Human-in-the-loop Data Integration

Guoliang Li
Department of Computer Science, Tsinghua University, Beijing, China
liguoliang@tsinghua.edu.cn

ABSTRACT

Data integration aims to integrate data in different sources and provide users with a unified view. However, data integration cannot be completely addressed by purely automated methods. We propose a hybrid human-machine data integration framework that harnesses human ability to address this problem, and apply it initially to the problem of entity matching. The framework first uses rule-based algorithms to identify possible matching pairs and then utilizes the crowd to refine these candidate pairs in order to compute actual matching pairs. In the first step, we propose similarity-based rules and knowledge-based rules to obtain some candidate matching pairs, and develop effective algorithms to learn these rules based on some given positive and negative examples. We build a distributed in-memory system DIMA to efficiently apply these rules. In the second step, we propose a selection-inference-refine framework that uses the crowd to verify the candidate pairs. We first select some "beneficial" tasks to ask the crowd and then use transitivity and partial order to infer the answers of unasked tasks based on the crowdsourcing results of the asked tasks. Next we refine the inferred answers with high uncertainty due to the disagreement from the crowd. We develop a crowd-powered database system CDB and deploy it on real crowdsourcing platforms. CDB allows users to utilize a SQL-like language for processing crowd-based queries. Lastly, we provide emerging challenges in human-in-the-loop data integration.

1. INTRODUCTION

In big data era, data are full of errors and inconsistencies and bring many difficulties in data analysis. As reported in a New York Times article, 80% time of a typical data science project is to clean and integrate the data, while the remaining 20% is actual data analysis[1]. Thus data science pipeline should include data acquisition, data extraction, data cleaning, data integration, data analytics and data visualization. Data integration is an important step to integrate data in different sources and provide users with a unified view. However, data integration cannot be completely addressed by purely automated methods [35]. Therefore, it is demanding to develop effective techniques and systems to serve the data integration problem.

We propose a hybrid human-machine data integration method that harnesses human ability to address the data integration problem [60, 69, 34], and apply it initially to the problem of entity matching, which, given two sets of records, aims to identify the pairs of records that refer to the same entity. For example, we want to find the products from Amazon and eBay that refer to the same entity. We also want to find the publications from DBLP and Google scholar that refer to the same entity. Our framework first uses rule-based algorithms to identify possible matching pairs as candidates and then utilizes the crowd (also called workers) to refine the candidate pairs so as to identify actual matching pairs.

The first step aims to design effective algorithms to efficiently identify candidate matching pairs. We propose rule-based methods to obtain the candidate matching pairs, where a pair of records is a candidate if the two records satisfy a rule. For example, consider the product integration problem. A possible rule is if two products have similar name and similar model, then they may refer to the same entity. There are three challenges in this step. Firstly, it is challenging to generate high-quality rules. Existing techniques use string similarity functions (e.g., edit distance, Jaccard) to define the similarity. For example, two product names are similar if their edit distance is smaller than a given threshold (e.g., 2). However, these functions neglect the knowledge behind the data. We propose to use knowledge bases, e.g., Freebase and Yago, to quantify the knowledge-aware similarity [57, 23, 56], and our rule can support both string similarity and knowledge-aware similarity. We develop effective algorithms to learn these rules based on some given positive and negative examples. Secondly, it is rather inefficient to apply these rules, because it is prohibitively expensive to enumerate every pair of records. To address this problem, we propose a signature-based method such that if two records refer to the same entity, they must share a common signature. We utilize the signatures to generate the candidates, develop efficient techniques to select high-quality signatures, and propose an algorithm to minimize the number of selected signatures. Thirdly, it is challenging to build a distributed system to support rule-based entity matching. We build a distributed in-memory system DIMA to learn and apply these rules and seamlessly integrate our

[1] http://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

techniques into Spark SQL [60]. To balance the workload, we propose selectable signatures, which can be adaptively selected based on the workload. Based on selectable signatures, we build global and local indexes, and devise efficient algorithms and cost-based query optimization techniques. Moreover, besides entity matching, our system can support most of similarity-based operations, e.g., similarity selection, similarity join, and top-k similarity selection and join.

The second step verifies the candidates by utilizing the crowd to identify the actual matching pairs. The big challenge of using the crowd to address the data integration problem is to save monetary cost, improve the quality and reduce the latency. We propose a selection-inference-refine framework to balance the cost, latency and quality. We first select some "beneficial" tasks to ask the crowd, and then infer the answers of unasked tasks based on the crowd results of the asked tasks. Next we refine the inferred answers with high uncertainty due to the disagreement from different workers. We propose to use the transitivity and partial order to reduce the cost [68]. For example, given three records r, s, t. If we get the answers from the crowd that, $r = s$ (refer to the same entity) and $s = t$, then we can infer $r = t$ and thus the pair (r, t) does not need to be asked. We iteratively select pairs without answers to ask and use the transitivity to infer the answers of unasked pairs, until we get the answers of all pairs. We devise effective algorithms to judiciously select pairs to ask in order to minimize the number of asked pairs. To reduce the latency, we devise a parallel algorithm to ask the tasks in parallel. To control the quality, we propose an online task assignment and truth inference framework to improve the quality. The truth inference module builds task model and worker model, and infers the truth of each task based on workers' answers. The adaptive task-assignment module on-the-fly estimates accuracies of a worker by evaluating her performance on the completed tasks, and assigns the worker with the tasks she is well acquainted with. To further reduce the monetary cost and improve the quality, we propose to use partial order to do inference [7]. We develop a crowdsourcing database system `CDB` [31], which provides declarative programming interfaces and allows users to utilize a SQL-like language for posing queries that involve crowdsourced operations, and encapsulates the complexities of interacting with the crowd. `CDB` has fundamental advantages compared with existing crowdsourcing database systems. Firstly, in order to optimize a query, existing systems often adopt the traditional query-tree model to select an optimized table-level join order. However, the query-tree model provides a coarse-grained optimization, which generates the same order for different joined records and limits the optimization potential that different joined records should be optimized by different orders. `CDB` employs a graph-based query model that provides more fine-grained query optimization. Secondly, existing systems focus on optimizing the monetary cost. `CDB` adopts a unified framework to perform the multi-goal optimization based on the graph model to optimize cost, quality and latency. We have deployed `CDB` on real crowdsourcing platforms. Our systems and source codes are available at `https://github.com/TsinghuaDatabaseGroup`.

Finally, we provide emerging challenges in human-in-the-loop data integration, including entity matching on unstructured data, data integration on complex data (e.g., graphs and spatial data), and fine-grained human involvement.

2. HYBRID HUMAN-MACHINE ENTITY MATCHING FRAMEWORK

2.1 Problem Definition

Consider two relations R and S. R has x attributes $a_1, a_2, \cdots, a_x$ and m records $r_1, r_2, \cdots, r_m$. S has y attributes $b_1, b_2, \cdots, b_y$ and n records $s_1, s_2, \cdots, s_n$. The entity matching problem aims to find all record pairs $(r_i, s_j) \in R \times S$ such that r_i and s_j refer to the same entity.

There are two main challenges in entity matching. First, it is not easy to evaluate whether two records refer to the same entity. Second, it is expensive to check every record pair. To address these two challenges, we propose a hybrid human-machine framework. We first use the rules to generate the candidate pairs and then utilize the crowd to verify the candidate pairs. The first step aims to devise algorithms to automatically prune a large number of dissimilar pairs. As the first step may generate false positives, the second step focuses on utilizing the crowd to refine the candidates in order to remove the false positives.

2.2 Rule Definition

Consider attribute a in R and attribute b in S, and record $r \in R$ and record $s \in S$. Let $r[a]$(resp. $s[b]$) denote the cell value of r(resp. s) on attribute a (resp. b). We use similarity functions to quantify the similarity of $r[a]$ and $s[b]$.

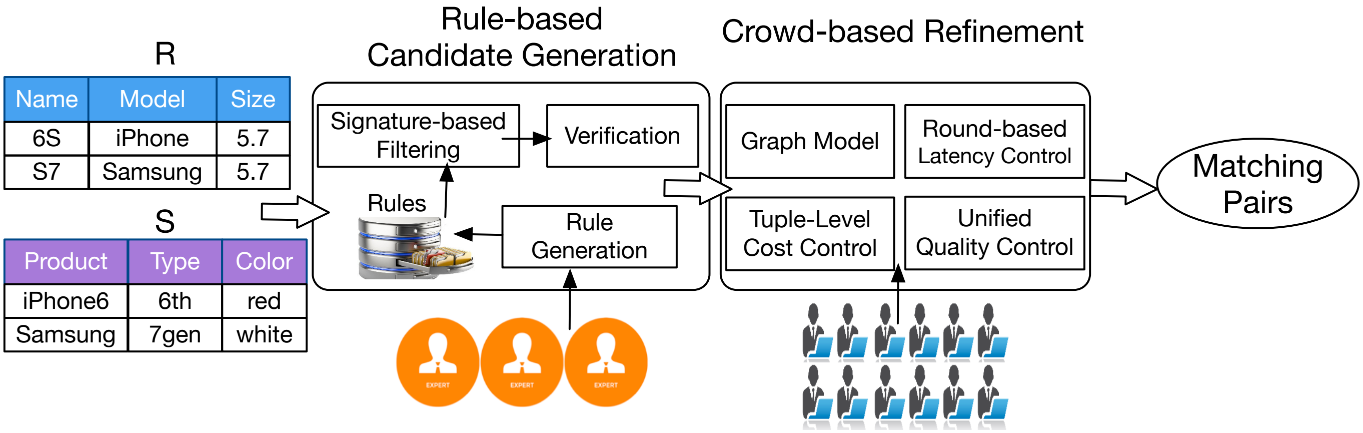
Set-Based Similarity. It tokenizes records as sets of tokens (or q-grams[2]) and computes the similarity based on the sets, e.g., Overlap, Jaccard, Cosine, DICE. We take Jaccard as an example, and our method can be easily extended to support other set-based similarities. The Jaccard similarity between r and s is $\texttt{Jac}(r, s) = \frac{|r \cap s|}{|r \cup s|}$, where $r \cap s$ and $r \cup s$ are the overlap and union of r and s respectively by tokenizing r and s as two sets of tokens. Two values are similar w.r.t. Jaccard if their Jaccard similarity is not smaller than a threshold τ. For example, the Jaccard similarity between {`VLDB, 2017`} and {`VLDB, 2016`} is 1/3.

Character-Based Similarity. It transforms a record to the other based on character transformations and computes the similarity/distance[3] by the number of character transformations. The well-known character-based similarity function is edit distance, which transforms a record to the other by three atomic operations, deletion, insertion and substitution, and takes the minimum number of edit operations as the edit distance. Two values are similar w.r.t. edit distance if their edit distance is not larger than a threshold τ. For example, the edit distance between `SIGMOD` and `SIGMD` is $\texttt{ED}(\texttt{SIGMOD}, \texttt{SIGMD}) = 1$.

Knowledge-Based Similarity. A knowledge hierarchy can be modeled as a tree structure and how to support the directed acyclic graph (DAG) structure can be found in [56]. Given two values $r[a]$, $s[b]$, we first find the tree nodes that exactly match the two values. Here we assume that each value matches a single tree node and how to support the case that each value matches multiple tree nodes is discussed in [56]. If the context is clear, we also use $r[a]$ and $s[b]$ to denote the corresponding matched nodes. Let $\texttt{LCA}_{r[a],s[b]}$ denote their lowest common ancestor (i.e., the

[2] A substring with length q is called a q-gram.

[3] Note that similarity and distance can be transformed to each other.



Name	Model	Size
6S	iPhone	5.7
S7	Samsung	5.7

Product	Type	Color
iPhone6	6th	red
Samsung	7gen	white

Figure 1: Hybrid Human-Machine Framework.

common ancestor of the two nodes and any of its descendant will not be a common ancestor of the two nodes), and $d_{r[a]}$ denote the depth of node $r[a]$ (the depth of the root is 0). Intuitively, the larger $d_{r[a],s[b]} = d_{\texttt{LCA}_{r[a],s[b]}}$ is, the more similar two values are. Thus the knowledge-aware similarity between $r[a]$ and $s[b]$ is defined as $\texttt{KS}(r[a], s[b]) = \frac{2d_{r[a],s[b]}}{d_{r[a]}+d_{s[b]}}$.

Based on the similarity functions, we define *attribute-level matching rules.*

Definition 1 (Attribute-level Matching Rule). *An attribute-level matching rule is a quadruple $\lambda(a, b, f, \tau)$, where a, b are attributes, f is a similarity function, and τ is a threshold. $r[a]$ and $s[b]$ are considered to be the same, if $f(r[a], s[b]) \geq \tau$ for similarity functions (or $f(r[a], s[b]) \leq \tau$ for distance functions).*

An attribute-level matching rule includes two attributes, a similarity function and a threshold. Two cell values are similar if they satisfy an attribute-level matching rule. For example, in Figure 1, $(\texttt{Model}, \texttt{Product}, \texttt{ED}, 2)$ is an attribute-level rule. Note that two records refer to the same entity if they have similar cell values on multiple attributes. Thus we define *record-level matching rules.*

Definition 2 (Record-level Matching Rule). *A record-level matching rule is a set of attribute-level matching rules, denoted by $\psi = \bigwedge_{i=1}^{z} \lambda_i$. A record pair (r, s) satisfies the record-matching rule ψ, if (r, s) satisfies every attribute-level matching rule $\lambda_i \in \psi$.*

For example, in Figure 1, $(\texttt{Model}, \texttt{Product}, \texttt{ED}, 2) \wedge (\texttt{Name}, \texttt{Type}, \texttt{ED}, 2)$ is an attribute-level matching rule. We can use record-level matching rules to deduce whether two records refer to the same entity. However, in most cases users do not know how to select the attributes, how to choose the similarly functions, and how to determine the appropriate thresholds. Thus we need to automatically learn the rules and we will discuss the rule generation in Section 3.

2.3 Hybrid Human-Machine Framework

Our hybrid human-machine framework contains the following two steps.

Rule-based Candidate Generation. Given a set of rules Ψ, it identifies the record pairs that satisfy a rule $\psi \in \Psi$. A straightforward method enumerates every record pair and every rule, and checks whether the record pair satisfies a rule. If yes, we take the pair as a candidate; otherwise we prune the pair. Obviously this method is rather expensive as it requires to enumerate every record pair. For example, if there are 1 million records, it is prohibitively expensive to enumerate every pair. To address this issue, we propose an effective signature-based method in Section 3.

Crowd-based Refinement. Given a set of candidates, it aims to identify the candidate pair that actually refers to the same entity by utilizing the crowd. A straightforward method enumerates every candidate pair (r, s) and asks the crowd to check whether r and s refer to the same entity. If yes, we take the pair as an answer; otherwise we prune the pair. Obviously it incurs high monetary cost if we ask every candidate pair. We propose effective techniques to reduce the cost in Section 4.

3. RULE-BASED CANDIDATE GENERATION

We first discuss how to generate the rules and then propose a signature-based algorithm to efficiently apply these rules. Lastly, we introduce our system `DIMA`.

3.1 Rule Generation

Give a set of examples, denoted by E, including positive examples, e.g., which records are known to be the same entity, denoted by M, and negative examples, e.g., which records are known to be not the same entity, denoted by N, we use these given examples to generate the rules. Obviously $M \cup N = E$ and $M \cap N = \emptyset$. The examples can be provided by experts or the crowd (also called workers).

Next we discuss how to evaluate the quality of a rule set Ψ. Given a rule $\psi \in \Psi$, let $\hat{M}_\psi$ denote the set of record pairs that satisfy ψ. Let $\hat{M}_\Psi = \bigcup_{\psi \in \Psi} \hat{M}_\psi$ be the set of record pairs that satisfy Ψ. Ideally, we hope that $\hat{M}_\Psi$ is exactly equal to M. However, in reality $\hat{M}_\Psi$ may contain negative examples. To evaluate the quality of Ψ, we focus on a general case of objective functions $\mathcal{F}(\Psi, M, N)$: the larger $|\hat{M}_\Psi \cap M|$, the larger $\mathcal{F}(\Psi, M, N)$; the smaller $|\hat{M}_\Psi \cap N|$, the larger $\mathcal{F}(\Psi, M, N)$. Many functions belong to this general class. For example, the well-know F-measure function in information retrieval is a general objective function $\frac{2}{\frac{1}{\mathsf{p}}+\frac{1}{\mathsf{r}}}$, where $\mathsf{p} = \frac{|\hat{M}_\Psi \cap M|}{|\hat{M}_\Psi \cap M| + |\hat{M}_\Psi \cap N|}$ is the precision, and $\mathsf{r} = \frac{|\hat{M}_\Psi \cap M|}{|M|}$ is the recall. Users can tune the weights of precision or recall to select their preferences: preferring to recall or precision.

Now we can formalize the rule generation problem.

Definition 3 (Rule Generation). *Given two relations R, S, a set of positive examples M, a set of negative examples N, and a set of similarity functions F, find a set of rules Ψ to maximize a pre-defined objective function $\mathcal{F}(\Psi, M, N)$.*

Note that the rule generation problem is NP-hard, which can be proved by a reduction from the maximum-coverage problem [69]. A brute-force method enumerates all possible record-level matching rule sets and selects the set that maximizes $\mathcal{F}(\Psi, M, N)$. Note that each record-level rule contains multiple attribute-level matching rules, and each attribute-level rule contains two attributes, a similarity function and a threshold. We can enumerate the attributes and similarity functions, but we cannot enumerate the threshold as the threshold is infinite. Fortunately, we do not need to consider all possible thresholds, and instead we only need to consider a limited number of thresholds, as discussed below.

From Infinite Thresholds to Finite Thresholds. Consider two attribute-level rules $\lambda_1(a, b, f, \tau_1)$ and $\lambda_2(a, b, f, \tau_2)$. Without loss of generality, suppose $\tau_1 < \tau_2$. Obviously the set of examples that satisfy λ_2 is a subset of that of λ_1, that is, $\hat{M}_{\lambda_2} \subseteq \hat{M}_{\lambda_1}$ (where $\hat{M}_{\lambda}$ denotes the record pairs that satisfy λ). If there is no positive example in $\hat{M}_{\lambda_1} - \hat{M}_{\lambda_2}$, for any record-level matching rules that contain λ_1, we will not use the rule λ_1 since they cannot get a better objective value than the rule λ_2. In other words, for $\lambda(a, b, f, \tau)$, we can only consider the threshold in $f(r[a], s[b])$ for a positive example $(r, s) \in M$ [69]. Thus we can only consider a limited number of thresholds. Note that different similarity functions and thresholds have redundancy. For example, a threshold τ_1 has no higher objective function value than another threshold $\tau_2 > \tau_1$, we can only keep τ_2. We devise efficient techniques to eliminate the redundancy.

Greedy Algorithm. Based on the similarity functions and a finite set of thresholds, we propose a greedy algorithm to identify the rules. We first evaluate the *quality* (the objective function value) of each possible rule based on the example set E, and then pick the rule with the highest quality. Next we update the example set E by removing the examples that satisfy the selected rule. The algorithm terminates when there is no example in E.

3.2 Signature-Based Method for Applying Rules

3.2.1 Algorithm Overview

Indexing. Consider a set of rules Ψ. Given a rule $\psi \in \Psi$, for each attribute-level rule $\lambda = (a, b, f, \tau) \in \psi$, we build an index $\mathcal{L}^{\lambda}$ for table R. For each record r in R, we generate a set of indexing signature $\texttt{iSig}^{\lambda}(r)$ for values $r[a]$. For each indexing signature $g \in \texttt{iSig}^{\lambda}(r)$, we build an inverted list $\mathcal{L}^{\lambda}(g)$, which keeps a list of records whose signature set w.r.t. λ contains g.

Filtering. Given a rule ψ, for each record $s \in S$, we generate a set of probing signature $\texttt{pSig}^{\lambda}(s)$ w.r.t. $\lambda \in \psi$ and $s[b]$. For each probing signature $g \in \texttt{pSig}^{\lambda}(s)$, the records on inverted list $\mathcal{L}^{\lambda}(g)$ are candidates of s, denoted by $\mathcal{C}^{\lambda}(s)$. If a records r is similar to s, they must satisfy every attribute-level matching rule $\lambda \in \psi$, thus we can get a candidate set $\mathcal{C}^{\psi}(s) = \cap_{\lambda \in \psi} \mathcal{C}^{\lambda}(s)$.

Verification. For each candidate pair (r, s), we verify whether they satisfy a rule ψ as follows. Consider a rule $\lambda = (a, b, f, \tau) \in \psi$, we compute $f(r[a], s[b])$. If $(r[a], s[b])$ is not similar w.r.t. function f and threshold τ, r and s are pruned.

3.2.2 Signature-based Filtering

We discuss how to generate signatures for different similarity functions. The basic idea is that we partition two values into disjoint segments, such that if two values are similar, they must share a common segment based on the pigeonhole principle. Formally, given an attribute-level rule $\lambda = (a, b, f, \tau)$, next we discuss how to generate signatures for a given similarity function f.

f is Overlap. Two values $v = r[a]$ and $v' = s[b]$ are similar, if they share at least τ tokens. We partition v and v' into $\theta = \max(|v|, |v'|) - \tau + 1$ disjoint segments based on a same partition strategy: if a token is partitioned into the i-th segment, it will be assigned into this segment for all the data. The partition with this property is called *a homomorphism partition*. To get a homomorphism partition, we can use a hash based method. For each token $e \in v$, we put it to the $\big((hash(e) \mod \theta) + 1\big)$-th segment where $hash(e)$ maps a token e to an integer. Thus any token will be partitioned into the same segment. However, this method may introduce a skewed problem: some segments have many tokens while some segments have few tokens [15]. To address this issue, we can use the token frequency to get a good partition [15].

Then based on the pigeonhole principle, if v and v' are similar, they must share a same segment. Thus the indexing signature set of v is the set of segments of v. The probing signature set of v' is also the set of segments of v'.

f is Edit Distance. Two values $v = r[a]$ and $v' = s[b]$ are similar if their edit distance is not larger than τ. Given a value $v = r[a]$, we partition it into $\tau + 1$ disjoint segments, and the length of each segment is not smaller than one[4]. For example, consider v=“`sigmod`”. Suppose $\tau = 2$. We have multiple ways to partition v into $\tau + 1 = 3$ segments, such as {“`si`”,“`gm`”,“`od`”}. To achieve high pruning power, each segment should have nearly the same length. To achieve this goal, the first $|v|\%(\tau + 1)$ segments have length of $\lceil \frac{|v|}{\tau+1} \rceil$, and other segments have length of $\lfloor \frac{|v|}{\tau+1} \rfloor$.

Consider another value $v' = s[b]$. If v' has no substring that matches a segment of v, v' cannot be similar to v based on the pigeonhole principle. In other words, if v' is similar to v, v' must contain a substring matching a segment of v. For example, consider v'=“`sgmd`”. It has a substring `gm` matching a segment of v. Thus they could be similar.

The indexing signature set of v is the set of segments of v. To find the similar values of v', we need to consider the values with length between $|v'| - \tau$ and $|v'| + \tau$. Thus for each length $l \in [|v'| - \tau, |v'| + \tau]$, we generate the probing signature set of v', which is the set of substrings of v' with length of $\lceil \frac{l}{\tau+1} \rceil$ or $\lfloor \frac{l}{\tau+1} \rfloor$, and use each probing signature to compute candidates based on the index. We propose effective techniques to minimize the number of probing signatures [33].

f is Jaccard. Two values $v = r[a]$ and $v' = s[b]$ are similar if their Jaccard similarity is not smaller than τ. We partition each value to several disjoint segments such that two values are similar only if they share a common segment. To achieve this goal, we need to decide (1) the number of partitions and (2) how to partition the tokens into different segments.

[4] The length of string $v(|v|)$ should be larger than τ, i.e., $|v| \geq \tau + 1$.

If $\texttt{Jac}(v, v') \geq \tau$, we have $1 - \frac{|v \cap v'|}{|v \cup v'|} \leq 1 - \tau$, $|v \cup v' - v \cap v'| \leq (1-\tau)|v \cup v'| \leq \frac{1-\tau}{\tau}|v \cap v'| \leq \frac{1-\tau}{\tau}|v|$. Thus if v' is similar to v, they should have no more than $\frac{1-\tau}{\tau}|v|$ mismatch tokens. Let $\eta_\tau(v) = \lfloor \frac{1-\tau}{\tau}|v| \rfloor + 1$. We partition v into $\eta_\tau(v)$ disjoint segments. We also partition v' into $\eta_\tau(v)$ disjoint segments. If they have no common segments, then they have at least $\eta_\tau(v)$ mismatch tokens and thus they cannot be similar. Note that we also need to use a homomorphism partition strategy to generate the segments.

The indexing signature set of v is the set of segments of v. To find the similar values of v', we need to consider the values with length between $|v'|\tau$ and $\frac{|v'|}{\tau}$. For each length $l \in [|v'|\tau, \frac{|v'|}{\tau}]$, we generate the probing signature set of v', which is the set of segments of v' by length l, and use the probing signatures to find candidates using the index.

f is Knolwedge-Aware Similarity. We can generate the signatures similar to Jaccard and more details are referred to [56]. Given a value v (v'), we can generate a set s_v (s'_v) of nodes from the root to the corresponding tree node on the knowledge hierarchy. Then if the knowledge similarity between v and v' is not smaller than τ, we have $\frac{2s_v \cap s'_v}{s_v \cup s'_v + s_v \cap s'_v} \geq \texttt{KS}(r[a], s[b]) = \frac{2d_{r[a],s[b]}}{d_{r[a]} + d_{s[b]}} \geq \tau$. If $\texttt{Jac}(s_v, s'_v) = \frac{s_v \cap s'_v}{s_v \cup s'_v} < \frac{\tau}{2}$, we can prune this pair. Thus we can transform the knowledge similarity to Jaccard and use the techniques of Jaccard to generate the signatures.

Based on the signatures and index, we can efficiently identify the candidates and do not need to enumerate every pair.

3.2.3 Verification Techniques

For overlap, Jaccard and knowledge-aware similarity, it takes $\mathcal{O}(|v| + |v'|)$ to verify a candidate pair for an attribute-level matching rule. For edit distance it is $\mathcal{O}(\tau \min(|v|, |v'|))$. For a record-level matching rule, it may contain multiple attributes, and we need to decide a good verification order, because if a pair is pruned based on an attribute, then other attributes do not need to be verified. Thus we compute the verification cost and pruning power for each attribute-level matching rule (i.e., the probability of non-matching on this attribute such that we do not need to check other attributes). Then we can get the benefit of verifying an attribute-level matching rule, which is the ratio of the pruning power to the cost. Next given a record-level matching rule, we verify its attribute-level rules following this order.

3.3 DIMA System

We build a distributed in-memory system DIMA to efficiently apply the rules [5].

3.3.1 Overview

Extended SQL. We extend SQL and define simSQL by adding an operation to support entity matching.

Entity Matching (Similarity Join). Users utilize the following simSQL query to find the records in tables T_1 and T_2 where T_1's column S is similar to T_2's column R w.r.t. a similarity function f and threshold τ.

SELECT * FROM T_1 SIMJOIN T_2 ON $f(T_1.S, T_2.R) \geq \tau$

Besides this operation, simSQL also supports similarity selection and top-k similarity selection and join.

Similarity Selection. Users utilize the following simSQL query to find records in table T whose column S is similar to query q w.r.t. a similarity function f and threshold τ.

SELECT * FROM T WHERE $f(T.S, q) \geq \tau$

Top-k Similarity Selection. Users utilize the following simSQL query to find k records in table T whose column S has the largest similarity to query q w.r.t. a similarity function f and integer k.

SELECT * FROM T WHERE KNN$(f, T.S, q, k)$

Top-k Similarity Join. Users utilize the following simSQL query to find k records in tables T_1 and T_2 with the largest similarity on table T_1's column S and table T_2's column R w.r.t. a similarity function f and integer k.

SELECT * FROM T_1 SIMJOIN T_2 ON KNN$(f, T_1.S, T_2.R, k)$

DataFrame. In addition to simSQL, users can also perform these operations over DataFrame objects using a domain-specific language similar to data frames in R. We also extend Spark's DataFrame API to support the above operations similar to the aforementioned extended simSQL.

Index. Users can also create index to support these operations. Our system supports both global indexes and local indexes (the details will be discussed later). Users can utilize the following simSQL query to create indexes on the column S of table T using our segment-based indexing scheme.

CREATE Index ON T.S USE SEGINDEX.

Query Processing. We utilize the above signature-based index to process the similarity queries. For selection, we utilize the global index to prune irrelevant partitions and send the query request to relevant partitions. In each local partition, we utilize the local index to compute the local results. For join query, we utilize the global index to make the similar pairs in the same partition and this can avoid expensive data transmission. In each partition, we utilize local index to compute local answers. For top-k, we progressively generate the signatures and use the signatures to compute top-k pairs, avoiding to generate all the signatures.

Query Optimization. A SQL query may contain multiple operations, and it is important to estimate the cost of each operation and thus the query engine can utilize the cost to select a query plan, e.g., join order. Since Spark SQL has the cost model for exact selection and join, we focus on estimating the cost for similarity join and search. If there are multiple join predicates, we also need to estimate the result size. In our system, we adopt cost-based model to optimize a SQL query. Dima extends the Catalyst optimizer of Spark SQL and introduces a cost-based optimization (CBO) module to optimize the approximation queries. The CBO module leverages the (global and local) index to optimize complex simSQL queries with multiple operations.

Dima Workflow. Next we give the query processing workflow of Dima. Given a simSQL query or a DataFrame object, Dima first constructs a tree model by the simSQL parser or a DataFrame object by the DataFrame API. Then Dima builds a logical plan using Catalyst rules. Next, the logical optimizer applies standard rule-based optimization to optimize the logical plan. Based on the logical plan, Dima applies cost-based optimizations based on signature-based indexes and statistics to generate the most efficient physical plan. Dima supports analytical jobs on various data sources such as CVS, JSON and Parquet.

[5] https://github.com/TsinghuaDatabaseGroup/DIMA

3.3.2 *Indexing*

To improve the performance in distributed environments, we propose a selectable signature that provides multiple signature options and we can judiciously select the signatures to balance the workload.

(1) Selectable Signatures

Basic Idea. Given a value $v = r[a]$ and a value $v' = s[b]$, we generate two types of indexing signature set for v, $\texttt{iSig}^+$ and $\texttt{iSig}^-$, and two types of probing signature set for v', $\texttt{pSig}^+$ and $\texttt{pSig}^-$, where $\texttt{iSig}^+$ and $\texttt{pSig}^+$ are segment and probing signatures as discussed in Section 3.2. $\texttt{iSig}^-$ (resp. $\texttt{pSig}^-$) is generated from $\texttt{iSig}^+$ (resp. $\texttt{pSig}^+$) by deleting a token for set-based similarity (or a character for character-based similarity). If $\texttt{iSig}^+ \cap \texttt{pSig}^+ = \emptyset$, we can deduce that v and v' have at least 1 mismatched token (or character). If $\texttt{iSig}^+ \cap \texttt{pSig}^+ = \emptyset$ & $\texttt{iSig}^+ \cap \texttt{pSig}^- = \emptyset$ & $\texttt{iSig}^- \cap \texttt{pSig}^+ = \emptyset$, we can infer that v and v' have at least 2 mismatched tokens (or characters). We can utilize either of the two properties to do pruning and thus we can select a better way to reduce the cost. Moreover, in distributed computing we can select a better way to balance the workload. The details on how to generate the signatures are referred to [14, 33].

Signature Selection. We have two options in selecting the probing signatures.

(1) Selecting the segment signature $\texttt{iSig}^+$ and $\texttt{pSig}^+$. If there is no matching segment signature in the i-th segment, v and v' have at least 1 mismatched token on the i-th segment.

(2) Selecting the deletion signature $\texttt{iSig}^-$ and $\texttt{pSig}^-$. If there is no matching segment signature and no matching deletion signature in the i-th segment, v and v' have at least 2 mismatched tokens on the i-th segment.

Suppose we select p probing segment signatures and q probing deletion signatures of v' such that $p + 2q \geq \theta_{|v|,|v'|}$, where $\theta_{|v|,|v'|} = \tau{+}1$ for edit distance and $\theta_{|v|,|v'|} = \lfloor \frac{1-\tau}{\tau}|v| \rfloor +$ 1 for Jaccard. If there is no matching on the selected signatures, v and v' have at least $\theta_{|v|,|v'|}$ mismatched tokens, then v and v' cannot be similar. Based on this property, we can select different signatures (segment or deletion signatures) for similarity operations to balance the workload [14, 33].

(2) Distributed Indexing

Given a table R, we build a global index and a local index offline. Then given an online query, we select the probing signatures, utilize the global index to locate the partitions that contain the query's probing signatures, and send the probing signature to such partitions. The executor that monitors such partitions does a local search to compute the local results.

Offline Indexing. Note that different queries may have different thresholds and we require to support queries with any choice of threshold. To achieve this goal, we utilize a threshold bound to generate the index. For example, the threshold bound for Jaccard is the smallest threshold for all queries that the system can support, e.g., 0.6. Using this threshold bound, we can select the indexing segment/deletion signatures and build a *local index*. In addition, we also keep the *frequency table* of each signature to keep each signature's frequency and build a *global index* that keeps a mapping from the signature to partitions that contain this signature.

Local Index. Next we shuffle the indexing signatures such that (1) each signature and its inverted list of records that contain this signature are shuffled to one and only one partition, i.e., the same signature will be in the same partition and (2) the same partition may contain multiple signatures and their corresponding records. For each partition, we construct an IndexRDD $\mathcal{I}_i^{\mathcal{R}}$ for indexing signatures in this partition. Each IndexRDD $\mathcal{I}_i^{\mathcal{R}}$ contains several signatures and the corresponding records, which includes two parts. The first part is a hash-map which keeps the mapping from a signature g to two lists of records: $\mathcal{L}^+[g]$ keeps the records whose indexing segment signatures contain g and $\mathcal{L}^-[g]$ keeps the records whose indexing deletion signatures contain g. The second part is all the records in this RDD, i.e., $\mathcal{D}_i = \cup_{g \in \mathcal{I}_i^{\mathcal{R}}} \mathcal{L}[g]$. Note that the records are stored in the data list $\mathcal{D}_i$ and $\mathcal{L}^+[g]$ and $\mathcal{L}^-[g]$ only keep a list of pointers to the data list $\mathcal{D}_i$.

Global Index. For each signature, we keep the mapping from the signature to the partitions that contain this signature. We maintain a global function $\mathcal{P}$ that maps a signature g to a partition p, i.e., $\mathcal{P}(g) = p$. Note that the global index is rather small and can be stored in each partition.

3.3.3 *Query Processing*

Similarity Selection. Given an online query, Dima utilizes the proposed indexes to support similarity search operation in three steps. (1) It first conducts a global search by utilizing the frequency tables to select the probing signatures. Specifically, we propose an optimal signature selection method to achieve a balance-aware selection by using a dynamic-programing algorithm. (2) For each selected probing signature, it utilizes the global hash function to compute the partition that contains the signature and sends the search request to the corresponding partition. (3) Each partition exploits a local search to retrieve the inverted lists of probing signatures and verify the records on the inverted lists to get local answers. Finally, it returns local answers.

Similarity Join. To join two tables R, S, a straightforward approach is to first build the index for R, then take each record in S as a query and invoke the search algorithm to compute its results. However, it is rather expensive for the driver, because it is costly to select signatures for huge number of queries. To address this issue, we propose an algorithm for similarity join consisting of four steps. (1) It generates signatures and builds the segment index for one table. (2) Then it selects probing signatures for each length l. Since it is expensive to utilize the dynamic-programming algorithm to compute the optimal signatures, we propose a greedy algorithm to efficiently compute the high-quality signatures. (3) For each selected signature it builds the probe index for the other table. Since the matched probing and indexing signatures are in the same executor, it can avoid data transmission among different partitions. (4) It computes the local join results in each executor based on the segment index and probe index, and the master collects the results from different local executors.

Top-k Selection. For similarity selection, we use a given threshold to generate segments and deletions as signatures. Top-k selection, however, has no threshold. To address this problem, we propose a progressive method to compute top-k results. We first generate a signature, use the signature to generate some candidates and put k best candidates in a priority queue. We use τ_k to denote the minimal similarity among the candidates in the priority queue. Then we estimate an upper bound ub for other unseen records. If

$\tau_k \geq$ ub, we can guarantee that the candidates in the queue are the final results and the algorithm can terminate. If $\tau_k <$ ub, we generate next signatures, compute candidates and update the priority queue and τ_k. We aim to first identify the candidates with the largest similarity and add them into the queue in order to get a larger τ_k. To achieve this goal, we first split value v into two segments, and take the first one as the first signature. Then we recursively split the second segment of v into two sub-segments and take the first sub-segment as the second signature. Given a query v', we use the same method to generate its first signature g_1. We utilize the global index to get the relevant partitions. For each relevant partition, we use the local inverted index to get candidates $\mathcal{L}[g_1]$ and send top-k local candidates to the server. The server collects all the local candidates, puts them into the priority queue and computes τ_k. Based on the first segment, we can also estimate an upper bound of the similarities of other records to the query. Since other records do not share the same first signature with v', they have at least one mismatch token with v'. Based on this idea, we can estimate an upper bound ub. When we use more signatures, we can get a tighter ub and early terminate if $\tau_k \geq$ ub.

Top-k Join. We still progressively generate the signatures. We first generate the first signatures of the two tables and use zip-partition to shuffle the same signature into the same partition. In each partition, we compute the candidates. Then the server collects all the candidates, puts the candidate into the priority queue and computes τ_k. Next for each record, we decide whether to generate its second segments or not based on the upper bound ub. If $\tau_k \geq$ ub, we do not need to generate its signatures; otherwise, we generate its second signature. If we do not need to generate the next signatures for all records, the algorithm terminates.

4. CROWD-BASED REFINEMENT

Given a set of candidate pairs, we utilize the crowd to check whether the two records in each pair refer to the same entity. To avoid enumerating every candidate pair, we propose a selection-inference-refine framework.

Task Selection. We first select a set of "beneficial" tasks, where the "beneficial" means that if we ask these tasks, we can reduce the monetary cost based on *result inference*. We then ask the crowd to answer these selected tasks.

Result Inference. Based on the results of asked tasks, we can infer the answers of unasked tasks. Thus we can reduce the monetary cost by result inference.

Answer Refinement. If the crowd answers are not correct (e.g., answers given by malicious workers), the inferred results may not be correct. Thus the inference may sacrifice the quality. To address this issue, we need to refine the answers with high uncertainty that different workers give conflict answers.

4.1 Transitivity-Based Method

We propose to use the transitivity to do result inference [68]. For example, if $r = s$ (r and s refer to the same entity) and $s = t$, we can deduce that $r = t$. If $r = s$ and $s \neq t$, we can deduce that $r \neq t$. In both of the two cases, we do not need to ask the task of (r, t), and thus can reduce the monetary cost. However given $s \neq t$ and $r \neq t$, we cannot deduce whether r and s refer to the same entity. Thus the order of asking tasks is very important.

In our method we first select some tasks to ask, where a task is a pair of records and asks the crowd to check whether the two records refer to the same entity. Then based on the answers of the asked tasks, we use the transitivity to infer the answers of some unasked tasks. If there are some tasks without answers (neither asked to the crowd nor inferred by the transitivity), we need to repeat the first two steps (task selection and result inference). In this framework, we aim to address the following problems.

Minimizing the monetary cost. We want to minimize the number of asked tasks. We prove that the optimal order is to first ask the matching pair and then non-matching pair. However before asking the crowd, we do not know which are matching pairs and which are not. Fortunately, we can use the similarity to estimate the matching probability. The larger the similarity between two records is, the larger the probability of the two records referring to the same entity is. Thus we should first ask the pairs with larger probabilities. However this method involves high latency, because it needs to wait for the answers of the asked tasks from the crowd to infer the answers of unasked tasks. Next we discuss how to reduce the latency.

Minimizing both monetary cost and latency. We use the number of rounds to model the latency. In each round, we ask some tasks. After getting the answers of these tasks, we do inference using the transitivity and select the tasks to ask in the next round. Thus we aim to minimize the number of rounds but without asking more tasks than the above method. In other words, we aim to ask the maximum number of tasks in each round, where each task cannot be inferred based on the answers of other tasks. To this end, we model the tasks as a graph, where nodes are records and edges are candidate record pairs. Following the order sorted by the candidate probability, we select the maximum spanning tree of the graph in each round and ask the edges on the tree in parallel.

Improving the quality. Due to the openness of crowdsourcing, the crowd workers may yield low-quality or even noisy answers. Thus it is important to control the quality in crowdsourcing. To address this problem, most of existing crowdsourcing studies employ a redundancy-based strategy, which assigns each task to multiple workers and then aggregates the answers given by different workers to infer the correct answer (called *truth*) of each task. A fundamental problem is *Truth Inference*, which decides how to effectively infer the truth for each task by aggregating the answers from multiple workers. Another important problem is *Task Assignment*, which judiciously assigns each task to appropriate workers. Task assignment aims to assign tasks to appropriate workers and truth inference attempts to infer the quality of workers and the truth of tasks simultaneously.

Truth Inference System [6]. We develop a truth inference system in crowdsourcing [81]. Our system implements most of existing truth inference algorithms (e.g., [11], [74], [10], [83], [28], [62], [51],[37], [3, 38], [71], [27], [39], [39], [51]) and users do not need to write truth inference algorithms on their own. Given a set of tasks and a set of answers for these tasks, our system can automatically suggest appropriate inference algorithms for users. For different types of tasks, our system can judiciously suggest best inference

[6] https://github.com/TsinghuaDatabaseGroup/CrowdTI

algorithms and infer the high-quality answers of each task. In addition, our system can automatically infer the quality of workers that answer these tasks. If users iteratively publish tasks to crowdsourcing platforms, they can eliminate low-quality workers based on the inferred worker quality.

Online Task Assignment System[7]**.** Existing studies assume that workers have the same quality on different tasks. However we find that in practice many tasks are domain specific, and workers are usually good at tasks in domains they are familiar with but provide low-quality answers in unfamiliar domains. To address this problem, we propose a domain-aware task assignment method [17, 82], which assigns tasks to appropriate workers who are familiar with the domains of the tasks. We develop an online task assignment system. When a worker requests for tasks, our system on-the-fly selects k tasks to the worker. We have deployed our system on top of Amazon Mechanical Turk, which can automatically compute the worker quality and judiciously assign tasks to appropriate workers. Thus users do not need to write codes to interact with the underlying crowdsourcing platforms, and they can use our systems for both task assignment and truth inference.

4.2 Partial-Order-Based Approach

The transitivity-based approach has several limitations. Firstly, the transitivity may not hold for some entities. Secondly, it still involves large cost for some datasets, especially if there are only few records satisfying the transitivity. Thirdly, it works well on self-join, e.g., $R = S$ but it cannot significantly reduce the cost for R-S join. To address these limitations, we propose a partial-order-based method [7].

Partial Order. We define a partial order on candidate record pairs. Given two pairs $p_{ij} = (r_i, s_j)$ and $p_{i'j'} = (r_{i'}, s_{j'})$, we define $p_{ij} \succeq p_{i'j'}$, if (r_i, s_j) has no smaller similarities than $(r_{i'}, s_{j'})$ on every attribute. We define $p_{ij} \succ p_{i'j'}$, if $p_{ij} \succeq p_{i'j'}$ and (r_i, s_j) has larger similarities on at least one attribute than $(r_{i'}, s_{j'})$.

We model the candidate pairs as a graph $\mathcal{G}$ based on the partial order, where each vertex is a candidate pair. Given two pairs p_{ij} and $p_{i'j'}$, if $p_{ij} \succ p_{i'j'}$, there is a directed edge from p_{ij} to $p_{i'j'}$.

Graph Coloring. Each vertex in $\mathcal{G}$ has two possibilities: (1) they refer to the same entity and we color it Green (matching); (2) they refer to different entities and we color it Red (non-matching). Initially each vertex is uncolored. Our goal is to utilize the crowd to color all vertices. A straightforward method is to take the record pair on each vertex as a task and ask workers to answer the task, i.e. whether the two records in the pair refer to the same entity. If a worker thinks that the two records refer to the same entity, the worker returns `Yes`; `No` otherwise. For each pair, to tolerate the noisy results from workers, we assign it to multiple workers, say 5. Based on the workers' results, we get a voted answer on each vertex. If majority workers vote `Yes`, we color it Green; otherwise we color it Red.

Obviously this method is rather expensive as there are many vertices on the graph. To address this issue, we propose an effective coloring framework to reduce the number of tasks. It first computes the partial orders between pairs and constructs a graph. Then it selects an uncolored vertex p_{ij} and asks workers to answer `Yes` or `No` on the vertex,
(1) If majority workers vote `Yes`, we not only color p_{ij} Green, but also color all of its ancestors Green. In other words, for $p_{i'j'} \succ p_{ij}$, we also take $r_{i'}$ and $s_{j'}$ as the same entity. This is because $p_{i'j'}$ has larger similarity on every attribute than p_{ij}, and since r_i and s_j refer to the same entity (denoted by $r_i = s_j$), we deduce that $r_{i'} = s_{j'}$.
(2) If majority workers vote `No`, we not only color p_{ij} Red, but also color all of its descendants Red. In other words, for $p_{ij} \succ p_{i'j'}$, we also take $r_{i'}$ and $s_{j'}$ as different entities. This is because $p_{i'j'}$ has smaller similarity on every attribute than p_{ij}, and since r_i and s_j refer to different entities (denoted by $r_i \neq s_j$), we deduce that $r_{i'} \neq s_{j'}$.

If all the vertices have been colored, the algorithm terminates; otherwise, it selects an uncolored vertex and repeats the above steps. Obviously, this method can reduce the cost as we can avoid asking many unnecessary vertices.

There are several challenges in this algorithm.

(1) Graph Construction. As there are large numbers of pairs, how to efficiently construct the graph? Can we reduce the graph size so as to reduce the number of tasks? We propose a range-search-tree to address this problem [7].

(2) Task Selection. How to select the minimum number of vertices to ask in order to color all vertices? We propose an expectation based method [7].

(3) Error Tolerant. The coloring strategy and the workers may introduce errors. So how to tolerate the errors? We propose to not to color a vertex if the confidence of the vertex color is low (e.g., there is a tie between `Yes` and `No` answers) and infer its answer based on other vertices. This strategy can reduce the inference error [7].

4.3 Crowd-Powered System

We develop a crowd-powered database system `CDB` that supports crowd-based query optimizations[8]. `CDB` provides declarative programming interfaces and allows requesters to use a SQL-like language for posing queries that involve crowdsourced operations. On the other hand, `CDB` leverages the crowd-powered operations to encapsulate the complexities of interacting with the crowd. `CDB` has fundamental differences from existing systems. First, existing systems adopt a query-tree model, which aims at selecting an optimized table-level join order to optimize a query. However, the tree model provides a coarse-grained optimization, which generates the same order for different joined records and limits the optimization potential that different joined record should be optimized by different orders. To address this problem, `CDB` employs a graph-based query model that provides more fine-grained query optimization. Given a `CQL` query, `CDB` builds a graph based on the query and the data. This graph model has the advantage of making the fine-grained record-level optimization applicable. Second, most of the existing systems (i.e., CrowdDB [20], Qurk [43], and Deco [46]) only focus on optimizing monetary cost, and they adopt the *majority voting* strategy for quality control, and do not consider to model the latency control. `CDB` adopts a unified framework to perform the multi-goal optimization based on the graph model. (*i*) For cost control, our

[7] https://github.com/TsinghuaDatabaseGroup/CrowdOTA

[8] https://github.com/TsinghuaDatabaseGroup/CDB

goal is to minimize the number of tasks to find all the answers. We prove that this problem is NP-hard and propose an expectation-based method to select tasks. (*ii*) For latency control, we adopt the round-based model which aims to reduce the number of rounds for interacting with crowdsourcing platforms. We identify the most "beneficial" tasks which can be used to prune other tasks, and ask such tasks in parallel to reduce the latency. (*iii*) We optimize the quality by devising quality-control strategies (i.e., truth inference and task assignment) for different types of tasks, i.e., single-choice, multi-choice, fill-in-blank and collection tasks. We have implemented our system and deployed it on Amazon Mechanical Turk (AMT), CrowdFlower and ChinaCrowd. Users can utilize our system to support entity matching and any other crowd-based queries.

Next we introduce the components of our system `CDB`.

Declarative Query Language. We extend SQL by adding crowd-powered operations and propose crowd SQL (`CQL`). `CQL` contains both data definition language (DDL) and data manipulation language (DML). A requester can use `CQL` DDL to define her data by asking the crowd to collect or fill the data, or use `CQL` DML to manipulate the data based on crowdsourced operations, e.g., crowdsourced selection, join, top-k, and grouping.

Graph Query Model. A requester can submit her tasks and collect the answers using relational tables. To provide a fine-grained optimization on the relational data, we define a graph-based query model. Given a `CQL` query, we construct a graph, where each vertex is a record of a table in the `CQL` and each edge connects two records based on the join/selection predicates in the `CQL`. We utilize the graph model to provide the record-level optimization.

Query Optimization. Query optimization includes cost control, latency control and quality control. (*i*) Cost control aims to optimize the monetary cost by reducing the numbers of tasks to ask the crowd. We formulate the task selection problem using the graph model, prove that this problem is NP-hard, and propose effective expectation-based algorithms to reduce the cost. (*ii*) Latency control focuses on reducing the latency. We utilize the number of rounds to model the latency and aim to reduce the number of rounds. To reduce the cost, we need to utilize the answers of some asked tasks to infer those of the unasked tasks, and the inference will lead to more rounds. Thus there is a tradeoff between cost and latency. Our goal is to simultaneously ask the tasks that cannot be inferred by others in the same round. (*iii*) Quality control is to improve the quality, which includes two main components: truth inference and task assignment. Task assignment assigns each task to multiple workers and truth inference infers task answers based on the results from multiple assigned workers. We propose a holistic framework for task assignment and truth inference for different types of tasks.

Crowd UI Designer. Our system supports four types of UIs. (1) Fill-in-the-blank task: it asks the crowd to fill missing information, e.g., the affiliation of a professor. (2) Collection task: it asks the crowd to collect new information, e.g., the top-100 universities. (3) Single-choice task: it asks the crowd to select a single answer from multiple choices, e.g, selecting the country of a university from 100 given countries. (4) Multiple-choice task: it asks the crowd to select multiple answers from multiple choices, e.g., selecting the research topics of a professor from 20 given topics. Another goal is to automatically publish the tasks to crowdsourcing platforms. We have deployed our system on top of AMT, ChinaCrowd and CrowdFlower. There is a main difference between AMT/ChinaCrowd and CrowdFlower. In CrowdFlower, it does not allow a requester to control the task assignment while AMT/ChinaCrowd have a development model in which the requester can control the task assignment. Thus in AMT/ChinaCrowd, we utilize the development model and enable the online tasks assignment.

MetaData & Statistics. We maintain three types of metadata. (1) Task. We utilize relational tables to maintain tasks, where there may exist empty columns which need to be crowdsourced. (2) Worker. We maintain worker's quality in the history and the current task. (3) Assignment. We maintain the assignment of a task to a worker as well as the corresponding result. We also maintain statistics, such as selectivity, graph edge weights, etc., to facilitate our graph-based query optimization techniques.

Workflow. A requester defines her data and submits her query using `CQL`, which will be parsed by `CQL Parser`. Then `Graph-based Query Model` builds a graph model based on the parsed result. Next `Query Optimization` generates an optimized query plan, where cost control selects a set of tasks with the minimal cost, latency control identifies the tasks that can be asked in parallel, and quality control decides how to assign each task to appropriate workers and infers the final answer. `Crowd UI Designer` designs various interfaces and interacts with underlying crowdsourcing platforms. It periodically pulls the answers from the crowdsourcing platforms in order to evaluate worker's quality, and when a worker requests tasks, it returns beneficial tasks. `Result Collection` reports the results to the requester.

5. RELATED WORK

There are many studies on data integration from different perspectives, e.g., schema matching [49, 40, 5, 80], entity matching [21, 50, 8, 16, 30, 58]. In this work, we focus on entity matching and we briefly classify existing studies into the following categories.

Human-in-the-loop Entity Matching. Doan et al develop two entity matching systems Magellan [30] and Corleone/Falcon [21, 50, 8, 16, 36]. The former focuses on generating rules from the crowd and applying the rules efficiently, and builds a system situated within the PyData eco-system. The latter aims to build a cloud-based entity-matching service without requiring developer in the loop. Different from them, our system has two steps: rule-based pruning and crowd-based refinement, and both of the two steps involve human in the loop to improve the quality of entity matching.

Crowd-based Entity Matching. Many studies utilize the crowd for entity matching to improve the quality. An important problem is to design tasks for workers. A straightforward method is to generate *pair-comparison-based tasks*, where each task is a pair of records and asks workers to check whether the two records refer to the same entity. This method may generate a large number of tasks. To address this problem, the *clustering-based tasks* are proposed [42, 65], where each task is a group of records and asks workers to classify the records into different clusters such that records in the same cluster refer to the same entity and records in different clusters refer to different entities. As the clustering-based method does not need to enumerate every pair, it can reduce the monetary cost. Wang et al. [65] propose a similarity-based method, which computes the similarity of record pairs and prunes the pairs with small similarities. As this method can prune many dissimilar pairs without sacrificing the quality of final answers, most of existing studies use this technique to reduce the cost. Wang et al. [70] propose a correlation-clustering method. It first prunes dissimilar pairs with small similarities, and then selects some pairs to ask and divides the records into a set of clusters based on the workers' results of these asked pairs. Finally, it refines the clusters by selecting more pairs to ask, checking whether their answers are consistent with the initial clusters, and adjusting the clusters based on the inconsistencies. This method improves the accuracy at the expense of huge monetary costs. Whang et al. [72] propose a probabilistic model to select high-quality tasks. Verroios et al. [64] improve the model by tolerating workers' errors. Gokhale et al. [21] study the crowdsourced record linkage problem, which focuses on linking records from two tables.

Blocking-based Method. There are also some blocking-based method [73, 59, 53], which partitions the records into different groups based on "keys". Then they only consider the records in the same group and prune the records in different groups. This method can significantly improve the performance since it does not need to enumerate every pair. However, it is hard to generate high-quality blocking strategies and they may involve false negatives.

Similarity Join. There have been many studies on similarity joins [4, 77, 2, 67, 76, 66, 13, 63, 52]. Jiang et al. [26, 19, 19, 25, 32, 12] conduct a comprehensive experimental study on similarity joins. Existing studies usually employ a signature-based framework, which generates some signatures for each record such that two records are similar if they share at least one common signature. There are two effective signatures, prefix filtering [4, 77, 67, 75, 67] and segment-based filtering [33, 14]. The former sorts the elements and selects several infrequent elements as signatures such that if two records do not share a common signature, they cannot be similar. The latter partitions each record into different segments and takes the segments as signatures such that if two records are similar they must share a common signature. In addition, some work [55, 24, 79, 6] focus on probabilistic techniques for set-based similarity joins. However, they cannot find the exact answer and need to tune parameters which are tedious and less effective [4]. There are some works on supporting similarity join using Map-Reduce framework [63, 44, 63, 1, 12, 13]. Vernica et al. [63] utilize the prefix filtering to support set-based similarity functions. Metwally et al. [44] propose a 2-stage algorithm for sets, multi-sets and vectors. Afrati et al. [1] optimize the map, reduce and communication cost. However, it is rather expensive to transmit long strings using a single MapReduce stage. Kim et al. [29] address the top-k similarity join problem using MapReduce. Deng et al [13] address the similarity joins with edit-distance constraints on MapReduce using the segment-based index.

Crowdsourcing Systems & Operators. Recently there are many studies to develop crowdsourcing database systems, such as CrowdDB [20], Qurk [43], Deco [47], and CrowdOP [18]. For achieving high crowdsourcing query processing performance, the systems focus on optimizing cost (cheap), latency (fast) and quality (good). Moreover, there are also some crowdsourcing techniques that focus on designing individual crowdsourced operators, including crowdsourced selection [45, 54, 78], join [68, 7], top-k/sort [9], aggregation [41, 22], and collect [61, 48]. Li et al.[35] give a survey on crowdsourced data management.

6. CHALLENGES AND OPPTUNITIES

6.1 Entity Matching on Unstructured Data

Most of existing studies focus on entity matching on structured data, where the attributes are well aligned. However in real applications, the data are not well structured, e.g., a product name at Amazon "Apple MacBook Pro 13.3-Inch Laptop black Multi-Touch Bar/Core i5/8GB/256GB MLH12CH/A OS X El Capitan". The similarity join methods cannot work well because different tokens should have different importance and it is hard to compute accurate similarities. The blocking-based method cannot find appropriate blocking strategies. The crowd-based method involves huge monetary cost. Thus it calls for new methods for entity matching on unstructured data.

6.2 Knowledge From Human

Existing works utilize the human to provide the answers of tasks (e.g., whether two records refer to the same entity) but cannot use the knowledge behind the answers, e.g., why these two records refer to the same entity, why the other two records do not refer to the same entity, which information plays an important role in the decision. Thus if we can detect the knowledge from the human, then we can use the knowledge to optimize data integration. This is similar to knowledge bases, which obtain the knowledge (entities, concepts, relations) from the big data. However for data integration, we need to obtain more "deep" knowledge, e.g., rules, decision factors, etc.

6.3 Fine-Grained Human Involvement

Existing techniques usually resort to experts to generate the high-quality rules, and utilize the crowd to verify the candidate pairs, which are easy to check. However, the expert and crowd have different skilled domains, capability and price requirement. Moreover the tasks also have diverse domains and should be assigned to qualified workers. Thus we need to prioritize the expert and crowd based on their quality, skilled field, and provide more fine-grained human ability scheduling to save the monetary cost and improve the quality.

6.4 Entity Matching on Complex Data

There are more and more complex data, such as social data, trajectory data, and spatial data. Existing techniques cannot efficiently and effectively handle such complex data. For example, given two social networks, e.g., Facebook and Twitter, if we can correlate the users from the two networks, we can obtain more complete profiles for users, e.g., user preferences, friends, and hobbies. There are many challenges that arise in this problem. The first is how to determine whether two users refer to the same entity, which is rather hard for graph data. The second is to efficiently link the users across graphs. As existing graph data are rather large (for example there are more than a billion users in Facebook), it is fairly important to devise efficient algorithms to achieve high performance. Take trajectories as an example.

Uber generates more and more trajectories by running vehicles. However the trajectories generated from different cars are not consistent due to the measurement and sampling errors. It is important to integrate the trajectories from different vehicles. However it is rather challenging to integrate them because the trajectories are more complicated and have high dimensions, e.g., time, speed, and location. It calls for new techniques and systems for complex data.

6.5 Machine Learning for Data Integration

Machine learning techniques have been widely used in many areas, e.g., computer vision, machine translation, national language processing. We can try to use machine learning techniques to benefit the data integration problem. For example, we can use deep learning and embedding techniques to find candidate pairs. We can also use active learning techniques to reduce the monetary cost. There are two big challenges in using machine learning techniques. The first is to find a large training data to feed the machine learning algorithms. The second is to design new machine learning models for data integration.

6.6 Other Problems in Data Integration

Besides entity matching, there are many other problems in data integration, e.g., schema matching. Schema matching aims to find correspondences between two schemas. It is hard to automatically find the correspondences because different schemas have different semantics. Note it is also hard for the crowd to label the correspondences based on the column name, because the name may have multiple representations, e.g., country, location, region, and the crowd may have no domain knowledge to correctly label the schemas they are not familiar with. Thus it requires to use the instances (cell values in the table) to help the crowd to better understand the data. However it is not user-friendly (and unnecessary) to show all the instances to the crowd. Thus the challenging problems include (1) how to design crowd tasks for schema matching and (2) how to assign the tasks to appropriate workers.

7. CONCLUSION

In this paper, we propose a hybrid human-machine data integration framework that harnesses human ability to address the data integration problem. We apply it initially to the entity-matching problem. We first use similarity-based rules to identify possible matching pairs and then utilize the crowd to compute actual matching pairs by refining these candidate pairs. In the machine step, we define the rules based on similarity functions and develop effective algorithms to learn rules based on positive and negative examples. We build a distributed in-memory system DIMA to efficiently apply these rules. In the crowd step, we propose a selection-inference-refine framework that uses the crowd to verify the candidate pairs. We first select beneficial tasks to ask the crowd, and then use transitivity and partial order to infer the answers of unasked tasks based on the crowd results of the asked tasks. Next we refine the inferred answers with low confidence to improve the quality. We also study how to reduce the latency by asking the questions in parallel. We develop a crowd-powered database system CDB that provides a SQL-like language for processing crowd-based queries. Lastly, we provide emerging challenges in human-in-the-loop data integration.

8. ACKNOWLEDGMENT.

Thanks very much to my outstanding collaborators, Jiannan Wang, Dong Deng, Yudian Zheng, Ju Fan, Chengliang Chai, Nan Tang, and Jianhua Feng. This work was supported by the 973 Program of China (2015CB358700), NSF of China (61632016, 61373024,61602488,61422205,61472198), ARC DP170102726, and FDCT/007/2016/AFJ.

9. REFERENCES

[1] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE*, pages 498–509, 2012.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[3] B. I. Aydin, Y. S. Yilmaz, Y. Li, Q. Li, J. Gao, and M. Demirbas. Crowdsourcing for multiple-choice question answering. In *AAAI*, pages 2946–2953, 2014.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[5] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster. Putting context into schema matching. In *VLDB*, pages 307–318, 2006.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *ACM STOC*, pages 327–336, 1998.

[7] C. Chai, G. Li, J. Li, D. Deng, and J. Feng. Cost-effective crowdsourced entity resolution: A partial-order approach. In *SIGMOD*, pages 969–984, 2016.

[8] S. Das, P. S. G. C., A. Doan, J. F. Naughton, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, and Y. Park. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*, pages 1431–1446, 2017.

[9] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *ICDT*, pages 225–236, 2013.

[10] A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the em algorithm. *Applied statistics*, pages 20–28, 1979.

[11] 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.

[12] D. Deng, Y. Jiang, G. Li, J. Li, and C. Yu. Scalable column concept determination for web tables using large knowledge bases. *PVLDB*, 6(13):1606–1617, 2013.

[13] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, pages 340–351, 2014.

[14] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.

[15] D. Deng, G. Li, H. Wen, H. V. Jagadish, and J. Feng. META: an efficient matching-based method for error-tolerant autocompletion. *PVLDB*, 9(10):828–839, 2016.

[16] A. Doan, A. Ardalan, J. R. Ballard, S. Das, Y. Govind, P. Konda, H. Li, S. Mudgal, E. Paulson, P. S. G. C., and H. Zhang. Human-in-the-loop challenges for entity matching: A midterm report. In *HILDA@SIGMOD*, pages 12:1–12:6, 2017.

[17] J. Fan, G. Li, B. C. Ooi, K. Tan, and J. Feng. icrowd: An adaptive crowdsourcing framework. In *SIGMOD*, pages 1015–1030, 2015.

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

[19] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.

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

[21] 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 *SIGMOD*, pages 601–612, 2014.

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

[23] S. Hao, N. Tang, G. Li, and J. Li. Cleaning relations using knowledge bases. In *ICDE*, pages 933–944, 2017.

[24] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM STOC*, pages 604–613, 1998.

[25] Y. Jiang, D. Deng, J. Wang, G. Li, and J. Feng. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In *EDBT*, pages 341–348, 2013.

[26] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[27] D. R. Karger, S. Oh, and D. Shah. Iterative learning for reliable crowdsourcing systems. In *NIPS*, pages 1953–1961, 2011.

[28] H.-C. Kim and Z. Ghahramani. Bayesian classifier combination. In *AISTATS*, pages 619–627, 2012.

[29] Y. Kim and K. Shim. Parallel top-k similarity join algorithms using mapreduce. In *ICDE*, pages 510–521, 2012.

[30] P. Konda, S. Das, P. S. G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. F. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[31] G. Li, C. Chai, J. Fan, X. Weng, J. Li, Y. Zheng, Y. Li, X. Yu, X. Zhang, and H. Yuan. CDB: optimizing queries with crowd-based selections and joins. In *SIGMOD*, pages 1463–1478, 2017.

[32] G. Li, D. Deng, and J. Feng. A partition-based method for string similarity joins with edit-distance constraints. *ACM Trans. Database Syst.*, 38(2):9, 2013.

[33] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

[34] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In *SIGMOD*, pages 1137–1151, 2015.

[35] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. *IEEE Trans. Knowl. Data Eng.*, 28(9):2296–2319, 2016.

[36] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. In *ICDE*, pages 39–40, 2017.

[37] Q. Li, Y. Li, J. Gao, L. Su, B. Zhao, M. Demirbas, W. Fan, and J. Han. A confidence-aware approach for truth discovery on long-tail data. *PVLDB*, 2014.

[38] Q. Li, Y. Li, J. Gao, B. Zhao, W. Fan, and J. Han. Resolving conflicts in heterogeneous data by truth discovery and source reliability estimation. In *SIGMOD*, pages 1187–1198, 2014.

[39] Q. Liu, J. Peng, and A. T. Ihler. Variational inference for crowdsourcing. In *NIPS*, pages 692–700, 2012.

[40] J. Madhavan, P. A. Bernstein, A. Doan, and A. Y. Halevy. Corpus-based schema matching. In *ICDE*, pages 57–68, 2005.

[41] A. Marcus, D. R. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. *PVLDB*, 6(2):109–120, 2012.

[42] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

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

[44] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[45] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD*, pages 361–372, 2012.

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

[47] H. Park, R. Pang, A. G. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: A system for declarative crowdsourcing. *PVLDB*, 5(12):1990–1993, 2012.

[48] H. Park and J. Widom. Crowdfill: collecting structured data from the crowd. In *SIGMOD*, pages 577–588, 2014.

[49] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[50] V. Rastogi, N. N. Dalvi, and M. N. Garofalakis. Large-scale collective entity matching. *PVLDB*, 4(4):208–218, 2011.

[51] V. C. Raykar, S. Yu, L. H. Zhao, G. H. Valadez, C. Florin, L. Bogoni, and L. Moy. Learning from crowds. *JMLR*, 2010.

[52] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[53] A. D. Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, pages 1055–1064, 2012.

[54] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and A. Y. Halevy. Crowd-powered find algorithms. In *ICDE*, 2014.

[55] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

[56] Z. Shang, Y. Liu, G. Li, and J. Feng. K-join: Knowledge-aware similarity join. *IEEE Trans. Knowl. Data Eng.*, 28(12):3293–3308, 2016.

[57] Z. Shang, Y. Liu, G. Li, and J. Feng. K-join: Knowledge-aware similarity join. In *ICDE*, pages 23–24, 2017.

[58] W. Shen, P. DeRose, L. Vu, A. Doan, and R. Ramakrishnan. Source-aware entity matching: A compositional approach. In *ICDE*, pages 196–205, 2007.

[59] G. Simonini, S. Bergamaschi, and H. V. Jagadish. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *PVLDB*, 9(12):1173–1184, 2016.

[60] J. Sun, Z. Shang, G. Li, D. Deng, and Z. Bao. Dima: A distributed in-memory similarity-based query processing system. *PVLDB*, 10(5), 2017.

[61] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, pages 673–684, 2013.

[62] M. Venanzi, J. Guiver, G. Kazai, P. Kohli, and M. Shokouhi. Community-based bayesian aggregation models for crowdsourcing. In *WWW*, pages 155–164, 2014.

[63] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[64] V. Verroios and H. Garcia-Molina. Entity resolution with crowd errors. In *ICDE*, pages 219–230, 2015.

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

[66] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[67] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.

[68] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, pages 229–240, 2013.

[69] J. Wang, G. Li, J. X. Yu, and J. Feng. Entity matching: How similar is similar. *PVLDB*, 4(10):622–633, 2011.

[70] S. Wang, X. Xiao, and C. Lee. Crowd-based deduplication: An adaptive approach. In *SIGMOD*, pages 1263–1277, 2015.

[71] P. Welinder, S. Branson, P. Perona, and S. J. Belongie. The multidimensional wisdom of crowds. In *NIPS*, 2010.

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

[73] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. In *SIGMOD*, pages 219–232, 2009.

[74] J. Whitehill, T.-f. Wu, J. Bergsma, J. R. Movellan, and P. L. Ruvolo. Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In *NIPS*, 2009.

[75] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[76] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[77] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[78] T. Yan, V. Kumar, and D. Ganesan. Crowdsearch: exploiting crowds for accurate real-time image search on mobile phones. In *MobiSys*, pages 77–90, 2010.

[79] J. Zhai, Y. Lou, and J. Gehrke. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *ACM SIGMOD*, pages 997–1008, 2011.

[80] C. J. Zhang, L. Chen, H. V. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 6(9):757–768, 2013.

[81] Y. Zheng, G. Li, Y. Li, C. Shan, and R. Cheng. Truth inference in crowdsourcing: Is the problem solved? *PVLDB*, 10(5):541–552, 2017.

[82] Y. Zheng, J. Wang, G. Li, R. Cheng, and J. Feng. QASCA: A quality-aware task assignment system for crowdsourcing applications. In *SIGMOD*, pages 1031–1046, 2015.

[83] D. Zhou, S. Basu, Y. Mao, and J. C. Platt. Learning from the wisdom of crowds by minimax entropy. In *NIPS*, 2012.