC-Refine* terminates and returns $\mathcal{C}$ (Lines 19-20).

It remains to clarify how we set the threshold T on the total cost of the operations in O^i. Intuitively, T controls the tradeoff between *PC-Refine*'s processing time and crowdsourcing cost. When T is large, *PC-Refine* issues fewer batches of questions to the crowd, which leads to smaller processing time. However, a large T also makes *PC-Refine* pack more operations into O^i, which results in higher crowdsourcing overheads. To strike a balance between processing time and crowdsourcing cost, we set T as follows. First, given a set $\mathcal{C}$ of clusters, we first compute $\frac{|R|^2}{2 \cdot |\mathcal{C}|}$, which is the maximum number of record pairs that we need to crowdsource, if we are to perform all operations on $\mathcal{C}$ in one batch. (This maximum number is achieved when each cluster contains $|R|/|\mathcal{C}|$ records, and we conduct $|\mathcal{C}|/2$ merger operations simultaneously.) After that, we retrieve the number N_u of record pairs whose crowd-based similarity scores are currently unknown. Let N_m be the smaller one between $\frac{|R|^2}{2 \cdot |\mathcal{C}|}$ and N_u. We set $T = N_m/x$, where x is a pre-defined constant. In Section 6, we will decide x with experiments.

6. EXPERIMENTS

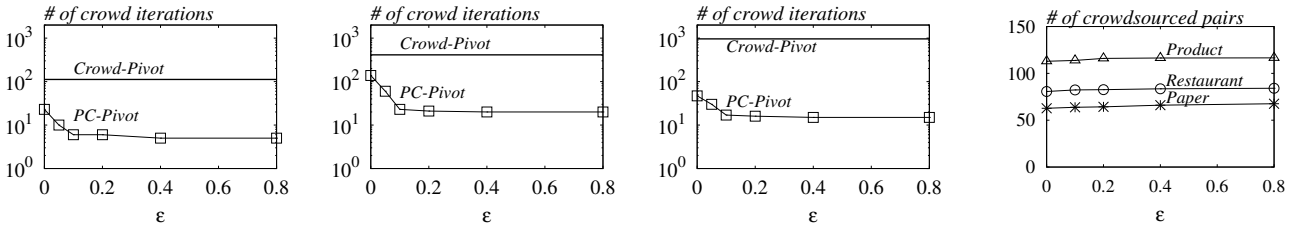
This section experimentally evaluates *ACD* against the states of the art in terms of deduplication accuracy, the number of record pairs crowdsourced, and the number of crowd iterations required.

6.1 Experimental Setup

Datasets and Baselines: We use three benchmark datasets in the literature of crowd-based data deduplication [46–48], namely, *Pa-*

datasets	# of records	# of entities	# of candidate pairs	crowd error rate (3w)	crowd error rate (5w)
Paper	997	191	29,581	23%	21%
Restaurant	858	752	4,788	0.8%	0.2%
Product	3,073	1,076	3,154	9%	5%

Table 3: Characteristics of datasets and crowd answers.



(a) # of iterations on *Paper*. (b) # of iterations on *Restaurant*. (c) # of iterations on *Product*. (d) # of crowdsourced record pairs

Figure 5: Impacts of ε.

per [1], *Restaurant* [2], and *Product* [3]. Table 3 shows the number of records and entities in each dataset.

We compare our *ACD* method against existing four state-of-the-art algorithms: *TransNode* [44], *TransM* [47], *CrowdER* [46], and *GCER* [48]. Note that *CrowdER* does not specify the algorithm for clustering crowdsourced record pairs. In our implementation, we use the *sorted neighborhood* algorithm [48] to produce clusters from crowd answers for *CrowdER*, following previous work [48]. We denote this implementation as *CrowdER+*. Meanwhile, *GCER* requires a user-specified parameter on the number of record pairs to be issued to the crowd. We set this parameter for *GCER*, so that it crowdsources the same number of record pairs as *ACD* does. This enables us to compare the deduplication accuracy of *ACD* and *GCER* under the same crowdsourcing budget. In addition, we also evaluate *ACD* against *PC-Pivot*, i.e., a "crippled" version of *ACD* that incorporates the pruning phase and cluster generation phase of *ACD* but omits the cluster refinement phase.

Following previous work [47], we use the F1-measure to gauge the deduplication accuracy of each method, and we evaluate the crowdsourcing overhead of each method in terms of the number of record pairs crowdsourced, as well as the number of crowd iterations (i.e., the number of batches of HITs that are issued to the crowd). For *ACD* and *PC-Pivot*, we repeat each of them 5 times in each experiment and report the average measurements, since they are both randomized algorithms. Note that *TransNode* [44] does not incorporate any parallel mechanism to issue HITs in a batch manner. Therefore, we omit *TransNode* from the experiments on the number of crowd iterations.

Pruning Phase Setting: All methods that we test require a machine-based approach in their pruning phases to generate a set S of candidate pairs. Following previous work [47], we compute the machine-based similarity score for each record pair using the *Jaccard* similarity metric, and we set the threshold value for machined-based similarity scores as $\tau = 0.3$. That is, only the record pairs with Jaccard similarity above 0.3 are retained in the candidate set S. Table 3 shows the size of S on each dataset.

AMT Setting: We use the Amazon Mechanical Turk (AMT) as our crowdsourcing platform. For each dataset, we post all record pairs in the candidate set S to AMT, and record the crowd's answers in local file F. Then, during our experiments, whenever a method requests to crowdsource a record pair (r, r'), we retrieve the answers for (r, r') from F instead of posting (r, r') to AMT. This ensures that all methods utilize the same set of crowdsourced results, for fairness of comparison.

Note that the above evaluation approach is adopted in previous work [46, 47]. We obtain, from the authors of [46, 47], the crowd answers used in their experiments, and reuse them in our evaluation. The AMT setting pertinent to those results are as follows. Each HIT contains 20 record pairs, and each record pair requires inputs from three workers. Each worker must pass a qualification test before she can work on the HITs, and she is paid 2 cents for completing each HIT.

In addition to reusing data from [46, 47], we also obtain an additional set of crowd answers from AMT under a more stringent setting[1]. In particular, we require each worker to not only pass a qualification test, but also has completed 100 approved HITs and has an approval rate at least 95%. (Previous work [24] adopts the same setting.) This setting is intended to ensure that all workers provide reasonably accurate answers to the HITs. To further reduce the impact of human errors, we require inputs from 5 workers for each record pair, and we only pack 10 record pairs into each HIT. The monetary reward for each HIT is set to 2 cents. For each dataset D, we denote the crowd answers obtained from this 5-worker setting as $D(5w)$, and those from the previous 3-worker setting as $D(3w)$.

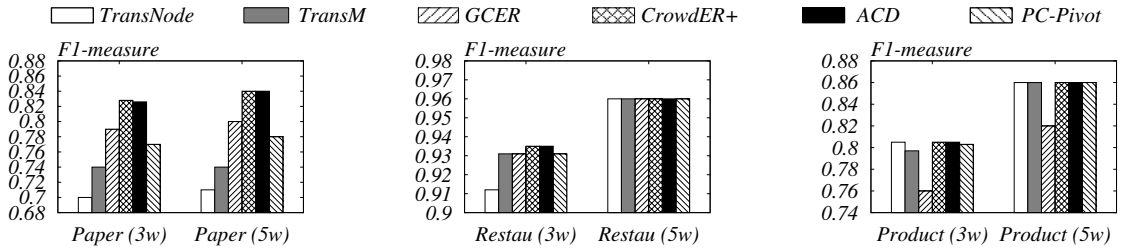
Table 3 shows, for each dataset, the percentages of crowd answers that are incorrect under the 3-worker and 5-worker settings. Evidently, the 5-worker setting results in more accurate outputs from the crowd, at a higher monetary cost. Furthermore, the crowd's error rate for *Paper* is above 20% under both settings, which indicate that the records in *Paper* are more difficult to deduplicate.

6.2 Tuning Parameters for *ACD*

Recall from Section 4.3 that, in the cluster generation phase of *ACD*, the *PC-Pivot* algorithm requires a parameter ε that decides the maximum allowed amount of wasted pairs. In the first set of experiments, we study the effect of ε on the efficiency and crowdsourcing cost of *PC-Pivot*. We only show results obtained from the 3-worker setting, but the results from the 5-worker setting are similar.

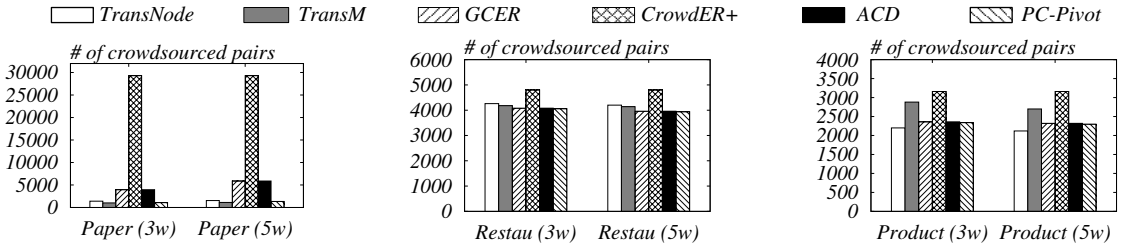
Figure 5(a), 5(b), and 5(c) show the number of crowd iterations required by *PC-Pivot*, when ε varies. For comparison, we also show the number of crowd iterations required by *Crowd-Pivot*, i.e., the sequential version of *PC-Pivot*. Observe that *PC-Pivot* incurs a

[1]We make this set of crowd answers available at [4].



(a) *Paper* dataset (b) *Restaurant* dataset (c) *Product* dataset

Figure 6: Comparison of deduplication accuracy.



(a) *Paper* dataset (b) *Restaurant* dataset (c) *Product* dataset

Figure 7: Comparison of crowdsourcing costs.

much smaller number of crowd iterations than *Crowd-Pivot* does. For example, even when we set ϵ as small as 0.1, *PC-Pivot*'s number of crowd iterations is 20 times smaller than that of *Crowd-Pivot* on the *Restaurant* dataset.

When ε increases, the number of crowd iterations decreases. This is because, with a larger ε, *PC-Pivot* can choose more pivots in each iteration, and hence, the clustering process can be accelerated. However, the decrement of crowd iterations is more significant when ε increases from 0.0 to 0.1, comparing to the case from 0.1 to 0.8. The reason is that the larger ε is, the more pivots are selected, in which case there is a higher chance that the remaining pairs are predicted as wasted pairs by the *PC-Pivot* algorithm.

Meanwhile, a larger ε leads to higher crowdsourcing costs, as shown in Figure 5(d). In particular, when ε increases, the number of HITs required by *PC-Pivot* also increases. In the following experiments, we set ε to 0.1 since it strikes a good balance between efficiency and crowdsourcing cost.

In addition to the ε, we have also conducted an experiment to evaluate the effect of T, i.e., which is a threshold used in the cluster refinement phase of *ACD* to restrict the total crowdsourcing cost of the independent operations to be performed in each iteration (see Algorithm 5). In particular, we vary T from $N_m/16$ to $N_m/2$, where N_m is as defined in Section 5.4. We observe that $T = N_m/8$ nicely balances the efficiency and cost of cluster refinement. Interested readers are referred to Appendix C for the experimental results on T. In the rest of our experiments, we set $T = N_m/8$.

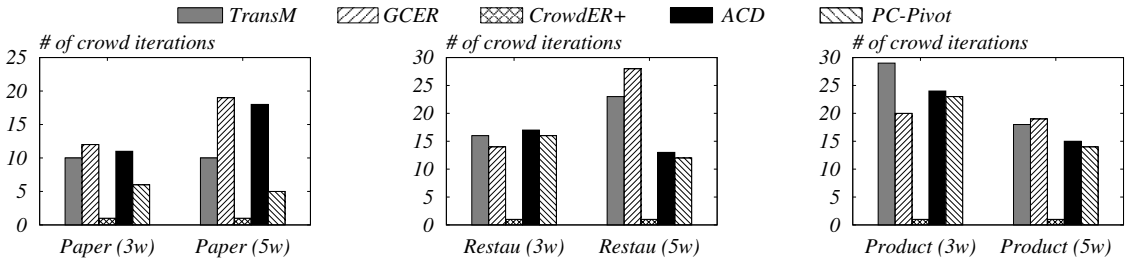
6.3 Comparison of All Methods

In our next set of experiments, we evaluate all methods on their deduplication accuracy (in terms of F1-measure), crowdsourcing costs (in terms of the number of record pairs crowdsourced), and crowdsourcing efficiency (in terms of the number of crowd iterations). Figure 6 (resp. Figure 7) shows the F1-measures (resp. numbers of crowdsourced record pairs) of each method on each dataset under both the 3-worker and 5-worker settings. *CrowdER+* consistently provides the highest accuracy, but incurs significant crowdsourcing costs (especially on *Paper*, since it requires issuing all record pairs in the candidate set S to the crowd). In contrast, *ACD* provides an accuracy that is highly comparable to that of *CrowdER+*, but incurs considerably lower crowdsourcing costs. In particular, on *paper*, the number of record pairs crowdsourced by *ACD* is less than $1/7$ (resp. $1/5$) the number by *CrowdER+* under the 3-worker (resp. 5-worker) setting. Meanwhile, *PC-Pivot* is much less accurate than *ACD* on *Paper*, which demonstrates the effectiveness of the cluster refinement phase of our solution. On the other hand, on *Restaurant* and *Product*, *PC-Pivot* provides similar accuracy to *ACD*. The reason is that, on those two datasets, the crowd workers make a relatively smaller number mistakes in processing record pairs. As a consequence, *PC-Pivot* alone can produce high-quality clustering results, which leaves little room for the cluster refinement phase to improve.

GCER's accuracy is lower than *ACD* in all cases, except for *Restaurant* under the 5-worker setting. This indicates that *GCER* provides lower-quality clustering results than *ACD* does, when both of them crowdsource the same number of record pairs. Both *TransNode* and *TransM* provide inferior deduplication accuracy on *Paper*, since they are susceptible to errors made by the crowd workers, as we analyzed in Sections 1 and 2.2. Their performance are much better on *Restaurant* and *Product*, due to the smaller amount of errors made by crowd workers on those datasets. However, on *Restaurant* and *Product*, the numbers of record pairs crowdsourced by *TransNode* and *TransM* are almost the same as that by *ACD*, i.e., they do not provide any advantage over *ACD* on those two datasets.

Overall, all methods yield higher accuracy in the 5-worker setting than the 3-worker setting, which shows that all methods benefit from more accurate results from the crowd. However, we observe that the performance of *TransNode* and *TransM* degrades more significantly than other methods in the 3-worker setting. This, again, is consistent with our analysis that *TransNode* and *TransM* are vulnerable to human errors.

Figure 8 illustrates the number of crowd iterations incurred by each method. The results of all methods are roughly comparable in all cases, except that *CrowdER+* requires only one crowd iterations, as it issues all record pairs in S in one batch. This shows that



(a) *Paper* dataset (b) *Restaurant* dataset (c) *Product* dataset

Figure 8: Comparison of crowdsourcing efficiency

crowdsourcing efficiency of *ACD* is comparable to that of *TransM* and *GCER*.

In summary, our experimental results show that *ACD* provides high deduplication accuracy at moderate costs of crowdsourcing, and its crowdsourcing efficiency is comparable to the states of the art. In contrast, the existing methods for crowd-based deduplication either provide inferior clustering results, or incur significant crowdsourcing overheads. Therefore, *ACD* is a favorable method for crowd-based data deduplication.

7. RELATED WORK

Correlation clustering is a classic method used in large scale data deduplication [8, 27]. Optimal correlation clustering is shown to be NP-hard [9], which motivates a line of research [5, 10, 22] to seek good approximation algorithms. In particular, Ailon et al. [5] propose the *Pivot* algorithm, which returns a 5-approximation when the similarity score of each record pair is a fractional number in $[0, 1]$. Meanwhile, if the similarity scores are either in $\{0, 1\}$, the algorithm yields a 3-approximation. This 3-approximation result is improved to a 2.5-approximation [5] with a linear programming approach, which, however, incurs a larger computational overhead than *Pivot*. [42] proposes a deterministic version of *Pivot* that provides the same approximation guarantee, but it also relies on linear programming. Gionis et al. [22] propose the *BOEM* algorithm for postprocessing the results of correlation clustering to improve accuracy. As we point out in Section 5.1, however, *BOEM* is unsuitable under the crowd-based setting, as it would incur a significant overhead in crowdsourcing. Besides correlation clustering, there exists a large body of literature on machine-based data deduplication algorithms (see [17] for a survey).

As we review in Section 2.2, there exist five crowd-based solutions [25, 44, 46–48] for data deduplication. Among them, *Crowd-clustering* [25] is designed for data categorization, which is a variant of data deduplication that aims to identify records belonging to the same category, instead of identifying duplicates. Meanwhile, *CrowdER* [46] provides a high precision of deduplication, but incurs a significant crowdsourcing overhead. The other three methods, namely, *GCER* [48], *TransM* [47], and *TransNode* [44], reduce the costs of crowdsourcing but compromise deuplication accuracy, as shown in our experiments. In contrast, our *ACD* approach yields as accurate results as *CrowdER* does, only at moderate costs of crowsourcing.

Besides data deduplication, there also exist many research works that incorporate crowdsourcing into record linkage, which targets to link records representing the same entity across databases. However, in such tasks, it is usually assumed that, given three records r_1, r_2, and r_3, if $r_1 = r_2$, then r_3 cannot represent the same entity as r_1 or r_2 does. Nonetheless, if r_1 does not represent the same entity as r_2 does, then r_3 can represent the same entity as r_1 or r_2 does but not both. Existing work on record linkage uses this assumption to prune a plenty of record pairs. Nevertheless, such an assumption does not hold for data deduplication scenario, which makes them inapplicable under data deduplication scenarios. Arasu et al. [7] mainly use human power to build up the classifier for the dataset, while as shown in previous work [46], these techniques often face difficulties when processing records that represent different entities but look highly similar. On the contrary, our solution can still work in such scenarios. Demartini et al. [16] propose *ZenCrowd* to do record linkage for large collections of online pages with the crowd. Gokhale et al. [24] study how to do hands-off crowdsourcing record linkage which requires no involvement of developers.

In addition, recent work has developed several crowdsourcing platforms for various tasks. Franklin et al. [19, 20] propose a *CrowdSQL* query language, and develop the *CrowdDB* database. Marcus et al. [35] propose *Qurk*, a query processing system for allowing crowd-powered processing of relational databases. Parameswaran et al. [38] develop *Deco*, a database system that answers, with crowd assistance, declarative queries posted over relational data. Jeffery et al. [30, 31] construct a hybrid crowd-machine data integration system. Furthermore, several database operations are also studied under the crowd setting, such as JOIN [35], MAX [26], TOP-K, and GROUP BY queries [15]. Others have also investigated crowd-based techniques for data mining [6], web table matching [18], and data analysis [33]. Finally, previous work [29, 37, 43, 45] have studied how one can extract high-quality answers from crowdsourcing platforms.

8. CONCLUSION

This paper presents *ACD*, a crowd-based algorithm for data deduplication. Compared with the existing solutions, *ACD* distinguishes itself in that it is more robust to errors made by the crowd in identifying duplicates. In addition, *ACD* incorporates several advanced techniques to improve efficiency without compromising deduplication accuracy. We experimentally evaluate *ACD* on the Amazon Mechanical Turk, and show that *ACD* outperforms the states of the art by providing high-precision deduplication with moderate crowdsourcing overheads. For future work, we plan to further improve the performance of *ACD*, by investigating techniques for adaptively assigning more crowd workers to more *difficult* record pairs in data deduplication.

9. ACKNOWLEDGMENTS

This work was supported by AcRF Tier 2 Grant ARC19/14 from the Ministry of Education, Singapore, an SUG Grant from the Nanyang Technological University, and a gift from the Microsoft Research Asia. The authors would like to thank the anonymous reviewers for their constructive and insightful comments.

10. REFERENCES

[1] http://www.cs.umass.edu/~mccallum/data/cora-refs.tar.gz.

[2] http://www.cs.utexas.edu/users/ml/riddle/data/restaurant.tar.gz.

[3] http://dbs.uni-leipzig.de/Abt-Buy.zip.

[4] https://sourceforge.net/p/acd2015/.

[5] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: ranking and clustering. *Journal of the ACM (JACM)*, 55(5):23, 2008.

[6] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 241–252. ACM, 2013.

[7] A. Arasu, M. Götz, and R. Kaushik. On active learning of record matching packages. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 783–794, 2010.

[8] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 952–963. IEEE, 2009.

[9] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. In *FOCS*, pages 238–, 2002.

[10] M. Charikar, V. Guruswami, and A. Wirth. Clustering with qualitative information. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 524–533. IEEE, 2003.

[11] Z. Chen, D. V. Kalashnikov, and S. Mehrotra. Exploiting context analysis for combining multiple entity resolution systems. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 207–218. ACM, 2009.

[12] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *ACM SIGMOD Record*, volume 27, pages 201–212. ACM, 1998.

[13] W. W. Cohen, H. Kautz, and D. McAllester. Hardening soft information sources. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 255–259. ACM, 2000.

[14] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 475–480. ACM, 2002.

[15] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *Proceedings of the 16th International Conference on Database Theory*, pages 225–236. ACM, 2013.

[16] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. Zencrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *WWW*, pages 469–478, 2012.

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

[18] J. Fan, M. Lu, B. C. Ooi, W.-C. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. Technical report, Technical report, National University of Singapore, 2013.

[19] A. Feng, M. Franklin, D. Kossmann, T. Kraska, S. Madden, S. Ramesh, A. Wang, and R. Xin. Crowddb: Query processing with the vldb crowd. *Proceedings of the VLDB Endowment*, 4(12), 2011.

[20] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 61–72. ACM, 2011.

[21] J. Gemmell, B. I. Rubinstein, and A. K. Chandra. Improving entity resolution with global constraints. *arXiv preprint arXiv:1108.6016*, 2011.

[22] A. Gionis, H. Mannila, and P. Tsaparas. Clustering aggregation. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):4, 2007.

[23] A. Goder and V. Filkov. Consensus clustering algorithms: Comparison and refinement. In *Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments, ALENEX 2008, San Francisco, California, USA, January 19, 2008*, 2008.

[24] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[25] R. Gomes, P. Welinder, A. Krause, and P. Perona. Crowdclustering. In *NIPS*, pages 558–566, 2011.

[26] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 385–396. ACM, 2012.

[27] O. Hassanzadeh, F. Chiang, H. C. Lee, and R. J. Miller. Framework for evaluating clustering algorithms in duplicate detection. *Proceedings of the VLDB Endowment*, 2(1):1282–1293, 2009.

[28] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *ACM SIGMOD Record*, volume 24, pages 127–138. ACM, 1995.

[29] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD workshop on human computation*, pages 64–67. ACM, 2010.

[30] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 847–860. ACM, 2008.

[31] S. R. Jeffery, L. Sun, M. DeLand, N. Pendar, R. Barber, and A. Galdi. Arnold: Declarative crowd-machine data integration. In *CIDR*, 2013.

[32] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[33] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. Cdas: a crowdsourcing data analytics system. *Proceedings of the VLDB Endowment*, 5(10):1040–1051, 2012.

[34] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[35] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *Proceedings of the VLDB Endowment*, 5(1):13–24, 2011.

[36] A. McCallum and B. Wellner. Conditional models of identity uncertainty with application to noun coreference. In *NIPS*, 2004.

[37] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: Algorithms for filtering data with humans. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 361–372. ACM, 2012.

[38] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1203–1212. ACM, 2012.

[39] L. Philips. Hanging on the metaphone. *Computer Language*, 7(12 (December)), 1990.

[40] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[41] W. M. Soon, H. T. Ng, and D. C. Y. Lim. A machine learning approach to coreference resolution of noun phrases. *Computational linguistics*, 27(4):521–544, 2001.

[42] A. Van Zuylen and D. P. Williamson. Deterministic pivoting algorithms for constrained ranking and clustering problems. *Mathematics of Operations Research*, 34(3):594–620, 2009.

[43] P. Venetis and H. Garcia-Molina. Quality control for comparison microtasks. In *Proceedings of the First International Workshop on Crowdsourcing and Data Mining*, pages 15–21. ACM, 2012.

[44] N. Vesdapunt, K. Bellare, and N. Dalvi. Crowdsourcing algorithms for entity resolution. *Proceedings of the VLDB Endowment*, 7(12), 2014.

[45] P. Wais, S. Lingamneni, D. Cook, J. Fennell, B. Goldenberg, D. Lubarov, D. Marin, and H. Simons. Towards building a high-quality workforce with mechanical turk. *Proceedings of computational social science and the wisdom of crowds (NIPS)*, pages 1–5, 2010.

[46] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5(11):1483–1494, 2012.

[47] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *Proceedings of the 2013 international conference on Management of data*, pages 229–240. ACM, 2013.

[48] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *Proceedings of the VLDB Endowment*, 6(6):349–360, 2013.

APPENDIX

A. PROOFS

Proof of Lemma 2. We prove this lemma by induction. Observe that in the first iteration, $G_1 = G$, and r_k is exactly the record who ranks k-th in $\mathcal{M}$. In *Crowd-Pivot* algorithm, assume that r_i is clustered into C_j where $j \leq i$ and r_j is the record first added into the cluster, then it can be concluded that if there exist incident edges between r_i and records with permutation ranks smaller than r_j, then the crowd infers them as non-deduplicates. Otherwise, it will be clustered into a cluster $C_{j'}$ with $\mathcal{M}(r_{j'}) < \mathcal{M}(r_j)$. With the constraints in our Algorithm 2 (Lines 7-11), r_i will be also clustered to C_j. So for all records with permutation rank no higher than r_k, *Crowd-Pivot* and *Partial-Pivot* will output the same cluster.

Then, assume that Lemma 2 still holds in the m-th iteration, and let r_w be the record with k-th smallest permutation rank in G_m, in the $(m+1)$-th iteration, G_{m+1} will be the same as the input graph of *Crowd-Pivot* after removing $r_1, r_2, \cdots, r_w$. Given the same input graph, and the same permutation order for the records, it can be similarly proved that *Crowd-Pivot* and *Partial-Pivot* will provide the same cluster result. So for all iterations, we guarantee that, given G_i, k, and $\mathcal{M}$, *Partial-Pivot* outputs $\mathcal{C}_i = \mathcal{C}'_i$. □

Proof of Lemma 3. Recall that a wasted pair is a pair of records that is issued in *Partial-Pivot* algorithm but not issued in *Crowd-Pivot* algorithm. The reason why we may issue wasted pairs in *Partial-Pivot* algorithm is that *Crowd-Pivot* algorithm removes records that have been clustered by $r_1, \cdots, r_{k-1}$ before issuing pairs incident to r_k, while *Partial-Pivot* algorithm will issue all the incident pairs to $r_1, r_2, \cdots, r_k$ in a batch.

If there exists an $x < j$, such that $d(r_j, r_x) = 1$, then r_j can be clustered by r_x. As a result, for all vertices r' incident to r_j, it can be a wasted pair. The exception is that if r' is a vertex in $\{r_1, r_2, \cdots, r_{j-1}\}$, then the pair is not a wasted pair for r_j. The reason is that, if pair (r', r_j) is not estimated as a wasted pair before, then it means that r' is the vertex with the smallest permutation rank that is incident to r_j and r' is not incident to pivots chosen earlier than it. So (r', r_j) will be issued in the *Crowd-Pivot* algorithm as well, which indicates that (r', r_j) is not a wasted pair. However, if (r', r_j) is estimated as a wasted pair before, then recounting this pair will overestimate the wasted pair, which will not guarantee tight bound.

If there does not exist such an $x < j$ such that $d(r_j, r_x) = 1$, then r_j will not be clustered by $r_1, r_2, \cdots, r_{j-1}$. As a result, for any records that are incident to r_j but not to any records in $r_1, r_2, \cdots, r_{j-1}$, then it is not a wasted pair, since in *Crowd-Pivot* algorithm, this record pair will also be issued. At the meantime, if it is incident to a record in $r_1, r_2, \cdots, r_{j-1}$, then it can be a wasted pair and we estimate it as a wasted pair.

By summing up all the wasted pairs incurred by $r_1, ..., r_k$, we get the upper bound of the total number of wasted pairs as $\sum_{j=1}^{k} w_j$. We next show that this upper bound is tight. If there exists an $x < j$, such that $d(r_j, r_x) = 1$, then if r_x and r_j are inferred as duplicates, then all the record pairs that incident to r_j become wasted pairs except the case in which the adjacent record is from $r_1, r_2, \cdots r_{j-1}$, which is exactly the value defined in Equation 3 with the first case. If there does not exist such an $x < j$ such that $d(r_j, r_x) = 1$, the worst case is that all the records adjacent to some records r_i with $i < j$ are clustered. Then all such pairs are wasted pairs, which is exactly the value defined in Equation 3 with the second case. Thus the total number of wasted pairs is exactly $\sum_{j=1}^{k} w_j$ in the worst case and the upper bound is tight. □

Proof of Lemma 4. We first prove that *PC-Pivot* returns a 5-approximation for the minimization of $\Lambda'(R)$ in expected case. Based on Lemma 2, it can be concluded that given a permutation $\mathcal{M}$, *PC-Pivot* and *Crowd-Pivot* return the same clustering result, which means both algorithms will have the same $\Lambda'(R)$ value. As a result, *Crowd-Pivot* and *PC-Pivot* have the same expected value of $\Lambda'(R)$ from all possible permutations, which means *PC-Pivot* returns a 5-approximation for the minimization of $\Lambda'(R)$ in expected case.

Next we prove that *PC-Pivot* has the following guarantee: among the record pairs crowdsourced by *PC-Pivot*, at most a fraction of ϵ is wasted. Given a random permutation $\mathcal{M}$, we let I_W^i be the number of issued wasted pairs in the i-th iteration of *PC-Pivot*, and I_{PCP}^i be the number of pairs issued by *PC-Pivot* in the i-th iteration. Then,

$$\epsilon \cdot I_W^i \leq I_{PCP}^i.$$

By summing the pairs in all iterations, we have $\epsilon \cdot I_W \leq I_{PCP}$ for the given permutation $\mathcal{M}$, where I_W and I_{PCP} denote the number of wasted pairs and issued pairs of *PC-Pivot* algorithm, respectively. By summing for all possible permutations, we have:

$$\epsilon \cdot E(I_W) \leq E(I_{PCP}).$$

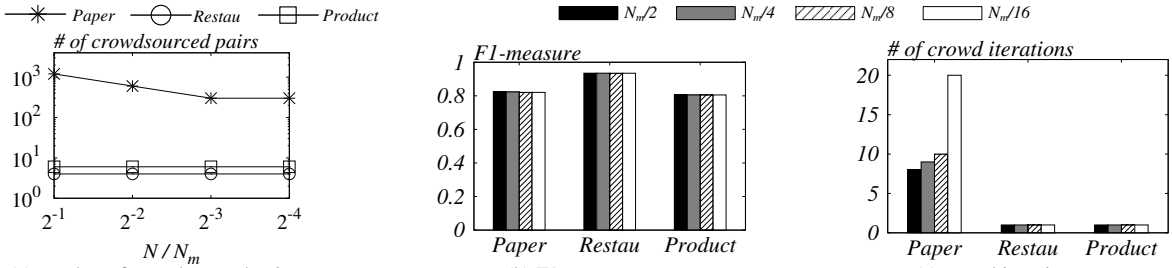
This finishes the proof. □

Proof of Lemma 5. We prove it by reduction from the maximal independent set problem [34]. A maximal independent set in an undirected graph is a maximal collection of vertices I subject to the restriction that no pair of vertices in I are adjacent. To reduce this problem to our problem of finding a set O^i of independent operations that maximizes $\Psi(O^i)$, we take each vertex of the graph as an operation. If there is an edge between two vertices, we regard the operation corresponding to the two vertices as a dependent operation. For each operation o corresponding to the vertex in the graph, we assume the $b(o)$, $c(o)$ are all the same and N is larger than the cost of all operations.

Now assume that we can obtain such a set O^i of the independent operation maximizing $\Psi(O^i)$ in polynomial time. Then by solving the above constructed problem, we can find a maximal independent set in polynomial time as well. This contradicts the fact that the maximal independent set problem is NP-hard. So by contradiction, we prove that finding a set O^i of independent operations that maximizes $\Psi(O^i)$ is NP-hard. □

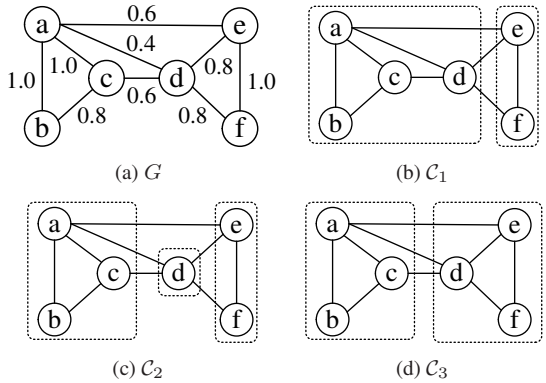
B. ADDITIONAL EXAMPLE

EXAMPLE 3. Given a set of six records $R = \{a, b, c, d, e, f\}$, assume that after the pruning phase, the candidate pairs are as shown in Figure 9a. The number on each edge (x, y) indicates the similarity score $f_c(x, y)$ that we would obtain if we issue (x, y) to the crowd. At the meantime, $b^*(o)$ is identical to $b(o)$. Then given a random permutation $P = \{c, e, b, d, a, f\}$ with $\epsilon = 0.4$, in cluster generation phase, *PC-Pivot* selects incident edges to c and e to the crowd in a batch, since the fraction of wasted pairs would not exceed 0.4. The issued pairs are $(a, c), (b, c), (c, d), (e, a), (e, d)$ and (e, f). With the crowd answers, two clusters $\{a, b, c, d\}$ and $\{e, f\}$



(a) number of crowdsourced pairs (b) F1-measure (c) crowd iterations

Figure 10: *PC-Refine*: impact of T on crowdsourced pairs, F1-measure, and crowd iterations on three datasets.



(a) G (b) $\mathcal{C}_1$ (c) $\mathcal{C}_2$ (d) $\mathcal{C}_3$

Figure 9: Example of cluster refinement phase.

are generated in one iteration. As the records are clustered after one iteration, *PC-Pivot* finishes. The clustering result after cluster generation phase is as shown in Figure 9b, where each dashed box corresponds to a cluster. Afterwards, in the postprocessing phase, *Crowd-Refine* algorithm first inspects each operation and calculates the benefit-cost ratio correspondingly. It can be verified that the operation o_s to split vertex d from cluster $\{a, b, c, d\}$ has the highest benefit-cost ratio. Then *Crowd-Refine* issues pair (a, d) to the crowd and obtains $f_c(a, d)$. As $f_c(a, d) = 0.4$, $f_c(b, d) = 0$, and $f_c(c, d) = 0.6$, $b(o_s)$ can be calculated with Equation 5 and equals 1. So o_s will be proceeded. After operation o_s proceeded, the clustering result is as shown in Figure 9c. Next, *Crowd-Refine* recomputes the benefit-cost ratio, and the merger operation o_m to merge $\{d\}$ and $\{e, f\}$ incurs the highest benefit-cost ratio. Only pair (d, f) is required to be issued to the crowd since (d, e) was issued in the cluster generation phase. After issuing pair (d, f) to the crowd, we obtain the crowd similarity $f_c(d, f) = 0.8$. The benefit of o_m can be calculated with Equation 6 and equals 1.2. The operation hence is proceeded. The clustering result is as shown in Figure 9d. After the two operations, $b^*(o)$ for all operations are no larger than zero. Hence *Crowd-Refine* stops and returns the clustering result. □

C. ADDITIONAL EXPERIMENTS

Recall that in the cluster refinement phase of *ACD*, *PC-Refine* algorithm requires a parameter T to specify the number of record pairs to issue in a crowd iteration. In this set of experiments, we study the impact of T to our *ACD* algorithm in terms of the crowdsourcing cost, accuracy, and efficiency.

Figure 10(a) demonstrates the impact of T in terms of the number of crowdsourced pairs. For the *Restaurant* and *Product* datasets, as the clustering result provided by the cluster generation phase is already of high quality, there are very few operations with expected benefit greater than zero to reduce $\Lambda'(R)$. Hence, the issued crowdsourced pairs can be less than the pre-defined threshold T in the first crowd-iteration. On *Paper* dataset, the number of crowdsourced pairs is reduced as T decreases. When T is small enough (e.g, $N_m/8$), it decreases smoothly or shows the same value. This is because for a large T, our algorithm packs many unnecessary operations with low benefit-cost ratio; in contrast, for a small enough T, it does not pack such operations further.

Figure 10(b) shows the impact of parameter T in terms of the accuracy to the deduplication result. Observe that the F1-measure value for each dataset is insensitive to T. The reason is that our algorithm stops when optimization goal $\Lambda'(R)$ cannot be further reduced, and this stopping condition is not significantly affected by the choice of T.

Figure 10(c) demonstrates the impact of T in terms of the number of crowd iterations. Notice that for the *Restaurant* and *Product* datasets, the result of the cluster generation phase is already of high quality. Therefore, the algorithm stops after one crowd iteration, regardless how we change N. On the other hand, on the *Paper* dataset, the number of crowd iterations increases with T. However, when T changes from $N_m/2$ to $N_m/8$, the increase of the number of crowd iterations is not significant. The main reason is that the number of crowd iterations is effected only by the postprocessing operations that can reduce $\Lambda'(R)$; when T is large, *PC-Refine* algorithm packs many operations with low benefit-cost ratio, which have low probability of reducing $\Lambda'(R)$. However, when we further decrease the size of T from $N_m/8$ to $N_m/16$, we can observe that the number of crowd iterations is doubled. The reason is that when T is small enough, the packed operations are mainly the operations that can reduce $\Lambda'(R)$. Therefore, the number of iterations in $N_m/16$ is 2 times more than that in $N_m/8$.

So in *ACD*, we set $T = N_m/8$, as it is the setting under which the *PC-Refine* algorithm provides good clustering accuracy while uses only a small number of crowdsourced pairs and crowd iterations.