Crowdsourced Selection on Multi-Attribute Data

Xueping Weng, Guoliang Li, Huiqi Hu, Jianhua Feng
Department of Computer Science, Tsinghua University
wxp15@mails.tsinghua.edu.cn, liguoliang@tsinghua.edu.cn, hqhu@sei.ecnu.edu.cn, fengjh@tsinghua.edu.cn

ABSTRACT

Crowdsourced selection asks the crowd to select entities that satisfy a query condition, e.g., selecting the photos of people wearing sunglasses from a given set of photos. Existing studies focus on a single query predicate and in this paper we study the crowdsourced selection problem on multi-attribute data, e.g., selecting the female photos with dark eyes and wearing sunglasses. A straightforward method asks the crowd to answer every entity by checking every predicate in the query. Obviously, this method involves huge monetary cost. Instead, we can select an optimized predicate order and ask the crowd to answer the entities following the order. Since if an entity does not satisfy a predicate, we can prune this entity without needing to ask other predicates and thus this method can reduce the cost. There are two challenges in finding the optimized predicate order. The first is how to detect the predicate order and the second is to capture correlation among different predicates. To address this problem, we propose predicate order based framework to reduce monetary cost. Firstly, we define an expectation tree to store selectivities on predicates and estimate the best predicate order. In each iteration, we estimate the best predicate order from the expectation tree, and then choose a predicate as a question to ask the crowd. After getting the result of the current predicate, we choose next predicate to ask until we get the result. We will update the expectation tree using the answer obtained from the crowd and continue to the next iteration. We also study the problem of answering multiple queries simultaneously, and reduce its cost using the correlation between queries. Finally, we propose a confidence based method to improve the quality. The experiment result shows that our predicate order based algorithm is effective and can reduce cost significantly compared with baseline approaches.

1 INTRODUCTION

Crowdsourced selection aims to find entities that satisfy a given query condition. For example, consider the 8 entities in Figure 1. Given a query Q which aims to find male people with white skin and wearing sunglasses, entities e_2 and e_4 are the results while e_1, e_3, e_5, e_6, e_7, e_8 are not because they do not satisfy all the query predicates. Crowdsourced selection is widely used in many applications, such as image search and entity matching.

As the machine-based algorithm cannot achieve high quality, crowdsourced selection solutions are widely studied that leverage the crowd's ability to solve this problem [4, 17, 20, 24]. Parameswaran et al.[20] focused on the single-predicate query selection problem. To support multi-predicate query, it needs to enumerate every predicate and involves huge cost. Marcus et al. [17] proposed sample-based methods to estimate the selectivity of predicates and Fan et al. [4] proposed a sampling method to minimize the cost. However, the sampled-based method has several weaknesses. First, if we use a low sample rate, it will introduce high errors. Second, if we use a high sample rate, it takes high monetary cost for sampling. Third, they only consider the independent selectivities and do not consider the correlation between different predicates.

To address these limitations, we propose a predicate order based crowdsourced selection framework, called Pows, which can significantly reduce the monetary cost while keeping high quality. The basic idea is that we define a predicate order on query predicates and use this order to check whether an entity meets predicates by asking as few questions as possible. Particularly, we first define a predicate order, and then select predicates in order. Given an entity, we ask workers to check whether this entity meets a predicate. (1) If workers give No, we do not need check other predicates and label it as NEG. (2) If workers give Yes, we keep checking the next predicate and add it into result set if all predicates are Yes. We build a predicate order expectation tree to compute the conditional selectivities between predicates. Then in each iteration, we choose the predicate order with minimal estimated cost as the current predicate order and use it check an entity. After we get answers from workers, we update selectivities on the expectation tree. Thus our goal is to estimate selectivities as precisely as possible and find the optimal order in each iteration. Because the size of possible predicates order is too large, we prune the tree with limited height and propose parallel algorithm to reduce latency. We extend our framework and expectation tree to answer multiple queries simultaneously. As workers and the framework may introduce errors, we develop confidence based error tolerance method to tolerate errors.

To summarize, we make the following contributions in this paper. (1) We propose a predicate-order based crowdsourced selection framework. We define a predicate order on query predicates and utilize it to check entities which can reduce monetary cost. (2) We build an expectation tree based on the predicates and utilize this tree to compute predicates' conditional selectivities. We devise efficient algorithms to estimate the expectation of predicates orders and choose the best estimate predicates order. (3) We extend the expectation tree to answer multiple crowdsourced selection queries effectively. (4) We develop a confidence-based error-tolerance algorithm based on Bayes voting to tolerate errors. (5) We conduct experiments using real-world dataset on CrowdFlower. Experimental results show that our method saves monetary cost and significantly outperforms state-of-the-art approaches.

2 PRELIMINARIES

2.1 Problem Formulation

Data Model. Our work focuses on finding all entities that meet a given query from a set of entities $\mathbb{E} = \{e_1, e_2, ..., e_n\}$. Every entity $e \in \mathbb{E}$ has k attributes. We denote its attribute set as $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_k\}$, where $\mathcal{A}_i(e)$ denotes the i-th attribute's value of

CIKM'17 , November 6–10, 2017, Singapore, Singapore

ACM ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3132891

Figure 1: Eight Entities In LFW Dataset.

entity e. We consider crowdsourced selection query Q that consists of a set of predicates. Each predicate $p_i = (\mathcal{A}_i = v_i)$ is a pair of attribute and value, where $\mathcal{A}_i$ is the attribute and v_i is the value. We model a crowdsourced selection query as a set of predicates $Q = \{p_1, p_2, ..., p_m\}$.

Definition 2.1 (Crowdsourced Selection Query). Given a set of entities $\mathbb{E}$ and a multi-predicate query $Q = \{\mathcal{A}_1 = v_1, \mathcal{A}_2 = v_2, ..., \mathcal{A}_m = v_m\}$. The crowdsourced selection problem aims to find the result set $\mathcal{R} \subseteq \mathbb{E}$ with the minimum cost C, where each entity $e \in \mathcal{R}$ satisfies Q, and those entities $e \notin \mathcal{R}$ do not satisfy the query.

For example, Figure 1 shows 8 entities and 3 queries Q_1, Q_2, Q_3. Entities e_2 and e_4 meet all the predicates in Q_1, and entities e_1, e_3, e_5, e_6, e_7, e_8 cannot meet them. Thus the result set $\mathcal{R}$ is $\{e_2, e_4\}$. We label those entities in $\mathcal{R}$ as POS, and label others as NEG. Crowdsourced selection problem asks questions to the crowd for detecting whether each entity could meet a predicate. As we need to pay the workers for answering a question, the objective is to reduce the number of questions while keeping high quality.

Definition 2.2 (Multiple Selection Query). Given a set of selection queries $\bar{Q} = \{Q_1, Q_2, ..., Q_t\}$ and an entity set $\mathbb{E}$, it aims to find result set $\mathcal{R}_i$ for each query Q_i.

For example, the result sets for queries Q_1, Q_2 and Q_3 in Figure 1 are $\mathcal{R}_1 = \{e_2, e_4\}$, $\mathcal{R}_2 = \{e_6\}$ and $\mathcal{R}_3 = \{\}$. For multiple selection queries, we aim to share the computation among these queries.

For a crowdsourced selection query, we ask workers to determine whether an entity e_i meets a predicate p_i and then label e_i as POS or NEG based on collected answers. If there is at least a predicate that entity e_i cannot meet, we will label e_i as NEG. So we aim to resolve the entity with the minimum number of questions.

A straightforward approach for selection query is to check every predicate and it takes a lot of monetary cost. For query Q_1 and entity e_3, it requires 4 questions. Furthermore, there is no need to check all predicates since we could stop checking when finding a NEG predicate. The key point is how to find the optimal predicate checking order which has significant influence on questions. If we first check $\mathcal{A}_4$ = sunglasses, we will get No result and stop without checking remained predicates. If we first check "$\mathcal{A}_1$ = male", it needs to check other predicates. For different predicate orders, the worst case asks 3 questions and the optimal cost is 1.

2.2 Related Works

2.2.1 Crowdsourced Query Optimization. Parameswaran et al. [20] studied the crowdsourced selection query with a single predicate and proposed heuristics to identify the result with the minimum expected cost. They assumed that each worker had false positive error rate e_0 and false negative rate e_1 independently, and aimed to find a strategy with minimum expected cost C under the error threshold τ and budget threshold m for each single item. They discussed their deterministic strategy and probabilistic strategy. Marcus et al.[17] proposed the basic idea of selectivities estimation on crowdsourcing, and they set the frequency as the selectivity using sampling theory. Sarma et al. [24] aimed to find k items satisfying a given boolean predicate from a set of items to optimize the monetary-cost and latency. They denoted the latency $\mathcal{T}$ as the number of phases and cost C as the questions' number to find those k items. In the algorithm process, they maintained a state of knowledge ε to determine the current state and determined how many questions should be asked in the next phase. Finally, they proposed *OptSeq*(a cost-optimal algorithm) and *UncOptCost*(a cost-optimal, phase-optimal algorithm) based on the prior probability of selectivity and error rate. Fan et al.[4] focused on the general crowd query optimization including selection query and join query. In selection query, they proposed *CSelect* algorithm to minimize the cost and budget-based latency. If without the requirements of latency, they estimated the selectivities using sample method and sorted those conditions. Hellerstein et al.[9] proved that sorting those predicates based on selectivities using rank is the optimal way for selection query. Different from existing works, we optimize selection queries with multiple predicates. We compared with existing algorithms and our method significantly outperforms them.

2.2.2 Other Related Work. There are several crowdsourced databases developed, like Deco[21–23] from Stanford, CrowdDB[7] from UC Berkeley and Qurk [19] from MIT and CDB [14] from Tsinghua. They provide SQL-like query operation and declarative interfaces, which can implement and optimize a lot of database operations, like filter, join etc. There are also a lot work on the crowdsourced database operation, like crowdsourcing selection[4, 14, 17, 20, 24], crowdsourced join[1, 14, 18, 26], crowdsourced sort or top-k[8, 18, 25, 27]. They focus on implementing those operations based on monetary-cost and latency. Some works on worker quality, accuracy[3, 6, 10, 11, 13, 20] and spammer detection[17] are proposed to improve the quality. Most of their works build model for workers and analyze the workers' quality and confidence. Some of them propose the task assignment method to improve results' quality [1, 5, 15, 27–30].

3 PREDICATE-ORDER-BASED-FRAMEWORK

We firstly define predicate order (Section 3.1) and then propose Pows-S framework (Section 3.2). We show how to update the expectation tree (Section 3.3), and estimate the optimal order (Section 3.4).

3.1 Predicate Order

We formally define the predicate order and use it to check entities.

Predicate Order. Given a crowdsourced selection query represented as $Q = \{p_1, p_2, ..., p_m\}$, we denote a permutation of predicates in Q as a predicate order π, where $\pi_i = \langle p_i^1, p_i^2, ..., p_i^m \rangle$.

Conditional Selectivity. Given a predicate order $\pi_i = \langle p_i^1, p_i^2, ..., p_i^m \rangle$, we denote s_i^j as the conditional selectivity of predicate p_i^j, which is the probability that an entity meets p_i^j if it also meets all previous predicates. s_i^j can be represented as:

$$s_i^j = \Pr(p_i^j | p_i^1, p_i^2, ..., p_i^{j-1}) \quad (1)$$

For example, consider the query Q_1 in Figure1. Its predicate set is $\{p_1, p_2, p_3, p_4\}$, both permutation $\langle p_4, p_2, p_3, p_1\rangle$ and $\langle p_3, p_1, p_2, p_4\rangle$ are two possible predicate orders of Q.

Predicate Order Cost. Given an entity e and a predicate order π_i, we denote the predicate order cost as the number of predicates need to check by π_i, which is represented as $C_{\pi_i}(e)$.

Optimal Predicate Order. Given an entity e and a query Q, the optimal predicate order π^* is the order such that $C_{\pi^*}(e) \le C_{\pi_i}(e)$ holds for any other order π_i.

For example, consider two predicate orders $\pi_1 = \langle p_3, p_2, p_4, p_1\rangle$, $\pi_2 = \langle p_1, p_2, p_4, p_3\rangle$. For entity e_5, if we use π_1 to check it, we will get No result from workers when checking the first predicate $\mathcal{A}_3$ = sunglasses and label e_5 as NEG, thus the cost is $C_{\pi_1}(e_5) = 1$. If we use π_2 to check e_5, we will get result $\langle$Yes, Yes, No$\rangle$ for first 3 predicates thus the cost is $C_{\pi_2}(e_5) = 3$. Since π_1 only needs 1 question (the minimal cost), it is an optimal predicate order for e_5.

Estimated Cost of Predicate Order. Given a predicate order $\pi_i = \langle p_i^1, p_i^2, ..., p_i^m\rangle$ and each predicate p_i^j's selectivity s_i^j, let's denote $\bar{C}_{\pi_i}(e_i)$ as the estimated cost if we take order π_i to check entity e_i.

If we check an entity with the predicate p_i^1, it has the probability $1 - s_i^1$ to get No result and we can skip checking, and probability s_i^1 to get Yes. For j-th predicate ($j \neq m$), if all previous predicates get Yes and the j-th predicate gets No, we exactly check j and only j predicates. Thus the probability is $(1 - s_i^j) * \prod_{k=1}^{j-1} s_i^k$. For the last predicate ($j = m$), whatever result it will get from workers, we have checked j predicate and should stop. The probability is $\prod_{k=1}^{m-1} s_i^k$.

Lemma 3.1. *The probability we need to check j and only j predicates with order π_i, that is the cost $C_{\pi_i} = j$ can be computed as:*

$$\Pr(C_{\pi_i} = j) = \begin{cases} (1 - s_i^j) * \prod_{k=1}^{j-1} s_i^k & j < m \\ \prod_{k=1}^{m-1} s_i^k & j = m \end{cases} \quad (2)$$

Based on Lemma 3.1, we could compute estimated cost $\bar{C}_{\pi_i}(e)$ of predicate order π_i as the expectation of cost.

$$\bar{C}_{\pi_i}(e) = \sum_{j=1}^{m-1} (j * (1 - s_i^j) * \prod_{k=1}^{j-1} s_i^k) + m * \prod_{k=1}^{m-1} s_i^k \quad (3)$$

Optimal Estimated Predicate Order. Given a predicate order $\pi_i = \langle p_i^1, p_i^2, ..., p_i^m\rangle$ and every predicate p_i^j's conditional selectivity, let's denote π^o as the optimal estimated predicate order, where $\bar{C}_{\pi^o}(e) \le \bar{C}_{\pi_i}(e)$ holds for any other order π_i.

3.2 Predicate Order Based Algorithm

To choose the optimal predicate order, a native way is to take a random order in each iteration whom we name as RDM. A further approach is CSEL[4], which samples a part of entities and sorts predicates by their selectivities in descending order. Those two methods both have limitations: the first cannot use selectivities and the second has large sampling cost. Thus we propose a predicate order based framework to estimate the optimal order iteratively.

3.2.1 E-Tree. We model the relationship of predicates and their selectivities as a multiway tree model.

Definition 3.2 (Expectation Tree). Given a query Q, we build a multiway tree $\{\mathcal{V}, \mathcal{E}\}$ named E-Tree.

Node. Each node v_i^j in E-Tree stores a predicate p_i^j with two fields: (1) $\hat{n}_i^j$ is the number of times p_i^j has been answered by workers; (2) $\bar{n}_i^j$ is the number of times p_i^j is labeled as Yes by workers;

Algorithm 1: A Predicate-Order Based Framework

Input: $\mathbb{E} = \{e_1, e_2, ..., e_n\}$, $Q = \{p_1, p_2, ..., p_m\}$
Output: Results set $\mathcal{R}$

```
1  Construct an empty E-Tree T;
2  for every e ∈ E do
3      Estimate the optimal predicate order π^o;
4      for every p_i^j ∈ π^o do
5          if majority workers vote it as No then
6              Label e as NEG and Break;
7      if all predicates are labeled as Yes then
8          Label e as POS and put into R;
9      Extend and update E-Tree T;
10 return R;
```

Edge. For every possible adjoining predicates pair p_i^j and p_i^{j-1} in predicate order π_i, there is a directed edge e_i^j from v_i^{j-1} to v_i^j.

Let $f_i^j = \frac{\bar{n}_i^j}{\hat{n}_i^j}$ denote the frequency that p_i^j is labeled as Yes when all its previous predicates are labeled as Yes. With the growth of $\hat{n}_i^j$, f_i^j is asymptotically close to s_i^j. So we use f_i^j as the estimation of conditional selectivity s_i^j. Figure 2 shows an example of E-Tree for query Q_1, which is not a complete tree since we dynamically build it to reduce space. Every node stores a predicate of p_1, p_2, p_3 and p_4 and every path from root to leaf is a prefix of predicate order based on definition of E-Tree.

3.2.2 Pows-S *Framework.* Based on the definition of E-Tree, we can estimate a predicate's conditional selectivity from it. Then we could compute the estimated cost $\bar{C}_{\pi_i}$ if given a predicate order π_i, whom we will use to estimate optimal predicate order π^o. So we propose our Pows-S framework. In each iteration of this framework, we fetch an entity e and estimate the optimal order π^o to check entity, and then update the E-Tree with results from workers.

Algorithm 1 shows the pseudo code for Pows-S framework. It firstly constructs an empty E-Tree $\mathcal{T}$ (line 1). Then for every entity $e \in \mathbb{E}$, it will estimate the optimal order π^o(line 3). In the next step, it takes predicate from π^o one by one to ask workers to answer it. (1) If majority workers vote it as No, we do not need to check remained predicates since we could label e as NEG now(line 6). (2) If majority workers vote Yes, we check the next predicate.

After all predicates have been checked and we label it as POS and put into result set $\mathcal{R}$ if all of them are labeled as Yes (line 7). Then we take answers got from workers to update E-Tree $\mathcal{T}$ (line 9). Obviously, Pows-S framework can reduce the cost as we always choose the current optimal predicate order. For example, considering the example in Figure 1, the worst case needs to ask 3 + 4 + 3 + 4 + 3+3 + 2 + 2 = 24 questions to get the result set. But in the optimal case, if we always check e with the optimal order, we only need to check a predicate for those negative entities and four predicates for positive entities. Thus the number of questions are $6 * 1 + 2 * 4 = 14$. In the next subsection, we will demonstrate how to construct E-Tree and estimate the optimal order.

3.3 E-Tree Construction and Updating

A straightforward way constructs a complete E-Tree with all possible predicate orders before the iterations begin. Since some predicate orders will not be updated in early iterations, there is no need to maintain them at the beginning. Thus we take the dynamic

Algorithm 2: E-Tree Extending and Updating

Input: E-Tree $\mathcal{T}$, entity result $\mathcal{L}_e^{\pi^o}$
1 Compute the predicate set $\mathtt{P}_y$ and $\mathtt{P}_n$;
2 Generate $\mathcal{S}_\pi^p$ from $\mathtt{P}_y$ and $\mathtt{P}_n$;
3 **for** *each predicate order prefix* π_i^p *in* $\mathcal{S}_\pi^p$ **do**
4 Set current node v_c as root;
5 **for** *every* p_i^j *in* π_i **do**
6 **if** p_i^j *is not in* v_c*'s children* **then**
7 Create node with p_i^j and append to v_c;
8 Update v_c as current node;
9 **if** v_c *has not been updated* **then**
10 Increase $\hat{n}_i^j$ by 1;
11 **if** $\mathcal{L}_{e,j}^{\pi^o}$ *is* Yes **then** Increase $\bar{n}_i^j$ by 1;

construction strategy and extend the predicate orders only when needed. In each iteration, we generate potential predicate orders from acquired answers and append them into E-Tree dynamically.

Entity Result. Given a predicate order π_i and an entity e, we use $\mathcal{L}_e^{\pi_i}$ to denote the entity result got from workers, which can be represented as $\mathcal{L}_e^{\pi_i} = \langle \mathcal{L}_{e,1}^{\pi_i}, \mathcal{L}_{e,2}^{\pi_i}, ..., \mathcal{L}_{e,m}^{\pi_i} \rangle$. Each element $\mathcal{L}_{e,j}^{\pi_i}$ is an answer or Unknown which means we haven't checked it. As the domain of predicate is {Yes, No}, thus $\mathcal{L}_{e,j}^{\pi_i}$ is one of {Yes, No, Unknown}.

For example, if π^o is $\langle p_2, p_4, p_1, p_3 \rangle$ and we try to check entity e_3, the entity result will be $\mathcal{L}_{e_3}^{\pi^o} = \langle$Yes, No, Unknown, Unknown$\rangle$.

Predicate Order Prefix. Given a predicate order $\pi_i = \langle p_i^1, p_i^2, ..., p_i^m \rangle$, we denote $\pi_i^p = \langle p_i^1, p_i^2, ..., p_i^k \rangle$ as the prefix of π_i where $0 \le k \le m$.

We generate the prefixes of potential predicate orders that meet the definition of conditional selectivity from the entity result $\mathcal{L}_e^{\pi_i}$. π_i^p is a potential predicate order prefix if it meets those two rules:
(1) First $k-1$ predicates are Yes, $\mathcal{L}_{e,j}^{\pi_i} =$ Yes for $1 \le j \le k-1$;
(2) The last predicate is Yes or No, $\mathcal{L}_{e,k}^{\pi_i} =$ No *or* Yes;

We generate potential predicate order prefixes from $\mathcal{L}_e^{\pi_i}$, and take $\mathcal{S}_\pi^p$ as the set of π^p produced by $\mathcal{L}_e^{\pi_i}$. Let $\mathtt{P}_y$ denote the maximum set where all predicates inside are Yes from $\mathcal{L}_e^{\pi_i}$, and $\mathtt{P}_n$ denote the set where all predicates are No. Since we break checking process once finding a No answer, $\mathtt{P}_n$ has a single predicate at most. We generate $\mathcal{S}_\pi^p$ from $\mathtt{P}_y$ and $\mathtt{P}_n$ by the following steps:
Step 1: Generate $\mathtt{P}_y$'s permutations of length 1 to $|\mathtt{P}_y|$ and put them into $\mathcal{S}_\pi^p$;
Step 2: If $\mathtt{P}_n$ is empty, skip this step. Let's take p^n as the only predicate in $\mathtt{P}_n$. (2.1) For every $\pi_i^p \in \mathtt{P}_y$, append p^n to its end; (2.2) Put $\langle p^n \rangle$ into $\mathcal{S}_\pi^p$.

For instance, $\mathtt{P}_y$ is $\{p_2\}$ and $\mathtt{P}_n$ is $\{p_4\}$ for $\mathcal{L}_{e_3}^{\pi^o}$. In the first step, we get $\mathcal{S}_\pi^p = \{\langle p_2 \rangle\}$. Next, we append p_4 to the end of $\langle p_2 \rangle$ and put $\langle p_4 \rangle$ into $\mathcal{S}_\pi^p$. Finally, $\mathcal{S}_\pi^p$ is $\{\langle p_4 \rangle \langle p_2, p_4 \rangle\}$.

Algorithm 2 shows the pseudo code for E-Tree extending and updating. It firstly computes the positive predicate set $\mathtt{P}_y$ and negative set $\mathtt{P}_n$ (line 1). Then it generates permutations $\mathcal{S}_\pi^p$ with $\mathtt{P}_y$ and $\mathtt{P}_n$ by the above rules(line 2). We enumerate predicate p_i^j in π_i^p and traverse E-Tree from top to down. If the current predicate does not exist, then we create it and append to its parent (line 7). If its result in $\mathcal{L}_e^{\pi_i}$ is Yes, we will both increase $\hat{n}$ and $\bar{n}$ by 1; otherwise, only increase $\hat{n}$ by 1(line 10).

Algorithm 3: Predicate Order Selection: Single

Input: E-Tree $\mathcal{T}$, Query Q
Output: Optimal estimated predicate order π^o
1 **for** *every path in* $\mathcal{T}$ **do**
2 Generate predicate order prefix π_i^p from top to down;
3 Append other predicates in Q to π_i^p to get π_i;
4 **if** $\bar{C}_{\pi_i} < \bar{C}_{\pi^o}$ **then** Choose π_i as π^o;
5 **return** π^o

Figure 2 shows the E-Tree extending and updating process with Q_1 as shown in Figure 1. At first, E-Tree is empty, so we generate random predicate order $\langle p_2, p_4, p_1, p_3 \rangle$ and choose entity e_3. After that, we get entity result $\mathcal{L}_e^{\pi_i} = \langle$Yes, No, Unknown, Unknown$\rangle$ and generate $\mathtt{P}_y$, $\mathtt{P}_n$ as $\{p_2\}$ and $\{p_4\}$ respectively. Thus we generate permutation set as $\mathcal{S}_\pi^p = \{\langle p_4 \rangle, \langle p_2, p_4 \rangle\}$. Firstly we fetch $\langle p_4 \rangle$, since the node with p_4 is not a child of root, we create it and set $\hat{n}$ as 1, $\bar{n}$ as 0. Next we take out $\langle p_2, p_4 \rangle$, since there is no child p_2 for root, no child p_4 for p_2, we create both nodes with p_2 and p_4. Then we update $\hat{n}$ and $\bar{n}$ both to 1 for p_2 and only set $\hat{n}$ for p_4. The E-Tree has 3 nodes after that (iteration-1 in Figure 2). Next we continue the above process and update E-Tree step by step.

3.4 Predicate Order Estimation

3.4.1 Predicate Order Selection. In E-Tree, every root to leaf path is a possible predicate order prefix. To estimate π^o, we traverse every π_i^p and compute its estimated cost. If the length of π_i^p is smaller than m, we randomly append remained predicates to it and get π_i. Since there is no selectivity for appended predicates, we set it as default value 0.5. After computing its estimated cost $\bar{C}_{\pi_i}$, we choose the order with minimal estimated cost as π^o. Algorithm 3 shows details of E-Tree order selection. For example, consider E-Tree of iteration-4 in Figure 2. For the left most branch, i.e., $\pi_i^p = \langle p_1, p_2, p_4 \rangle$, we randomly append p_3 to it and generate π_1 as $\langle p_1, p_2, p_4, p_3 \rangle$. Thus the estimated cost is $\bar{C}_{\pi_1} = (1-0.33) * 1 + 0.33 * (1-1.0) * 2 + 0.33 * 1.0 * (1-0.0) * 3 + 0.33 * 1.0 * 0.0 * 4 = 1.66$. Similarly, we compute $\bar{C}_{\pi_2} = 1.33$, $\bar{C}_{\pi_3} = 3$, $\bar{C}_{\pi_4} = 2$, $\bar{C}_{\pi_5} = 1.5$ (number from left to right). Finally, we take π_2 as π^o as its cost is minimum 1.33.

3.4.2 E-Tree Pruning. Based on Equation 3, those predicates in the front of π_i make more contribution to estimated cost than those in the behind position. If given a query Q, it has $O(m!)$ possible predicate orders. The time and space complexity are unacceptable when m is large. So we limit the height of E-Tree as $\hbar$. Now E-Tree has $\hbar$ layers, and the maximal numbers of possible predicates order is $\mathcal{A}_m^\hbar$. So the time complexity of E-Tree updating and predicates order selection is $O(\mathcal{A}_m^\hbar) \approx O(m^\hbar)$. As we have $|\mathbb{E}|$ entities need to check, Pows-S's time complexity is $O(|\mathbb{E}| * m^\hbar)$. As $\hbar$ is a trade-off between cost and efficiency, we should choose $\hbar$ carefully.

3.4.3 E-Tree Parallel. We parallel Pows-S by grouping entities.

Grouping. Let $\mathcal{B}$ denote a group of entities, where all entities will be checked with the same predicate order π_i. In each iteration, after estimating the optimal predicate order π^o and grouping $|\mathcal{B}|$ entities, we ask workers to answer those predicates in π^o in turn and drop NEG entities until checking all predicates or $\mathcal{B}$ becomes empty. Apparently, the size of $\mathcal{B}$ has influence on the numbers of questions to ask. As in parallel algorithm, we estimate π^o for a set of entities together rather than one by one respectively. And we update E-Tree after collecting a batch of answers rather than updating it immediately. Thus we should choose the group size

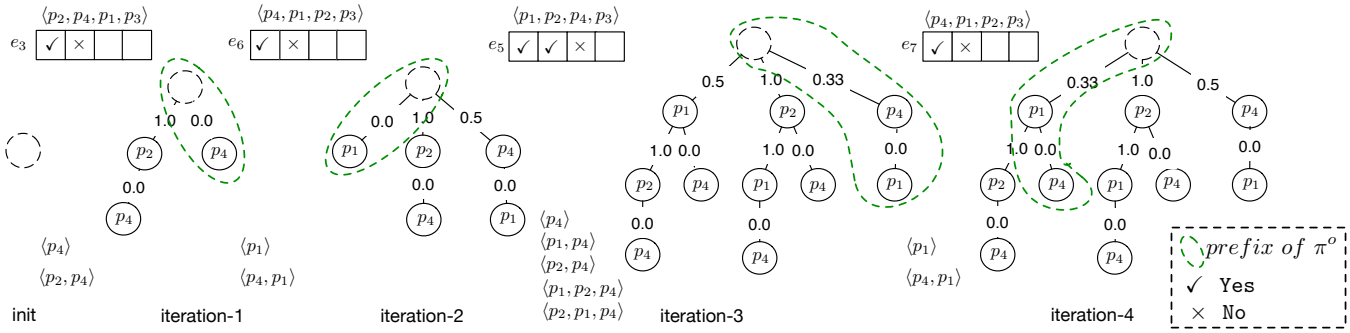

Figure 2: E-Tree Extending and Updating Example.

$|\mathcal{B}|$ carefully. Since we estimate π^o for every $|\mathcal{B}|$ entities, we could merge those predicate orders' prefix to update and update them together. The time complexity can be reduced to $O(\frac{|\mathbb{E}|*m^h}{|\mathcal{B}|})$.

4 MULTIPLE SELECTION QUERY

We extend our techniques to support multiple queries.

4.1 ME-Tree

A native approach is to apply Pows-S to every query independently. However, there may be some duplicated attributes in queries, and it checks them independently which needs to ask a lot of questions. We name it as RAW. Another approach is to merge those predicates with the same attribute into a predicate with multiple values and check them together with a question. Similarly, how to choose predicate order is important. A baseline approach is to choose the predicate order randomly whom we name as RAND. However, it cannot utilize selectivities to reduce cost. Thus we propose our Pows-M to merge queries and estimate optimal predicate order.

Multi-Value Predicate. We extend our predicate as multi-value predicate $\hat{p}$ which can be represented as $\mathcal{A} = (v_1, v_2, ..., v_{m'})$. Each v_i is a possible value and we need workers to answer a single-choice question to determine it. We denote $\mathcal{D}_{\hat{p}}$ as the possible values set of $\hat{p}$, that is the domain of $\hat{p}$. To distinguish with multi-value predicate, we name predicate in section 3 as single-value predicate.

Merged Query. We denote $\widehat{Q}$ as a merged query for $\bar{Q}$, which can be represented as $\widehat{Q} = \{\hat{p}_1, \hat{p}_2, ...\}$. The attribute of $\hat{p}_i$ is a unique attribute for all predicates of $\bar{Q}$, the domain set is a union of all predicate values for attribute $\mathcal{A}_i$.

Taking the multi-query examples in Figure 1, we have 3 queries Q_1, Q_2 and Q_3. Thus $\widehat{Q}$ can be represented as $\widehat{Q} = \{\mathcal{A}_1 = (\texttt{male}, \texttt{female}), \mathcal{A}_2 = (\texttt{white}, \texttt{black}), \mathcal{A}_3 = (\texttt{no}, \texttt{eyeglasses}, \texttt{sunglasses}), \mathcal{A}_4 = \texttt{false}, \mathcal{A}_5 = \texttt{wavy}, \mathcal{A}_6 = \texttt{true}\}$.

Predicate Conflict. We denote predicate conflict on single-value predicate pairs. Given a set of queries $\bar{Q}$ and two single-value predicate p_i and p_j which can be represented as $\mathcal{A}_i = v_i$, $\mathcal{A}_j = v_j$. If all the queries containing p_j also contain a predicate p_i' with $\mathcal{A}_i$ but $v_i \neq v_i'$, we call p_j is conflicting with p_i representing as $p_i \triangleright p_j$. Otherwise, $p_i \not\triangleright p_j$. For example, $(\mathcal{A}_1 = \texttt{male}) \triangleright (\mathcal{A}_5 = \texttt{wavy})$ since only Q_2 includes $(\mathcal{A}_5 = \texttt{wavy})$ and there is a predicate $(\mathcal{A}_1 = \texttt{female})$ in it which has opposite value with $(\mathcal{A}_1 = \texttt{male})$. However, $(\mathcal{A}_5 = \texttt{wavy}) \not\triangleright (\mathcal{A}_1 = \texttt{male})$, as Q_1 and Q_3 contain $\mathcal{A}_1 = \texttt{male}$ but don't contain a predicate with $\mathcal{A}_5$.

Definition 4.1 (Multiple Expectation Tree). Given a merged query $\widehat{Q}$, we build a multi-way tree $\ddot{\mathcal{T}} = \{\mathcal{V}, \mathcal{E}\}$ named ME-Tree.

Node. Every node $\ddot{v}_i^j$ in $\ddot{\mathcal{T}}$ contains a single-value predicate p_i^j with two fields:

(1) $\hat{n}_i^j$ is the number of times p_i^j has been answered;

(2) $\bar{n}_i^j$ is the number of times p_i^j has been labeled as Yes;

Edge. Given two nodes $\ddot{v}_i^j$ and $\ddot{v}_{i'}^{j'}$, if $p_i^j \not\triangleright p_{i'}^{j'}$ and $\ddot{v}_{i'}^{j'}$ is not an ancestor of $\ddot{v}_i^j$, there is a directed edge from $\ddot{v}_i^j$ to $\ddot{v}_{i'}^{j'}$.

Similar with E-Tree, we take $\frac{\bar{n}_i^j}{\hat{n}_i^j}$ to estimate the conditional selectivity for predicate p_i^j. Based on ME-Tree, we could estimate cost of multi-value predicate order and choose the optimal one π^o. For a given multi-value predicate $\hat{p}_i$, its truth must be in two cases: (1) one value $v_i \in \mathcal{D}_{\hat{p}_i}$ (2) None of them. Thus for every predicate in merged query, we always append a other into domain representing other values. Figure 3 shows the ME-Tree for queries in Figure 1. We don't show all nodes for illustration purpose. For example, since $(\mathcal{A}_3 = \texttt{eyeglasses})$ conflicts with all remained predicates but $(\mathcal{A}_1 = \texttt{male})$, $(\mathcal{A}_2 = \texttt{black})$, $(\mathcal{A}_6 = \texttt{true})$. Thus, there are edges from $(\mathcal{A}_3 = \texttt{eyeglasses})$ to those 3 predicates.

4.2 Pows-M Framework

RAW and RAND have their limitations and cannot reduce cost effectively. To address the issue, we propose an effective Pows-M framework to reduce the number of questions. Algorithm 4 shows pseudo code for Pows-M. It firstly constructs ME-Tree. For each entity e, it puts all queries into query set S_Q and estimates the optimal predicate order π^o. When S_Q is not empty and not all predicates have been checked, we continuously pop predicate from π^o and determine whether or not to check. If all remained queries don't contain any attribute $\mathcal{A}_j$ of $\hat{p}_i^j$, we skip it. Otherwise, we ask workers to answer it and update $\mathcal{L}_e^{\pi_i}$. Then for every query Q_i in S_Q, we remove it from S_Q if there is a predicate $p_{i'}^{j'}$ in Q_i with the same attribute but different value compared with p_i^j. If S_Q is not empty finally, we put e into corresponding result sets and update $\ddot{\mathcal{T}}$.

For example, assume we get e_6 as the current entity and optimal estimated predicate order $\pi^o = \langle \hat{p}_3, \hat{p}_4, \hat{p}_1, \hat{p}_5, \hat{p}_2, \hat{p}_6 \rangle$. At the beginning, query set is $S_Q = \{Q_1, Q_2, Q_3\}$. We pop the first predicate $\hat{p}_3$ $(\mathcal{A}_3 = (\texttt{sunglasses}, \texttt{eyeglasses}, \texttt{no}))$ and get no from workers. Since Q_1 and Q_3 have attribute $\mathcal{A}_3$ but different values, we remove them and S_Q is $\{Q_2\}$ now. In the next step, we pop $\hat{p}_4$ and no need to check since S_Q doesn't contain a predicate with $\mathcal{A}_4$. Then we continuously pop $\hat{p}_1$, $\hat{p}_5$, $\hat{p}_2$ and check them. Finally e_6 passes all predicates in Q_2 and thus we put it into R_2. We use 4 single-choice questions to finish all queries. For RAW, it needs 1+1+4=6 questions in the best case and 3+4+1=8 questions in the worst case. And for RAND, it requires 6 questions to finish it (with predicate order $\langle \hat{p}_5, \hat{p}_6, \hat{p}_4, \hat{p}_2, \hat{p}_1, \hat{p}_3 \rangle$). It's obvious Pows-M can reduce cost.

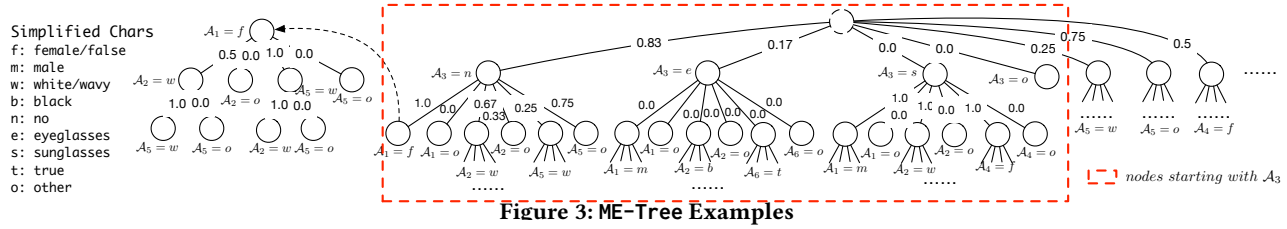

Figure 3: ME-Tree Examples

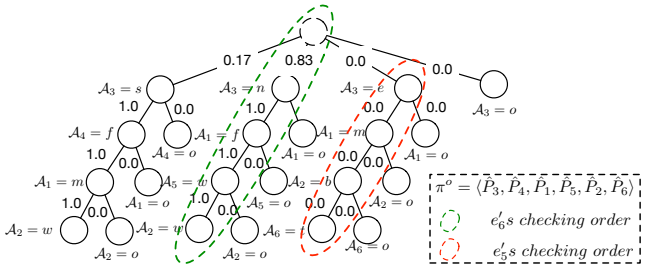

Figure 4: Possible predicates checking orders.

Algorithm 4: Pows-M Framework

Input: $\mathbb{E} = \{e_1, e_2, ..., e_n\}$, $\widehat{Q} = \{Q_1, Q_2, ..., Q_t\}$
Output: Result sets $\bar{R}$

1 Construct ME-Tree $\ddot{\mathcal{T}}$;
2 **for** *every* $e \in \mathbb{E}$ **do**
3 Init query set S_Q from $\bar{Q}$;
4 Estimate π^o and init entity result $\mathcal{L}_e^{\pi^o}$;
5 **while** *there exist queries in* S_Q *and* $\hat{p}_i^j \in \pi^o$ **do**
6 Pop the first predicate $\hat{p}_i^j$ in π^o;
7 **if** $\hat{p}_i^j$ *should be checked* **then**
8 Get workers' result with $\hat{p}_i^j$ and update $\mathcal{L}_e^{\pi^o}$;
9 Traverse S_Q and pop conflicting queries with $\hat{p}_i^j$;
10 **if** S_Q *is not empty* **then** Put e to $\mathcal{R}_i$ with $Q_i \in S_Q$;
11 Update ME-Tree $\ddot{\mathcal{T}}$;
12 **return** $\bar{R}$;

Algorithm 5: ME-Tree Construction

Input: Merged Query $\widehat{Q}$
Output: ME-Tree $\ddot{\mathcal{T}}$

1 Create root node for $\ddot{\mathcal{T}}$;
2 Init queue Q as Q = {root};
3 **while** Q *is not empty* **do**
4 Pop current node $\ddot{v}_i^j$ from Q;
5 **for** *every predicate* $p_{i'}^{j'}$ *in* $\bar{Q}$ **do**
6 **if** $p_i^j \not\vdash p_{i'}^{j'}$ *and* $\ddot{v}_{i'}^{j'}$ *is not* $\ddot{v}_i^j$*'s ancestor* **then**
7 Create node $\ddot{v}_{i'}^{j'}$ with $p_{i'}^{j'}$ as $\ddot{v}_i^j$'s child;
8 Push $\ddot{v}_{i'}^{j'}$ into Q;
9 **return** $\ddot{\mathcal{T}}$

4.3 ME-Tree Construction and Updating

4.3.1 ME-Tree *Construction.* Based on definition 4.1, we show ME-Tree construction in Algorithm 5. It firstly inits the root and puts into an empty queue (line 2). Then it pops every node in queue and traverses queries to find non-conflicting predicates. After that, it will create a new node for every predicate and put into queue.(line 7-8). Figure 3 shows parts of ME-Tree for 3 example queries after checking 8 entities and the subtree in red rectangle is the part with $\mathcal{A}_3$ = (no, eyeglasses, sunglasses) as the first predicate.

Algorithm 6: Predicate Order Selection: Mutiple

Input: ME-Tree $\ddot{\mathcal{T}}$, Queries $\bar{Q}$
Output: Optimal estimated predicate order π^o

1 **for** *every predicate order* π_i **do**
2 Extract possible checking paths for π_i;
3 **for** *every path* **do**
4 Accumulate probability times questions to $\bar{C}_{\pi_i}$;
5 **if** $\bar{C}_{\pi_i}$ *is least* **then** Choose π_i as π^o;
6 **return** π^o

4.3.2 ME-Tree *Updating.* Every value in entity result $\mathcal{L}_e^{\pi_i}$ must be in $\{d_i, \texttt{other}, \texttt{unknown}\}$ where $d_i \in \mathcal{D}_{p_i^j}$. Thus we extract possible single predicate orders and update them. Let P_l denote the set of labeled single predicates. We enumerate all possible any-length's predicate orders' prefix and use them to update. For any prefix π_i^p, we update it in the same way with Pows-S. The only difference is we always increase $\hat{n}_i^j$ and $\hat{n}_i^j$ by 1 without considering the result label. For example, taking the predicate order $\langle \hat{p}_3, \hat{p}_4, \hat{p}_1, \hat{p}_5, \hat{p}_2, \hat{p}_6 \rangle$ and e_6, result will be $\mathcal{L}_e^{\pi_i} = \langle$no, unknown, female, wavy, white, unknown$\rangle$. Thus labeled predicate set is $\{\mathcal{A}_3 = \texttt{no}, \mathcal{A}_1 = \texttt{female}, \mathcal{A}_5 = \texttt{wavy}, \mathcal{A}_2 = \texttt{white}\}$, and we enumerate all possible predicate orders' prefix as $\mathsf{P}_l = \{\langle \mathcal{A}_3 = \texttt{no} \rangle, ..., \langle \mathcal{A}_3 = \texttt{no}, \mathcal{A}_1 = \texttt{female} \rangle, ..., \langle \mathcal{A}_3 = \texttt{no}, \mathcal{A}_1 = \texttt{female}, \mathcal{A}_5 = \texttt{wavy} \rangle ..., \langle \mathcal{A}_3 = \texttt{no}, \mathcal{A}_1 = \texttt{female}, \mathcal{A}_5 = \texttt{wavy}, \mathcal{A}_2 = \texttt{white} \rangle\}$. Then we update every predicate order prefix in P_l. We can limit the ME-Tree height to reduce the complexity.

4.4 Predicate Order Selection

Every path in ME-Tree is a possible real predicate checking order since we may skip some predicates based on the answers. Taking the predicate order $\pi^o = \langle \hat{p}_3, \hat{p}_4, \hat{p}_1, \hat{p}_5, \hat{p}_2, \hat{p}_6 \rangle$ as example. If we get sunglasses from workers for $\hat{p}_3$, we will skip $\hat{p}_4$ because of conflicting. Thus given a predicate order, we can enumerate all possible paths of it and collect their probability and cost. After that, we could use them to estimate the cost of predicate order π_i. Figure 4 shows all possible paths for π^o. The path in green rectangle is the real checking path for e_6, and red rectangle is for e_5. Algorithm 6 shows how to estimate the optimal predicate order. For every predicate order, it firstly extracts possible checking paths from ME-Tree and computes its estimated cost. It finally chooses the one with least cost as the optimal order. For example, path $\langle \mathcal{A}_3 = \texttt{sunglasses}, \mathcal{A}_4 = \texttt{false}, \mathcal{A}_1 = \texttt{male}, \mathcal{A}_2 = \texttt{white} \rangle$'s probability is $0.17 * 1.0 * 1.0 * 1.0 = 0.17$ and needs 4 questions, $\langle \mathcal{A}_3 = \texttt{no}, \mathcal{A}_1 = \texttt{female}, \mathcal{A}_5 = \texttt{wavy}, \mathcal{A}_2 = \texttt{white} \rangle$'s probability is $0.83 * 1.0 * 1.0 * 1.0 = 0.83$ and also needs 4 questions. The probabilities of all remained paths are 0. The estimated cost for order π^o is 0.17*4+0.83*4=4.

5 TOLERATING ERRORS

In this section, we use Bayes Voting to aggregate answers based on estimated workers' weight in 5.1 and tolerate errors in 5.2.

$\mathcal{M}^{\cdot,P}$	$\begin{bmatrix}0.6 & 0.4\\0.45 & 0.55\end{bmatrix}$	$\begin{bmatrix}0.65 & 0.35\\0.3 & 0.7\end{bmatrix}$	$\begin{bmatrix}0.75 & 0.25\\0.2 & 0.8\end{bmatrix}$	$\begin{bmatrix}0.8 & 0.2\\0.3 & 0.7\end{bmatrix}$	$\begin{bmatrix}0.8 & 0.2\\0.15 & 0.85\end{bmatrix}$
Workers	w_1	w_2	w_3	w_4	w_5
$\mathcal{L}$	Yes	Yes	Yes	No	No

Figure 5: Workers' answers and quality matrix.

5.1 Bayes Voting Framework

We assign each predicate to multiple workers and aggregate their answers. Traditional method uses `Majority Voting` to aggregate answers, and we take `Bayes Voting` [16] to maximize the worker's quality utilization. In this paper, we leverage `EM` algorithm to compute workers weight. We will reassign workers' weight after finishing a batch of entities represented as τ_e since it is time-consuming process. To estimate workers' weight distribution, we construct confusion matrix for workers on different predicates.

Confusion Matrix. Confusion matrix is used to model a worker's quality for answering single-choice tasks. Suppose predicate p's domain set is $\mathcal{D}_p$ with size ζ, then the confusion matrix $\mathcal{M}^{w,p}$ is an $\zeta \times \zeta$ matrix. In $\mathcal{M}^{w,p}$, the i-th ($1 \le i \le \zeta$) row, that is $\left[\mathcal{M}^{w,p}_{i,1}, \mathcal{M}^{w,p}_{i,2}, ..., \mathcal{M}^{w,p}_{i,\zeta}\right]$, represents the probability distribution of worker w's possible answers when predicate p's truth is i-th value in $\mathcal{D}_p$, that is d_i. Every element in $\mathcal{M}^{w,p}_{i,j}$ means the probability that worker w gives j-th domain d_j when the truth for task is d_i. For example, the domain for crowd selection is $\mathcal{D}_p = \{\texttt{Yes}, \texttt{No}\}$. An example confusion matrix is $\mathcal{M}^{w,p} = \begin{bmatrix}0.6 & 0.4\\0.45 & 0.55\end{bmatrix}$. $\mathcal{M}^{w,p}_{1,2} = 0.4$ means that if the truth for a task is `Yes`, the probability is 0.4 that the worker gives answer `No`.

Suppose we get workers' labels set for predicate p and represent it as $\mathcal{L}_p = \{l_i, l_2, ..., l_\lambda\}$. It means there are λ workers providing answers for p. For every domain $d_i \in \mathcal{D}_p$, the probability that ground-truth is d_i can be computed as blow based on `Bayes Voting`.

$$\Pr(d_i|\mathcal{L}_p) = \frac{\Pr(\mathcal{L}_p|d_i)}{\sum_{d_j \in \mathcal{D}_p} \Pr(\mathcal{L}_p|d_j)} \tag{4}$$

We have estimated worker's weight as confusion matrix $\mathcal{M}^{w,p}$, therefore, the probability $\Pr(\mathcal{L}_p|d_i)$ can be computed as:

$$\Pr(\mathcal{L}_p|d_i) = \prod_{l_k = d_i} \mathcal{M}^{w,p}_{i,i} \prod_{l_k = d_j} \mathcal{M}^{w,p}_{i,j} \tag{5}$$

Based on equations 4 and 5, we can compute the probability that the truth is d_i from our observed answers $\mathcal{L}_p$.

Figure 5 shows an example of workers' confusion matrix and their answers. The domain of predicate is $\mathcal{D}_p = \{\texttt{Yes}, \texttt{No}\}$, and 5 workers provide answers. By `Majority Voting`, we aggregate the truth is `Yes` since there are 3 `Yes` s of 5 answers. Based on equation 5, we compute the probabilities $\Pr(\texttt{Yes}|\mathcal{L}_p) = 0.6 * 0.65 * 0.75 * 0.2 * 0.2 = 0.0117$ and $\Pr(\texttt{No}|\mathcal{L}_p) = 0.7 * 0.85 * 0.45 * 0.3 * 0.2 = 0.0161$. We get $\Pr(\texttt{Yes}|\mathcal{L}_p) = \frac{0.0117}{0.0117+0.0161} = 0.42$ and $\Pr(\texttt{No}|\mathcal{L}_p) = \frac{0.0161}{0.0117+0.0161} = 0.58$ after normalization. Since $\Pr(\texttt{No}|\mathcal{L}_p)$ is greater than $\Pr(\texttt{Yes}|\mathcal{L}_p)$, we aggregate the result as `No`.

Algorithm 7 shows details of our `Pows-S+` with error tolerance.

5.2 Error Tolerating

Predicate Confidence. Let $z_e^{p_i^j}$ denote the confidence that predicate p_i^j has been answered correctly on e. We take the maximum probability of all domains as it, thus $z_e^{p_i^j} = \max_{d_k \in \mathcal{D}_{p_i^j}} \Pr(d_k|\mathcal{L}_{p_i^j})$. For example, as $\Pr(\texttt{Yes}|\mathcal{L}_p)$ is 0.42 and $\Pr(\texttt{No}|\mathcal{L}_p)$ is 0.58, $z_e^{p_i^j}$ is 0.58.

Algorithm 7: `Pows-S+` Framework

Input: $\mathbb{E} = \{e_1, e_2, ..., e_n\}$, $Q = \{p_1, p_2, ..., p_m\}$
Output: Results set $\mathcal{R}$

```
1 Construct empty E-Tree;
2 Init M^{w,p} for workers on different predicates.
3 for each entity e ∈ 𝔼 do
4     if the number of checked entities > τ_e then
5         Conduct EM and update M^{w,p};
6     Estimate order π^o and check e with Bayes Voting;
7     if e meet Q then Append e to R;
8 Tolerate errors;
9 return R
```

Algorithm 8: Error-Tolerance: `Pows-S+`

Input: Result set $\mathcal{R}$, $Q = \{p_1, p_2, ..., p_m\}$, Budget $\mathbb{B}$
Output: Result set $\mathcal{R}$

```
1 Sort entities by z_e in ascending order;
2 while there remain budget 𝔹 do
3     Pop the first entity e and get π^o;
4     for p_i^j in π^o do
5         if z_e^{p_i^j} < 0.8 then
6             Ask more questions on p_i^j util z_e^{p_i^j} ≥ 0.8;
7             Reduce budget 𝔹;
8     if L_e^{π^o} changes then Update result set R;
9 return R
```

We get all answered predicates' confidence after checking entity e, thus we compute e's confidence by every predicate's confidence.

Entity Confidence. Let z_e denote entity e's confidence, so we compute it as:

$$z_e = \prod_{p_i^j \in \pi_i} z_e^{p_i^j} \tag{6}$$

To tolerate errors in crowdsourced selection query, we conduct extra budget to improve the quality. Our general idea is to let more workers answer low-confidence entities. Here we propose a greedy confidence-based approach. Algorithm 8 shows the pseudo code for our error tolerance approach. Firstly, we sort all entities by its confidence in ascending order, where low confidence entities are placed in the front(line 1). Based on Equation 6, there must be some low confidence predicates in those entities. Every time we take an entity e from entities set, and ask more questions on its low confidence predicates until $z_e^{p_i^j}$ is greater than given threshold or reaches the maximal number of questions for a predicate(line 3-6). We reduce the budget $\mathbb{B}$ once finishing a predicate(line 7). If the e's answer changes, we will update the result set(line 8).

We take error tolerance strategy on crowd selection query and it's similar to extend it to multiple selection queries.

6 EXPERIMENT

6.1 Experimental Setting

Datasets. We use two real-world datasets to evaluate different algorithms. (1) `LFW` [12] is a face dataset consisting of 13143 persons' upper part images. It has 72 binary attributes, and we merge those attributes with same type (For example, merging is_male, is_female to sex) and choose 12 attributes. (2) `Cloth` [2] is a dataset of cloth attributes, which contains 1814 images and 27 attributes. Similar to `LFW`, we merge them and choose 11 attributes. In simulation

experiments, we use all images with their ground-truth and sample 500 images in real experiments. Table 1 shows the details.

Baselines. We compare Pows-S with RDM, CSEL(sample rate is 10%) and two other baselines. (1) OPT. It assumes we always know the real optimal order which means we only need 1 question if the entity is NEG; otherwise m questions. It's the theoretical lower bound. (2) BOD. It sorts predicates by selectivities (which are assumed to be known) and takes it as the optimal predicate order.

CrowdFlower Setting. We use CrowdFlower as the crowdsourcing platform. We assign each question to 3 workers and pack every 5 questions into a unit paying 5 cents for it.

Evaluation Metrics. We compare the number of questions and the quality in different approaches. For quality, we use F-measure. Let S_T denote the exact result set, S_P denote the result set of an algorithm. The precision is $p = \frac{|S_T \bigcap S_P|}{|S_P|}$, the recall rate is $r = \frac{|S_T \bigcap S_P|}{|S_T|}$, and the F-measure is $\frac{2pr}{p+r}$.

Worker Generation. To show the effectiveness of our method, we conduct simulation experiments and generate workers with quality 70%, 80% and 90%. We assume the accuracy of workers fit Guassian Distribution and generate workers' accuracy based on average values of $\mu = 70\%, 80\%, 90\%$, and variance $\sigma = 0.15$.

Queries Generation. For single selection problem, we randomly generate 5 queries with length from 4-8 and take their average as the final result. For multiple selection problem, we generate 8 queries with length 4 and execute those queries for 5 times.

Tree Height. In the experiments, we limit E-Tree and ME-Tree's height to 4 simultaneously.

6.2 Evaluating Single Selection

We compare our algorithm Pows-S with three baseline methods (OPT, BOD, RDM) and one state-of-the-art approach CSEL in simulation experiments, and compare with RDM and CSEL in real experiments.

6.2.1 Simulation Exp: Evaluating Worker Accuracy. Taking into account that workers' accuracy in real experiments only reflects their historical accuracy, we conduct simulation experiments with the exact accuracy. Assuming the ground truth is known, we generate workers whose accuracy is between 70%-80%, 80%-90% and above 90%. Figures 6-7 show the simulation results.

#Questions. We compare Pows-S with OPT and BOD. Firstly, Pows-S has similar performance with BOD even without knowing selectivities in advance, and it takes about 25% more monetary cost than OPT(the theoretical lower bound). Compared with BOD, which is the best solution with knowing all predicates' selectivities, Pows-S can estimate selectivities precisely and take conditional selectivities into consideration, which is a significant improvement. Compared with OPT on LFW, both BOD and Pows-S need 25% more cost, but Pows-S does not need know the selectivities. At the same time, CSEL needs 52% more cost, and it's 72% for RDM.

Quality. Firstly, those methods always get the similar performance with the same workers' accuracy. That's because those methods take the similar way to check entities. Secondly, the F-measure is not so high when workers' accuracy is 70% since workers can not provide accurate answers. Finally, the relationship between F-measure and workers' quality is not linear. Although the probability without errors declines exponentially with the growth of numbers of questions. As some of those errors has no influence on the final result, the quality does not decline exponentially.

6.2.2 Real Exp: Evaluating Worker Accuracy. It's impossible to know predicates' selectivities in advance, so we can't conduct BOD in real experiments. As we don't know the optimal predicate order before asking workers, it's impossible to evaluate OPT. In CrowdFlower, we can specify worker's quality by choosing worker's level. We select three groups of workers, 70%-80%(Level 1), 80%-90%(Level 2) and above 90%(level 3). For every group of workers, we ask them to answer our questions and compare different approaches. We make the following observations from Figures 8-9.

	#Images	#Binary Attrs	#Attrs	#Workers/Pred
LFW	13143	72	12	3
Cloth	1814	27	11	3

Table 1: Two real Datasets.

#Questions. Our method Pows-S asks fewer questions than the state-of-the-art method and the baseline method since our method can make the best of conditional selectivities and does not need to sample entities which is expensive. CSEL can also utilize selectivities to sort predicates, but it only uses the independent selectivities and spends a lot of cost on sampling. RDM does not need sample, but it cannot maximize the use of selectivities. For example, RDM requires 2671 questions on LFW, and CSEL requires 2313 questions. But the number of questions for Pows-S is only 1995. Thus our method saves 25% monetary cost than RDM and saves 14% than CSEL. On Cloth, our method saves 26% and 16% cost than RDM and CSEL.

Quality. Pows-S, CSEL and RDM get the similar F-measure under the same worker accuracy as we mentioned in simulation experiments. All those methods can get over 80% F-measure even workers' accuracy is 70% since workers are able to provide high-quality answers in real world on those simple judgement questions.

6.3 Evaluating Multiple Selection

6.3.1 Simulation Exp: Evaluating Worker Accuracy. We compare Pows-M with two baseline methods RAW, RAND in simulation experiments. Figure 10-11 show the results of different approaches.

#Questions. Firstly, Pows-M outperforms RAW and RAND, because (1) Pows-M utilizes the selectivities to estimate the optimal predicate order; (2) Pows-M also merges several judgement questions to a single choice question when checking an attribute, which will reduce cost. For Cloth, Pows-M only needs 31848, 29236 and 26206 questions under workers accuracy 70%, 80%, 90% respectively, and RAW requires 50044, 48165 and 44440 questions. Compared with RAW, Pows-M saves 40% monetary cost because it considers integrated selectivities and merges questions. Compared with Pows-M, RAND still needs 30% more monetary cost although it also merges questions.

Quality. Firstly, those methods all get the similar F-measure under the same worker accuracy, because they always take the same checking method. Secondly, similar with the crowdsourced selection problem, the questions we ask workers to answer are simple single choice questions. So workers provide high quality answers.

6.3.2 Real Exp: Evaluating Worker Accuracy. Figure 12-13 shows results of real experiments.

#Questions We have similar observation with simulation results.

Quality Since real-world's workers have high accuracy and our questions are simple single-choice questions, F-measure in real experiments is higher than simulation experiments. Pows-M, RAW and RAND methods all get above 80% F-measure when setting workers' accuracy as 70%. When workers' accuracy is set as 90%, F-measure of those methods reaches above 90%.

6.4 Evaluating Error Tolerance

Settings. We evaluate Pows-S and Pows-S+ in simulation experiments and real experiments. We take $\mu_{\mathbb{B}}$ as the percentage of additional questions to tolerate errors. When we vary the percentage

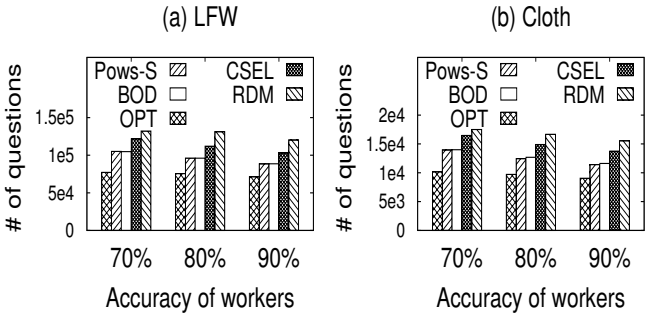

Figure 6: Selection Query Cost (Simulation Experiments).

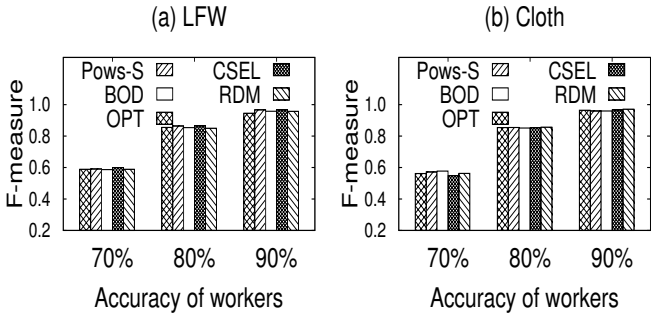

Figure 7: Selection Query Quality (Simulation Experiments).

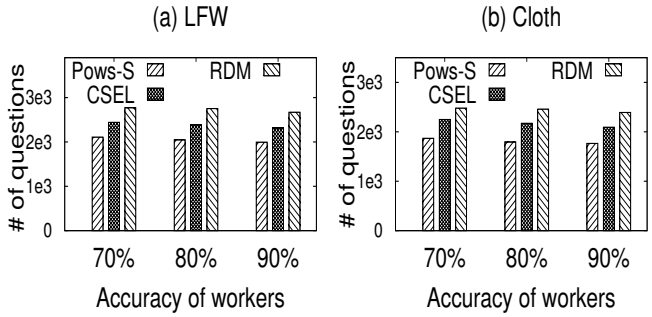

Figure 8: Selection Query Cost (Real Experiments).

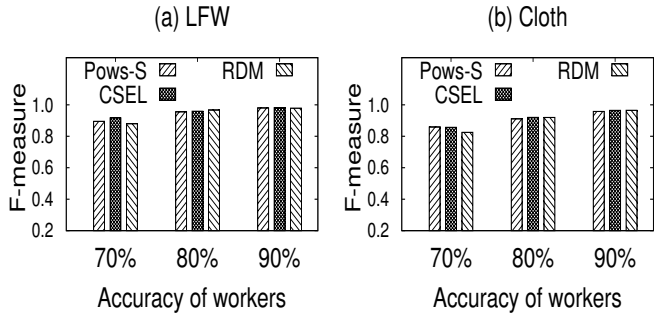

Figure 9: Selection Query Quality (Real Experiments).

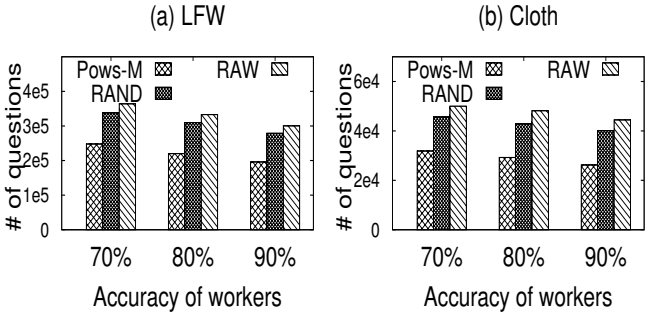

Figure 10: Multiple Selection Cost (Simulation Experiments).

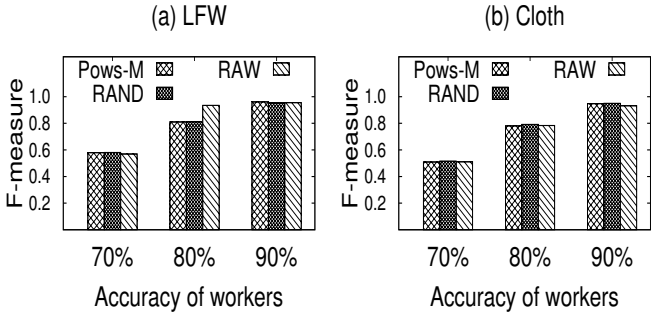

Figure 11: Multiple Selection Quality(Simulation Experiments).

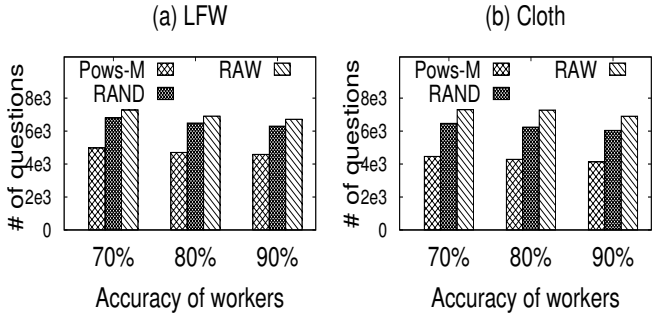

Figure 12: Multiple Selection Cost (Real Experiments).

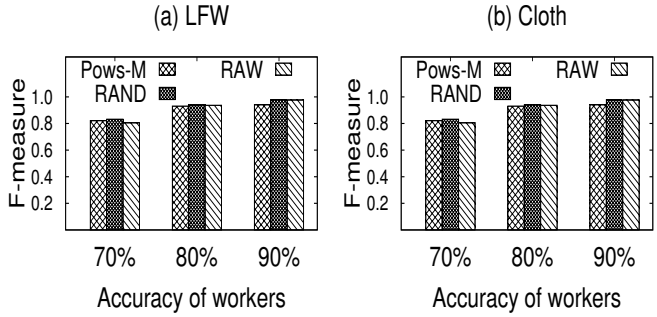

Figure 13: Multiple Selection Quality (Real Experiments).

of budget, we use the workers' accuracy as 80%. When varying accuracy of workers, we set budget percentage $\mu_{\mathbb{B}}$ as 6%, which will be showed as a proper rate in our experiments. Figures 14-17 show the results. We make the following observations. Firstly, Pows-S+ significantly outperforms Pows-S with various budget percentages both in simulation and real experiments. In the simulation experiments, when we use 2% additional cost to tolerate errors, Pows-S+ has 3% improvement than Pows-S. With the improvement of $\mu_{\mathbb{B}}$, Pows-S+ is more effective. When we set the budget percentage as 10%, the improvement is 9.4%. On Cloth, it has 11% improvement when we use 10% additional budget. In both datasets, if we use 10% additional budget to tolerate errors, the final F-measure will reach 95% under workers' accuracy 80%. Secondly, the improvement with different budget percentage in the real experiments is not so significant as the simulation experiments. This is because (1) we always get the high confidence answers from workers. So the confidence of every task is always high, we don't have too many low-confidence tasks to improve; (2) The quality without error tolerance is already

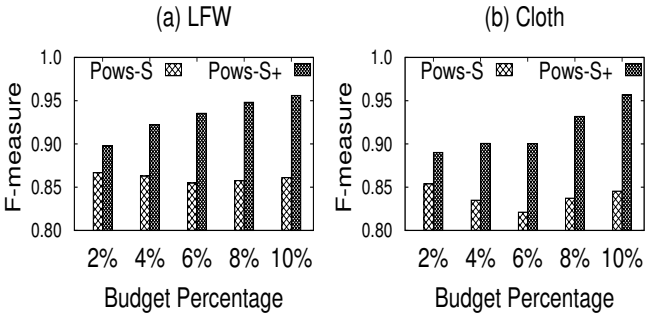

Figure 14: Error Tolerance (Simulation Experiments).

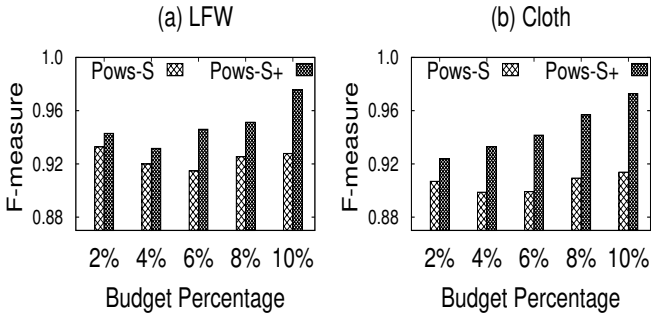

Figure 15: Error Tolerance (Real Experiments).

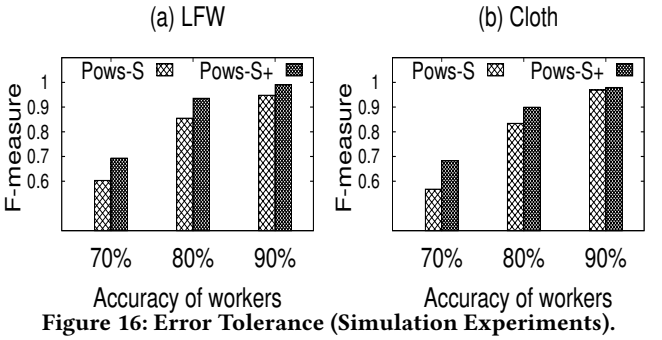

Figure 16: Error Tolerance (Simulation Experiments).

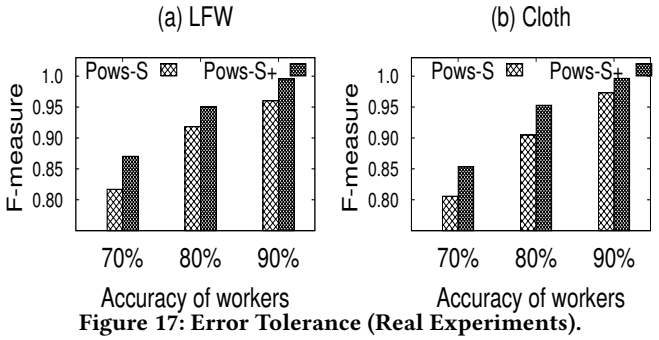

Figure 17: Error Tolerance (Real Experiments).

high, so it's hard to improve more significantly. Finally, when varying workers' quality, the improvement decreases with the growth of workers quality both in simulation and real experiments. For `Cloth`, when workers accuracy is 70%, `Pows-S+` is 10% higher than `Pows-S`; when workers accuracy is 90%, the improvement becomes 3%, because (1) It's hard to find the low confidence entities with high workers quality; (2) The workers accuracy is high, so it's hard to improve. Generally, `Pows-S+` significantly improves quality.

7 CONCLUSION

We studied the crowdsourced selection problem with multiple predicates. We proposed a predicate-order based crowdsourced selection framework. We defined a predicate order based on the conditional selectivities. We proposed an expectation-tree model to estimate conditional selectivities and predicate orders's cost. We extended our framework to answer multiple selection queries and developed confidence based error-tolerant algorithm. Experimental results show that our method outperforms existing algorithms.

Acknolwedgement. This work was supported by 973 Program of China (2015CB358700), NSF of China (61632016,61373024,61602488, 61422205,61472198), and FDCT/007/2016/AFJ.

REFERENCES

[1] 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.
[2] H. Chen, A. Gallagher, and B. Girod. Describing clothing by semantic attributes. *ECCV*, pages 609–623, 2012.
[3] J. Fan, G. Li, B. C. Ooi, K.-l. Tan, and J. Feng. icrowd: An adaptive crowdsourcing framework. In *SIGMOD*, pages 1015–1030. ACM, 2015.
[4] J. Fan, M. Zhang, S. Kok, M. Lu, and B. C. Ooi. Crowdop: Query optimization for declarative crowdsourcing systems. *IEEE TKDE*, 27(8):2078–2092, 2015.
[5] Y. Fang, H. Sun, G. Li, R. Zhang, and J. Huai. Effective result inference for context-sensitive tasks in crowdsourcing. In *DASFAA*, pages 33–48, 2016.
[6] J. Feng, G. Li, H. Wang, and J. Feng. Incremental quality inference in crowdsourcing. In *DASFAA*, pages 453–467, 2014.
[7] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72. ACM, 2011.
[8] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD*, pages 385–396. ACM, 2012.
[9] J. M. Hellerstein and M. Stonebraker. *Predicate migration: Optimizing queries with expensive predicates*, volume 22. ACM, 1993.
[10] H. Hu, G. Li, Z. Bao, Y. Cui, and J. Feng. Crowdsourcing-based real-time urban traffic speed estimation: From trends to speeds. In *ICDE*, pages 883–894, 2016.
[11] H. Hu, Y. Zheng, Z. Bao, G. Li, J. Feng, and R. Cheng. Crowdsourced POI labelling: Location-aware result inference and task assignment. In *ICDE*, pages 61–72, 2016.
[12] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.
[13] G. Li. Human-in-the-loop data integration. *PVLDB*, 10(12):2006–2017, 2017.
[14] 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. ACM, 2017.
[15] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. *IEEE TKDE.*, 28(9):2296–2319, 2016.
[16] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. Cdas: a crowdsourcing data analytics system. *VLDB*, 5(10):1040–1051, 2012.
[17] A. Marcus, D. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. In *VLDB*, volume 6, pages 109–120. VLDB Endowment, 2012.
[18] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *VLDB*, 5(1):13–24, 2011.
[19] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Demonstration of qurk: a query processor for humanoperators. In *SIGMOD*, pages 1315–1318. ACM, 2011.
[20] 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. ACM, 2012.
[21] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *CIKM*, pages 1203–1212. ACM, 2012.
[22] H. Park, H. Garcia-Molina, R. Pang, N. Polyzotis, A. Parameswaran, and J. Widom. Deco: A system for declarative crowdsourcing. *VLDB*, 5(12):1990–1993, 2012.
[23] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. An overview of the deco system: data model and query language; query processing and optimization. *SIGMOD Record*, 41(4):22–27, 2013.
[24] A. D. Sarma, A. Parameswaran, H. Garcia-Molina, and A. Halevy. Crowd-powered find algorithms. In *ICDE*, pages 964–975. IEEE, 2014.
[25] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, pages 989–998. ACM, 2012.
[26] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, pages 229–240. ACM, 2013.
[27] X. Zhang, G. Li, and J. Feng. Crowdsourced top-k algorithms: An experimental evaluation. *PVLDB*, 9(8):612–623, 2016.
[28] Y. Zheng, G. Li, and R. Cheng. DOCS: domain-aware crowdsourcing system. *PVLDB*, 10(4):361–372, 2016.
[29] 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.
[30] 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.