# CDB: Optimizing Queries with Crowd-Based Selections and Joins

Guoliang Li[†], Chengliang Chai[†], Ju Fan[*], Xueping Weng[†], Jian Li[‡], Yudian Zheng[#]
Yuanbing Li[†], Xiang Yu[†], Xiaohang Zhang[†], Haitao Yuan[†]
[†]Department of Computer Science, Tsinghua University, Beijing, China
[‡]Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
[*]DEKE Lab & School of Information, Renmin University of China, China
[#]Department of Computer Science, The University of Hong Kong, Hong Kong, China
{liguoliang,lijian83}@tsinghua.edu.cn;chaicl15@mails.thu.edu.cn,fanj@ruc.edu.cn,ydzheng2@cs.hku.hk

## ABSTRACT

Crowdsourcing database systems have been proposed to leverage crowd-powered operations to encapsulate the complexities of interacting with the crowd. Existing systems suffer from two major limitations. Firstly, in order to optimize a query, they often adopt the traditional tree model to select an optimized table-level join order. However, the tree model provides a coarse-grained optimization, which generates the same order for different joined tuples and limits the optimization potential that different joined tuples can be optimized by different orders. Secondly, they mainly focus on optimizing the monetary cost. In fact, there are three optimization goals (i.e., smaller monetary cost, lower latency, and higher quality) in crowdsourcing, and it calls for a system to enable multi-goal optimization.

To address the limitations, we develop a crowd-powered database system CDB that supports crowd-based query optimizations, with focus on join and selection. CDB has fundamental differences from existing systems. First, CDB employs a graph-based query model that provides more fine-grained query optimization. Second, CDB adopts a unified framework to perform the multi-goal optimization based on the graph model. We have implemented our system and deployed it on AMT, CrowdFlower and ChinaCrowd. We have also created a benchmark for evaluating crowd-powered databases. We have conducted both simulated and real experiments, and the experimental results demonstrate the performance superiority of CDB on cost, latency and quality.

## Keywords

Crowdsourcing; Crowdsourcing Optimization; Crowd-based Selection; Crowd-based Join

## 1. INTRODUCTION

Crowdsourcing aims at soliciting human intelligence to solve machine-hard problems, and has a variety of real applications such as entity resolution. Due to the wide adoption of crowdsourcing, the recent years have witnessed growing research interests in devising crowd-powered operations in the database community, such as join [56, 57, 55, 25, 13], filter [51, 44], max [26, 54], selection [45, 51], top-$k$ [16, 65], sort [42, 14], counting [41], enumeration [53], as well as quality-control techniques [35, 37, 40, 66, 49, 70, 59].

Inspired by traditional DBMS, crowdsourcing database systems, such as CrowdDB [24], Qurk [43], Deco [46], and CrowdOP [23], have been recently developed. On one hand, these systems provide declarative programming interfaces and allow requesters to use an SQL-like language for posing queries that involve crowdsourced operations. On the other hand, the systems leverage the crowd-powered operations to encapsulate the complexities of interacting with the crowd. Under these design principles, given an SQL-like query from a requester, the systems first parse the query into a query plan with crowd-powered operations, then generate tasks to be published in crowdsourcing markets, and finally collect the crowd's inputs for producing the result. However, existing systems suffer from two major limitations.

**Coarse-Grained Query Model.** Existing systems adopt a 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 tuples and limits the optimization potential that different joined tuples can be optimized by different orders. For example, consider the tuples in Figure 1. Each edge between two tuples is a task, which asks the crowd whether they can be joined. A BLUE solid (RED dotted) edge denotes that the two tuples can (cannot) be successfully joined. Before asking the crowd, we do not know the result of each edge. We aim to ask the minimum number of tasks to find the BLUE solid chains as answers. The tree model asks at least 9+5+1=15 tasks for any join order. However, the optimal solution is to ask the 3 RED dotted edges, and the tasks on other 24 edges can be saved. We can observe that the tree model provides a coarse-grained table-level optimization, possibly because the optimization goal of traditional databases is to reduce random accesses. While in crowdsourcing, one optimization goal is to reduce the number of tasks that reflects the monetary cost, and a fine-grained tuple-level optimization is preferred.

**Single-Goal Optimization.** In crowdsourcing, there are three optimization goals, i.e., smaller monetary cost, lower latency, and higher quality. There exists trade-off among these optimization goals. For example, a smaller cost leads

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*SIGMOD'17, May 14-19, 2017, Chicago, IL, USA*

© 2017 ACM. ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3064036

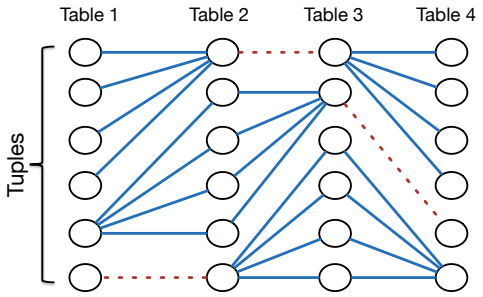


**Figure 1: An example of tuple-level optimization.**

to a lower quality and higher latency, and a higher quality leads to a larger cost and higher latency. Therefore, the multi-goal optimization that effectively balances these goals is highly desirable. However, most of the existing systems (i.e., CrowdDB [24], 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. Thus it calls for a new crowd-powered system to enable the multi-goal optimization.

To address these limitations, we have developed a crowd-powered database system CDB, which provides a declarative query language CQL (an extended SQL) that supports crowd-based data definition and manipulation. CDB has the following fundamental differences compared with existing systems.

**(1) Fine-Grained Query Model.** We propose a graph-based query model that supports crowd-based query optimizations. Given a CQL query, we build a graph based on the query and the data (e.g., Figure 1). This graph model has the advantage of making the tuple-level optimization applicable and providing the potential of the multi-goal optimization at the same time.

**(2) Multi-Goal Optimization in One Framework.** We devise a unified framework to perform the multi-goal optimization based on the graph model, with a focus on crowd-based join and selection. (i) For cost control, our goal is to minimize the number of tasks to find all the answers. For example, our method targets at selecting the three dotted RED edges to ask the crowd. 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. For example, the three dotted RED edges can be asked in parallel. (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.

To summarize, we make the following contributions.
(1) We develop a crowd-powered database CDB (Section 2) and define a declarative query language CQL (Section 3).
(2) We propose a graph-based query model that supports tuple-level optimization, which can save a large amount of cost than the traditional tree model (Section 4).
(3) We introduce a unified multi-goal optimization framework for balancing cost, latency and quality (Section 5).
(4) We have implemented and deployed our system on Amazon Mechanical Turk (AMT), CrowdFlower and ChinaCrowd. We created a benchmark for evaluating crowd-powered databases. We have conducted both simulated and real experiments, and the experimental results demonstrate the performance superiority of CDB on cost, latency and quality (Section 6).

# 2. OVERVIEW OF CDB

We introduce our CDB framework in Section 2.1, and discuss the differences from existing systems in Section 2.2.

## 2.1 CDB Framework

**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 and join (see Section 3 for more details of CQL).

**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 tuple of a table in the CQL and each edge connects two tuples based on the join/selection predicates in the CQL. We utilize the graph model to provide the tuple-level optimization (see Section 4 for details).

**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 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. Note to reduce the cost, we need to utilize the answers of some tasks to infer those of the others, and the inference will lead to more rounds. Thus there is a tradeoff between cost and latency. Our goal is to simultaneously ask the tasked 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. The details on query optimization will be presented in Section 5.

**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 and CrowdFlower. There is a main difference between AMT and CrowdFlower. In CrowdFlower, it does not allow a requester to control the task assignment while AMT has a development model in which the requester can control the task assignment. Thus in AMT, 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

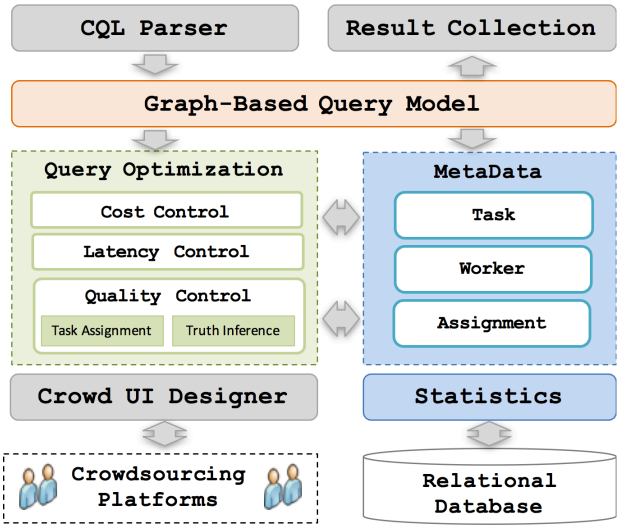


**Figure 2: CDB Framework**

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. Finally, `Result Collection` reports the results to the requester.

## 2.2 Differences from Existing Systems

This section compares our CDB with recent crowdsourcing database systems, CrowdDB [24], Qurk [43], Deco [46], and CrowdOP [23], as illustrated in Figure 3.

*(1) Optimization models.* Query optimization in the existing systems can be classified into rule-based and cost-based. CrowdDB [24] used rule-based optimization, e.g., pushing down selection predicates and determining join order, which may not be able to find the query plan with low cost. The other systems [46, 43, 23] designed cost model that aims to find query plan with the minimum cost. However, these systems still adopted a tree model that selects an optimized table-level join order to optimize the query. As analyzed above, the tree model gives the same order for different joined tuples and limits the optimization potential that different joined tuples can be optimized for different orders. While CDB devises graph-based query optimization to perform a fine-grained tuple-level optimization, which has the potential to save a huge amount of cost.

*(2) Optimization objectives.* Crowdsourcing query optimization should consider trade-offs among cost, latency and quality, because any single-objective optimization, such as smaller cost with lower quality, higher quality with larger latency, etc., is not desirable. As shown in Figure 3, most of existing systems optimized monetary cost, utilized majority voting

| | | CrowdDB | Qurk | Deco | CrowdOP | CDB |
|---|---|---|---|---|---|---|
| Optimized Crowd Operators | COLLECT | √ | × | √ | × | √ |
| | FILL | √ | × | √ | √ | √ |
| | SELECT | √ | √ | √ | √ | √ |
| | JOIN | √ | √ | √ | √ | √ |
| Optimization Objectives | Cost | √ | √ | √ | √ | √ |
| | Latency | × | × | × | √ | √ |
| | Quality | MV | MV | MV | MV | √ |
| Optimization Strategies | Cost-Model | × | √ | √ | √ | √ |
| | Tuple-Level | × | × | × | × | √ |
| | Budget-Supported | × | × | × | × | √ |
| Task Deployment | Cross-Market | × | × | × | × | √ |

**Figure 3: Comparison of crowdsourcing systems.**

(MV) for quality control, and did not optimize the latency. CrowdOP [23] optimized latency by simply considering data dependencies. In contrast, our CDB system develops techniques based on data inference to reduce latency in a more effective way. Considering the quality concern, existing studies leverage existing majority voting or its variants, which is only applicable in single-choice tasks. However, CDB also takes quality into consideration and devises more sophisticated quality-control strategies (i.e., truth inference and task assignment) for either single-choice, multiple-choice, fill-in-blank and collection tasks.

*(3) Optimized crowd operators.* We consider the commonly used crowd-powered operators, and examine whether they are optimized in the existing systems, as shown in Figure 3. CrowdDB [24] optimized `SELECT`, `JOIN`, `COLLECT` and `FILL`. Qurk [43] focused on crowd-powered `SELECT` and `JOIN`. Deco [46] considered more on `FILL` and `COLLECT` (i.e., the *fetch* operator in Deco) while also supporting `SELECT` and `JOIN`. CrowdOP [23] only supported `SELECT`, `JOIN` and `FILL` operators. Compared with these systems, CDB optimizes all of the operators by introducing query language CQL, which can fulfill more crowdsourcing requirements.

*(4) Task deployment.* Existing systems published human-intelligence tasks (HITs) on one individual crowdsourcing market, such as AMT [1], and the results may be affected by the bias of the market. In contrast, CDB has the flexibility of cross-market HITs deployment by simultaneously publishing HITs to AMT [1], ChinaCrowd [2], CrowdFlower [3], etc.

# 3. CROWD SQL IN CDB: CQL

This section presents CQL, a declarative programming interface for requesters to define the crowdsourced data and invoke crowd-powered manipulations. CQL follows standard SQL syntax and semantics, and extends SQL by adding new features to support *crowd-powered* operators, i.e., `CROWDEQUAL`, `CROWDJOIN`, `FILL` and `COLLECT`, which are analogical to `SELECTION`, `JOIN`, `UPDATE` and `INSERT` in SQL. One advantage of this design is that SQL programmers can easily learn CQL and user-friendly SQL query formulation tool can be easily applied for inexperienced users. This section highlights differences between CQL and existing languages of CrowdDB [24], Qurk [43], Deco [46], and CrowdOP [23]. The details of CQL are introduced in Appendix A.

**Crowd-Powered Collection**: CQL works under the open-world assumption and either columns or tables can be crowdsourced by introducing a keyword `CROWD`. CDB introduces two built-in keywords for data collection: `FILL` and `COLLECT`. `FILL` asks the crowd to fill the values of a column, e.g., filling missing values of affiliation of a professor. `COLLECT` asks the crowd to collect a table, e.g., collecting top-100 universities.

One practical issue in FILL and COLLECT is the cleansing of the crowd-collected data, in particular, the entity resolution problem. For example, two workers may contribute the same university with different representations, e.g., "Massachusetts Institute of Technology" and "MIT", or the attribute values like "United States" and "USA". In CDB, this issue is solved by the following methods. First, CDB provides an *autocompletion* interface to the crowd workers to choose an existing value of an attribute, e.g., first typing a character "m" and then choosing "MIT" from a suggested list. If no existing value fits a worker's tuple, the worker can also input a new value, which will be added to the existing value set. This type of interface can reduce the variety of collected tuples or attributes, since workers could choose from a same set of values or learn how to represent new values. Second, there may still exist the cleansing issue, even if the aforementioned autocompletion interface is used. CDB then solves this problem in a crowdsourcing manner, such as the crowdsourced entity resolution techniques [19], by leveraging the query semantics of CQL, which is discussed below.

**Crowd-Powered Query**: CQL defines crowdsourced operations, CROWDEQUAL and CROWDJOIN to solicit the crowd to perform selection and join on crowdsourced or ordinary attributes. Moreover, CQL has a highlight feature of introducing a *budget* mechanism to allow requesters to configure the cost constraint of crowdsourcing. On the one hand, with respect to the collection semantics, it is often unclear how many tuples or values can be collected due to the open-world assumption. Thus, a budget should be naturally introduced to bound the number of COLLECT or FILL tasks. On the other hand, with respect to the query semantics, as the amount of data may be huge, the requester often wants to set a budget to avoid running out of money when evaluating a CQL query.

To achieve this goal, CQL introduces a keyword BUDGET, which can be attached in either collection or query semantics to set the number of tasks. The requester only needs to provide the budget, and our query optimization and plan generation components will design algorithms to fully utilize the budget for producing better results. BUDGET is different from LIMIT in CrowdDB [24] and MinTuples in Deco [46], which introduce a constraint on the number of the results. First, the result number may not reflect the budget, as prices of each crowdsourcing task and number of tasks may vary due to our query plan generation strategies (see Section 5.1.3). Second, the requester may not know how to configure the result number: a higher number results in overrun, while a lower number may not fully utilize the budget.

# 4. GRAPH QUERY MODEL

We first consider that a CQL query only contains join predicates (Section 4.1). Then we discuss how to extend our method to support selection in Section 4.2.

## 4.1 Graph Model for Join Predicates

Given a CQL query with only CROWDJOIN predicates, we construct a graph, where each vertex is a tuple of a table in the CQL query. For each join predicate $\mathcal{T}.\mathcal{C}_i$ CROWDJOIN $\mathcal{T}'.\mathcal{C}_j$ between two tables $\mathcal{T}$ and $\mathcal{T}'$ in the query, we add an edge between two tuples $t_x \in \mathcal{T}$ and $t_y \in \mathcal{T}'$, and the weight of this edge is the *matching probability* that the two values $t_x[\mathcal{C}_i]$ and $t_y[\mathcal{C}_j]$ can be matched, where $t_x[\mathcal{C}_i]/t_y[\mathcal{C}_j]$ is the value of $t_x/t_y$ on attribute $\mathcal{C}_i/\mathcal{C}_j$.

Next, we discuss how to estimate the matching probability. The estimation is trivial for traditional join without crowdsourcing. For example, considering equi-join, we simply set $\omega(e)$ to 1 if $t_x[\mathcal{C}_i] = t_y[\mathcal{C}_j]$, and 0 otherwise. The probability is difficult to estimate for CROWDJOIN, as it is hard to know whether the values can be matched before crowdsourcing. We use the *similarity*-based estimation, following the standard practice in prior studies [57, 56, 58, 60]. We can adopt similarity functions, e.g., edit distance and Jaccard, to compute the similarity between $t_x[\mathcal{C}_i]$ and $t_y[\mathcal{C}_j]$. Usually, the larger similarity $t_x[\mathcal{C}_i]$ and $t_y[\mathcal{C}_j]$ have, the larger probability they can be matched. Thus we can take the similarity as the matching probability $\omega(e)$ [57]. There also exist other sophisticated methods to transform similarities to probabilities based on a training set [60]. Note that if two tuples have very small similarity, i.e., the similarity is smaller than a threshold $\varepsilon$ (e.g., $\varepsilon$=0.3), they have very small likelihood to be matched [56, 57, 58], and we do not keep such edges. We also note that we do not enumerate every tuple pair to compute similarities. Instead, we use existing techniques for similarity join[10, 56, 33] to efficiently identify the pairs with similarities not smaller than threshold $\varepsilon$.

We assume that the matching probabilities of the edges are independent. Note that the independence assumption may not always hold in practice, because probabilities of the edges may have positive/negative correlations. However, such correlations seem to be very complex to compute. Thus in this paper we adopt the independence assumption for simplicity, and leave a more comprehensive study on considering probabilistic correlations as future work.

Next we formally formulate the graph model.

Definition 1 (Graph Query Model). *Given a* CQL *query and a database $\mathcal{D}$, the graph model is a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ such that: (1) For each table $\mathcal{T}$ in the* CQL *query, there is a vertex for each tuple in this table; (2) For each crowd join predicate $\mathcal{T}.\mathcal{C}_i$* CROWDJOIN *$\mathcal{T}'.\mathcal{C}_j$ in the* CQL *query, there is an edge $e$ between $t \in \mathcal{T}$ and $t' \in \mathcal{T}'$ with weight $\omega(e) \geq \varepsilon$, where $\omega(e)$ is the matching probability between cell values $t[\mathcal{C}_i]$ and $t'[\mathcal{C}_j]$ and $\varepsilon$ is a threshold; (3) For each traditional join predicate between $\mathcal{T}.\mathcal{C}_i$ and $\mathcal{T}'.\mathcal{C}_j$ in the* CQL *query, there is an edge between $t \in \mathcal{T}$ and $t' \in \mathcal{T}'$ if the tuples satisfy the join predicate and the weight is 1.*

For example, consider the four tables in Table 1 and the CQL query in Figure 4. The graph is illustrated in the figure.

Given a graph and a CQL query, we want to find the substructure that satisfies every query predicate. Thus a candidate structure is substructure that contains a corresponding edge for every predicate, which is formally defined as below.

Definition 2 (CQL Query Candidate). *Given a* CQL *query with N join predicates, suppose $\mathcal{G}$ is the corresponding graph. A connected substructure of the graph $\mathcal{G}$ with N edges is called a candidate if it contains a corresponding edge for every query predicate in the* CQL *query.*

For example, substructure $(u_1, r_1, p_1, c_1)$ is a candidate as it contains an edge for every query predicate. $(u_1, r_2, p_1, r_1, c_1)$ is not as it contains more than 3 edges. $(u_1, r_2, p_3, c_5)$ is not as it is not connected. $(p_1, r_1, r_2, r_3)$ is not as it misses edges for join predicate Paper.Title CROWDJOIN Citation.Title.

Definition 3 (Invalid Edges). *An edge is an invalid edge if it is not contained in any candidate.*

To check whether an edge is invalid, we can use a depth-first traversal algorithm to check whether there is a candidate from this edge. Obviously, all the invalid edges can be removed from the graph first.

**Table 1: Four Relational Tables (The Attribute Pairs That Can Be Joined Are Highlighted).**

(a) Paper

| | Author | Title | Conference |
|---|---|---|---|
| $p_1$ | Michael J. Franklin | APrivateClean: Data Cleaning and Differential Privacy. | sigmod16 |
| $p_2$ | Samuel Madden | Querying continuous functions in a database system. | sigmod08 |
| $p_3$ | David J. DeWitt | Query processing on smart SSDs: opportunities and challenges. | acm sigmod |
| $p_4$ | W. Bruce Croft | Optimization strategies for complex queries | sigir |
| $p_5$ | H. V. Jagadish | CrowdMatcher: crowd-assisted schema matching | sigmod14 |
| $p_6$ | Hector Garcia-Molina | Exploiting Correlations for Expensive Predicate Evaluation. | sigmod15 |
| $p_7$ | Aditya G. Parameswaran | DataSift: a crowd-powered search toolkit | sigmod14 |
| $p_8$ | Surajit Chaudhuri | Dynamically generating portals for entity-oriented web queries. | sigmod10 |

(b) Researcher

| | *Affiliation* | Name |
|---|---|---|
| $r_1$ | University of California | Michael I. Jordan |
| $r_2$ | University of California Berkery | Michael Dahlin |
| $r_3$ | University of Chicago | Michael Franklin |
| $r_4$ | Duke Uni. | David J. Madden |
| $r_5$ | University of Minnesota | David D. Thomas |
| $r_6$ | University of Wisconsin | David DeWitt |
| $r_7$ | Department of Nutrition | David J. Hunter |
| $r_8$ | University of Massachusetts | Bruce W Croft |
| $r_9$ | University of Michigan | H. Jagadish |
| $r_{10}$ | University of Stanford | Molina Hector |
| $r_{11}$ | University of Cambridge | Nandan Parameswaran |
| $r_{12}$ | Microsoft Cambridge | S. Chaudhuri |

(c) Citation

| | **Title** | Number |
|---|---|---|
| $c_1$ | Towards a Unified Framework for Data Cleaning and Data Privacy. | 0 |
| $c_2$ | Query continuous functions in database system | 56 |
| $c_3$ | ConQuer: A System for Efficient Querying Over Inconsistent Database. | 13 |
| $c_4$ | Webfind: An Architecture and System for Querying Web Database. | 17 |
| $c_5$ | Adaptive Query Processing and the Grid: Opportunities and Challenges. | 27 |
| $c_6$ | Optimal strategy for complex queries | 94 |
| $c_7$ | CrowdMatcher: crowd-assisted schema match | 9 |
| $c_8$ | Exploit Correlations for Expensive Predicate Evaluation | 0 |
| $c_9$ | DataSift: An Expressive and Accurate Crowd-Powered Search Toolkit. | 16 |
| $c_{10}$ | A crowd powered search toolkit | 4 |
| $c_{11}$ | A Crowd Powered System for Similarity Search | 0 |
| $c_{12}$ | Query portals: dynamically generating portals for entity-oriented web queries. | 1 |

(d) University

| | *Name* | Country |
|---|---|---|
| $u_1$ | Univ. of California | USA |
| $u_2$ | Univ. of California Berkery | USA |
| $u_3$ | Univ. of Chicago | USA |
| $u_4$ | Duke Univ. | USA |
| $u_5$ | Univ. of Minnesota | US |
| $u_6$ | Univ. of Wisconsin | US |
| $u_7$ | Depart of Nutrition | US |
| $u_8$ | Univ. of Massachusetts | US |
| $u_9$ | Univ. of Michigan | US |
| $u_{10}$ | Univ. of Stanford | USA |
| $u_{11}$ | Univ. of Cambridge | UK |
| $u_{12}$ | Microsoft | US |

```
SELECT * FROM Paper, Researcher, Citation, University
WHERE Paper.Author CROWDJOIN Researcher.Name AND
Paper.Title CROWDJOIN Citation.Title AND
Researcher.Affiliation CROWDJOIN University.Name
```

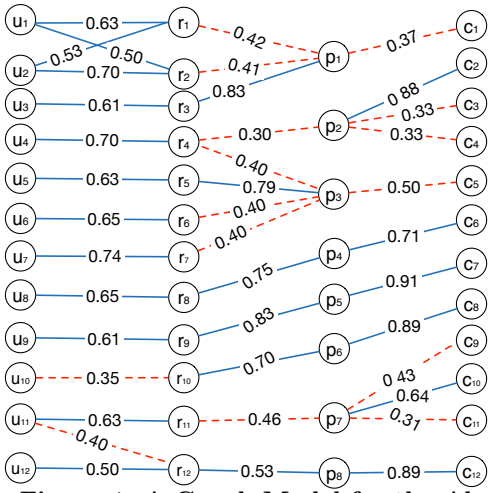


**Figure 4: A Graph Model for the Above CQL.**

In the graph, each crowd edge has a probability (i.e., the weight) to be connected or disconnected before we ask the crowd. Thus we want to ask the crowd to check each edge w.r.t. a predicate such that whether the two cell values satisfy the predicate. If yes, it is connected and we mark it Blue (solid edge); otherwise it is disconnected and we mark it Red (dotted edge). Note we directly mark the edges w.r.t. traditional join predicate Blue without needing to ask the crowd. Then the candidates with $N$ Blue edges are answers.

Definition 4 (CQL Query Answer). *Given a CQL query with $N$ join predicates, a candidate is an answer if each edge in the candidate is* Blue.

For example, candidate $(u_1, r_1, p_1, c_1)$ is not an answer as $(p_1, c_1)$ is Red. There are three answers, i.e.,$(u_{12}, r_{12}, p_8, c_{12})$, $(u_8, r_8, p_4, c_6)$ and $(u_9, r_9, p_5, c_7)$ in Figure 4.

To find all the answers, a straightforward method randomly asks the edges until all the edges are colored as Blue or Red. However some Red edges may make the Blue edges disconnected, and we do not need to ask such Blue edges. For example, if we first ask $(p_1, c_1)$ and it is Red, thus the edges $(u_1, r_1)$, $(u_2, r_1)$, $(u_1, r_2)$, $(u_2, r_2)$, $(u_3, r_3)$, $(r_1, p_1)$, $(r_2, p_1)$, $(r_3, p_1)$ become invalid edges after removing $(p_1, c_1)$, which do not need to be asked. Thus we can avoid asking 8 edges here. In the following sections, we will present how to select the minimum number of edges to ask.

## 4.2 Supporting Selection Predicates

For each crowd-powered selection $\mathcal{T}.\mathcal{C}_i$ `CROWDEQUAL value`, we add a new vertex into the graph. For each tuple $t \in \mathcal{T}$, we take the similarity between $t[\mathcal{C}_i]$ and `value` as the matching probability $\omega(t[\mathcal{C}_i], \texttt{value})$. If $\omega(t[\mathcal{C}_i], \texttt{value}) \geq \varepsilon$, we add an edge between this vertex and $t$ with weight $\omega(t[\mathcal{C}_i], \texttt{value})$. For a traditional selection predicate, we add the edge if they satisfy the predicate and the weight is 1. In this way, we can use the graph model to support the selection operation. For example, in the above query, if we want to select the paper published in `SIGMOD`, we will add a selection predicate (`Paper.Conference CROWDEQUAL` "`SIGMOD`".) In the above graph, we add a vertex "`SIGMOD`" and for each vertex $p_1, p_2, \cdots, p_8$, we add an edge to this new vertex.

**Remark.** If a requester still wants to use crowd-powered group or sort operations, `CDB` can adopt existing techniques [57, 13, 42, 14] to support them. For example, given a query with crowd-powered group, we first execute the crowd-based selection and join operations using our proposed techniques, and then group the results by applying existing crowdsourced entity resolution approaches [57, 13]. Similar strategy can also be applied to crowd-powered sort [42, 14].

For ease of presentation, in the following section, we do not distinguish traditional and crowd predicates. Instead, we consider a graph, where each edge is associated with a weight. If the weight is 1, we directly color it as Blue without crowdsourcing. Otherwise we need to ask the crowd.

# 5. QUERY OPTIMIZATION

We first present cost-control techniques, which select an optimized list of tasks (Section 5.1). Then we discuss which tasks can be asked simultaneously to reduce latency (Section 5.2). Finally we assign tasks to appropriate workers and infer results to improve quality (Section 5.3).

## 5.1 Cost Control

Given a graph, we aim to ask the minimum number of edges to find all the answers. We first discuss the case that if the colors of every edge is known, how to select the edges (Section 5.1.1) and then extend it to support the case that the colors are unknown (Section 5.1.2). Finally, we present how to select tasks if a budget is given (Section 5.1.3).

### 5.1.1 Task Selection with Known Edge Color

We consider the case that the colors of edges are known.

**Chain Join Structure.** We first consider a simple case that the tables are joined by a chain structure, i.e., each table is joined with at most two other tables and there is no cycle. We first use the depth-first algorithm to find all BLUE chains with $N$ BLUE edges. We can prove that these edges must be asked because they are in the final answers and their colors cannot be deduced based on other edges. Next we find a set of RED edges using the min-cut algorithm in the following flow graph $\mathcal{G}'$, and we can prove that these RED edges must be asked and others can be pruned, because others are disconnected by these RED edges. We make a minor change on graph $\mathcal{G}$ and generate a new graph $\mathcal{G}'$ as follows. (1) We remove the edges on BLUE chains. (2) We add a source vertex $s$ and a sink vertex $s^*$. For each vertex $t$ on a BLUE chain, we duplicate it by another vertex $t^*$. We add BLUE edges $(s,t)$ and $(t^*,s^*)$. (3) Suppose the tuples in the chain are sorted based on their corresponding tables from left to right. $t$ keeps its left edges and its right edges are moved to $t*$. We assign the weight of BLUE edges as $\infty$ and the weight of RED edges as 1. Then we find the min-cut of this graph. We can prove that the RED edges in the min-cut must be asked and other edges can be deduced.

LEMMA 1. *It is enough to ask the* RED *edges in the min-cut and edges on* BLUE *chains, and other edges can be pruned. This method is optimal (asks the minimum number of edges).*

PROOF. See Appendix C for the proofs of all lemmas. □

For example, in Figure 5, since path $(u_{12}, r_{12}, p_8, c_{12})$ is a BLUE path, each edge in the path must be asked. So we duplicate each node in the path and construct Figure 5(b). We compute the min-cut of Figure 5(b) and get edges $(p_7, r_{11})$, $(r_{12}, u_{11})$. We can see that these two edges must be asked and others can be pruned.

**Star Join Structure.** In this case, there is a center table and all other tables are joined with this table. For each tuple in the center table, if it has BLUE edges to tuples in every other table, then all the edges must be asked. If it has no BLUE edge (it has only RED edges) to tuples in some tables, we select the table with the least number of RED edges to ask. For example, in Figure 4, consider a CQL query

```
SELECT Researcher.Name,Paper.Title,Citation.Number
FROM Paper, Citation, Researcher
WHERE Paper.Title CROWDJOIN Citation.Title AND
Paper.Author CROWDJOIN Researcher.Name AND
Paper.Conference CROWDEQUAL "SIGMOD"
```

which aims to calculate the number of citation of papers published at SIGMOD. Figure 6 shows its join structure, i.e., a star join. We can see that $(p_1, c_1)$ is RED and we ask it, and then $(p_1, r_1)$, $(p_1, r_2)$, $(p_1, r_3)$ and edges on the selection operation can be pruned.

**Tree Join Structure.** The tables are joined by a tree structure and there is no cycle. We can transform it into a chain structure as follows. We first find the longest chain in the tree. Suppose the chain is $T_1, T_2, \cdots, T_x$. Then for each vertex $T_i$ on the chain, which (indirectly) connects other vertices $T'_1, T'_2, \cdots, T'_y$ that are not on the chain, we insert these vertices into the chain using a recursive algorithm. If these vertices are on a chain, i.e., $T_i, T'_1, T'_2, \cdots, T'_y$, then we insert them into the chain by replacing $T_i$ with $T_i, T'_1, T'_2, \cdots, T'_{y-1}, T'_y, T'_{y-1}, \cdots, T_i$. If these vertices are not on a chain, we find the longest chain and insert other vertices not on the chain into this chain using the above method. In this way, we can transform a tree join structure to a chain structure. Suppose that there are $|E|$ edges on the tree. It needs $O(|E|)$ time complexity to transform it into a chain structure. Note that the resulting chain has some duplicated tables. Hence, joining those tables may result in invalid join tuples (e.g., a join tuple that uses one tuple in the first copy of $T_i$, and a different tuple in the second copy of $T_i$). We need to remove those invalid join tuples.

**Graph Join Structure.** The tables are joined by a graph structure, i.e., there exist cycles in the join structure. We can transform it into a tree structure. For example, given a cycle $(T_1, T_2, \cdots, T_x, T_1)$, we can break the cycle by inserting a new vertex $T'_1$ and replacing it with $T_1$. Thus we can transform a cycle to a tree structure. We can first find a spanning tree of the graph using breadth first search in $O(|E|)$ time where $|E|$ is the number of edges, and break all non-tree edges. Similar to the tree structure, it takes $O(|E|)$ to transfer the graph structure to a new chain structure.

### 5.1.2 Task Selection without Known Edge Color

We consider the case where the colors of edges are unknown. In a high level, our goal is to ask fewer edges to find all answers with high probability.

Given a graph with $N$ edges, since the color of each edge is BLUE or RED, there are $2^N$ possible graphs. One possible formulation is to find the minimum of edges to satisfy all possible graphs, i.e., the answers in each possible graph can be computed based on these edges. However, this method unfortunately would lead to a rather trivial and expensive solution, since the possible graph with all BLUE edges (assuming each BLUE edge is in some BLUE chain) requires us to ask all edges. Hence, instead of satisfying all possible graphs, we consider a relaxation where we want to satisfy a random possible graph with high probability. Using the sample average method in stochastic optimization (see e.g., [27]), we can sample some possible graphs, e.g., $S$ samples, as follows. To generate a sample, we check each edge $e$ and set the color of the edge as BLUE with probability $\omega(e)$, and RED with probability $1 - \omega(e)$.

We consider the following problem: given $S$ sample graphs, select the minimum number of edges to resolve all sample graphs. Unfortunately, this problem is NP-hard, which can be proven by a reduction from the set cover problem.

LEMMA 2. *The problem of selecting the minimum edges that cover $S$ sample graphs is NP-hard.*

**Greedy Algorithm.** To address this problem, we propose a greedy algorithm. For each sample graph, we select its edges using the min-cut algorithms. Then we compute the

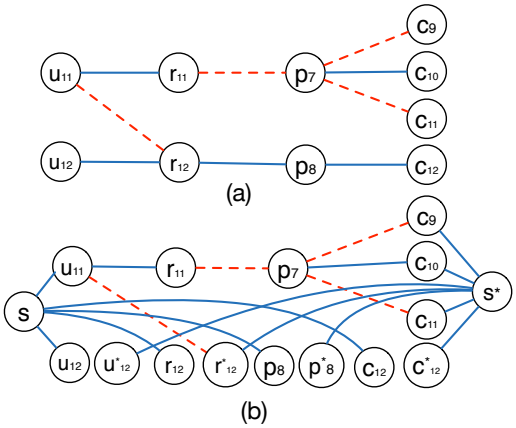


**Figure 5: Min-cut example.**

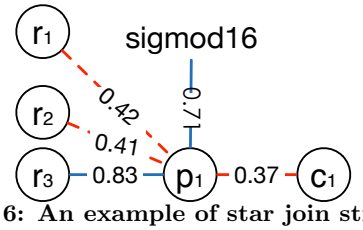


**Figure 6: An example of star join structure.**

union of these edges and sort the edges by the number of occurrences in the samples. We ask the edges in this order.

For example, consider the graph in Figure 7(a), and suppose we sample 5 graphs. Then we compute the min-cut of each graph. Edges $(u_{11}, r_{12})$, $(u_{12}, r_{12})$ and $(r_{11}, p_7)$ are the min-cut of Figure 7(b). Edges $(u_{11}, r_{12})$, $(c_{11}, p_7)$ and $(c_9, p_7)$ are the min-cut of Figure 7(c). Edges $(u_{11}, r_{11})$ and $(p_8, r_{12})$ are the min-cut of Figure 7(d). Edges $(u_{12}, r_{12})$ and $(r_{11}, p_7)$ are the min-cut of Figure 7(e). Edges $(r_{12}, p_8)$, $(c_{11}, p_7)$and $(c_9, p_7)$ are the min-cut of Figure 7(f). Then we compute the union of these edges and sort edges by the number of occurrences in these samples. We can see that $(c_9, p_7)$ and $(u_{11}, r_{12})$ occurs 2 times, and then we get the order: $(u_{11}, r_{12})$, $(c_9, p_7)$, $(p_7, r_{11})$, $(p_8, r_{12})$, $(c_{11}, p_7)$, $(u_{12}, r_{12})$, $(c_{10}, p_7)$, $(c_{12}, p_8)$ and $(u_{11}, r_{12})$. We ask the edges in this order: $(u_{11}, r_{12})$, $(c_9, p_7)$, $(p_7, r_{11})$, $(p_8, r_{12})$, $(u_{12}, r_{12})$ and $(c_{12}, p_8)$. This method asks an unnecessary edge $(c_9, p_7)$.

The time complexity of finding a min-cut is $O(|\mathcal{V}|^2 \log |\mathcal{V}|)$. Sampling a graph is $O(|\mathcal{E}|)$ and sorting the edges is $O(|\mathcal{E}| \log |\mathcal{E}|)$. Thus the time complexity is $O(|\mathcal{V}|^2 \log |\mathcal{V}| + |\mathcal{E}| \log |\mathcal{E}|)$. Note that this method is expensive since it requires to generate many samples and selects the edges in every sample. Moreover, it may select many unnecessary edges. To address this issue, we propose an expectation-based method.

**Expectation-Based Method.** Consider an edge $e = (t, t')$ where $t$ and $t'$ are from tables $T$ and $T'$ respectively. If cutting the edge can make some edges invalid, we can compute its pruning expectation by the probability to cut the edge $(1 - \omega(e))$ times the number of invalid edges introduced by this cutting. However cutting an edge may not make the any edge invalid. Thus this method will take the expectation as 0. To address this issue, we can consider the edges $(t, t'_1), (t, t'_2), \cdots, (t, t'_x)$ that from $t$ to all tuples in $T'$. If we cut all such edges, this must make some edges invalid, i.e., edges from tuples in other tables to $t$. The probability to cut all of these edges is $\prod_{i=1}^{x}(1 - \omega(t, t'_i))$ and thus the pruning expectation is $\prod_{i=1}^{x}(1-\omega(t, t_i))$ times the number of invalid edges (e.g., $\alpha$). As the graph is cut by these $x$ edges, the expectation of each edge should be $\frac{\prod_{i=1}^{x}(1-\omega(t,t_i))}{x}\alpha$. Similarly, we consider edges $(t_1, t), (t_2, t'), \cdots, (t_y, t')$ that from $t'$ to all tuples in $T$. The pruning expectation is $\prod_{i=1}^{y}(1-\omega(t_i, t'))$

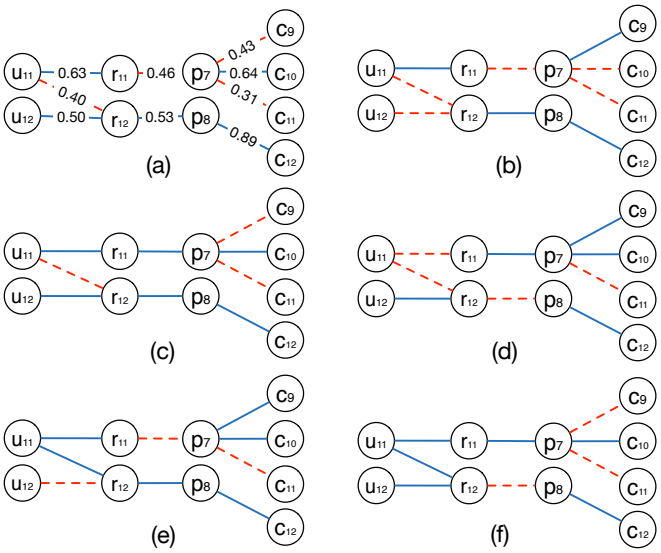


**Figure 7: Min-cut Greedy Algorithm.**

times the number of invalid edges (e.g., $\beta$) dividing $x$. Although the edge can cut the graph by combining with other edges, it is expensive to enumerate all such edges. Thus we only use the above two types of edges. Next we formally define the pruning expectation:

$$\mathbb{E}(t, t') = \frac{\prod_{i=1}^{x}(1 - \omega(t, t_i))}{x}\alpha + \frac{\prod_{i=1}^{y}(1 - \omega(t_i, t'))}{y}\beta. \quad (1)$$

Then we compute the pruning expectation for every edge, sort them by the expectation in descending order, and select the edge in order. Note we can efficiently compute $\alpha$ and $\beta$ by a depth-first algorithm starting from the cut edges. Since for each edge, we need to compute the pruning expectation, the time complexity is $O(|\mathcal{E}|)$. And then we have to sort these edges, which results in $O(|\mathcal{E}| \log |\mathcal{E}|)$ time complexity.

For example, we discuss how to compute $\mathbb{E}(p_1, r_1)$. We can see from Figure 4 that there is only one edge $(r_1, p_1)$ from $r_1$ to all tuples in table `Paper` and there are three edges $(p_1, r_1)$, $(p_1, r_2)$ and $(p_1, r_3)$ from $p_1$ to all tuples in table `Researcher`. Therefore, $\mathbb{E}(p_1, r_1) = (1 - 0.42) * 2 + \frac{(1-0.42)(1-0.41)(1-0.83)*6}{3} = 1.27$. Then we compute the expectation for every edge and sort them in descending order: $(p_1, c_1)$, $(p_3, c_5)$, $(p_7, r_{11})$, $(p_2, r_4)$, $(r_{12}, p_8)$, $(r_{10}, u_{10})$, $\cdots$. Then we select the edge in order. After we ask $(p_1, c_1)$ and get a red color, edges $(p_1, r_1)$, $(p_1, r_2)$, $(p_1, r_3)$, $(u_1, r_1)$, $(u_1, r_2)$, $(u_2, r_1)$, $(u_2, r_2)$ and $(u_3, r_3)$ are unnecessary to ask. Finally, we ask 15 edges: $(p_1, c_1)$, $(p_3, c_5)$, $(p_7, r_{11})$, $(p_2, r_4)$, $(r_{12}, p_8)$, $(r_{10}, u_{10})$, $\cdots$. Note if we use traditional tree-based method, which selects a table join order to ask, there are 3 join orders, which ask 24, 23 and 33 tasks respectively.

### 5.1.3 *Budget-Aware Task Selection*

In many applications, there is a hard budget constraint. Hence, we consider the following problem: given a budget $B$, how to select $B$ tasks to maximize the number of found answers. This problem is a variant of the previous problem and thus is likely to be intractable as well. We propose a greedy heuristic. Different from computing the pruning expectation of each edge, we should ask the edge that has large probability to be in an answer. To this end, we compute the answer expectation of becoming an answer for each edge. We first find all the candidate and compute the probability of each candidate $C$ that can become an answer, which is the product of edge similarity in the candidate: $\texttt{Pr}(C) = \prod_{e \in C} w(e)$.

Next we discuss how to select $B$ tasks based on the expectation. Firstly, we select the candidate with the largest expectation and add each edge $e$ in the candidate into the selected set $S$. Then we ask the edges in $S$ by their weights in descending order, because for a small weight, we have larger probability to prune unnecessary edges. For example consider a chain. If we ask the edge with smaller similarity, it has large probability to be RED and in this way we can prune other edges in the chain. After asking the edges in $S$, we update the graph with the answers of these edges, recompute the expectation, and repeat the above steps until we ask $B$ tasks. For example, suppose that $B = 6$, candidate $(u_9, r_9, p_5, c_7)$ has the largest product of the edge similarity:$0.61 * 0.83 * 0.91 = 0.46$. So we ask these three edges in a descending order of their similarity and we obtain a true answer. Then we ask $(u_9, r_9)$, $(r_9, p_5)$ and $(p_5, c_7)$ because the product of their similarities is the largest in the remainder candidates. Now we have used the budget and obtain two answers without wasting any task.

It is hard to fully utilize the budget to get as many answers as possible in the tree model. For example suppose it finds a join order, e.g., `University`, `Researcher`, `Paper` and `Conference`. Then it selects the edge with the largest weight based on this order. It first selects edge $(u_7, r_7)$ and then selects edges $(r_7, p_3)$ that has the largest weight from $r_7$ to tuples in `Paper`. Next it asks $(p_3, c_5)$. We can see that it cannot find any answer. Thus our graph model has significant superiority than the tree model.

## 5.2 Latency Control: Reducing #Rounds

Given two edges $e$ and $e'$, we check whether they are in the same candidate. If they are in the same candidate, we call that they are conflict, because asking an edge may prune the other edges; otherwise we call that they are non-conflict. Obviously we can ask non-conflict edges simultaneously but cannot ask conflict edges. For example, consider two conflict edges. If an edge is colored RED, then the other edge does not need to be asked. To check whether two edges are in a some candidate, we can enumerate all the candidates of an edge, e.g., $e$, and check whether they contain $e'$. If yes, they are in a same candidate; no otherwise. However this method is rather expensive. Next we propose several effective rules to detect whether two edges can be asked simultaneously.

**Connected Components.** We first compute the connected components in the graph. Obviously, the tasks in different connect components can be asked simultaneously, because they are non-conflict. For example, $(p_1, c_1)$ and $(p_2, c_2)$ are in different connected components and they are non-conflict.

**Edges Containing Tuples from the Same Table.** The edges that contain two different tuples from the same table can be asked simultaneously, because they cannot be in the same candidate. For example, $(p_1, r_1)$ and $(p_1, r_2)$ are non-conflict as they contain tuples in the same table.

**Overall Algorithm.** We first compute the connected components. For each component, we selected an ordered list of tasks sorted by the expectation in descending order. Next we select the longest "prefix" of this list such that every two edges in the prefix are non-conflict. Then all of these non-conflict edges in the prefixes of these components can be asked simultaneously. Suppose $S$ is the current prefix (initialized as empty). We access the next edge with the largest expectation and check whether $e$ has conflict edges in $S$ as follows. For each edge $e' \in S$, if $e$ and $e'$ contain different tuples from the same table, $e'$ is not conflict with $e$; otherwise, we check whether they are in the same candidate. If $e'$ is not conflict with any edge in $S$, we add $e'$ to $S$ and check the next edge with the largest expectation. If $e$ has conflict edges in $S$, we terminate and $S$ is the longest prefix.

In Figure 4, $(p_1, c_1)$ is a non-conflict edge of the first component and $(p_2, r_4)$ and $(p_3, c_5)$ are the non-conflict edges of the second component. Therefore, at the first round, we select $(p_1, c_1)$, $(p_2, r_4)$, $(p_3, c_5)$, $(r_8, u_8)$, $(r_9, u_9)$, $(r_{10}, u_{10})$, $(r_{11}, p_7)$ and $(r_{12}, p_8)$ in parallel. Then, we select $(p_4, c_6)$, $(p_5, r_9)$, $(r_{12}, u_{11})$ and $(r_{12}, u_{12})$ to ask at the second round. At last, we ask $(p_4, r_8)$, $(p_5, c_7)$ and $(p_8, c_{12})$ in parallel. Compared with the serial algorithm, we ask the same number of questions but only take 3 rounds.

## 5.3 Quality Control

In order to derive high-quality results based on workers' answers, it is important to do quality control. `CDB` controls quality at two timestamps: (1) when a worker answers task, we estimate the worker's quality and infer the truth of the answered task, called *"truth inference"*; (2) when a worker comes and requests for new tasks, we consider the worker's quality and assign tasks with the highest improvement in quality to the worker, called *"task assignment"*. `CDB` supports four types of tasks: single-choice, multiple-choice, fill-in-blank and collection tasks. Next we illustrate how `CDB` addresses truth inference (Section 5.3.1) and task assignment (Section 5.3.2) on different types of tasks, respectively.

### 5.3.1 Truth Inference

Let us denote $W = \{w\}$ as a set of workers, $T = \{t\}$ as a set of tasks, and $V_t = \{(w, a)\}$ as a set of workers' answers for task $t \in T$ where each tuple $(w, a) \in V_t$ means that worker $w$ provides answer $a$ for task $t$. Note that truth inference has been proven to be an effective way to do quality control [66, 20, 32, 35, 70, 49, 9, 68, 67].

**Single-Choice Task.** Similar to [39, 17, 63, 34], we model each worker $w \in W$ as a quality $q_w \in [0, 1]$, which indicates the probability that $w$ answers a task correctly. Based on workers' answers for all tasks, we compute each worker's quality via the Expectation-Maximization (EM) algorithm [18], which iteratively updates those parameters until convergence. Assume task $t \in T$ has $\ell$ choices, labeled as $1, 2, \ldots, \ell$, then based on each worker $w$'s computed quality $q_w$, we adopt the *Bayesian Voting* to derive the truth of task $t$, which has been proven to be optimal in [66] with known workers' qualities. That is, the probability of the $i$-th choice being the truth for task $t$ is computed as

$$p_i = \frac{\prod_{(w,a)\in V_t} (q_w)^{\mathbb{1}_{\{i=a\}}} \cdot \left(\frac{1-q_w}{\ell-1}\right)^{\mathbb{1}_{\{i\neq a\}}}}{\sum_{j=1}^{\ell} \prod_{(w,a)\in V_t} (q_w)^{\mathbb{1}_{\{j=a\}}} \cdot \left(\frac{1-q_w}{\ell-1}\right)^{\mathbb{1}_{\{j\neq a\}}}}, \qquad (2)$$

where $\mathbb{1}_{\{\cdot\}}$ is an indicator function, which returns 1 if the argument is true; 0, otherwise. For example, $\mathbb{1}_{\{5=3\}} = 0$ and $\mathbb{1}_{\{5=5\}} = 1$. Then the truth for $t$ can be estimated as the choice with the highest probability, i.e., $\text{argmax}_{1\le i\le \ell}\ p_i$.

**Multiple-Choice Task.** Since a task is possible to have multiple choices as truth, we can apply the above method that deals with single-choice task to multiple choice task, by decomposing each multiple-choice task (with $\ell$ choices) into a set of $\ell$ single-choice tasks, where each one addresses whether a specific choice is true or not. Then we can estimate the truth of a multiple-choice task as all the choices that are estimated as the truth in each single-choice task.

**Fill-in-blank Task.** It is hard to model a worker's quality in such kind of *open task*. Thus we do not model a worker's quality in this task. Alternatively, given workers' answers for a task, we estimate its truth by considering which answer is the *"pivot"*, i.e., closest to all workers' answers. To implement this idea, we first define the similarity between two answers, i.e., $sim(a, a')$, which we resort to string similarity measures [62], e.g., *Jaccard*, *Edit Distance*, *Cosine*. Then for each answer $a$, we compute its aggregated similarity w.r.t. other answers, i.e., $s_a = \sum_{(w,a')\in V_t} sim(a, a')$. Finally we estimate the truth of $t$ as the answer that attains the highest aggregated similarity, i.e., $\text{argmax}_{(*,a)\in V_t} s_a$.

**Collection Task.** `Collection` tasks can be implemented using fill-in-blank tasks, and we perform quality control with two phases: (1) while workers type in the user interface, we provide the auto-completion feature, which gives suggestions as the workers may type. This may help workers know what similar information has been recorded in `CDB` by other workers. (2) After workers give inputs to `CDB`, we can either leverage existing machine-based [12] or crowd-based [57, 56, 55] entity resolution methods to do disambiguation.

### 5.3.2 Task Assignment

When a worker comes and requests for new tasks, existing platforms provide interfaces for programmers to dynamically assign tasks to the coming worker. For example, AMT [1] identifies each worker via a unique ID. When a worker comes, AMT passes the worker ID to our sever, which dynamically assigns a set of tasks to the coming worker. Following this way, we can record the worker's information in `CDB`, and when the worker comes again, `CDB` leverages the worker's information and dynamically assigns a set of (say $k$) tasks to the worker, such that the quality will be improved the most. Existing works [69, 29, 20, 29, 48, 31, 11] mainly focus on a limited number of task types, and we are more general in handling multiple task types, as shown below.

**Single-Choice Task.** Recall that we model each worker $w$ as $q_w \in [0, 1]$, and we already store the worker's estimated quality if the worker answered tasks before (for a new worker, we can set its quality as the default value, say 0.7). Then when a worker $w$ comes with quality $q_w$, our objective is to "*assign a set of k tasks to worker w, such that the quality can be improved the most after the worker w answers the assigned tasks*". However, there are two problems: $(i)$ we do not know the ground truth of each task, thus the quality is hard to know; $(ii)$ we have no idea about how the worker can answer each task. We first focus on assigning $k = 1$ task, and then generalize it to $k > 1$ tasks.

For problem $(i)$, despite the unknown ground truth, we obtain a distribution of choices being true for each task $t$ based on workers' answers, i.e., $\vec{p} = (p_1, p_2, \ldots, p_\ell)$. Intuitively, the more consistent the distribution is (e.g., $p_i$ approximates 1 while others $p_j$ approximates 0 for $j \neq i$), the higher the quality will be achieved. In order to capture such consistency, we use the entropy function [52], i.e., $\mathcal{H}(\vec{p}) = -\sum_{i=1}^{\ell} p_i \cdot \log p_i$, which quantifies *the amount of inconsistency*, i.e., the lower $\mathcal{H}(\vec{p})$ is, the more consistent $\vec{p}$ is, the higher quality will be achieved. For problem $(ii)$, we can leverage the coming worker $w$'s quality and the task $t$'s distribution $\vec{p}$ to estimate the probability that the $i$-th choice will be answered by $w$, i.e., $p_i \cdot q_w + (1-p_i) \cdot \frac{1-q_w}{\ell-1}$. Then after worker $w$ answers task $t$ with the $i$-th choice, the distribution $\vec{p}$ becomes $\vec{p'} = (\frac{p_1 \cdot \frac{1-q_w}{\ell-1}}{\Delta}, \ldots, \frac{p_i \cdot q_w}{\Delta}, \ldots, \frac{p_\ell \cdot \frac{1-q_w}{\ell-1}}{\Delta})$, where $\Delta$ is the normalization factor, i.e., $\Delta = p_i \cdot q_w + (1-p_i) \cdot \frac{1-q_w}{\ell-1}$.

Based on the above solutions to problems $(i)$ and $(ii)$, we now can estimate the expected quality of improvement (or the expected decrease of inconsistency) if worker $w$ answers task $t$, denoted as $\mathcal{I}(t)$, which is shown below:

$$\mathcal{I}(t) = \mathcal{H}(\vec{p}) - \sum_{i=1}^{\ell} \left[ p_i \cdot q_w + (1-p_i) \cdot \frac{1-q_w}{\ell-1} \right] \cdot \mathcal{H}(\vec{p'}). \quad (3)$$

Thus we can select the task with the highest improvement in quality, i.e., $\text{argmax}_{t\in T} \mathcal{I}(t)$. In the above approach, we focus on assigning $k = 1$ task. It can be generalized to assign multiple ($k > 1$) tasks, where we select top-$k$ tasks with the highest improvement in quality $\mathcal{I}(t)$.

**Multiple-Choice Task.** Similar to the method in truth inference, we can decompose each multiple choice task (with $\ell$ choices) into a set of $\ell$ single-choice tasks, where each one asks whether the $i$-th choice is correct or not. In this way, we can define the quality (or consistency) of the multiple-choice task as the summation of all the consistencies (captured by entropy) in each decomposed single-choice task. Then following the above approach for single-choice task, we can generalize it to assigning top $k$ tasks to the coming worker.

**Fill-in-blank Task.** As it is hard to model a worker's quality in answering fill-in-blank task, we define the quality in each task as the consistency of workers' answers for the task. To be specific, suppose task $t$ obtains a set of answers $V_t$, then we define the consistency of task $t$, i.e., $\mathcal{C}(t)$, as the normalized similarities of all its obtained pairwise answers:

$$\mathcal{C}(t) = \sum_{\{(w,a)\in V_t\}\wedge\{(w',a')\in V_t\}\wedge\{w\neq w'\}} \frac{sim(a, a')}{\binom{|V_t|}{2}}. \quad (4)$$

Then we can select the task $t$ with the least consistency, i.e., $\text{argmin}_{t\in T} \mathcal{C}(t)$. We can also generalize it to selecting $k > 1$ tasks, by assigning top-$k$ tasks with the least consistencies.

**Collection Task.** To assign collection tasks, there are two factors to consider: (1) although we have developed autocomplete features, we have to disambiguate workers' answers, which we refer to the entity resolution technique [57]; (2) we also need to estimate the cardinality of results as workers gradually give answers, which we refer to the techniques that address crowd enumeration queries [53]. Suppose the number of distinct tuples collected is denoted as $M$ [57], and the number of estimated cardinality is denoted as $N$ [53], then the *completeness score* is defined as $\frac{N-M}{N}$, where we assign the collection tasks with the least completeness score, i.e., the tasks that are far from complete.

# 6. EXPERIMENTS

We have implemented `CDB` in Java on a Ubuntu server with Intel 2.40GHz Processor and 32GB memory, and deployed `CDB` with two popular crowdsourcing platforms, AMT [1], CrowdFlower [3] and ChinaCrowd [2]. This section presents the evaluation of `CDB`. We first introduce the experiment settings (Section 6.1), and then report the results of both simulated (Section 6.2) and real experiments (Section 6.3).

## 6.1 Experimental Settings

**Datasets**. We evaluate `CDB` using two datasets: (1) `paper` is a publication dataset crawled from ACM and DBLP. The dataset has four tables, `Paper`, `Citation`, `Researcher` and `University`, and each table (e.g., `Citation`) contains several attributes (e.g., 'title' and 'number'). Note that attributes

**Table 2: Paper Datasets.**

| Table | #Records | Attributes |
|---|---|---|
| Paper | 676 | **author**,*title*,conference |
| Citation | 1239 | *title*, number |
| Researcher | 911 | affiliation, **name**, gender |
| University | 830 | name, city, country |

**Table 3: Award Datasets.**

| Table | #Records | Attributes |
|---|---|---|
| Celebrity | 1498 | **name**, *birthplace*, birthday |
| City | 3220 | *birthplace*, country |
| Winner | 2669 | **name**, award |
| Award | 1192 | name, place |

with the same format represent that the tables can be joined through these attributes. For example, tables `Paper` and `Researcher` can be joined through attributes '**author**' and '**name**'. (2) `award` contains award information, which is crawled from Dbpedia [4] and Yago [5]. It also has four tables, `Celebrity`, `City`, `Winner` and `Award`. Specification and statistics of the datasets are shown in Tables 2 and 3. More details can be found in Appendix D.

**Queries.** As our `CQL` supports both *query* and *collection* semantics of crowdsourcing, we design the following queries for evaluation. (1) For *query semantics in* `CQL`, we design 5 representative queries on `paper` and `award` datasets respectively, as shown in Table 4 (Appendix D). These queries have covered different kinds of operators, such as `CROWDSELECT` and `CROWDJOIN`. For example, the query labeled with "2J1S" (i.e., two Joins and one Selection) on the `paper` dataset joins three tables, `Paper`, `Citation`, `Researcher` ("2J") and filters `Paper` tuples such that attribute 'conference' equals *sigmod* ("1S"). (2) For *collection semantics in* `CQL`, we design a `COLLECT` query that collects 100 universities' names with the highest rankings in the world, and a `FILL` query that fills the state of the collected universities.

**Competitors.** We compare `CDB` with existing methods. Crowdsourced entity resolution methods: (1) `Trans` [57] utilizes transitivity to prune dissimilar pairs; (2) `ACD` [58] leverages the correlation clustering to identify matching pairs. Crowdsourcing database systems: (3) `CrowdDB` [24] uses traditional rule-based techniques to estimate a query plan; (4) `Qurk` [43] optimizes a single join and uses rule-based techniques to estimate a query plan; (5) `Deco` [47] uses the cost-based model to estimate a query plan; (6) `OptTree` reports the optimal results of the tree model based method. We assume the colors of edges are known, enumerate all possible join orders, select the order with the minimum cost, and report the result of this optimal order. Our methods: (7) `MinCut` optimizes a query using the chain-join structure (Section 5.1.1) and uses greedy algorithm to select tasks to the crowd; (8) `CDB` leverages the expectation-based method to assign tasks to ask the crowd; (9) `CDB+` improves `CDB` by incorporating the quality control techniques in Section 5.3. `Trans` and `ACD` use a cost-based method to select a query plan based on the number of non-pruned pairs, and adopt their techniques to do crowdsourced entity resolution for a single join. `CrowdDB` and `Deco` do not optimize a single join and we use crowdsourced entity resolution techniques in [56] to support a single join for them. In all of these methods, we used 2-gram Jaccard similarity to compute the probability as follows. Given a value, say $t_x[\mathcal{C}_i]$, we split it into a set of 2-grams (i.e., substrings with length of 2). Then given an edge with two values $t_x[\mathcal{C}_i]$ and $t_y[\mathcal{C}_j]$, we computed the Jaccard similarity between their 2-gram sets (i.e., computing the ratio of their intersection size to their union size) as the probability. We utilized the prefix-filtering techniques to efficiently identify the tuple pairs with similarities not smaller than $\varepsilon = 0.3$ [10, 56]. We also evaluated other similarity functions in Appendix D.

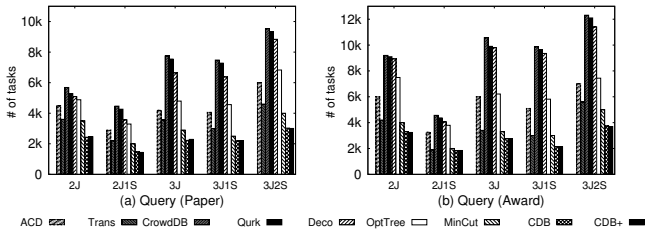


**Figure 8: Varying Query: # Tasks**

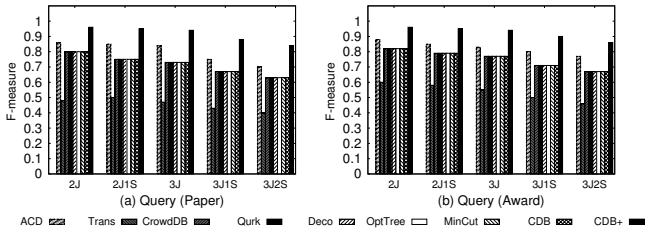


**Figure 9: Varying Query: Fmeasure**

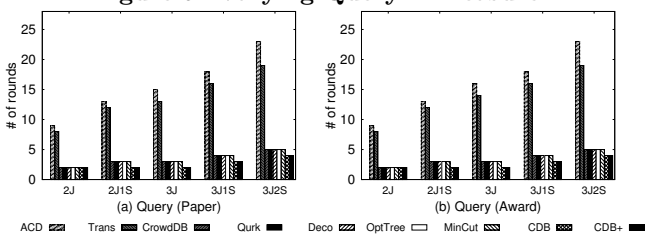


**Figure 10: Varying Query: # Rounds**

**Evaluation Metrics**. We evaluate `CDB` using the following three metrics in crowdsourcing [35] given a query. (1) Cost: since monetary cost largely depends on #tasks, we use the #tasks needed to ask the crowd to measure the cost. (2) Quality: we use the well-known F-measure to measure quality of the result returned by different systems. F-measure is the harmonic mean of *precision* and *recall*, i.e., F-measure $= \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$, where *precision* is the fraction of #returned tuples that are correct, and *recall* is the fraction of #correct tuples that are accurately returned. (3) Latency: we assign tasks to the crowd in different rounds, where tasks of any round would not be assigned until all tasks in the last round are completed. By following the settings of existing works [35] that assume each round takes the same amount of time, we use #rounds to measure latency.

## 6.2 Simulated Experiments

We conduct simulated experiments that examine performance of queries in Table 4 (Appendix D) and vary quality of the workers. For statistical significance, we repeat each experiment for $1K$ times and report the average result.

### 6.2.1 The Performance of Different Queries

We evaluate the performance of the five queries in Table 4 on nine methods: `Trans`, `ACD`, `CrowdDB`, `Qurk`, `Deco`, `OptTree`, `MinCut`, `CDB` and `CDB+`. To make a fair comparison, we assign each task to five simulated workers that are generated from the same Gaussian distribution $\mathcal{N}(0.8, 0.01)$, which means that each worker has an average chance of 80% to correctly answer a task. We first examine the monetary cost, as shown in Figure 8. Firstly, crowdsourcing entity resolution methods `ACD` and `Trans` outperform `Deco`, `Qurk`, `CrowdDB` because `ACD` and `Trans` can optimize a single join and prune many unmatched pairs. However `ACD` and `Trans` involve more rounds, because they need to use multiple rounds for a single join to do pruning. Even if `Trans` costs less than `ACD`,

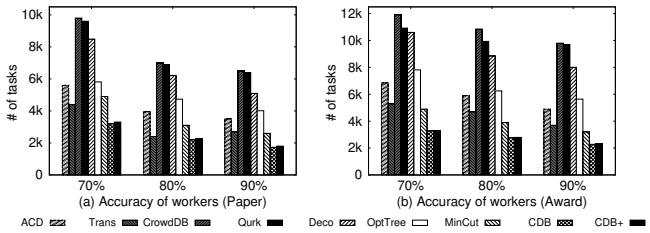


**Figure 11: Varying Quality: # Tasks**

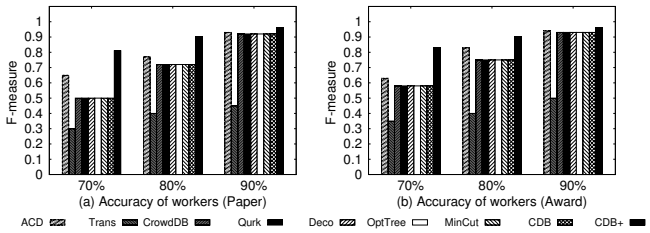


**Figure 12: Varying Quality: Fmeasure**

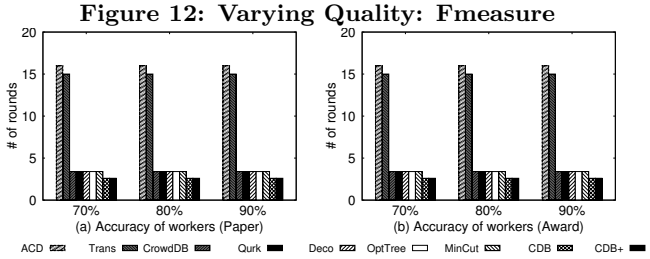


**Figure 13: Varying Quality: # Rounds**

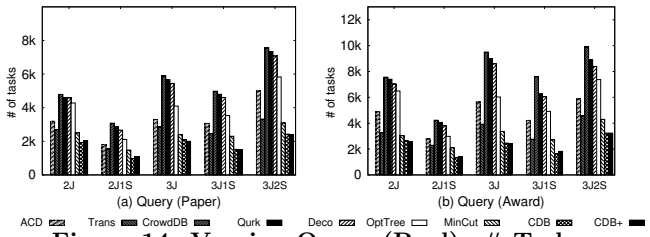


**Figure 14: Varying Query (Real): # Tasks**

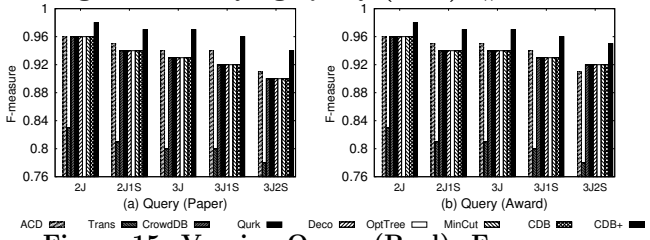


**Figure 15: Varying Query (Real): Fmeasure**

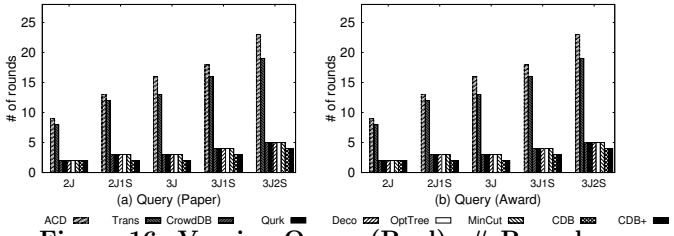


**Figure 16: Varying Query (Real): # Rounds**

`Trans` sacrifices quality (around 50%), because a transitivity error will affect many tuples. Secondly, `CDB` outperforms `ACD` and `Trans`, because `ACD` and `Trans` focus on optimizing join pairs between two tables rather than tuple-level pruning among multiple tables. This validates our claim that a fine-grained tuple-level optimization is much preferred in crowdsourcing. Thirdly, for tree model based methods, `OptTree` performs slightly better than `Deco`, `CrowdDB` and `Qurk`, since `OptTree` selects the table-level join order with the minimum cost by enumerating all possible orders. `Deco` is better than `Qurk` and `CrowdDB` because `Deco` uses a cost-based model while `Qurk` and `CrowdDB` use a rule-based model. Our graph model with tuple-level optimization achieves significant performance superiority compared with tree model-based methods, e.g., saving more than half of the cost. Fourthly, `CDB` and `CDB+` perform better than `MinCut`. The reason is that `MinCut` generates sample graphs for determining edge order. On one hand, it is expensive to enumerate many sample graphs, since it requires to select many edges to satisfy the samples. On the other hand, it cannot select high-quality edges if sampling a small number of samples. On the contrary, `CDB` employs an expectation-based method, which can effectively estimate the edge order. Although `CDB+` does not bring cost reduction compared with `CDB`, it improves quality.

Figure 9 shows the performance evaluation on quality. Obviously, `CDB+` performs significantly better than the other methods that apply the simple majority voting strategy. `CDB+` employs more sophisticated techniques to infer each worker's quality and obtain the correct results and assign tasks to appropriate workers.

Next, we evaluate the latency and report the results in Figure 10. Although our methods `MinCut`, `CDB` and `CDB+` significantly improve cost and quality, they do not bring high latency. Specifically, we can observe that all the methods except entity resolution take nearly the same number of rounds to complete the crowdsourcing process. For tree-based methods, this is natural as the round is exactly the number of join predicates where tuple pairs in each predicate are crowdsourced in parallel. For our graph-based methods, this is not easy because we need to consider the trade-off between the benefit of cost brought by task inference and the benefit of latency brought by asking tasks in parallel. For entity resolution methods, they take around 5 times more than graph-based methods because they need to take multiple rounds to perform crowdsourced entity resolution for a single join. This result shows that our methods can effectively balance the trade-off.

### 6.2.2 *Varying the Quality of Workers*

We vary the quality of simulated workers by tuning the underlying Gaussian distribution $\mathcal{N}(q, 0.01)$ as $q \in \{0.7, 0.8, 0.9\}$. Similar to the previous settings, we first assign each task to five workers, then run the five queries in Table 4 on each dataset, and finally report the results in Figure 11. First, it has similar trends among the nine methods, that is, in term of the cost, we have `Qurk` $\approx$ `CrowdDB` $\approx$ `Deco` $>$ `OptTree` $>$ `ACD` $>$ `Trans` $>$ `MinCut` $>$ `CDB` $\approx$ `CDB+`. Moreover, an interesting observation is that with the increase of worker quality, the number of tasks needed to ask the crowd decreases, and thus the cost is reduced. The reason is that with higher worker quality, the answers given by workers are more accurate, which may help the methods to infer more unknown tasks, so as to prune more tasks in the next round.

With respect to quality, we can observe that for workers with quality $\mathcal{N}(0.7, 0.01)$ and $\mathcal{N}(0.8, 0.01)$, `CDB+` outperforms the other methods with significant gains. The performance superiority comes from our result inference and task assignment techniques. These techniques can effectively reduce the uncertainty of answering crowdsourcing tasks, and thus improve the overall quality. For workers with high quality (e.g., $\mathcal{N}(0.9, 0.01)$), although all of the methods can achieve high quality, `CDB+` still outperforms the other methods due to its effective quality-control techniques.

As all methods adopt the round-based approach, the average number of rounds of different kinds of quality are the same (about 3). These results also show that our methods achieve quite low latency and can be used in real settings.

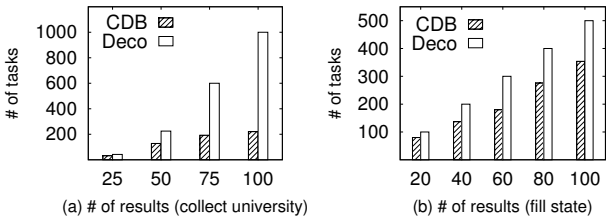


**Figure 17: Collect/Fill: Varying #Results (Real).**

## 6.3 Real Experiments

We also conduct experiments on the real crowdsourcing platform AMT. We pack 10 tasks in each HIT with \$0.1 as its price, and ask 5 workers on AMT to answer each HIT. Then, we show the comparison result of different methods.

### 6.3.1 The Performance of Different Queries

We first evaluate the cost for different queries. As shown in Figure 14, compared with `Qurk`, `CrowdDB`, `Deco`, `ACD` and `OptTree`, our proposed methods `MinCut`, `CDB` and `CDB+` reduce the number of selected tasks about 2-3 times, which is mainly attributed to our tuple-level optimization. `ACD` and `Trans` have more rounds because they need to use multiple rounds to do entity resolution. `Trans` has significant lower quality than `CDB` as a transitivity error will affect many tuples. As it is expensive to generate many samples for `MinCut`, we generate 100 samples and `MinCut` has low performance. `CDB` and `CDB+` outperform other competitors as we can select high-quality tasks and utilize them to prune many tasks. `CDB+` costs similarly as `CDB`, since workers in real crowdsourcing platforms are of high quality to answer the tasks.

With respect to quality, as shown in Figure 15, we can see that all the methods achieve high quality on all queries, i.e., higher than 90% on F-measure. That is because the essential task for join and selection is actually similar to entity resolution, which is not difficult for AMT workers. Despite that all methods achieve good performance, `CDB+` still achieves much higher quality than the other methods because it can tolerant errors and correct errors. The `CQL` queries with more predicates have slightly lower quality than those with less predicates, because the crowd may introduce errors and more predicates lead to larger chances of introducing errors.

Considering latency, as shown in Figure 16, our methods have smaller number of rounds than competitors, especially for the entity resolution methods. As discussed in Section 6.2, since we adopt a round-based approach, our methods can complete the tasks in few rounds, which results in less latency. We can see that for each query, our methods can complete within 4 rounds, which is acceptable in practice.

### 6.3.2 The Performance of Collection Queries

We evaluate the collection semantics (i.e., *collection* and *fill*) and compare with a baseline method without considering cleansing of the collected data and duplicate control.

Our collection operator evaluates collecting the names of top 100 universities in USA. We compare with `Deco`. As shown in Figure 17(a), `Deco` does not consider duplicate control, and thus many workers return the same answers, which is a waste of budget. `CDB` can reduce the numbers of questions by about 5 times compared with `Deco`, because its autocompletion mechanism can remind workers the similar universities that have been collected by other workers. Moreover, with the increase of the collection number, the improvement of our method is more significant as workers have small possibility to give duplicate answers for a small number but have large possibility for more tuples.

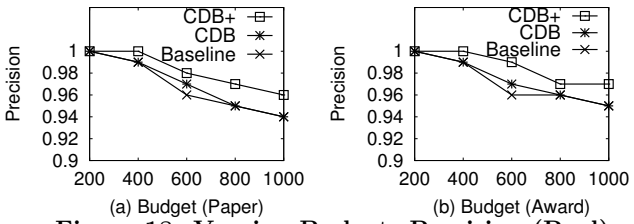


**Figure 18: Varying Budget: Precision (Real).**

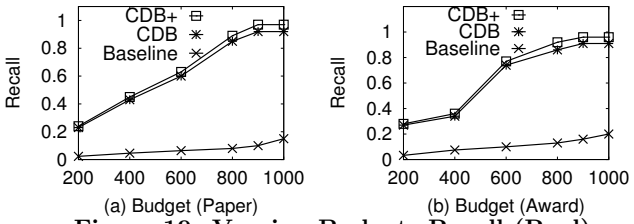


**Figure 19: Varying Budget: Recall (Real).**

Our fill operator evaluates asking the crowd to fill in the corresponding state of 100 given universities in USA. For each university, we ask 5 workers to fill in the task. If more than 3 results we obtain from workers have high similarity score in each other, we do not ask other two workers. As shown in Figure 17(b), `CDB` reduces 30% cost than `Deco`.

### 6.3.3 The Performance of Budget Setting

We vary the budget for crowdsourcing so as to ask the crowd to find as many answers as possible within the budget constraint. We still use the queries in Table 4. Since `Deco` does not support the budget-aware task selection, we compare with a well-designed baseline. Specifically, the baseline method first selects the edge with large probability in the first table (with respect to the best table order) and then uses a depth-first traversal to find answers joined with the other table. Figures 18 and 19 show the results. We have two observations. First, under each budget, both methods achieve high precision, e.g., the precision of both methods under each budget is around 98%. This is because the query is relatively easy for crowdsourcing workers. Second, `CDB` significantly improves recall. For example, for 600 tasks, our method achieves 60%-80% recall while the recall of baseline is only 10%. With the increase of the budget, the recall of `CDB` increases sharply and becomes flat after 800 questions, which indicates that nearly all answers have been obtained. Note that the recall of the baseline only increases slightly with the increasing budget, which indicates that this method obtains smaller numbers of correct results. `CDB` outperforms baseline significantly, because it picks up the candidate edges with high probability to collect the answers. Furthermore, we can see that `CDB+` outperforms `CDB` by 5% on recall and 3% on precision because it can improve quality by truth inference and task assignment.

# 7. CONCLUSION

We developed a crowd-powered database system `CDB`. `CDB` adopted a graph-based query model which can provide tuple-level graph optimization. With the graph model, `CDB` employed a unified framework to perform the multi-goal optimization. We formulated the task selection problem which minimizes the number of tasks to be crowdsourced, proved that the problem is NP-hard and devised greedy algorithms to select high-quality tasks. We developed effective algorithms to reduce latency. We optimized the quality by devising more sophisticated quality-control strategies. We have implemented `CDB` and deployed it into AMT and CrowdFlower. Experimental results demonstrate `CDB` outperformed existing studies in terms of cost, latency and quality.

# 8. REFERENCES


[1] Amazon mechanical turk. https://www.mturk.com/.

[2] Chinacrowd. http://www.chinacrowds.com/.

[3] Crowdflower. http://www.crowdflower.com.

[4] Dbpedia. https://www.dbpedia.org.

[5] Yago. https://www.mpi-inf.mpg.de.

[6] A. Alfarrarjeh, T. Emrich, and C. Shahabi. Scalable spatial crowdsourcing: A study of distributed algorithms. In *MDM*, pages 134–144, 2015.

[7] Y. Amsterdamer, S. B. Davidson, T. Milo, S. Novgorodov, and A. Somech. Ontology assisted crowd mining. *PVLDB*, 7(13):1597–1600, 2014.

[8] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD*, pages 241–252. ACM, 2013.

[9] A.P.Dawid and A.M.Skene. Maximum likelihood estimation of observer error-rates using em algorithm. *Appl.Statist.*, 28(1):20–28, 1979.

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

[11] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, and W.-C. Tan. Asking the right questions in crowd data sourcing. In *ICDE'12*.

[12] D. G. Brizan and A. U. Tansel. A survey of entity resolution and record linkage methodologies. *Communications of the IIMA*, 6(3):5, 2015.

[13] 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.

[14] X. Chen, P. N. Bennett, K. Collins-Thompson, and E. Horvitz. Pairwise ranking aggregation in a crowdsourced setting. In *WSDM*, pages 193–202, 2013.

[15] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD*, pages 1247–1261. ACM, 2015.

[16] 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.

[17] 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.

[18] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *J.R.Statist.Soc.B*, 30(1):1–38, 1977.

[19] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 19(1):1–16, 2007.

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

[21] J. Fan, M. Lu, B. C. Ooi, W.-C. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. In *ICDE*, pages 976–987. IEEE, 2014.

[22] J. Fan, Z. Wei, D. Zhang, J. Yang, and X. Du. Distribution-aware crowdsourced entity collection. *IEEE Trans. Knowl. Data Eng.*, 2017.

[23] 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.

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

[25] 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.

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

[27] A. Gupta, M. Pál, R. Ravi, and A. Sinha. Boosted sampling: approximation algorithms for stochastic optimization. In *ACM symposium on Theory of computing*, pages 417–426, 2004.

[28] C.-J. Ho, S. Jabbari, and J. W. Vaughan. Adaptive task assignment for crowdsourced classification. In *ICML*, pages 534–542, 2013.

[29] C.-J. Ho and J. W. Vaughan. Online task assignment in crowdsourcing markets. In *AAAI*, 2012.

[30] 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.

[31] 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*, 2016.

[32] P. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *SIGKDD Workshop*, 2010.

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

[34] M. Joglekar, H. Garcia-Molina, and A. G. Parameswaran. Evaluating the crowd with confidence. In *SIGKDD*, pages 686–694, 2013.

[35] G. Li, J. Wang, Y. Zheng, and M. J. Franklin. Crowdsourced data management: A survey. *TKDE*, 2015.

[36] 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.

[37] Y. Li, J. Gao, C. Meng, Q. Li, L. Su, B. Zhao, W. Fan, and J. Han. A survey on truth discovery. *SIGKDD Explor. Newsl.*, 2016.

[38] Q. Liu, J. Peng, and A. T. Ihler. Variational inference for crowdsourcing. In *NIPS*, pages 701–709, 2012.

[39] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. CDAS: A crowdsourcing data analytics system. *PVLDB*, 2012.

[40] F. Ma, Y. Li, Q. Li, M. Qiu, J. Gao, S. Zhi, L. Su, B. Zhao, H. Ji, and J. Han. Faitcrowd: Fine grained truth discovery for crowdsourced data aggregation. In *KDD*, 2015.

[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. G. Parameswaran, S. Boyd, H. Garcia-Molina, A. Gupta, N. Polyzotis, and J. Widom. Optimal crowd-powered rating and filtering algorithms. *PVLDB*, 7(9):685–696, 2014.

[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 and J. Widom. Query optimization over crowdsourced data. *PVLDB*, 6(10):781–792, 2013.

[48] L. Pournajaf, L. Xiong, V. Sunderam, and S. Goryczka. Spatial task assignment for crowd sensing with cloaked locations. In *MDM*, volume 1, pages 73–82. IEEE, 2014.

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

[50] S. B. Roy, I. Lykourentzou, S. Thirumuruganathan, S. Amer-Yahia, and G. Das. Task assignment optimization in knowledge-intensive crowdsourcing. *The VLDB Journal*, 24(4):467–491, 2015.

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

[52] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1), Jan. 2001.

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

[54] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, 2012.

[55] N. Vesdapunt, K. Bellare, and N. N. Dalvi. Crowdsourcing algorithms for entity resolution. *PVLDB*, 7(12):1071–1082, 2014.

[56] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: crowdsourcing entity resolution. *PVLDB*, 5(11), 2012.

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

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

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

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

[61] 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.

[62] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers of Computer Science*, 10(3):399–417, 2016.
[63] C. J. Zhang, L. Chen, H. V. Jagadish, and C. C. Cao. Reducing uncertainty of schema matching via crowdsourcing. *PVLDB*, 2013.
[64] C. J. Zhang, Y. Tong, and L. Chen. Where to: Crowd-aided path selection. *PVLDB*, 7(14):2005–2016, 2014.
[65] X. Zhang, G. Li, and J. Feng. Crowdsourced top-k algorithms: An experimental evaluation. *PVLDB*, 9(8):612–623, 2016.
[66] Y. Zheng, R. Cheng, S. Maniu, and L. Mo. On optimality of jury selection in crowdsourcing. In *EDBT*, pages 193–204, 2015.
[67] Y. Zheng, G. Li, and R. Cheng. DOCS: domain-aware crowdsourcing system. *PVLDB*, 10(4):361–372, 2016.
[68] 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.
[69] Y. Zheng, J. Wang, G. Li, R. Cheng, and J. Feng. QASCA: A quality-aware task assignment system for crowdsourcing applications. In *SIGMOD*, 2015.
[70] D. Zhou, S. Basu, Y. Mao, and J. C. Platt. Learning from the wisdom of crowds by minimax entropy. In *NIPS*, 2012.

# APPENDIX

# A. SPECIFICATION OF CQL

**Data Define Language (DDL).** To define attributes or tuples to be crowdsourced, CQL introduces keyword CROWD.

For example, a requester can define the following CQL to use CROWD to define that the crowd can be employed to "fill" the missing values of attributes, gender and affiliation.

```
CREATE TABLE Researcher (name varchar(64), gender CROWD
varchar(16), affiliation CROWD varchar(64));
```

Similarly, the requester can also define the use of crowd for collecting more tuples using the following CQL:

```
CREATE CROWD TABLE University (name varchar(64), city
varchar(64), country varchar(64));
```

Different from CrowdDB [24], CQL does not need that a CROWD table (e.g., University) must have a primary key to determine if two workers contribute the same tuple, because this is often not practical in crowdsourcing even if automatic entity resolution techniques are utilized [19].

**Data Manipulation Language (DML).** CQL supports two types of crowd-powered manipulations: 1) *Crowd-Powered Collection*: CQL formalizes crowd-powered collection semantics to crowdsource any CROWD column or table (Section A.1). 2) *Crowd-Powered Query*: CQL defines query semantics to solicit the crowd to perform selection and join(Section A.2). CQL also supports a *budget* mechanism of imposing cost constraint for crowdsourcing.

## A.1 Collection Semantics of CQL

CDB provides *collection semantics* to employ the crowd to contribute missing data in the defined tables. To this end, CQL introduces two built-in keywords, FILL and COLLECT, which are respectively for CROWD attributes and CROWD tables.

**Fill Semantics.** CQL introduces FILL, which can be considered as a crowd-powered UPDATE, to crowdsource missing attribute values (we introduce CNULL to indicate that an attribute value needs to be crowdsourced), e.g.,

```
FILL Researcher.gender;
```

Moreover, FILL also allows requester to fill a part of missing attribute values, e.g., female researchers.

```
FILL Researcher.affiliation
WHERE Researcher.gender = "female";
```

**Collection Semantics.** We design COLLECT in CQL to collect more tuples from the crowd for a CROWD table. For example, the following CQL wants to collect more universities.

**Algorithm 1**: CDB

**Input**: CQL query $q$, $\mathcal{D}$
**Output**: *Result*
1 SQLParser($q$);
2 $\mathcal{G}$ = ConstructGraph($\mathcal{D}$);
3 **while** *there exist uncolored edges in* $G$ **do**
4  $T$ = TaskSelection($\mathcal{G}$);
5  $P$=GenerateParallelTasks($G$);
6  **for** *each request from a worker* $w$ **do**
7  $A$=TaskAssignment($S \in PT$, $w$);
8  ColorGraph($A$,$G$);
9 **return** *Result*=ResultInference*($G$);*

```
COLLECT University.name, University.city
WHERE University.country = "US";
```

## A.2 Query Semantics of CQL

CDB provides *query semantics* to solicit the crowd to process the data, which are either requester-provided or contributed from the crowd, to fulfill a variety of requirements.

CQL introduces CROWDEQUAL to ask the crowd for filtering tuples according to some criteria. The crowd may recognize variety of the data. For example, the following CQL may retrieve tuples with country like "United States", "US", etc.

```
SELECT University.*
FROM University
WHERE University.country CROWDEQUAL "USA";
```

The CROWDJOIN can be used for joining tuples, e.g., joining researchers with their affiliated universities:

```
SELECT Researcher.*, University.*
FROM Researcher, Researcher
WHERE Researcher.affiliation CROWDJOIN University.name;
```

# B. ALGORITHM OVERVIEW

We give an overview of our method in Algorithm 1. Given a CQL query, CDB first parses the CQL query (line 1) and constructs a graph (line 2). Next it uses the expectation-based method to select a minimal set of tasks $T$ (line 4). Then we identify the set of tasks $P$ that can be asked in parallel (line 5). Next for each worker request, we select a subset of tasks $S$ and assign the tasks to the worker (line 7). After collecting the answers from the worker, we color the graphs based on the answer (line 8). If there exist uncolored edges, we repeat the above steps until all the edges are colored.

# C. PROOFS OF LEMMAS

**Proof of Lemma 1.** We denote the original graph as $\mathcal{G}$ (assuming $\mathcal{G}$ also contains $s$ and $s^*$) and the graph constructed above as $\mathcal{G}'$. Consider a BLUE chain from $s$ to $s^*$, with $N$ BLUE edges in the original graph $\mathcal{G}$. Since all edges on this chain are BLUE, the chain corresponds to a tuple in a final answer (we call such a chain BLUE chain). We call any edge on a BLUE chain a B-edge (note that we removed these edges from $\mathcal{G}$ in step (2)). Hence, every B-edge must be asked.

Next, we show that if we ask all edges in the min-cut of $\mathcal{G}'$ (in addition to the above BLUE edges), we do not need to ask any other edge. Indeed, we only need to refute the existence of any other BLUE chain except the above BLUE chains in $\mathcal{G}$. Hence, for any possible $s$-$s^*$ non-BLUE chain, we need to ask at least one RED edge on the chain. Consider an arbitrary $s$-$s^*$ non-BLUE chain $P$ in $\mathcal{G}$. Note that $P$ may contain zero or more B-edges.

(1) If $P$ contains zero B-edges, the min-cut we constructed above must contain at least 1 RED edge in $P$ (since other-

**Table 4: The 5 representative queries used on paper and award.**

| Query | Dataset paper | Dataset award |
|---|---|---|
| 2 Joins (2J) | SELECT Paper.title,Researcher.affiliation, Citation.number<br>FROM Paper, Citation, Researcher<br>WHERE Paper.title CROWDJOIN Citation.title AND<br>Paper.author CROWDJOIN Researcher.name | SELECT Winner.award, City.country<br>FROM Winner, City, Celebrity<br>WHERE Celebrity.name CROWDJOIN Winner.name AND<br>Celebrity.birthplace CROWDJOIN City.name |
| 2 Joins 1 Selection (2J1S) | SELECT Paper.title,Researcher.affiliation, Citation.number<br>FROM Paper, Citation, Researcher<br>WHERE Paper.title CROWDJOIN Citation.title AND<br>Paper.author CROWDJOIN Researcher.name AND<br>Paper.conference CROWDEQUAL "sigmod" | SELECT Winner.award, City.country<br>FROM Winner, City, Celebrity<br>WHERE Celebrity.name CROWDJOIN Winner.name AND<br>Celebrity.birthplace CROWDJOIN City.name AND<br>Celebrity.birthplace CROWDEQUAL "New York" |
| 3 Joins (3J) | SELECT Paper.title,Citation.number,University.country<br>FROM Paper, Citation, Researcher,University<br>WHERE Paper.title CROWDJOIN Citation.title AND<br>Paper.author CROWDJOIN Researcher.name AND<br>University.name CROWDJOIN Researcher.affiliation | SELECT Winner.name, Award.place, City.country<br>FROM Winner, City, Celebrity, Award<br>WHERE Celebrity.name CROWDJOIN Winner.name AND<br>Celebrity.birthplace CROWDJOIN City.name AND<br>Winner.award CROWDJOIN Award.name |
| 3 Joins 1 Selection (3J1S) | SELECT Paper.title,Citation.number<br>FROM Paper, Citation, Researcher,University<br>WHERE Paper.title CROWDJOIN Citation.title AND<br>Paper.author CROWDJOIN Researcher.name AND<br>University.name CROWDJOIN Researcher.affiliation AND<br>University.country CROWDEQUAL "USA" | SELECT Winner.name,City.country<br>FROM Winner, City, Celebrity, Award<br>WHERE Celebrity.name CROWDJOIN Winner.name AND<br>Celebrity.birthplace CROWDJOIN City.name AND<br>Winner.award CROWDJOIN Award.name AND<br>Award.place CROWDEQUAL "US" |
| 3 Joins 2 Selections (3J2S) | SELECT Paper.title,Citation.number<br>FROM Paper, Citation, Researcher,University<br>WHERE Paper.title CROWDJOIN Citation.title AND<br>Paper.author CROWDJOIN Researcher.name AND<br>University.name CROWDJOIN Researcher.affiliation AND<br>Paper.conference CROWDEQUAL "sigmod" AND<br>University.country CROWDEQUAL "USA" | SELECT Winner.name,City.country<br>FROM Winner, City, Celebrity, Award<br>WHERE Celebrity.name CROWDJOIN Winner.name AND<br>Celebrity.birthplace CROWDJOIN City.name AND<br>Winner.award CROWDJOIN Award.name AND<br>Celebrity.birthplace CROWDEQUAL "New York" AND<br>Award.place CROWDEQUAL "US" |

**Table 5: Efficiency of 5 queries (milliseconds).**

| Dataset | 2J | 2J1S | 3J | 3J1S | 3J2S |
|---|---|---|---|---|---|
| paper | 2 | 3 | 4 | 4 | 5 |
| award | 8 | 9 | 11 | 11 | 12 |

wise, we can go from $s$ to $s^*$ without crossing a cut-edge in $\mathcal{G}'$, which is a contradiction).

(2) Suppose $P$ contains one or more B-edges. Let $e = (u, v)$ be the first B-edge such that $u \neq s$ (i.e., closest to $s$). By the construction of $\mathcal{G}'$, we know that we can go from $u'$ to $s^*$ in $\mathcal{G}'$. Hence in the min-cut of $\mathcal{G}'$, it contain at least one RED edge on the $s$-$u$ chain in $\mathcal{G}'$, which corresponds to a $s$-$u$ chain of $\mathcal{G}$. Hence, we can refute the blueness of $P$ as well.

Now, we show that one cannot do better than the min-cut. Let $h$ denote the size of the min-cut. By the max-flow min-cut theorem, the value of a max flow is also $h$. Suppose that there is a solution $S$ using fewer than $h$ RED edges. Remove those edges from $\mathcal{G}'$. Removing each RED edge can only reduce the flow value by 1. So the remaining flow is at least 1, implying that there is still an $s$-$s^*$ non-blue chain, but no edge in this chain is in $S$. This is a contradiction, since we have to ask at least one RED edge from this chain.

**Proof of Lemma 2.** We provide a reduction from the set cover problem, which is a well-known NP-hard problem. In a set cover problem, we are given a ground set $U$ of elements, and a family $\mathcal{S}$ of subsets of $U$ (say $\mathcal{S} = \{S_1, \ldots, S_m\}$). The goal is to cover all elements in $U$ using as few sets in $\mathcal{S}$ as possible. Here is our reduction: For each element $e \in U$, we create a sample graph $G_e$, which is simply a chain. Each such chain $G_e$ contains $m = |\mathcal{S}|$ edges. If $S_i$ contains $e$, we let the color of the $i$th edge of $G_e$ be RED. All remaining edges are BLUE. We can see that the set cover instance has a solution with $k$ sets, if and only if we can ask $k$ edges to resolve all sample graphs.

# D. ADDITIONAL EXPERIMENTS

**Generation of Dataset paper**. Attributes 'author', 'title' and 'conference' in paper and 'affiliation' in Researcher were crawled from ACM. Attributes 'title' and 'number' in Citation, 'name' in Researcher and 'name' in University were crawled from DBLP. We crawled the attributes 'city' and 'country' in University from the web.

**Generation of Dataset award**. It is an award dataset with 4 tables Celebrity, City, Winner and Award. The attributes 'name', 'birthplace' and 'birthday' in Celebrity and the attribute 'award' in Winner were from dbpedia [4]. The attributes 'birthplace' and 'country' in City, 'name' in Winner and 'name' in Award were from Yago [5]. We crawled the attribute 'place' in Award from the web.

**Five Representative Queries**. Table 4 lists 5 representative queries used in our experiments.

**Efficiency.** Table 5 shows the efficiency of our method, which only takes about 10 milliseconds to select the tasks that can be asked in parallel.

**Tradeoffs Among Optimization Objectives.** We also evaluate tradeoffs among objectives of crowdsourcing optimization, i.e., quality, cost, and latency.

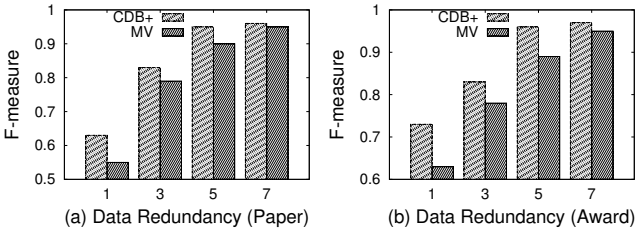


**Figure 20: Varying Costs: F-measure (Real).**

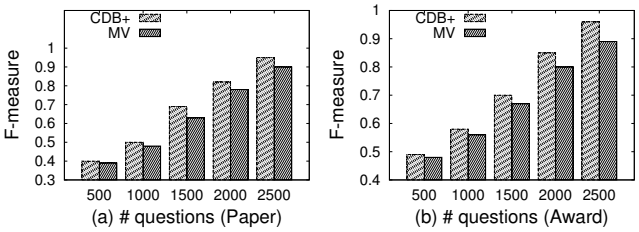


**Figure 21: Cost vs. Quality (Real).**

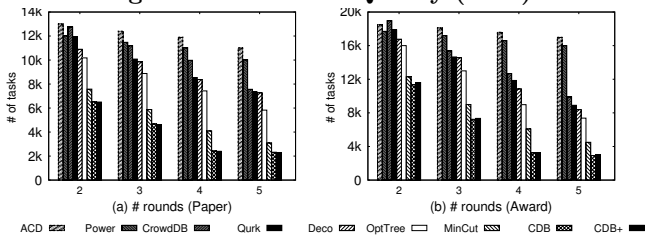


**Figure 22: Cost vs. Latency (Real).**

We first compare our quality optimization approach with majority voting used in the existing systems. We use the most complex query "3J2S" in Table 4, and compare the approaches by varying data redundancy (i.e., the number of assignments per task). As shown in Figure 20, CDB+ achieves higher F-measure than majority voting, which shows that our approach can tolerate errors well even when data redundancy is low. When increasing redundancy, the gap between the approaches becomes smaller, since majority voting can also perform well when assigning more workers. However, higher redundancy will induce higher crowdsourcing cost.

Then, we evaluate the tradeoff between quality and cost. We use the aforementioned query "3J2S" and examine quality by varying the number of questions (i.e., cost budget), where the data redundancy is set as 5. The experimental result is shown in Figure 21. We can see that, as the increase of cost, although quality of both approaches improves, CDB+ always outperforms majority voting. Moreover, with the increase of cost, the improvement on quality brought by CDB+ becomes higher. This is because CDB+ has more information to infer truth of questions, calculate workers' quality, and judiciously assign tasks to appropriate workers.

We also evaluate the tradeoff between cost and latency. We compare 9 methods by varying the number $r$ of rounds as latency constraint. Specifically, given a constraint $r$, each method applies its optimization techniques in the first $r-1$ rounds, and crowdsources all the remaining tasks in the last round. As shown in Figure 22, with the increase of number of rounds, the cost of all methods decreases. This reveals the tradeoff between cost and latency: when having loose latency constraints, one method has more opportunities to prune tuple pairs, leading to lower cost. Our proposed methods CDB and CDB+ achieve the lowest cost at every latency constraint due to the tuple-level optimization technique. For example, when the number of round is 5, the costs of CDB and CDB+ are 3 times less than those of other methods.

**Comparison of Different Similarity Functions.** We compared the methods by using different similarity functions to estimate the probability. (1) NoSim that does not use any similarity estimation and takes the probability of each edge as 0.5. (2) ED that uses the normalized edit distance to compute the probability. (3) JAC that tokenizes each attribute and uses Jaccard on token sets to compute the probability. (4) CDB that generates 2-gram sets of each attribute and uses Jaccard on 2-gram sets to compute the probability. We use the expectation-based method to select tasks to ask the crowd. Figure 23 shows the number of tasks and Figure 24 shows the F-measure for different similarity functions. We have two observations. First, most of similarity functions can indeed reduce the cost, and NoSim without similarity functions incurs higher cost than other methods with similarity functions. CDB, ED and JAC nearly take the same cost because all of the three similarity functions can estimate a good probability for each edge. Second, different similarity functions can slightly affect the quality, and a good similarity function may improve the quality. CDB is slightly better than ED and JAC in terms of quality, because Jaccard may not assign a good probability for attributes with short string, e.g., conference, and edit distance may not assign a good probability on attributes with long string, e.g., title, leading to that some edges with low similarity are wrongly pruned. CDB leverages 2-grams to handle both short and long strings, and thus achieves a little better quality. In our paper, we take the Jaccard similarity as an example and leave selecting good similarity functions as future work.

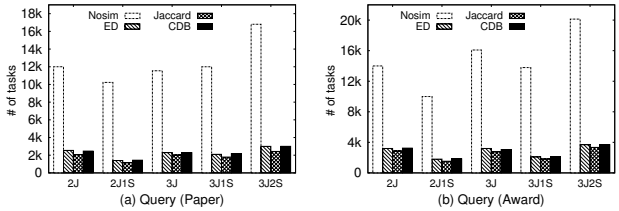


**Figure 23: Varying Similarity Functions: # Tasks**

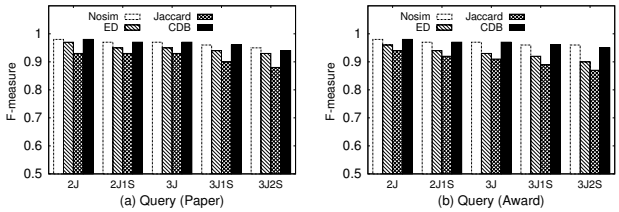


**Figure 24: Varying Similarity Functions: F-measure**

# E. MORE RELATED WORKS

**Quality Control.** To improve crowdsourcing quality, a lot of *quality control* strategies are proposed. Most of the strategies assign a crowdsourcing task to multiple workers and aggregate worker answers to infer the truth of each task, called "*truth inference*". (1) Truth inference has been studied a lot in existing works [35, 37, 40, 36, 66, 61, 49, 70, 17, 59]. The idea is to assign more weights to the answers given by more reliable workers. Thus the key step is to obtain each worker's reliability (or quality). Existing works typically adopt two types of approaches. (i) The first type of approaches [39, 20] leverage a small amount of crowdsourcing tasks with ground truth (called "golden tasks"). A worker is required to answer these golden tasks at first come, and then the worker's quality is initialized based on the answering performance for these tasks. (ii) The second type of approaches [40, 61, 49, 59, 49, 70, 38] purely leverage workers' answers to infer each worker's quality. They often adopt iterative algorithms (e.g., Expectation-Maximization [18, 9, 49, 61], gibbs sampling [40], variational inference [38]), which gradually update each worker's quality until convergence. Compared with the latter approaches, the first approaches make use of golden-tasks, which assume a set of tasks with known truth in advance. Thus CDB mainly uses the more general latter approaches in estimating workers' qualities (CDB also supports the first approaches if the requesters indicate the known golden tasks). CDB adopts similar approaches with existing works, however, we are more general in handling multiple task types, i.e., single choice tasks, multiple choice tasks and fill-in-blank tasks. (2) Task assignment techniques have also been widely studied [69, 29, 28, 48, 20, 11, 50, 22], which assign a set of tasks to the coming worker. The basic idea is to assign the most uncertainty tasks to the worker. In CDB, we consider the coming worker's quality, and assign tasks such that the quality can be improved the most; CDB also supports task assignments on multiple task types.

There are some studies on crowd mining[8, 7], data cleaning [15, 21], spatial crowdsourcing [6, 64, 30], etc.


## Acknowledgment

This work was supported by 973 Program of China(2015CB358700), NSF of China (61632016,61373024,61602488,61422205,61472198), CCF-Tencent RAGR20160108, and FDCT/007/2016/AFJ.