CrowdOp: Query Optimization for Declarative Crowdsourcing Systems (Extended Abstract)

Ju Fan†, Meihui Zhang§, Stanley Kok§, Meiyu Lu†, and Beng Chin Ooi†

†*National University of Singapore, Singapore 117417*
{fanj, lumeiyu, ooibc}@comp.nus.edu.sg

§*Singapore University of Technology and Design, Singapore 487372*
{meihui_zhang, stanleykok}@sutd.edu.sg

***Abstract*—We propose CROWDOP, a cost-based query optimization approach for declarative crowdsourcing systems. CROWDOP considers both cost and latency in the query optimization objectives and generates query plans that provide a good balance between the cost and latency. We develop efficient algorithms in CROWDOP for optimizing three types of queries: selection, join and complex selection-join queries. We validate our approach via extensive experiments by simulation as well as with the real crowd on Amazon Mechanical Turk.**

I. INTRODUCTION

Crowdsourcing has attracted growing interest in recent years as an effective tool for harnessing human intelligence to solve problems that computers cannot perform well [1], [3]. Recent crowdsourcing systems, Deco [4] for example, provide an SQL-like query language as a declarative interface to the crowd. While declarative querying improves the usability of the system, it requires the system to have the capability to optimize and provide a "near optimal" query execution plan for each query.

In paper [2], we propose a novel optimization approach CROWDOP to finding the most efficient query plan for answering a query. Compared to the query optimization techniques proposed in recent crowdsourcing systems and algorithms, CROWDOP has the following fundamental differences in its design principles: 1) *Supporting cost-based query optimization:* Unlike some other systems that employ a rule-based query optimizer based on several rewriting rules such as predicate push-down, join ordering, etc., CROWDOP, in contrast, adopts cost-based optimization that estimates costs of alternative query plans and uses the one with the lowest estimated cost (with respect to pre-defined cost functions). 2) *Optimizing multiple crowdsourcing operators:* CROWDOP considers three commonly used operators in crowdsouring systems, FILL, SELECT and JOIN. 3) *Tradeoff between monetary cost and latency:* CROWDOP incorporates the cost-latency tradeoff into its optimization objectives. It is capable of finding the query plan with low latency given a user-defined budget constraint, which nicely balances the cost and time requirement of users.

We study the research challenges that naturally arise in the system design of CROWDOP. The first challenge is the formalization of our optimization objectives that consider both monetary cost and latency. To address this challenge, we introduce two optimization objectives: 1) cost minimization without latency constraint; 2) budget-bounded latency minimization. The second challenge is to efficiently select the best query plan with respect to the defined optimization objectives. To this end, we develop a class of optimization algorithms for different types of queries.

II. THE CROWDOP APPROACH

Data Model and Query Language. CROWDOP employs the *relational data model*. Different from traditional databases, some attributes of tuples are *unknown* before executing crowdsourcing. A CROWDOP query Q is an SQL-like query. We consider three query types: 1) *Selection Query* applies human-recognized selection conditions over tuples. 2) *Join Query* leverages human intelligence to combine tuples from relations. 3) *Complex Query* containing both selections and joins, is to help users express more complex crowdsourcing intent.

CrowdOp Query Processing. Like traditional databases, an SQL query is issued by a user and is processed by QUERY OPTIMIZER, which produces an optimized query plan (tree). The key difference is that *crowd-powered operators* are applied: 1) CSELECT *(Crowd-Powered Selection)* abstracts human operation of object filtering. 2) CJOIN *(Crowd-Powered Join)* abstracts human operation of combining objects from sources. 3) CFILL *(Crowd-Powered Fill)* abstracts human operation of filling unknown tuple attributes. The query plan is executed by CROWDSOURCING EXECUTOR iteratively by a bottom-up traversal of the tree. In each iteration, it selects a batch of operators whose inputs are already prepared to generate human intelligence tasks (HITs) and publish these HITs on crowdsourcing platforms. Based on the HIT answers collected from the crowd, it evaluates the query and returns the results.

Overview of Query Optimization. We consider *Monetary cost* and *Latency* as the performance metrics in query optimization. 1) Monetary cost of a query plan is the overall rewards paid for crowdsourcing. It depends on the price of each question in operators and the number of questions. CROWDOP employs effective pricing schemes that consider both question difficulty and worker's effort of evaluating conditions/attribute values. 2) *Latency* is introduced to quantify the speed of query evaluation. However, it is non-trivial to predict and optimize latency due to the dynamic and uncertain property in crowdsourcing market. CROWDOP borrows the idea of existing work to make a simplification: we measure the latency as *the number of iterations* used in crowdsourcing execution, i.e., the height of the query plan. In addition, accuracy issue is addressed by employing our previous work on quality control [3].

CROWDOP balances the cost-latency tradeoff by defining two optimization objectives. 1) *Cost Minimization* finds the query plan with the minimum overall cost. 2) *Cost Bounded Latency Minimization* finds an economical query plan with the lowest latency under a user-provided cost budget.

Selection Query Optimization. Selection query optimization reveals a *cost-latency tradeoff*. Consider a selection query

with n selection conditions (e.g., `make` ="*Volvo*", `color` ="*black*"). On one end of the spectrum, a query plan can *sequentially* crowdsource these n selection conditions: in each iteration, it only examines one condition and propagates the selected tuples to the next iteration. On the other end, the query plan can pack all the selection conditions in a single CSELECT operator, and crowdsources the conditions in *parallel*. The sequential plan, which needs n iterations for crowdsourcing, obviously incurs larger latency than the parallel plan that needs only one iteration. However, while reducing the latency, the parallel plan may introduce more monetary cost. We design a dynamic-programming algorithm to find a proper solution that balances cost and latency. The design is based on two nice properties: 1) *Increasing selectivity*: a condition with lower selectivity must be in the preceding or the same operator compared with another condition with higher selectivity. 2) *Optimal subproblem*: the optimal solution can be computed from optimal solutions of its subproblems.

Join Query Optimization. We introduce a CFILL-CJOIN framework which produces join results in two steps. 1) *Fill step* employs CFILL operators to crowdsource the *unknown* values on attributes involved in the join query. It "clusters" tuples into a *partition tree* with two types of nodes: a partition node specifies some partition condition on a attribute (e.g., `make` = "*Volvo*"), while a leaf node represents the tuples satisfying all conditions along the path from the root to this leaf. 2) *Join step* performs join similarly to hash join. It applies CJOIN operators to the tuples in the same leaf node of the partition tree. Since filling itself incurs cost and more filling does not necessarily lead to overall cost reduction, the join query optimization essentially solves a problem of finding an *optimal partition tree that minimizes the overall cost*. The problem is proved to be NP-hard. We propose a greedy algorithm. It builds the partition tree in a *top-down* manner. At each node, it considers each filling attribute, and makes a local decision on choosing a filling attribute with the minimum estimated cost and compares the cost with that of direct join. If the former has lower cost, it executes the partitioning, generates the partitions (child nodes) and proceeds to the processing at child nodes. To achieve cost bounded latency minimization, we need to solve the problem of partition tree optimization with latency constraint $\bar{L}$. We design a dynamic-programming algorithm based on the property that *any subtree of an optimal partition tree must be optimal*. Based on this, the algorithm constructs the partition tree with height lower than $\bar{L}$ in a *bottom-up* manner by recursively computing optimal subtrees and assembling these subtrees.

Complex Query Optimization. The key challenge is *latency constraint allocation* across operators in the entire plan. We devise a dynamic-programming algorithm based on the optimal subproblem property, as shown in Figure 1. Given latency constraint $\bar{L}$, the optimal query plan rooted at operator o_2^j with latency l must satisfy the following property: *the subtrees of the root must achieve the minimum cost under the latency constraint* $\bar{L} - l$, which implies that the subtrees must also incorporate optimal operator ordering and latency allocation.

III. EXPERIMENTAL EVALUATION

Simulation Evaluation. We study the performance of our approach over complex query CQ_1 and CQ_2. CQ_1 performs one join over two relations with three selection conditions. CQ_2 has two joins over three relations with three selection conditions. We vary the budget requirement and report the latency and cost in Table I (see [2] for comparison with existing systems). The overall latency can be reduced with increasing

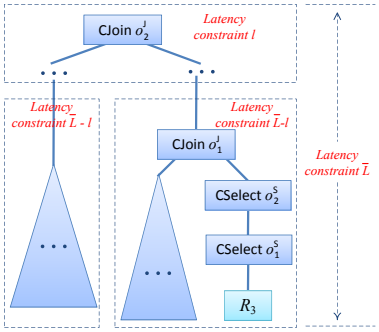
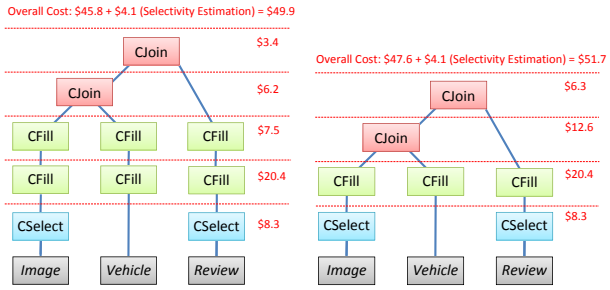


Fig. 1: Optimization for complex query.

Tab. I: Evaluation on complex queries with budgets.

	Budget	CSelect	CFill	CJoin	Total Cost	Latency
CQ_1	120	76.4	31.0	7.6	115	5
	130	85.1	31.0	7.6	123.7	4
	140	85.1	37.6	10.9	133.6	3
CQ_2	240	104.0	104.9	29.6	238.5	8
	250	112.8	104.9	29.6	247.3	7
	260	112.8	88.8	56.4	258.0	6
	270	112.8	88.8	56.4	258.0	6
	280	112.8	90.0	68.9	271.7	5



(a) Under budget \$50. (b) Under budget \$60.

Fig. 2: Query plans for real crowdsourcing.

budget constraint. CROWDOP can effectively allocate the latencies among the operators in the query plan to fit the budget requirements, and achieve better performance. For example, in CQ_2, with increasing the budget requirement, CROWDOP first reduces the latency of CSELECT, which will not incur large cost increase, and then it determines which CFILL attributes can be eliminated by considering the overall CFILL-CJOIN cost. When the budget is large enough, the optimal plan with the minimal latency can be obtained.

Real Crowdsourcing Query Evaluation. Figure 2 shows two query plans produced under different budgets for query CQ_2 on AMT. Under budget \$50, a plan with height 5 is generated, with overall cost \$49.9 and latency 545 minutes. With budget \$60, a different plan with height 4 is generated, with a lower latency 379 minutes but a larger monetary cost \$51.7.

Acknowledgement. Meihui Zhang was supported by SUTD Start-up Research Grant with No. SRG ISTD 2014 084.

REFERENCES

[1] J. Fan, M. Lu, B. C. Ooi, W. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. In *IEEE 30th International Conference on Data Engineering ICDE*, pages 976–987, 2014.

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

[3] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. CDAS: A crowdsourcing data analytics system. *PVLDB*, 5(10):1040–1051, 2012.

[4] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *CIKM*, pages 1203–1212, 2012.