CDB: A Crowd-Powered Database System

Guoliang Li[†], Chengliang Chai[†], Ju Fan[*], Xueping Weng[†], Jian Li[‡], Yudian Zheng[#], Yuanbing Li[†], Xiang Yu[†], Xiaohang Zhang[†], Haitao Yuan[†]
[†]Tsinghua University, [*]Renmin University, [#]Twitter
{liguoliang,lijian83}@tsinghua.edu.cn; chaicl15@mails.thu.edu.cn; fanj@ruc.edu.cn; yudianz@twitter.com

ABSTRACT

Crowd-powered database systems can leverage the crowd's ability to address machine-hard problems, e.g., data integration. Existing crowdsourcing systems adopt the traditional tree model to select a good query plan. However, the tree model can optimize the I/O cost but cannot optimize the monetary cost, latency and quality, which are three important optimization goals in crowdsourcing. To address this limitation, we demonstrate CDB, a crowd-powered database system. CDB proposes a new graph-based model that adopts a fine-grained tuple-level optimization model which significantly outperforms existing coarse-grained tree-based optimization models. Moreover, CDB provides a unified framework to simultaneously optimize the monetary cost, quality and latency. We have deployed CDB on well-known crowdsourcing platforms and users can easily use our system to deploy their applications. We will demonstrate how to use CDB to address real-world applications, including web table integration and entity collection.

PVLDB Reference Format:
Guoliang Li, Chengliang Chai, Ju Fan, Xueping Weng, Jian Li, Yudian Zheng, Yuanbing Li, Xiang Yu, Xiaohang Zhang, Haitao Yuan. CDB: A Crowd-Powered Database System. *PVLDB*, 11 (12): 1926 - 1929, 2018.
DOI: https://doi.org/10.14778/3229863.3236226

1. INTRODUCTION

Crowdsourcing aims to leverage the crowd's ability to solve the problems that are hard for the machines, e.g., entity resolution [2]. Inspired by traditional DBMS, crowdsourcing database systems, such as CrowdDB [4], Qurk [7], Deco [8], and CrowdOP [3], have been recently proposed and developed. There are two major limitations in existing crowdsourcing systems. First, to optimize a query, existing systems always adopt the traditional tree model to select a good query plan. The tree model can optimize the I/O cost but cannot optimize the monetary cost, latency and quality, which are three important optimization goals in crowdsourcing. Second, existing crowdsourcing systems mainly focus on optimizing the monetary cost while ignoring other two significant optimization goals (e.g., reducing the latency and improving the quality) in crowdsourcing. To address these limitations, we develop a novel crowd-powered database system, CDB, which has the following salient features.

Fine-Grained Query Model. To optimize a query, existing systems adopt a tree model to select an optimized table-level join order. However, it generates the same order for different joined tuples and limits the optimization potential that different joined tuples can be optimized by different orders, and thus the tree model only provides a coarse-grained table-level optimization. This is possibly because the optimization goal of traditional databases is to reduce random access. While in crowdsourcing, one optimization goal is to reduce the number of tasks that reflects the monetary cost. To this end, we propose a graph-based query model that provides the fine-grained tuple-level optimization [5].

Multi-Goals Optimization. In crowdsourcing, there are three optimization goals, i.e., smaller monetary cost, lower latency, and higher quality. However, most of the existing systems only focus on optimizing monetary cost, and they adopt the simplest *majority voting* strategy for quality control, without modeling the latency control. Thus it calls for a new crowd-powered system to enable the multi-goal optimization. We devise a unified framework to perform the multi-goal optimization based on the graph model [5].

Cross Platform Deployment. We have implemented and deployed our system on Amazon Mechanical Turk (AMT), CrowdFlower and ChinaCrowd. Given a query, our system can execute the crowd-based operations by publishing tasks to any of these platforms. Thus our system can be easily and user-friendly used to deploy crowdsourcing applications.

Demonstration Scenarios. *(1) Web Data Integration.* Web tables contain a large amount of data but with rather low quality. CDB utilizes crowd-powered operations to integrate the data. For example, CDB can identify the product with the lowest price from Amazon, eBay and Best_buy. CDB can also find the publications from ACM, DBLP, and Google scholar that refer to the same entity. We will demonstrate how CDB generates the query plan to reduce the cost while achieving high quality and low latency. The participants can pose online queries to find the products with the lowest price from different sources. *(2) Entity Collection.* CDB can collect entities with low cost, e.g., collecting NBA players born in California. As different crowd workers may return duplicated results, it is important to avoid duplicates to reduce the cost. We will demonstrate how CDB collects answers with few duplicates and how CDB interacts with the crowd.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/08.
DOI: https://doi.org/10.14778/3229863.3236226

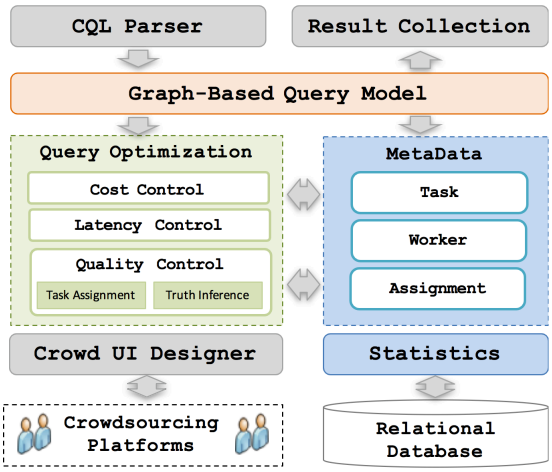


Figure 1: System Overview

		CrowdDB	Qurk	Deco	CrowdOP	CDB
Crowd Powered Operators	COLLECT	√	×	√	×	√
	FILL	√	×	√	√	√
	SELECT	√	√	√	√	√
	JOIN	√	√	√	√	√
	ORDER	√	√	×	×	√
	GROUP	×	×	×	×	√
Optimization Objectives	Cost	√	√	√	√	√
	Latency	×	×	×	√	√
	Quality	×	×	×	×	√
Optimization Strategies	Cost-model	×	√	√	√	√
	Tuple-level	×	×	×	×	√
	Budget-supported	×	×	×	×	√
Task Deployment	Cross-Market	×	×	×	×	√

Figure 2: Comparison of Crowdsourcing Systems.

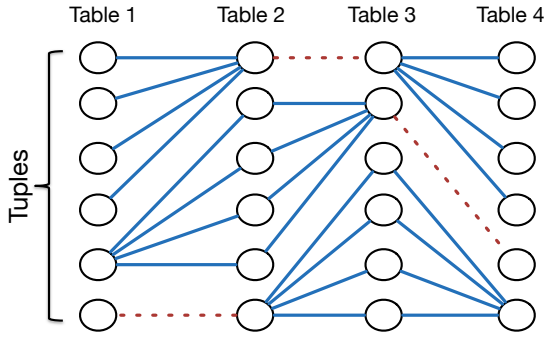


Figure 3: An example of tuple-level optimization

2. SYSTEM ARCHITECTURE

The architecture of CDB is shown in Figure 1. CDB still uses the relational data model and extends SQL to define a declarative query language CQL by adding some crowd-powered operations into SQL, e.g., CROWDEQUAL, CROWDJOIN. A requester (user) defines her data and submits her query with crowd-powered operations using CQL, which will be parsed by CQL Parser. Then Graph-based Query Model builds a graph model. Next Query Optimization generates an optimized query plan, where cost control selects a set of tasks with the minimal cost, latency control identifies tasks that can be asked in parallel, and quality control decides how to assign each task to appropriate workers and infer the truth. 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.

Figure 2 shows the differences between CDB and existing systems. CDB supports many more crowdsourcing operations, enables multiple optimization goals (cost, latency and quality), provides tuple-level fine-grained optimization and is deployed into multiple crowdsourcing markets. Please refer to [5] for more details. The source code is available at https://github.com/TsinghuaDatabaseGroup/CDB.

2.1 Graph Query Model

After parsing the CQL query, we obtain the operators that a requester wants to execute. CDB supports all relational operators such as join, selection, group-by, aggregation. We define a graph-based query model to provide a fine-grained optimization on CQL queries. Given a CQL query, we construct a graph, where each vertex is a tuple in a table and each edge connects two tuples based on the join/selection predicates in the CQL. For example, consider the tuples in Figure 3. 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 traditional tree-based optimization model first selects two tables to join (involving 9 tasks) and then joins with the third table and the fourth table. 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 could be saved.

Formally, given a CQL query with N join predicates, we construct a graph $\mathcal{G}$ according to the CROWDJOIN predicates (i.e., asking the crowd to check whether two tuples can be joined), 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 $\mathcal{C}_i/\mathcal{C}_j$. (The matching probability can be computed based on the similarity of $t_x[\mathcal{C}_i]/t_y[\mathcal{C}_j]$, e.g., Jaccard similarity [5]). Besides, in order to reduce the cost, we will remove some edges whose weights are below a threshold ε. In the graph $\mathcal{G}$, 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. And obviously, a candidate is an answer if each edge in the candidate is BLUE. 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 some BLUE or RED edges disconnected, and we do not need to ask such edges, which are called *invalid edges*.

Based on this observation, we propose an effective task selection algorithm in [5]. The intuitive idea is that we aim to first ask edges that can result in the maximum benefit. Consider an edge $e = (t, t')$ where t and t' are from tables T and T' respectively, and $\omega(e)$ is the matching probability of t and t'. If cutting the edge makes some edges invalid, we compute its pruning expectation by the probability of cutting the edge (i.e., $1-\omega(e)$) times the number of invalid edges introduced by this cutting. Then we compute the pruning

expectation for every edge, sort them by the expectation in descending order, and select the edge in order as tasks.

To support selection operation, we integrate it into our graph model instead of just pushing it down as existing systems. For each crowd-powered selection $\mathcal{T}.\mathcal{C}_i$ `CROWDEQUAL value`, we add a new vertex (with this value) 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 $\omega(t[\mathcal{C}_i], \texttt{value})$. We discuss how to support other operators in Section 2.3.

2.2 Query Optimization

2.2.1 Cost Optimization

We optimize the cost through the graph model in Section 2.1. We select the edge with the largest pruning expectation to ask and thus provide a fine-grained tuple-level optimization model compared with the traditional coarse-grained table-level model.

2.2.2 Latency Optimization

We use the round model to quantify the latency, i.e., the number of rounds to publish the tasks to a platform, and aim to minimize the number of rounds without increasing the number of tasks. Given two edges e and e', we check whether they are in the same candidate answer. 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. We propose several effective rules [5] to detect whether two edges can be asked simultaneously.

2.2.3 Quality Optimization

In order to derive high-quality results based on workers' answers, it is important to do quality control. `CDB` controls quality from two aspects. (1) When a worker first comes to answer tasks, we estimate the worker's quality and infer the truth of her 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. More details on how `CDB` addresses truth inference and task assignment on crowdsourcing tasks can be found in [12, 11, 13].

2.3 Optimizing Other Operators

2.3.1 Collection and Fill Operations

`CDB` also supports the open-world collection and fill operations, which are not well supported in other systems.

Fill Operation. `CQL` introduces `FILL`, which can be considered as a crowd-powered `UPDATE`, to crowdsource missing attribute values. Given a `CQL` query with fill operation, e.g., filling the affiliations of researchers, we first parse it and then publish a fill task associated with the missing attributes and other attributes to the crowdsourcing platform. After workers finish answering the tasks based on other attributes, we collect the results and fill them to the corresponding missing attributes. Moreover, `FILL` also allows a requester to fill a part of missing attribute values, e.g., filling the affiliations of female researchers.

Collection Operation. We also design `COLLECT` in `CQL` to collect more tuples from the crowd for a `CROWD` table. For example, if we want to collect NBA players using `CQL` *`collect NBAPlayer.name, NBAPlayer.birthplace`*. Our system will parse the `CQL` first, publish collection tasks and the workers will collect the answers for the requester. To avoid the duplicated entities provided by different workers, we propose an incentive-based strategy to reduce the cost [1].

2.3.2 Sort and Top-k Operation

`CDB` also supports sorting and top-k operations [10, 6], which contains two main steps. The first step infers top-k results based on the current answers of rating and ranking tasks, called *top-k inference*. We model the score of each object as a Gaussian distribution, utilize the rating and ranking results to estimate the Gaussian distribution, and infer the top-k results based on the distributions. The second step selects tasks for a coming worker, called *task selection*. Based on the probability of an object in the top-k results, we get two distributions: real top-k distribution and estimated top-k distribution. We propose an effective task selection strategy that selects tasks to minimize the distance between the real distribution and the estimated distribution. We propose effective algorithms to estimate the distribution so as to minimize the distance of the two distributions [6].

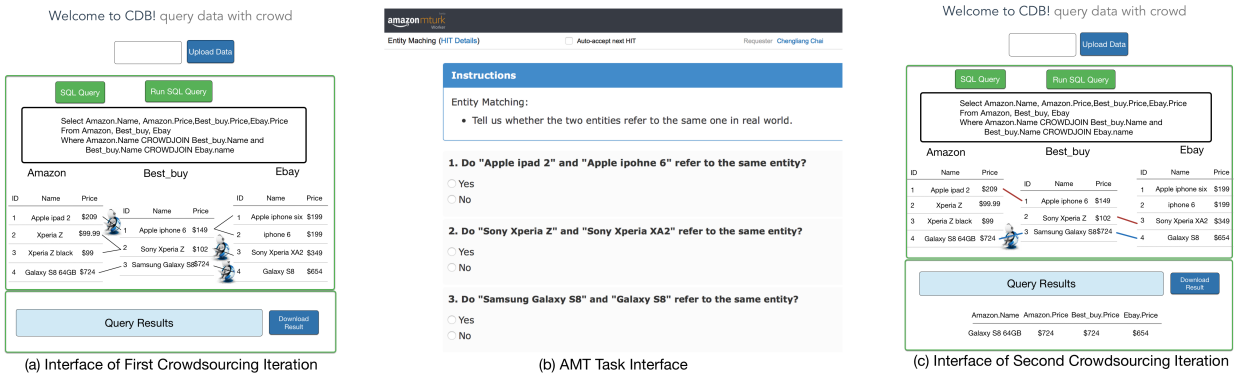
2.3.3 Groupby Operation

`CDB` supports the Groupby operation to aggregate the query results based on a given attribute. We use crowdsourcing entity resolution to support groupby [2]. Specifically, we first use a similarity-based method to identify the possible matching pairs and prune large numbers of un-matching pairs. Then we ask the crowd to verify the pairs. We can use transitivity and partial order to reduce the cost [9, 2].

3. DEMONSTRATION SCENARIOS

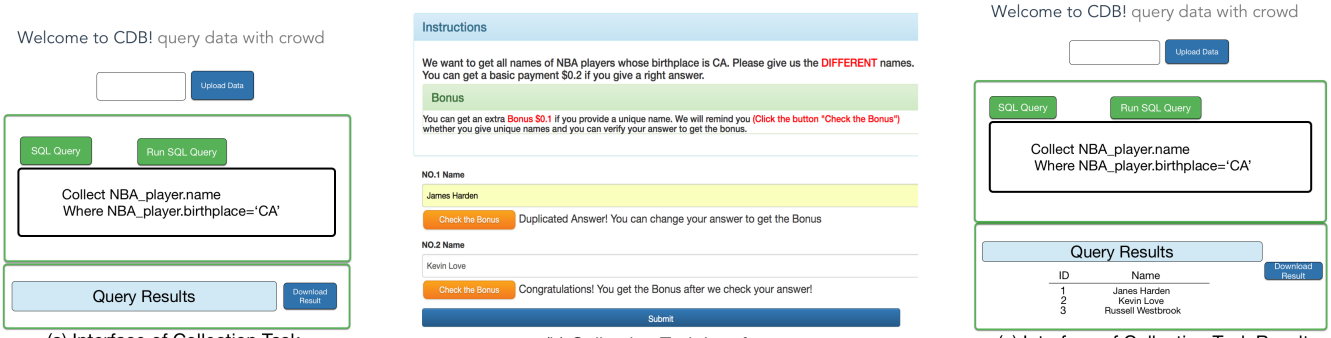
Web Table Integration. We first illustrate the web table integration scenario. Given three tables of electrical products from Amazon, Best_buy and eBay, we aim to compare prices of the same products from different sources. To use `CDB` to address this problem, the requester needs to upload the relational data (i.e., the products). Then, the requester can pose a `CQL` query to find the records that refer to the same entity. Next `CDB` parses the `CQL`, constructs the graph model and publishes entity resolution tasks to a crowdsourcing platform like AMT. In `CDB`, we use a tuple-level task selection approach based on the graph model to optimize the cost. We can see from Figure 4(a) that these tasks are selected from different adjacent tables rather than the same adjacent tables as other crowd-powered systems do. After workers finish a batch of tasks on the platform (like AMT) in Figure 4(b), `CDB` reconstructs the graph model, selects tasks which can achieve the most benefit and publishes another batch of tasks, as shown in Figure 4(c). `CDB` repeats this until getting all the results. At last, `CDB` shows the results to the requester and she can download the results from our platform as shown in the bottom part of Figure 4(c). In this scenario, we ask 4 tasks in total. However, if we apply other crowd-powered systems [4, 7, 8], they will select a table-level order to ask the crowd. Firstly, they will ask four tasks between the first two tables (or the last two). And then they have to ask two tasks between the remaining two tables. Therefore, they task 6 tasks in total, which have weaker pruning ability than `CDB`.

In our demo, we allow participants to pose product search queries, e.g., "iPhone X 256GB", and our system can automatically find the products with the lowest cost from different sources while keeping high quality and low latency.



(a) Interface of First Crowdsourcing Iteration (b) AMT Task Interface (c) Interface of Second Crowdsourcing Iteration

Figure 4: CDB Screenshot for Web Data Integration



(a) Interface of Collection Task (b) Collection Task Interface (c) Interface of Collection Task Result

Figure 5: CDB Screenshot for Entity Collection

We will show how CDB generates the query plan and selects tasks to ask, how CDB iteratively interacts with the crowd, and why CDB can reduce the cost.

Entity Collection. Entity collection aims to collect a set of "open-world" entities, e.g., NBA players born in California. Given this entity collection problem, CDB publishes a query in CDB, as shown in Figure 5(a). Then CDB generates some collection tasks and publishes the tasks on crowdsourcing platforms, e.g., AMT. Workers in AMT will answer our collection tasks like Figure 5(b). Then the requester can view the results and download them as shown in Figure 5(c) during the collection process.

Note that for entity collection tasks, we want to collect *complete entities with low cost*. However the workers may provide duplicate entities. For example, most workers will provide famous players, e.g., "James Harden", "Kevin Love", "Russell Westbrook", "Kawhi Leonard", and these entities will be provided by many workers. Thus existing methods will incur high cost. To address this challenge, CDB employs an incentive-based crowdsourcing model in the entity collection scenario. When a worker provides an answer, she can check whether the entity has been already collected. If it is a duplicated answer, the worker can choose to change other entities and check again until providing a distinct entity. If she provides a distinct entity, she will get some bonus. Otherwise she can only get the basic reward for answering a task. Since workers want to get the bonus, they will think over how to provide distinct entities. We devise effective techniques to encourage workers to provide distinct entities in [1]. Besides, we design strategies to eliminate those workers who provide duplicated answers or wrong answers.

In our demo, we allow participants to pose an entity collection task, e.g., collecting all universities in Rio, and our system will automatically generate the tasks, deploy them on AMT, and interact with workers to collect entities. We show how CDB encourages workers to provide distinct entities and find the results with the lowest cost.

Acknowledgement. This work was supported by the 973 Program of China (2015CB358700), NSF of China (61632016, 61472198,61521002,61661166012), and TAL education.

4. REFERENCES

[1] C. Chai, J. Fan, and G. Li. Incentive-based entity collection using crowdsourcing. In *ICDE*, 2018.
[2] 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.
[3] 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.
[4] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, 2011.
[5] G. Li, C. Chai, J. Fan, X. Weng, J. Li, and Y. Zheng. CDB: optimizing queries with crowd-based selections and joins. In *SIGMOD*, pages 1463–1478, 2017.
[6] K. Li, X. Z. G. Li, and J. Feng. A rating-ranking based framework for crowdsourced top-k computation. In *SIGMOD*, pages 1–16, 2018.
[7] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.
[8] A. G. Parameswaran, H. Park, H. Garcia-Molina, and J. Widom. Deco: declarative crowdsourcing. In *CIKM*, 2012.
[9] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.
[10] X. Zhang, G. Li, and J. Feng. Crowdsourced top-k algorithms: An experimental evaluation. *PVLDB*, 9(8):612–623, 2016.
[11] Y. Zheng, G. Li, and R. Cheng. DOCS: domain-aware crowdsourcing system. *PVLDB*, 10(4):361–372, 2016.
[12] 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.
[13] Y. Zheng, J. Wang, G. Li, R. Cheng, and J. Feng. QASCA: A quality-aware task assignment system for crowdsourcing applications. In *SIGMOD*, 2015.