CrowdOp: Query Optimization for Declarative Crowdsourcing Systems

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

Abstract—We study the query optimization problem in declarative crowdsourcing systems. Declarative crowdsourcing is designed to hide the complexities and relieve the user the burden of dealing with the crowd. The user is only required to submit an SQL-like query and the system takes the responsibility of compiling the query, generating the execution plan and evaluating in the crowdsourcing marketplace. A given query can have many alternative execution plans and the difference in crowdsourcing cost between the best and the worst plans may be several orders of magnitude. Therefore, as in relational database systems, query optimization is important to crowdsourcing systems that provide declarative query interfaces. In this paper, we propose CROWDOP, a cost-based query optimization approach for declarative crowdsourcing systems. CROWDOP considers both cost and latency in query optimization objectives and generates query plans that provide a good balance between the cost and latency. We develop efficient algorithms in the CROWDOP for optimizing three types of queries: selection queries, join queries 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.

Index Terms—Crowdsourcing, Query Optimization

1 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, such as translation, handwriting recognition, audio transcription and photo tagging. Various solutions have been proposed to perform common database operations over crowdsourced data, such as selection (filtering) [14], [18], join [12], [21], sort/rank [12], [6], [19] and count [11].

Recent crowdsourcing systems, such as CrowdDB [3], Qurk [13] and Deco [15], provide an SQL-like query language as a declarative interface to the crowd. An SQL-like declarative interface is designed to encapsulate the complexities of dealing with the crowd and provide the crowdsourcing system an interface that is familiar to most database users. Consequently, for a given query, a declarative system must first compile the query, generate an execution plan, post human intelligence tasks (HITs) to the crowd according to the plan, collect the answers, handle errors and resolve the inconsistencies in the answers.

Q_1: SELECT $R_2.*$, R_1.review, R_3.image
FROM REVIEW R_1, AUTOMOBILE R_2, IMAGE R_3
WHERE R_1.sentiment ="*pos*"
AND R_3.color ="*black*" AND R_3.quality ="*high*"
AND R_1.make =R_2.make AND R_1.model =R_2.model
AND R_2.make =R_3.make AND R_2.model =R_3.model

To illustrate a declarative crowdsourcing interface, we consider the three example relations shown in Figure 1: the REVIEW table contains car reviews from customers; the AUTOMOBILE table contains car specifications; the IMAGE table contains car pictures. An example query for finding cars with black color, high-quality images and positive reviews can be formulated as in Q_1.

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. Since a declarative crowdsourcing query can be evaluated in many ways, the choice of execution plan has a significant impact on overall performance, which includes the number of questions being asked, the types/difficulties of the questions and the monetary cost incurred. It is therefore important to design an efficient crowdsourcing query optimizer that is able to consider all potentially good query plans and select the "best" plan based on a cost model and optimization objectives. To address this challenge, 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:

Supporting cost-based query optimization. Like in traditional databases, optimization mechanisms in crowdsourcing systems can be broadly classified into rule-based and cost-based. A rule-based optimizer simply applies a set of rules instead of estimating the cost to determine the best query plan. CrowdDB [3] is an example system that employs a rule-based query optimizer based on several rewriting rules such as predicate push-down, join ordering, etc. While rule-based optimization is easy to implement, it has limited optimization capability and often leads to ineffective execution plans. CROWDOP, in contrast, adopts cost-based optimization that estimates costs of alternative query plans for evaluating a query and uses the one with the lowest estimated cost (with respect to pre-defined cost functions).

Optimizing multiple crowdsourcing operators. CROWDOP considers three commonly used operators in crowdsouring systems: FILL solicits the crowd to fill in missing values in databases; SELECT asks the crowd to filter items satisfying certain constraints; and JOIN leverages the crowd to match items according to some

- *Ju Fan, Meiyu Lu and Beng Chin Ooi are with School of Computing, National University of Singapore, Singapore 117417. E-mail:{fanj, lumeiyu, ooibc}@comp.nus.edu.sg*
- *Meihui Zhang and Stanley Kok are with Information Systems Technology and Design Pillar, Singapore University of Technology and Design, Singapore 487372. E-mail:{meihui_zhang, stanleykok}@sutd.edu.sg*

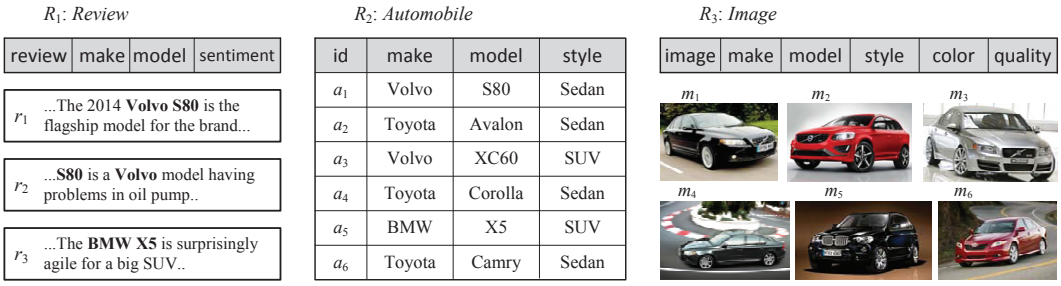


Fig. 1. A running example with three relations: AUTOMOBILE, REVIEW and IMAGE.

criteria. Considering the existing crowdsourcing database systems, Deco [16] focuses on crowdsourcing missing values/records in the database, Qurk [12] on studying the JOIN and SORT operators, and the two recent crowdsourcing algorithms, CrowdScreen [14] and CrowdFind [18], are designed for optimizing SELECT operator. CROWDOP supports cost-based optimization for all the three operators, optimizes the overall cost of all operators involved in a query, and derives the "best" query evaluation plan.

Tradeoff between monetary cost and latency. Two key performance concerns in crowdsourcing systems are monetary cost (how much people pay for crowdsourcing) and latency (how long people wait for results). A good query optimizer should consider the tradeoff between these factors and perform a multi-objective optimization. Neither single-objective solution, i.e., minimizing the cost but incurring heavy latency or reducing latency but incurring high cost, is desirable. We examine recent crowdsourcing works and find most query optimizers only search for query plans with the minimal monetary cost [16], [12], [3], [14]. The only approach taking latency into account is CrowdFind [18] that studies the tradeoff between cost and latency for finding a limited number of items. 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. The first minimizes cost without considering latency, and the second uses budget-bounded latency minimization to judiciously tradeoff cost and latency.

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 selection queries, we study how to balance between the cost and latency when selecting items by using multiple SELECT operators (e.g., R_3.`color` ="*black*", R_3.`make` ="*Volvo*", etc). We devise an algorithm to determine which SELECT operators should be crowdsourced in parallel for reducing the latency, and which ones should be applied over the results filtered by other operators in order to save cost. For join queries, we introduce a hybrid framework that combines FILL and JOIN operators: we leverage the crowd to first fill some missing attributes of items and then join the items having the same attributes. For example, to match cars' images with their reviews, we can first ask the crowd to fill makes of cars and then only match the cars with the same make. A key challenge in this framework is how to balance the costs from FILL and JOIN. To this end, we propose a partition-tree based strategy to select the most appropriate attributes to fill, and devise efficient algorithms for building the partition tree under latency constraints. Finally, we study the optimization of complex queries that involve all the three aforementioned operators with latency constraints.

Summary. We summarize our contributions. 1) We study cost-based query optimization that considers cost-latency tradeoffs and supports multiple crowdsourcing operators. 2) We formalize query optimization objectives to minimize the latency under user-defined cost budget. 3) We develop efficient algorithms for optimizing selection, join and complex queries. 4) We conduct extensive experiments and show that CROWDOP effectively balances the tradeoff between cost and latency and outperforms the state of the art.

The remainder of the paper is organized as follows. Section 2 introduces the data model, query language, and architecture of CROWDOP, and Section 3 overviews the query optimization. Detailed optimization strategies for selection query, join query and complex query are described in Section 4, Section 5 and Section 6 respectively. Section 7 presents the experiments. Section 8 reviews existing work. Section 9 concludes the paper.

2 OVERVIEW OF CROWDOP

2.1 Data Model and Query Language

Data model. CROWDOP employs *relational* data model, like previous work on crowdsourcing systems [3], [12], [15]. In CROWDOP, the data is specified as a schema that consists of a set of relations $\mathcal{R} = \{R_1, R_2, \ldots, R_{|\mathcal{R}|}\}$. These relations are designated by schema designers and can be queried by crowdsourcing users. Figure 1 provides an example schema with three relations. Each relation R_i has a set of attributes $\{A_1^i, A_2^i, \ldots, A_m^i\}$ describing properties of its tuples. Different from traditional databases, some attributes of tuples are *unknown* before executing crowdsourcing, such as REVIEW.`sentiment` and IMAGE.`make`[1].

Query language. A CROWDOP query Q is an SQL query over the designated relations, and its semantics represents the results of evaluating Q over the relations using crowdsourcing. We consider the following three query types.

1) Selection Query. A selection query applies one or more human-recognized selection conditions over the tuples in a single relation. Selection query has many applications in real crowdsourcing scenarios, such as filtering data [14] and finding certain items [18]. A simple example of finding high-quality images of black Volvo sedans is shown below, where selection conditions (e.g., `make` ="*Volvo*") are evaluated using crowdsourcing and the

1. Current version of CROWDOP does not support crowdsourcing new rows/tuples. Only unknown attribute values of existing tuples can be crowdsourced.

image m_1 satisfying all the conditions is returned as a result. More Details will be discussed in Section 4.

Q_2: SELECT R_3.image
FROM IMAGE R_3
WHERE make ="*Volvo*" AND style ="*Sedan*"
AND color ="*black*" AND quality ="*high*"

2) Join Query. A join query leverages human intelligence to combine tuples from two or more relations according to certain join conditions. One typical application of join query is crowdsourcing entity resolution [20], [22], which identifies pairs of records representing the same real-world entity. Other applications include subjective classification (e.g., sentimental analysis) [8] and schema matching [2]. An example join query Q_3 over the relations in Figure 1 is to link the automobile records in R_2 with the images in R_3, which is presented as follows.

Q_3: SELECT $R_2.*$, R_3.image
FROM AUTOMOBILE R_2, IMAGE R_3
WHERE R_2.make =R_3.make
AND R_2.model =R_3.model
JoinFilter R_2.style = R_3.style

Notice that CROWDOP can allow users to specify additional criteria that may help filter the possible join candidates by using the keyword JoinFilter. For example, R_2.style = R_3.style is a join-filter designated in Q_3, which means we do not need to consider the image-automobile pairs with different values on style. A simple strategy to execute Q_3 is to crowdsource the tuples in R_2 and R_3 and obtain the set of tuple pairs satisfying Q_3, such as $\langle a_1, m_1\rangle$, $\langle a_1, m_3\rangle$, etc. We describe more effective optimization strategies for join queries in Section 5.

3) Complex (Selection-Join) Query. CROWDOP supports more general queries containing both selections and joins. These queries can help users express more complex crowdsourcing requirements. Q_1 in Section 1 is an example of the complex query, which finds black cars with high-quality images and "positive" reviews. Optimization strategies for complex queries will be discussed in Section 6.

Discussion. Current version of CROWDOP is designed to support a basic set of SQL queries, i.e., Selection-Join (SJ) queries. While SJ queries are powerful and can express many crowdsourcing intents, they do not encompass other crowdsourcing requirements, such as subjective sorting/top-k [12], human-powered clustering [1], etc., which will be supported in next version.

By default, CROWDOP evaluates the predicate in WHERE clause using crowdsourcing if the predicate involves attributes with unknown values. CROWDOP also allows users to specify the predicates that they want to evaluate using only the values stored in the database by a bypass keyword. For instance, if make ="*Volvo*" is marked as *bypass predicate* in Q_2, the system will only search for the images that have "*Volvo*" as the existing value in the make attribute rather than also searching for images with unknown make using crowdsourcing. In this paper, we focus our discussion on processing the predicates that need crowdsourcing. The bypass predicates can be processed in the traditional way.

2.2 System Architecture

The architecture of query processing in CROWDOP is illustrated in Figure 2. An SQL query is issued by a crowdsourcing user and is firstly processed by QUERY OPTIMIZER, which parses the query and produces an optimized query plan. The query plan is then executed by CROWDSOURCING EXECUTOR to generate human intelligence tasks (or HITs) and publish these HITs

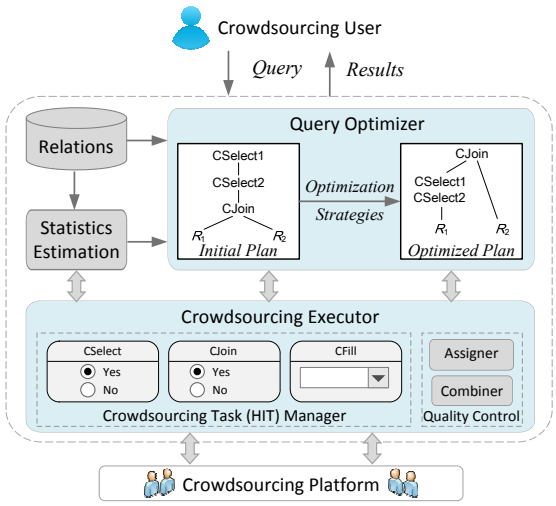


Fig. 2. Architecture of the CROWDOP system.

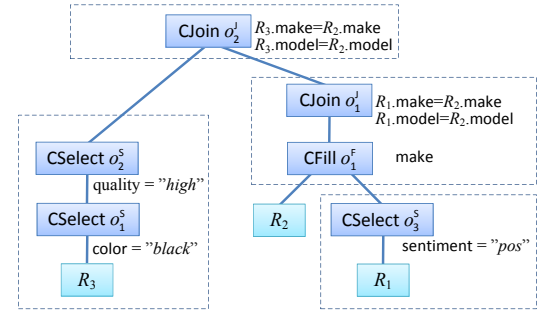
on crowdsourcing platforms, such as Amazon Mechanical Turk (AMT). Based on the HIT answers collected from the crowd, CROWDSOURCING EXECUTOR evaluates the query and returns the obtained results to the user.

Query optimizer. Like traditional databases, QUERY OPTIMIZER also parses an SQL query into a tree-structure query plan and applies optimization strategies to the initial plan. The difference is that tree nodes in a query plan in CROWDOP represent *crowd-powered* instead of machine-based, operators. Typically, a crowd-powered operator (or operator if there is no ambiguity) abstracts a specific type of operation that can be processed by humans. Figure 3(a) illustrates an example query plan for Q_1. In this query plan, three types of crowd-powered operators, i.e., CSELECT, CJOIN and CFILL, are applied for evaluating Q_1. Next, we shall formally define these operators as follows.

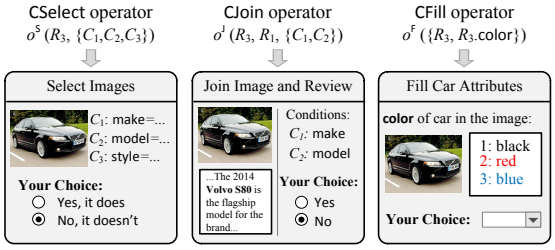
1) CSELECT *(Crowd-Powered Selection).* A CSELECT operator abstracts the human operation of selecting objects satisfying certain conditions. Formally, the input of a CSELECT operator o^{S} consists of a set $\mathcal{T}$ of tuples and a collection $\mathcal{C}$ of selection conditions, and the output is a subset $\mathcal{T}' \subseteq \mathcal{T}$ such that any tuple $t \in \mathcal{T}'$ satisfies all conditions in $\mathcal{C}$. For example, the operator o^{S}_1 in Figure 3(a) leverages the crowd to select the images of "*black*" cars from relation IMAGE R_3, and outputs tuples $\{m_1, m_4, m_5\}$.

2) CJOIN *(Crowd-Powered Join).* A CJOIN operator leverages the crowd to combine objects from two sources according to certain constraints. Formally, the input of a CJOIN operator o^{J} consists of two tuple sets $\mathcal{T}_1$ and $\mathcal{T}_2$ as well as a collection $\mathcal{C}$ of join conditions, and the output is a set $\{\langle t_1, t_2\rangle\} \subseteq \mathcal{T}_1 \times \mathcal{T}_2$ satisfying all conditions in $\mathcal{C}$. For instance, the operator o^{J}_1 combines car records in R_2 and reviews in R_1 using two equi-join conditions R_1.make =R_2.make and R_1.model =R_2.model.

3) CFILL *(Crowd-Powered Fill).* A CFILL operator crowdsources the tuple attributes that are unknown to machines but can be identified by humans. Formally, the input of a CFILL operator o^{F} is a collection of pairs of tuple sets and attributes $\{\langle \mathcal{T}_1, A_1\rangle, \langle \mathcal{T}_2, A_2\rangle, \ldots\}$, and the output is a collection of tuple sets $\{\mathcal{T}_1, \mathcal{T}_2, \ldots\}$ such that any tuple $t \in \mathcal{T}_i$ has its attribute $t.A_i$ filled by the crowd. To fill a tuple attribute, the crowd can either choose a value from attribute's value domain, or fill a new value if no value in the domain fits the tuple. The value domain can be either specified by the schema designer upfront or constructed using the crowdsourcing results on the fly. Take the operator o^{F}_1 in Figure 3(a) as an example. The operator fills make (e.g., "*Volvo*",



(a) Query plan generation.



(b) Crowdsourcing execution.

Fig. 3. Overview of evaluating query Q_1.

"*BMW*") of the tuples from R_1 that have passed o_3^S and the tuples from R_2. Obviously, CFILL can be exploited to facilitate CJOIN operators. In this example, given that o_1^F has filled `make` in R_1 and R_2, we only need to consider the tuples from two tables having the same `make`. More details can be found Section 5.

Crowdsourcing executor. CROWDSOURCING EXECUTOR iteratively executes the operators by a bottom-up traversal of the plan tree. In each iteration, the algorithm selects the operators $\mathcal{O}$ whose inputs are already prepared, and executes the operators by generating Human Intelligence Tasks (HITs) and publishing the HITs on crowdsourcing platforms. After collecting and evaluating the crowd's answers $\mathcal{A}(o)$ for each operator o, the algorithm propagates the answers to the parent operators, and further executes crowdsourcing in the next iteration. When all operators in the plan have been executed, the algorithm terminates and returns the tuples produced by the root operator.

The crowdsourcing executor employs a Crowdsourcing Task (HIT) Manager to generate the HITs to be published on crowdsourcing platforms. The HIT manager carefully design interfaces for each operator type, as shown in Figure 3(b): 1) For the CSELECT operator o^S, a Yes/No question is asked to determine whether a tuple satisfies selection conditions; 2) For the CJOIN operator o^J, a Yes/No question is asked to identify a tuple pair satisfying join conditions; 3) For the CFILL operator o^F, a drop-down list containing domain values (e.g., values for car `color`) is designed to fill the missing value for an attribute. The HIT Manager is also responsible for *HIT-level* optimization. For instance, it can apply effective batch strategies [12] to reduce the number of HITs, and thus further reduce the overall cost. We will not elaborate more details of the HIT manager, as it is out of the scope of this paper.

Selectivity estimation. Optimization strategies may rely on database statistics, e.g., selectivity. However, obtaining these statistics is not trivial. One may require schema designers and/or data providers to provide statistical information for unknown attributes (e.g., the number of car makers) based on domain knowledge [16]. As the data is crowdsourced and becomes more complete, the system can adaptively update the statistics. One may also look to crowdsourcing solutions. One way is to ask the crowd to fill in statistical information based on their general background knowledge. However, this method is data independent and could be highly inaccurate. Another option is to sample the data, perform crowdsourcing *count* tasks [11] and estimate the statistics based on the sample data. This approach is more reliable but may incur high crowdsourcing cost. Estimating selectivity is not the main focus of this paper and we assume selectivity is known in later sections. We use the sampling-based approach in our experiments.

3 Overview of Query Optimization

This section provides an overview of query optimization. We discuss performance metrics in optimization, optimization objectives, and an optimization framework.

3.1 Performance Metrics

Monetary cost. The monetary cost of query plan Q, denoted by $\texttt{cost}(Q)$, is the overall rewards paid for executing all crowdsourcing operators in the query plan $\texttt{P}_Q$, i.e.,$\texttt{cost}(\texttt{P}_Q) = \sum_{o\in\mathcal{O}} \texttt{cost}(o)$, where $\texttt{cost}(o)$ denotes the cost for executing each individual operator $o \in \mathcal{O}$. Intuitively, the cost of an operator depends on the price given to the crowd for each question generated for the operator, called unit cost u. We shall discuss how to define the unit cost for different types of operators as follows.

For a CSELECT with condition set $\mathcal{C}^{\text{S}}$, the unit cost u_{S} needs to be given based on the crowds workload of evaluating conditions $\mathcal{C}^{\text{S}}$. As such, the larger the $|\mathcal{C}^{\text{S}}|$, the higher the unit cost u_{S}. Similarly, a CJOIN operator should be given larger unit cost u_{J} if there are more join conditions $\mathcal{C}^{\text{J}}$ in the operator. For a CFILL operator with attribute A, we consider the unit cost u_{F} depends on the crowd's workload of scanning the value domain of A: the crowd should be given more rewards for filling an attribute with a larger value domain $Dom(A)$.

Based on the above intuition, we can define unit cost as a price function $u(x)$, where x the number of selection conditions/join conditions or the attribute domain size discussed above. Intuitively, one can use any monotonically increasing function (w.r.t. x) to compute the $u(x)$. In our work, we consider the linear price function $u(x) = b + wx$, where b is the base charge for answering one question, and w is the incremental charge of evaluating a condition or scanning an attribute value. For example, consider the operators in Figure 3(b), the CSELECT unit cost $u_{\text{S}} = b + 3 \cdot w$, the CJOIN unit cost $u_{\text{J}} = b + 2 \cdot w$, and the CFILL unit cost $u_{\text{F}} = b + 3 \cdot w$.

A crowdsourcing user can set the parameter b based on the difficulty of crowdsourced questions. For example, comparing two images is usually much harder than checking the equality of two simple strings. In this sense, the user can set a larger b to the former task than the latter. In addition, the user can set w based on worker's incremental effort of evaluating on additional conditions/attribute values.

Latency. As crowdsourcing takes time, *latency* is naturally introduced to quantify the speed of query evaluation. However, it is non-trivial to predict and optimize latency. On one hand, the crowdsourcing tasks are completed by a pool of workers in parallel and the size of the pool is always changing. In other words, workers may enter or leave the pool at any time. On the other hand, the latency can also be affected by other crowdsourcing tasks. There are a limited number of crowdsourcing workers on

public crowdsourcing platforms such as AMT. The workers are free to choose the HITs that interest them, such as those with high reward. In that sense, published crowdsourcing tasks compete against each other for the workers. As such, the latency in real crowdsourcing scenario has many uncertainties.

In this paper, we borrow the idea of [18] to make the following simplification: we measure the latency $L(\mathtt{P}_Q)$ of a query plan $\mathtt{P}_Q$ as *the number of iterations* (also called phases) used in $\mathtt{P}_Q$'s crowdsourcing execution. Notice that this latency model implicitly assumes that each iteration takes around the same amount of time. One iteration refers to one level of the query plan tree and results in one batch of HIT tasks, and the latency $L(\mathtt{P}_Q)$ is equivalent to the height of the query tree (assuming the height of leaves is 0).

We have conducted a set of crowdsourcing jobs to examine this assumption in the real crowdsourcing platform, Amazon Mechanical Turk (see Section 7.2.1). We have the observations that: although the jobs have various numbers of tasks and operator types, the crowd workers take a similar amount of time to complete most of the questions, while a small number of answers may arrive much later. This observation is consistent with [18]. Thus, we can employ the "outstanding cancellation" approach introduced in [18] to make latency of different crowdsourcing tasks being similar. As a first attempt towards studying the cost-latency tradeoffs of *multiple* crowdsourcing operators, we adopt this simplified model in this paper. Nevertheless, this simplification may result in inaccurate latency estimation. We plan to study the latency refinement in future work. As discussed, the uncertain nature of crowdsourcing makes it rather challenging to estimate the latency without some simplification. The most recent study [5] models the latency as a function of crowd arrival rate and the probability of the crowd to accept the tasks, under the assumption that the arrival rate/accept probability follows certain distribution.

Accuracy. Crowdsourcing may yield relatively low-quality results or even noise, if there are spammers or malicious workers. Thus, *accuracy* is taken as another important performance metric to measure the quality of crowdsourcing results. In our CROWDOP system, we address the accuracy issue by employing our previous work on quality control [10] as a building block. Specifically, the quality control model consists of a predicator and a verifier. Given a required accuracy, the predictor estimates the number of workers that are needed to achieve the requirement based on the worker's average accuracy. If no average accuracy is available in the system, a default value 3 is used. The verifier is to resolve the inconsistencies in the results returned by different workers and select the best answer. A probability-based verification model [10] is adopted if each worker's historical performance is monitored, otherwise a simple voting-based strategy is used. In this paper, we focus on studying the cost-latency optimization problems while assuming the accuracy issue has been adequately addressed.

3.2 Optimization Objectives

We define two optimization objectives considered in this paper. The first one only takes into account the monetary cost and aims to find the most economical query plan.

Objective 1 (Cost Minimization). Given a query Q, it aims to find a query plan $\mathtt{P}_Q^*$ that minimizes the monetary cost, i.e., $\mathtt{P}_Q^* = \arg_{\mathtt{P}_Q} \min \mathtt{cost}(\mathtt{P}_Q)$.

As an optimal plan under *Objective 1* may take a long time to complete, we introduce the second objective that also takes latency into consideration. It aims to find an economical query plan with a low latency under a user-provided cost budget.

Objective 2 (Cost Bounded Latency Minimization). Given a query Q and a cost budget $\bar{C}$, it finds a query plan $\mathtt{P}_Q^*$ with bounded cost $C(\mathtt{P}_Q^*) \leq \bar{C}$ and the minimum latency $\mathtt{P}_Q^* = \arg_{\mathtt{P}_Q} \min \mathtt{latency}(\mathtt{P}_Q)$. If there are multiple plans with the minimum latency, it finds the one with lowest cost.

For example, given cost constraint $\bar{C} = 200$, the Cost Bounded Latency Minimization problem aims to find a query plan such that the overall cost is no more than 200 and the query plan tree is of the minimum height. If there are multiple cost-bounded trees having the minimum height, the problem finds the tree with the minimum latency and the lowest cost.

Algorithm 1: OPTFRAMEWORK $\left(Q, \bar{C}\right)$

```
Input: Q: A query; C̄: A cost budget;
Output: P*: An optimized query plan
if C̄ = Nil then
    P* ← COSTOPT (Q)
else
    L̄_min ← COMPUTEMINLATENCY (Q)
    L̄_max ← COMPUTEMAXLATENCY (Q)
    while L̄_min ≤ L̄_max do
        L̄ ← (L̄_min + L̄_max)/2
        P ← LATENCYBOUNDOPT (Q, L̄)
        if P.cost ≤ C̄ then
            P* ← P
            L̄_max ← L̄ − 1
        else L̄_min ← L̄ + 1
return P*
```

3.3 An Optimization Framework

To achieve the objectives introduced in Section 3.2, we introduce an optimization framework in Algorithm 1. This algorithm takes a crowdsourcing query Q and a cost budget $\bar{C}$ as input, and produces the optimized query plan P^*. It considers the following two cases. If the cost budget is not specified ($\bar{C} = \mathtt{Nil}$), it calls Function COSTOPT to find a query plan of Q that achieves the minimum cost (i.e., Cost Minimization). Otherwise, the algorithm applies a *binary search* strategy to find the query plan with the lowest latency under budget $\bar{C}$ (i.e., Cost Bounded Latency Minimization), which will be elaborated as follows.

The basic idea of the algorithm is to first solve the *latency bounded cost minimization* problem, which, given the query Q and a latency constraint $\bar{L}$, finds the query plan with latency bounded by $\bar{L}$ and minimum cost. For example, given latency constraint $\bar{L} = 2$, it finds a query plan such that the height of the plan tree is at most 2 and the overall cost is minimized. Next, taking the solution of latency bounded cost minimization as a building block, denoted by LATENCYBOUNDOPT, the algorithm aims to find and return the lowest latency bound that can produce a query plan satisfying the cost budget $\bar{C}$. To this end, the algorithm exploits a binary search strategy: it first computes the highest and lowest possible latency bounds of Q ($\bar{L}_{max}$ and $\bar{L}_{min}$), which can be easily achieved by considering the number of CSELECT/CFILL/CJOIN operators in Q. Then, it computes the half $\bar{L}$ of the interval $[\bar{L}_{min}, \bar{L}_{max}]$. If it can find a query plan P under latency bound $\bar{L}$ satisfying cost budget $\bar{C}$, the algorithm examines the latency bounds smaller than $\bar{L}$ by setting $\bar{L}_{max} = \bar{L} - 1$.

Otherwise, it considers the latency bounds larger than $\bar{L}$. Applying this half-interval strategy iteratively, it finally finds an optimized plan P^*.

It is not difficult to prove that the above binary search algorithm can find the optimal solution of Objective 2, as cost of the solution of LATENCYBOUNDOPT is monotonically non-increasing with the increase of latency bound $\bar{L}$. We shall present the techniques devised to implement the two functions COSTOPT and LATENCYBOUNDOPT in the following sections.

4 SELECTION QUERY OPTIMIZATION

Cost model. Consider a CSELECT operator o^{S} with condition set $\mathcal{C}^{\mathrm{S}}$ over a set $\mathcal{T}^{\mathrm{S}}$ of tuples. Its cost can be computed as the input tuple size $|\mathcal{T}^{\mathrm{S}}|$ multiplying the unit cost u_{S}, i.e.,

$$\mathtt{cost}(o^{\mathrm{S}}) = |\mathcal{T}^{\mathrm{S}}| \cdot u_{\mathrm{S}}(|\mathcal{C}^{\mathrm{S}}|), \tag{1}$$

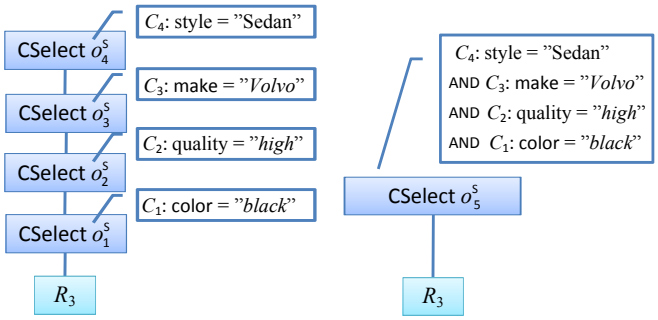
where $u_{\mathrm{S}}(|\mathcal{C}^{\mathrm{S}}|)$ is unit cost function (see Section 3.1). For simplicity, in the examples throughout this section, we shall use a simple function u_{S} as the number of conditions, i.e., $u_{\mathrm{S}}(|\mathcal{C}^{\mathrm{S}}|) = |\mathcal{C}^{\mathrm{S}}|$. For example, using this function, we pay 3 cost units for the CSELECT question in Figure 3(b) as there are three selection conditions.

Cost-latency tradeoff. Consider the selection query Q_2 in Section 2.1. Figure 4 shows two possible query plans for evaluating Q_2. The first one makes use of four CSELECT operators, $o_1^{\mathrm{S}} - o_4^{\mathrm{S}}$, each of which considers only one selection condition. When executing crowdsourcing, this plan *sequentially* crowdsources these operators. Specifically, in each iteration, it only examines one selection condition and propagates the selected tuples to the next iteration. The second query plan packs all the selection conditions in a single CSELECT operator o_5^{S}. Thus, it can crowdsource the conditions in parallel and evaluate Q_2 in one iteration. The sequential plan, which needs four 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. Based on the cost model defined above, the cost of the parallel plan is $|R_3| \cdot u_{\mathrm{S}}(|\mathcal{C}^{\mathrm{S}}|) = 6 \cdot u_{\mathrm{S}}(4)$, while the cost of the sequential plan is $(6+3+2+1) \cdot u_{\mathrm{S}}(1)$ as tuples are selected progressively[2].

Cost estimation. The cost of a query plan $\mathtt{P}_Q$ for a selection query is computed by summing the cost of each CSELECT operator in $\mathtt{P}_Q$. Given the pre-defined unit cost, the key issue is to estimate the size of input tuple set $\mathcal{T}^{\mathrm{S}}$ for each CSELECT operator o^{S} (refer to Equation (1)). As illustrated in Figure 4, the input of the bottom operator comes from the base table, while the upper ones take the tuples that pass the selection conditions in the previous operators. We can therefore estimate the input size using the *selectivity* of the selection conditions. We denote the *selectivity of a condition* C as $s(C)$, which represents the probability that condition C is satisfied, i.e., $Pr(C = \mathtt{true})$. Moreover, like optimization in traditional databases [7], we assume the independence of conditions and thus can calculate the selectivity of the conjunction of conditions as $s(\bigwedge_{i=1}^{n} C_i) = \prod_{i=1}^{n} s(C_i)$. The *selectivity of an operator* can then be defined as $s(o^{\mathrm{S}}) = \prod_{C \in \mathcal{C}^{\mathrm{S}}} s(C)$.

We are now ready to estimate the cost of a selection query plan $\mathtt{P}_Q$. Let R be the base table; the unit cost $u_{\mathrm{S}}(|\mathcal{C}_i|)$ is predefined; $\mathcal{O}$ is an ordered set of CSELECT operators in $\mathtt{P}_Q$ where the operators are ordered in the sequence that they are evaluated; each operator $o_i^{\mathrm{S}} \in \mathcal{O}$ has a set of conditions $\mathcal{C}_i$. The overall cost of $\mathtt{P}_Q$ can be calculated as

$$\mathtt{cost}(\mathtt{P}_Q) = \sum_{o_i^{\mathrm{S}} \in \mathcal{O}} u_{\mathrm{S}}(|\mathcal{C}_i|) \cdot |R| \cdot \prod_{j=1}^{i-1} s(o_j^{\mathrm{S}}) \tag{2}$$



(a) Sequential Plan. (b) Parallel Plan.

Fig. 4. Cost-latency tradeoff of selection queries.

Example 1. We compute the cost for the two query plans in Figure 4. Suppose that selectivity of selection conditions is known, i.e., $s(C_1) = 0.5$, $s(C_2) = 0.83$, $s(C_3) = 0.33$ and $s(C_4) = 0.66$. Then, the cost estimated for the sequential plan is $|R_3| \cdot (1 + 0.5 + 0.5 \cdot 0.83 + 0.5 \cdot 083 \cdot 0.33) = 12.3$. For the parallel plan, as all conditions reside in a single operator, the cost can be estimated as $|R_3| \cdot 4 = 24$.

Algorithm for cost minimization. We first consider the optimization objective that finds the query plan with minimum cost without latency constraint. It is easy to see that the optimal plan must be a sequential plan as illustrated in Example 1, because the cost of simultaneously evaluating multiple conditions in an operator is always larger than or equal to (w.r.t. the example price function $u_{\mathrm{S}}(|\mathcal{C}^{\mathrm{S}}|) = |\mathcal{C}^{\mathrm{S}}|$) the cost of evaluating the conditions one by one. Based on Equation (2), since the price function u_{S} is predefined and the base table R is fixed, we can prove that the optimal sequential plan can be obtained by evaluating the conditions in the selection query Q in increasing order of their selectivity [7].

Algorithm for latency-bounded cost minimization. We develop an algorithm to achieve latency-bounded cost minimization. We first define the optimization problem and then introduce two lemmas. Intuitively, the optimization needs to not only "pack" selection conditions into at most $\bar{L}$ groups, but also determine the ordering of the obtained groups.

Formally, given conditions $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$, it considers each ordered set of groups $\mathcal{G} = \langle G_1, G_2, \ldots \rangle$ satisfying all the properties below: 1) A group G_i is a subset of $\mathcal{C}$, i.e., $G_i \subset \mathcal{C}$; 2) The groups cover all the conditions and are mutually disjointed, i.e., $\bigcup_i G_i = \mathcal{C}$ and $\forall i \neq j, G_i \cap G_j = \emptyset$; 3) The size of the group set $|\mathcal{G}| \leq \bar{L}$. Obviously, a group set $\mathcal{G}$ uniquely corresponds to a query plan where each $G \in \mathcal{G}$ corresponds to a CSELECT operator. Thus, query optimization naturally becomes finding the optimal group set $\mathcal{G}^*$ with minimum estimated cost.

Lemma 1 (Increasing Selectivity). $\forall i, j$ $(i \neq j)$, if selectivity $s(C_i) < s(C_j)$, then, in the optimal group set $\mathcal{G}^*$, the group $G_{i'}$ containing C_i and the group $G_{j'}$ containing C_j must satisfy $i' \leq j'$[3].

2. Referring to R_3 in Figure 1 and 4(a), 6 tuples are fed to o_1^{S}, 3 of them satisfy C_1 and are passed to o_2^{S}, and 2 out of 3 satisfy C_2, and finally only 1 satisfies C_3.

3. See the appendices for all lemma proofs.

Algorithm 2: OPTSELECT ($\mathcal{C}, \bar{L}$)

Input: $\mathcal{C}$: Selection conditions ; $\bar{L}$: Latency constraint
Output: P: A query plan

Sort $\mathcal{C}$ in increasing order of *selectivity* ;
/* Initializing $G(i,j)$ */
for $i = 1 \dots |\mathcal{C}|$ **do**
  $G(i,1) \leftarrow \texttt{cost}(i, |\mathcal{C}|)$
  for $j = 2 \dots \bar{L}$ **do** $G(i,j) \leftarrow +\infty$
/* Iteratively computing $G(i,j)$ */
for $j = 2 \dots \bar{L}$ **do**
  $G(|\mathcal{C}|, j) \leftarrow \texttt{cost}(|\mathcal{C}|, |\mathcal{C}|)$
  for $i = 1 \dots |\mathcal{C}| - 1$ **do**
    $G(i,j) \leftarrow \min_{k=i}^{|\mathcal{C}|-1} \{\texttt{cost}(i,k) + G(k+1, j-1)\}$
    $p[i][j] \leftarrow k^*$ with the minimum cost
Generate query plan P w.r.t $G(1, \bar{L})$ from $p[\cdot][\cdot]$
return P

Lemma 1 provides us the following insight. To compute an optimal group set, we can first sort conditions in increasing order of their selectivity, and then determine how to group the conditions. Next, we formalize this grouping problem as follows. Given a sorted set of conditions $\langle C_1, C_2, \dots, C_m \rangle$, it finds a group set $\mathcal{G}^*$ with $|\mathcal{G}^*| \leq \bar{L}$ and the minimum cost. This problem has the optimal subproblem property, which enables us to develop a dynamic programming algorithm. For simplicity, we slightly abuse notation $G(i, L)$ to denote both the L-size group set over subset $\langle C_i, \dots, C_m \rangle$ and its cost. We use $\texttt{cost}(i,k)$ to denote the cost of the group containing the conditions $C_i, C_{i+1}, \dots, C_k$, which can be computed as $u_{\mathrm{S}}(k-i+1) \cdot |R| \cdot \prod_{l=1}^{i-1} s(C_l)$, according to our cost estimation method mentioned previously.

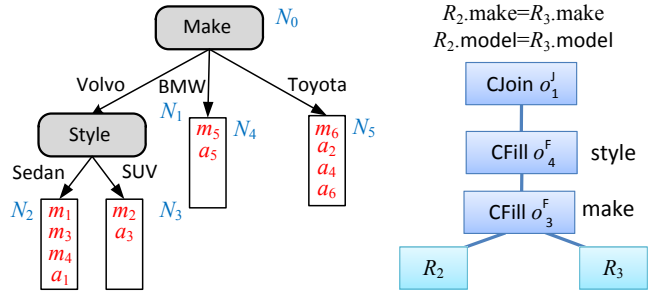
Lemma 2 (Optimal Subproblem Property). For any subset of conditions $\mathcal{C}' = \langle C_i, \dots, C_m \rangle$ with $1 \leq i \leq m$ and any $1 \leq L \leq \bar{L}$, the optimal group set $G^*(i, L)$ dividing $\mathcal{C}'$ into L groups satisfies:

$$G^*(i,L) = \min_{k=i}^{m-1} \{\texttt{cost}(i,k) + G^*(k+1, L-1)\}, \quad (3)$$

where $G^*(i,1) = \texttt{cost}(i,m)$.

Based on Lemma 2, we develop a dynamic programming algorithm, as shown in Algorithm 2. The basic idea is to compute the final solution $G^*(1, \bar{L})$ in a bottom-up manner: it first computes the solutions for its subproblems and then iteratively derives $G^*(1, \bar{L})$ by applying Lemma 2. Specifically, the algorithm takes as input the set $\mathcal{C}$ of conditions in query Q and latency constraint $\bar{L}$, and outputs a query plan. It first sorts $\mathcal{C}$ in increasing order of selectivity, and then computes $G(i,j)$ for $\forall i \in \{1, \dots, |\mathcal{C}|\}, j \in \{1, \dots, \bar{L}\}$ and employs $p[\cdot][\cdot]$ to materialize grouping indexes. Finally, the algorithm generates a query plan P w.r.t. $G(1, \bar{L})$ from $p[\cdot][\cdot]$ and returns it. The time complexity of the algorithm is $\mathcal{O}(\bar{L} \cdot |\mathcal{C}|^2)$.

Example 2. Consider the conditions $\{C_1, C_2, C_3, C_4\}$ in Figure 4 and a latency constraint $\bar{L} = 2$. Algorithm 2 first sorts the conditions based on selectivity (given in Example 1) into $\langle C_3, C_1, C_4, C_2 \rangle$. Next, to compute $G(1,2)$, the algorithm first computes $G(i,1)$ as $\texttt{cost}(i,4)$. For instance, $\texttt{cost}(3,4)$ is the cost of group $\{C_4, C_2\}$, which can be computed as $u_{\mathrm{S}}(2) \cdot |R| \cdot s(C_3) \cdot s(C_1)$. Then, it computes $G(1,2)$ as $\min_{k=1}^{3} \{\texttt{cost}(1,k) + G(k+1, 1)\}$, and outputs group set $\langle \{C_3\}, \{C_1, C_4, C_2\} \rangle$.



(a) A partition tree $\mathcal{S}$. (b) Plan based on $\mathcal{S}$.

Fig. 5. CFILL-CJOIN Framework for example query Q_3.

Discussion. We discuss general cases where u_{S} can be any monotonically increasing function. In such cases, grouping conditions may not only reduce the latency, but also lower the cost. Consider an extreme example that $u_{\mathrm{S}}(|\mathcal{C}|) = 1$ which means we pay 1 unit cost regardless the number of conditions. Then, a parallel plan as shown in Figure 4(b) can always achieve the lowest cost and minimum latency. Algorithm 2 applies to general cases as well.

5 JOIN QUERY OPTIMIZATION

5.1 A CFill-CJoin Framework

A naïve query plan that only uses CJOIN for comparing all tuple pairs is obviously not effective. To address the limitation of this CJOIN-only approach, we introduce a CFILL-CJOIN framework which produces join results in the following two steps.

Fill step: We employ CFILL operators to crowdsource the missing values on some attributes involved in either join conditions or join-filters, which are called *candidate* CFILL attributes. For instance, in query Q_3, there are three candidate CFILL attributes, `make`, `model` and `style`. The attributes that are actually filled by the crowd are called CFILL attributes. After the fill step, we have the knowledge of the values of some attributes and can "cluster" the tuples into a *partition tree*, as shown in Figure 5(a). A partition tree has two types of nodes, partition nodes and leaf nodes. A partition node specifies some condition on a single attribute (e.g., `make` = "*Volvo*"), while a leaf node represents the tuples satisfying all the conditions along the path from the root to this leaf. For example, the leftmost leaf N_2 contains four tuples $\{m_1, m_3, m_4, a_1\}$ that satisfy the two conditions `make` = "*Volvo*" and `style` = "*Sedan*".

Join step: This step performs join similarly to hash join. It applies CJOIN operators to the tuples in the same leaf node, as any join output tuple must have the same join key or join-filter value. For instance, given the partition tree in Figure 5(a), this step only compares the IMAGE and AUTOMOBILE tuples in the same partition, such as $\langle m_1, a_1 \rangle$, $\langle m_6, a_2 \rangle$, etc.

Figure 5(b) illustrates the corresponding query plan for answering Q_3 obtained through the above two steps. In this plan, we first employ CFILL operators to fill tuples from R_2 and R_3 on attributes `make` and `style`. After that, a CJOIN operator is used to join the tuples in the same leaf nodes. In fact, the query plan is generated based on the partition tree. The tree height implies the number of CFILL operators to be generated in the plan. One level of partition nodes in the tree corresponds to one CFILL operator in the query plan, but in the opposite direction. For instance, the root node `make` converts to the bottom CFILL operator in the query plan and the lower-level node `style` corresponds to the upper CFILL operator. We notice that not all of the candidate CFILL attributes are filled in the optimized plan. Further, it may

choose only a subset of tuples to fill in their missing values. This is because filling itself incurs cost and more filling does not necessarily lead to cost reduction in the join step. We next discuss the cost computation and partition tree optimization.

5.2 Cost Computation for the CFill-CJoin Framework

Intuitively, the cost can be computed from the partition tree, and it consists of the following two components: 1) the CFILL cost for building the partition tree, and 2) the CJOIN cost of joining the tuples in the same leaf in the partition tree. More formally, let $\mathcal{S}$ denote the partition tree where partition (intermediate) node and leaf node are respectively denoted by N_p and N_l. We slightly abuse the notation to use N_p to denote both a partition node and the attribute on this node. Similarly, we use N_l to represent both a leaf node and the set of tuples contained in this leaf.

CFILL cost. Cost of the fill step depends on the CFILL tasks to be crowdsourced. Consider the partition tree in Figure 5(a). Each partition node takes in its input tuples (e.g, the root node `make` takes tuples from R_2 and R_3 and the node `style` takes the tuples having "*Volvo*" as their `make`) and leverages the crowd to fill in a specific attribute value[4]. Formally, let us denote the set of input tuples of N_p as $\texttt{In}(N_p)$. The overall CFILL cost can be computed by summing the cost incurred at all partition nodes in the partition tree, i.e.,

$$\texttt{cost}^{\texttt{F}} = \sum_{N_p \in \mathcal{S}} |\texttt{In}(N_p)| \cdot u_{\texttt{F}}(|Dom(N_p)|), \tag{4}$$

where $u_{\texttt{F}}(|Dom(N_p)|)$ is the unit cost of filling N_p.

Example 3. Take the root node `make` in Figure 5(a) as an example. As it takes as input 12 tuples (6 from AUTOMOBILE and 6 from IMAGE) and the value domain size is 4, the cost of this node is $12 \times u_{\texttt{F}}(4)$. Similarly, the cost of node `style` is $6 \times u_{\texttt{F}}(2)$. Overall, the CFILL cost of the partition tree is $12 \times u_{\texttt{F}}(4) + 6 \times u_{\texttt{F}}(2)$.

CJOIN cost. The CJOIN cost depends on how many tuple pairs residing in the same leaf nodes in the partition tree. Recall that, for each of such pairs, we model the crowd's workload as the unit function $u_{\texttt{J}}(|\mathcal{C}^{\texttt{J}}|)$. Based on this unit cost, we compute the CJOIN cost as

$$\texttt{cost}^{\texttt{J}} = \sum_{N_l \in \mathcal{S}} |N_l.\mathcal{T}_1| \cdot |N_l.\mathcal{T}_2| \cdot u_{\texttt{J}}(|\mathcal{C}^{\texttt{J}}|). \tag{5}$$

where $N_l.\mathcal{T}_1$ and $N_l.\mathcal{T}_2$ are tuple sets from input tables.

Example 4. Considering all the leaves in Figure 5(a), we can compute the overall CJOIN cost: $(3 \times 1 + 1 \times 1 + 1 \times 1 + 1 \times 3) \times u_{\texttt{J}}(2) = 8 \times u_{\texttt{J}}(2)$.

Overall, the cost of the query plan generated from the partition tree $\mathcal{S}$ is computed by summing CFILL and CJOIN costs, i.e., $\texttt{cost}(\mathcal{S}) = \texttt{cost}^{\texttt{F}} + \texttt{cost}^{\texttt{J}}$.

5.3 Partition Tree Optimization

This section studies the problem of finding the optimal partition tree that minimizes the overall cost.

Definition 1 (Partition Tree Optimization). Given a join query Q with input tuple sets $\mathcal{T}_1$ and $\mathcal{T}_2$ and candidate CFILL attributes $\mathcal{A}^{\texttt{F}}$, it finds the best partition tree $\mathcal{S}^*$ with the minimum overall cost, i.e., $\mathcal{S}^* = \arg_{\mathcal{S}} \min \texttt{cost}(\mathcal{S})$.

4. Note that the input tuples are only those having missing attribute value. Thus, the crowdsourcing will not be applied to do redundant work.

Algorithm 3: OPTJOIN($\mathcal{T}_1, \mathcal{T}_2, Q^{\texttt{J}}, \bar{L}$)

Input: $\mathcal{T}_1, \mathcal{T}_2$: tuple sets to join ; $Q^{\texttt{J}}$: join query; $\bar{L}$: latency constraint;
Output: P: A query plan

```
S ← GENPARTREE (T1, T2, Q^J.A^F, Q^J.C^J, L̄)
P ← GENPLAN (S)
return P
```

Algorithm 4: GENPARTREE($\mathcal{T}_1, \mathcal{T}_2, \mathcal{A}^{\texttt{F}}, \mathcal{C}^{\texttt{J}}, \bar{L}$)

Input: $\mathcal{T}_1, \mathcal{T}_2$: tuple sets to join ; $\mathcal{A}^{\texttt{F}}$: CFILL attributes; $\mathcal{C}^{\texttt{J}}$: join conditions; $\bar{L}$: latency constraint;
Output: N: A partition tree rooted at node N

```
N ← INITIALIZE (T1, T2)
N.cost ← ESTCJOINCOST (N, C^J)
if L̄ = 1 then return N
for each attribute A ∈ A^F do
    cost(A) ← ESTCFILLCOST (N, A)
    for each value v ∈ Dom(A) do
        T1' ← ESTCARD(T1, A, v)
        T2' ← ESTCARD(T2, A, v)
        N_c ← GENPARTREE(T1', T2', A^F − {A}, C^J, L̄ − 1)
        cost(A) ← cost(A) + N_c.cost
        SETNODE(A, v, N_c)
A* ← the attribute with minimum cost(A)
if cost(A*) < N.cost then
    N.cost ← cost(A*)
    for each value v ∈ Dom(A*) do
        N_c ← GETNODE(A*, v)
        ADDCHILD(N, N_c)
return N
```

Lemma 3. The problem of finding a partition tree with minimum overall cost is NP-hard.

Algorithm for cost minimization. To solve the partition tree optimization problem, we use a greedy heuristic. The algorithm builds the partition tree in a top-down manner and iteratively constructs the tree nodes. It first initializes the root node N as a single partition containing all tuples from $\mathcal{T}_1$ and $\mathcal{T}_2$. Given the root node, it determines whether to partition tuples in N by an attribute in $\mathcal{A}^{\texttt{F}}$, or directly join these tuples without partitioning. To make the decision, it considers each attribute $A \in \mathcal{A}^{\texttt{F}}$ and estimates the cost of first partitioning tuples by A and then joining the tuples in the same partitions. We denote the cost as $\texttt{cost}(A)$. It then selects the attribute A^* with the minimum estimated cost, i.e., $A^* = \arg_{A \in \mathcal{A}^{\texttt{F}}} \min \texttt{cost}(A)$, and compares the cost with that of direct join. If the former has lower cost, the algorithm executes the partitioning, generates the partitions (child nodes) and proceeds to the processing at child nodes. Finally, it returns the partition tree rooted at node N.

Algorithm for latency-bounded cost minimization. We next study the partition tree optimization with latency constraint $\bar{L}$, which essentially finds the one with minimum cost from possible partition trees with heights not greater than $\bar{L}$. The problem can be solved by a dynamic-programming algorithm, based on the optimal subproblem property: *any subtree of an optimal partition tree must be optimal*. This property can be easily proved by contradiction: if a subtree of an optimal plan is not optimal, a better partition scheme can be applied to the subtree to further reduce

the cost of the entire tree, which results in a contradiction. Based on the optimal subproblem property, we devise an algorithm, as shown in Algorithm 4. The algorithm recursively constructs the partition tree. At each node N, it initializes the node with its input tuples and estimates the cost of performing a direct join. It then checks for the latency constraint: 1) If further partitioning is allowed, it examines each remaining candidate CFILL attribute and generates new partitions. And for each new partition it recursively looks for the optimal subtree; 2) Otherwise, it returns node N. When the optimal subtree is returned, it compares the cost of having the subtree with the cost of doing direct join, and performs the ADDCHILD function to assemble the nodes if the former is cheaper. This way, the partition tree is constructed bottom up. After that, Algorithm 3 generates a query plan based on the constructed partition tree and returns it.

6 COMPLEX QUERY OPTIMIZATION

We focus our discussion on latency-bounded cost minimization for complex query optimization. For the case where the latency constraint is not imposed, we can optimize the query plan similarly to traditional databases: apply some heuristic rules, such as pushing down selections and determining the join ordering, and then invoke the above-mentioned techniques for optimizing selections and joins.

6.1 Latency Constraint Allocation

As shown previously, latency constraints would largely affect the cost of the query plan. Therefore, to achieve latency-bounded cost minimization, we need to carefully allocate latency constraints across operators in the entire plan. We illustrate this via an example shown in Figure 6(a). Given an overall latency constraint $\bar{L} = 5$, this query plan allocates the latency as follows: it assigns 2 to the selections over R_3 (the selection over R_1 can be executed simultaneously with latency 1), 1 to the join between R_2 and R_3 and 2 to the join between R_1 and R_2. Another possible plan (not shown in the figure) may allocate 1 (instead of 2) to the selections over R_3 and preserve 1 for its subsequent join with R_2. The join can then be implemented using CFILL-CJOIN approach and potentially save cost.

Moreover, latency constraint allocation may be intertwined with the traditional join ordering problem, which makes the optimization more complicated. Consider an alternative query plan shown in Figure 6(b). The plan utilizes a better join order that first joins R_1 and R_2 and then joins the result with R_3. This new join order, combined with elaborately determined latency allocation, can further reduce the cost under latency constraint $\bar{L} = 5$.

6.2 Complex Query Optimization Algorithm

A straightforward method that enumerates all possible query plans with height smaller than latency constraint $\bar{L}$ is very expensive, since it has to explore a large search space. We devise a dynamic-programming algorithm based on the optimal subproblem property, as illustrated in Figure 6(c). Given a latency constraint $\bar{L}$, the optimal query plan rooted at operator o_2^{J} with latency l must satisfy the following suboptimal 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 the optimal operator ordering. Similar to the latency-bound partition tree optimization (Section 5.3), this property can also be proved by contradiction: if the subtree is not optimal, we can always find another entire plan to further reduce the cost. The property holds for nested subtrees as well.

Algorithm 5: LATENCYBOUNDOPT $\left(\mathcal{Q}^{\mathrm{S}}, \mathcal{Q}^{\mathrm{J}}, \mathcal{R}, \bar{L}\right)$

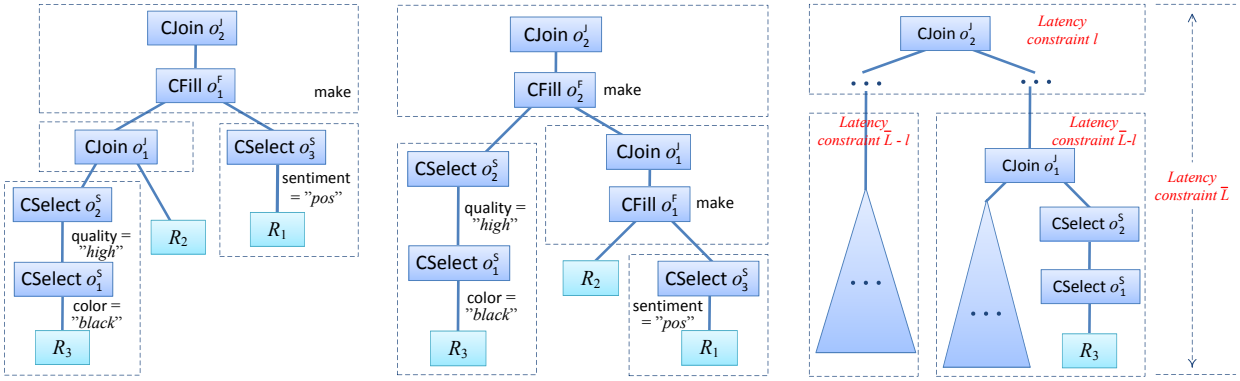
Input: $\mathcal{Q}^{\mathrm{S}}$: Select segments; $\mathcal{Q}^{\mathrm{J}}$: Join segments; $\mathcal{R}$: Relations; $\bar{L}$: Latency constraint
Output: P: An optimized query plan

for *each* $Q^{\mathrm{S}} \in \mathcal{Q}^{\mathrm{S}}$ **do**
  for $L = 1 \dots \min\{\bar{L} - |\mathcal{Q}^{\mathrm{J}}|, |Q^{\mathrm{S}}.\mathcal{C}|\}$ **do**
    $P^{\mathrm{S}} \leftarrow$ OPTSELECT $\left(Q^{\mathrm{S}}.\mathcal{C}, L\right)$
    SAVEPLAN $\left(\{Q^{\mathrm{S}}.R\}, P^{\mathrm{S}}\right)$
for $k = 1 \dots |\mathcal{Q}^{\mathrm{J}}|$ **do**
  $\{\langle \mathcal{R}_i, \mathcal{R}_j \rangle\} \leftarrow$ FINDCANDIDATES $\left(\mathcal{R}, \mathcal{Q}^{\mathrm{J}}, k\right)$
  for *each* $\langle \mathcal{R}_i, \mathcal{R}_j \rangle$ *joined via* Q^{J} **do**
    for $L = k, \dots, (\bar{L} - |\mathcal{Q}^{\mathrm{J}}| + k)$ **do**
      for $l = 1 \dots L - k$ **do**
        $P_i \leftarrow$ FINDPLAN $\left(\mathcal{R}_i, L - l\right)$
        $P_j \leftarrow$ FINDPLAN $\left(\mathcal{R}_j, L - l\right)$
        $P^{\mathrm{J}} \leftarrow$ OPTJOIN $\left(P_i.\mathcal{T}, P_j.\mathcal{T}, Q^{\mathrm{J}}, l\right)$
        $\mathtt{cost}_{P^{\mathrm{J}}} \leftarrow P^{\mathrm{J}}.\mathtt{cost} + P_i.\mathtt{cost} + P_j.\mathtt{cost}$
      $\overline{P}^{\mathrm{J}} \leftarrow \arg_{P^{\mathrm{J}}} \min \mathtt{cost}_{P^{\mathrm{J}}}$
      $P' \leftarrow$GENPLAN $(\overline{P}^{\mathrm{J}}, P_i, P_j)$
      SAVEPLAN $\left(\mathcal{R}_i \cup \mathcal{R}_j, P'\right)$
$P \leftarrow$ FINDPLAN $(\mathcal{R}, \bar{L})$
return P

The pseudo code is shown in Algorithm 5. Let $\mathcal{R}$ be the set of relations involved in the query. We parse the input complex query into *selection segments* $\mathcal{Q}^{\mathrm{S}}$ and *join segments* $\mathcal{Q}^{\mathrm{J}}$: a selection segment $Q^{\mathrm{S}} \in \mathcal{Q}^{\mathrm{S}}$ consists of a relation $Q^{\mathrm{S}}.R$, and all the selection conditions $Q^{\mathrm{S}}.\mathcal{C}$ over $Q^{\mathrm{S}}.R$; a join segment $Q^{\mathrm{J}} \in \mathcal{Q}^{\mathrm{J}}$ contains the join conditions $Q^{\mathrm{J}}.\mathcal{C}$ and the candidate CFILL attributes $Q^{\mathrm{J}}.\mathcal{A}^{\mathrm{F}}$ over relations $Q^{\mathrm{J}}.R_1$ and $Q^{\mathrm{J}}.R_2$. The algorithm takes as input $\mathcal{Q}^{\mathrm{S}}$, $\mathcal{Q}^{\mathrm{J}}$ and a latency constraint $\bar{L}$, and produces an optimized plan using a bottom-up strategy.

As the selections are always pushed down, the algorithm first invokes OPTSELECT (refer to Algorithm 2) and generates the optimal query plan of each selection segment $Q^{\mathrm{S}} \in \mathcal{Q}^{\mathrm{S}}$, for each possible latency constraint from 1 to $\min\{\bar{L} - |\mathcal{Q}^{\mathrm{J}}|, |Q^{\mathrm{S}}.\mathcal{C}|\}$. The maximum possible latency allocated to the selections is $\bar{L} - |\mathcal{Q}^{\mathrm{J}}|$ as each join needs as least one latency. On the other hand, each Q^{S} costs at most $|Q^{\mathrm{S}}.\mathcal{C}|$ latency when each condition in $Q^{\mathrm{S}}.\mathcal{C}$ takes up one latency. The algorithm materializes each generated optimal plan P^{S} and maintains the mapping between the plan and its base relation $\{Q^{\mathrm{S}}.R\}$. The latency of each plan is stored as $P^{\mathrm{S}}.L$ for later use.

It then iteratively joins the subtrees (sub-plans) generated previously. Each iteration performs one join, and thus it takes $K = |\mathcal{Q}^{\mathrm{J}}|$ iterations (K is the number of join segments). In iteration k, the algorithm generates subtrees that contain exact k joins by combining the sub-plans generated previously. It first invokes FINDCANDIDATES to look for candidate subtrees that are *joinable* in this iteration. Specifically, each pair of candidates are in the form of $\langle \mathcal{R}_i, \mathcal{R}_j \rangle$, where $\mathcal{R}_i$ and $\mathcal{R}_j$ represent sets of base relations, and they satisfy the following constrains: 1) $\mathcal{R}_i$ and $\mathcal{R}_j$ are disjoint sets; 2) there exists $Q^{\mathrm{J}} \in \mathcal{Q}^{\mathrm{J}}$ that joins the relations in $\mathcal{R}_i$ and $\mathcal{R}_j$; 3) the total number of relations contained in $\mathcal{R}_i$ and



(a) A possible plan of Q_1. (b) An optimized plan of Q_1. (c) Optimal subproblem of query optimization.

Fig. 6. Optimization for complex query Q_1 under latency constraint.

$\mathcal{R}_j$ is $k+1$. The algorithm then searches for the optimal plan of joining the two subtrees via Q^{J} (i.e., generates the optimal subtree having $\mathcal{R}_i \cup \mathcal{R}_j$ as base relations) as follows. For each possible latency allocation for Q^{J} and its subtrees (l for Q^{J} and $L-l$ for its subtrees), FINDPLAN retrieves the materialized optimal subtrees corresponding to $\mathcal{R}_i$ and $\mathcal{R}_j$ under latency constraint $L-l$, and OPTJOIN (refer to Algorithm 3) generates the join plan under latency constraint l. Then, the optimal plan associated with base relation set $\mathcal{R}_i \cup \mathcal{R}_j$ is generated and materialized. Finally, the algorithm returns the optimal plan with latency constraint $\bar{L}$ for the input complex query (with base relation set $\mathcal{R}$).

7 EXPERIMENTAL STUDY

In this section, we first evaluate the effectiveness of our proposed optimization schemes for the crowd-powered selection, join and complex queries in a simulated crowdsourcing environment, and then examine the latency model and query optimization via experiments on the real crowd on Amazon Mechanical Turk (AMT).

TABLE 1
Dataset description.

Dataset	Relation	# of Attributes	# of Tuples
Auto1	VEHICLE	8	205
	IMAGE	10	4100
	REVIEW	9	2050
Auto2	VEHICLE	2	111
	IMAGE	4	111
	REVIEW	3	111

Datasets. We use two datasets in our experiments. First, in the simulation evaluation, as we need to evaluate our optimization approaches under many queries with varying numbers of conditions, domain sizes, etc., we generate a synthetic dataset `Auto1` with a bulk of attributes. We first use the specification of 205 cars from UCI Automobile Dataset[5] to generate a relation VEHICLE. Then, we generate a relation IMAGE by duplicating each tuple in VEHICLE 20 times and add two attributes `color` and `quality` whose values are randomly generated. Similarly, we duplicate each tuple in VEHICLE 10 times to generate relation REVIEW and add an attribute `sentiment` with randomly generated values. Second, to evaluate our approaches in the real crowdsourcing platform AMT, we use a real dataset `Auto2` containing car models in 2014 (downloaded from Yahoo Auto[6]). This dataset contains car specification, images and reviews, which can be evaluated by real crowd. More details of the datasets can be found in Table 1.

Price function. In the experiments, we use the linear price function $b + wx$. For CSELECT and CJOIN, we set both base charge b and incremental charge w to \$0.005, while for CFILL, b and w are set to \$0.01 (because filling a missing value is generally more expensive) and \$0.002 (as some attributes have large domains).

7.1 Simulation Evaluation

We implemented a simulated crowdsourcing environment on top of the dataset `Auto1`. This environment has the knowledge of the complete database of `Auto1`. When a HIT arrives, it searches the complete database and returns the correct answer to the CROWDSOURCING EXECUTOR.

7.1.1 *Evaluating Selectivity Estimation*

We estimate the selectivity using a sampling-based method. Given a sampling rate $k\%$, we randomly select $k\% * |R|$ tuples from each relation R in `Auto1`, where $|\cdot|$ is the cardinality. Let $\mathcal{A}$ be the attribute set in the database, and $D_i = \{v_1, \ldots, v_m\}$ be the value domain of attribute A_i ($A_i \in \mathcal{A}$). For each attribute and value pair $\langle A_i, v_j \rangle$ where $v_j \in D_i$, we compute its selectivity based on the sampled tuples and denote it as s_{ij}^k.

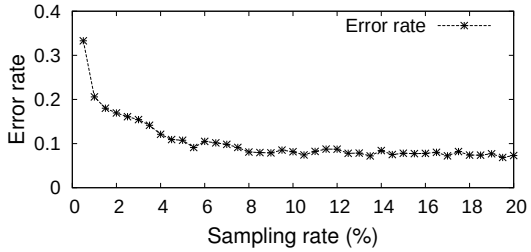


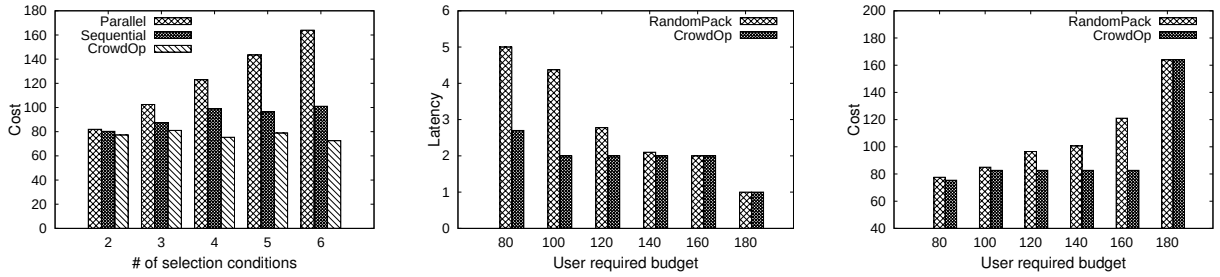
Fig. 9. Effect of sampling on selectivity estimation.

We vary the sampling rate from 0.5% to 20% and plot the relative error rate err^k in Figure 9, where err^k is defined as

$$err^k = \frac{1}{\sum_{i=1}^{|\mathcal{A}|} |D_i|} \sum_{i=1}^{|\mathcal{A}|} \sum_{j=1}^{|D_i|} \frac{|s_{ij}^k - s_{ij}^{100}|}{s_{ij}^{100}}. \quad (6)$$

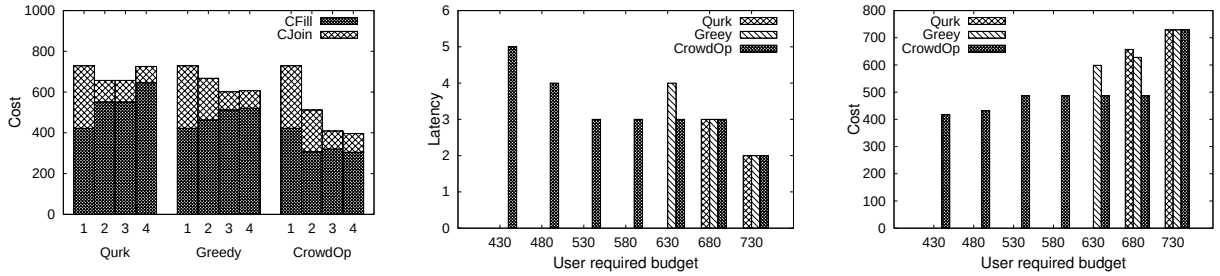
As shown in the result, the more samples we use, the lower the error rates we achieve. The error rates drop dramatically with a small sampling rate and tend to be stable when the sampling rate is larger than 5%.

5. https://archive.ics.uci.edu/ml/datasets/Automobile
6. https://autos.yahoo.com/



(a) Cost comparison w/o budget. (b) Latency on budget (6 select cond.). (c) Cost on budget (6 select cond.).

Fig. 7. Evaluation on simulated selection queries.



(a) Cost with # candidate CFill. (b) Latency on budget (# CFill = 4). (c) Cost with budget (# CFill = 4).

Fig. 8. Evaluation on simulated join queries.

7.1.2 Evaluating Selection Query Optimization

This section evaluates our optimization approach for selection queries. We first consider the objective of cost minimization where no budget constraint is imposed. We vary the number of selection conditions in a selection query from 2 to 6, and randomly generate 10 queries for each selection condition setting and report the average cost. We compare our optimization scheme against two alternatives: 1) `Parallel` packs all the selection conditions in one single CSELECT operator; 2) `Sequential` examines one selection condition in each phase according to its order in the query syntax (refer to Figure 4 for examples on parallel and sequential plan). Figure 7(a) shows the experimental results. Since `Parallel` does not make use of the selectivity information, it incurs the highest cost in all cases, especially when there are more selection conditions. In contrast, our approach CROWDOP incurs much lower cost. This is because we prioritize the conditions based on selectivity, and thus more irrelevant tuples are filtered out in the first few operators. The performance of `Sequential` lies somewhere in the middle, as it depends on the condition order in the query, which might not be optimal.

We evaluate our optimization with budget varying from 80 to 180. We consider the queries with 6 selection conditions and compare our approach (refer to Algorithm 2) with `RandomPack`, which first sorts the selection conditions in increasing order of selectivity and then randomly packs them into groups. Figures 7(b) and 7(c) provide the result on both latency and cost. The result shows that CROWDOP incurs much lower latency and cost than `RandomPack`. In addition, when a smaller budget (e.g., \$80) is allowed, the latency of CROWDOP can be further reduced (e.g., nearly half of that of `RandomPack`). The superiority of CROWDOP is attributed to our optimal grouping method, which can judiciously determine which conditions can be grouped to reduce latency and apply the conditions in an optimal order.

7.1.3 Evaluating Join Query Optimization

This section evaluates our proposed CFILL-CJOIN approach over VEHICLE-IMAGE join. We first investigate optimization approaches for cost minimization and compare our approach with two alternatives: 1) `Qurk` [12] eliminates ineffective attributes with large join selectivity[7], and fills ALL missing values on the remaining attributes; 2) `Greedy` generates the partition tree in a top-down manner: at each tree node, it selects the "local optimal" attribute to fill, which can reduce the most cost. In addition, it will stop applying CFill at the node if it is cheaper to directly join tuples.

`Qurk` fills 1, 2, 2, 3 attributes with candidate CFILL attributes varying from 1 to 4 (as shown in Figure 8(a)). With only 1 CFILL attribute, it incurs more CJOIN cost. When using more CFILL attributes, the CJOIN cost decreases, while the CFILL cost increases. As shown in the figure, with a large number of CFILL attributes (e.g., 4), the increased CFILL cost may offset its benefit, and thus increase the overall cost. This is because `Qurk` uses a fixed threshold and is not able to make the adjustment dynamically. In contrast, `Greedy` can avoid filling the *useless* attributes that cannot reduce the overall cost, and thus it achieves lower cost than `Qurk`. However, `Greedy` only looks for local optimal solution at each node, and that may not achieve the lowest overall cost. Our proposed approach CROWDOP can further improve the performance and incurs the lowest cost across all settings. For example, given 4 candidate CFILL attributes, CROWDOP respectively saves 45 and 35 percent of cost compared with `Qurk` and `Greedy`. Moreover, we also observe that: given more candidate CFILL attributes (e.g., from 1 to 2), CROWDOP can simultaneously reduce CFILL and CJOIN costs. This is because CROWDOP is able to perform CFILL at a finer degree, i.e. fill only a subset of tuples benefiting CJOIN and CFILL the most.

We next evaluate our approach with budget constraints in Figures 8(b) and 8(c), where the budget is varied from 430 to 730. From the result, we observe that the higher the budget is allowed, the lower the latency we can achieve. We can also see that CROWDOP outperforms the two baselines. Specifically, for budgets lower than \$580, `Qurk` and `Greedy` cannot generate any plan satisfying the budgets, while CROWDOP can find plans with short latency and low cost. On the other hand, for larger budgets, CROWDOP can optimize latency while achieving lower cost.

7. In the experiment, we eliminate attributes with join selectivity larger than 0.25, which achieves the best performance

7.1.4 Evaluating Complex Query Optimization

In this experiment, we study the performance of our approach over complex Select-Join query CQ_1 and CQ_2. CQ_1 performs VEHICLE-IMAGE join with three selection conditions. CQ_2 joins three relations VEHICLE-IMAGE-REVIEW with three selection conditions. We first report the optimization results without budget constraint in Table 2.

TABLE 2
Cost comparison over complex queries.

	Optimizer	CSelect	CJoin	CFill	Total
CQ_1	CrowdDB	84.15	307.8	0	391.95
	Qurk	84.15	**3.7**	43.0	130.85
	CrowdOp	**76.37**	7.56	**31.02**	**114.95**
CQ_2	CrowdDB	111.83	1252.44	0	1364.27
	Qurk	111.83	**29.34**	120.37	261.54
	CrowdOp	**104.04**	29.61	**104.88**	**238.53**

Here we compare with two alternatives: 1) CrowdDB [3] uses rule-based optimization to push down selection operations and determine the best join order; 2) `Qurk` [12] further reduces the cost by optimizing join operations via filling CFILL attributes. As the result shows, the optimizer of CROWDOP significantly reduces the cost. Compared to CrowdDB, we incur only 30 and 17 percent of the cost over CQ_1 and CQ_2 respectively, which is largely attributed to our CFILL-CJOIN optimization. With a small amount of CFILL cost, we can significantly reduce the CJOIN cost. Moreover, CROWDOP incurs lower cost than `Qurk`, which also employs a CFILL-CJOIN strategy. This is because CROWDOP not only applies the optimized filling scheme to reduce the overall CFILL-CJOIN cost, but also incurs lower cost on the selection queries in CQ_1 and CQ_2.

We next examine the performance under certain budget constraints. We vary the budget requirement and provide the latency as well as cost over CQ_1 and CQ_2 in Table 3.

TABLE 3
Evaluating CROWDOP 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

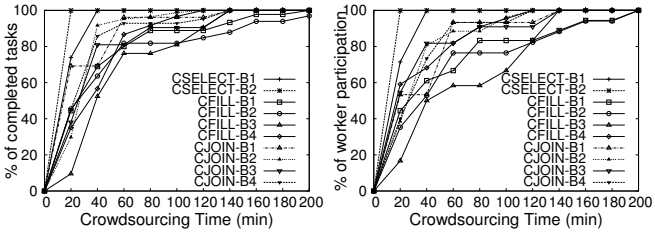
In general, the overall latency can be reduced with increasing budget constraint. More specifically, CROWDOP can effectively allocate the latencies among the operators in the query plan to fit the budget requirements. For example, in CQ_2, with increasing the budget requirement, CROWDOP first reduces 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.

7.2 Real Crowd Evaluation

To further evaluate our proposed approach, we conduct a set of experiments on the real crowdsourcing platform AMT.

7.2.1 Observation on Crowdsourcing Latency

We investigate crowdsourcing latency and examine our simplification of using the number of phases to approximate the latency.



(a) Task completion over time. (b) Worker participation over time.

Fig. 10. Observation on real crowdsourcing latency.

We generate 10 crowdsourcing jobs with different numbers of crowdsourcing tasks and operator types from the query plans of CQ_2, the details of which are listed in Table 4. We publish these jobs on AMT and measure latency (time) of completing the tasks.

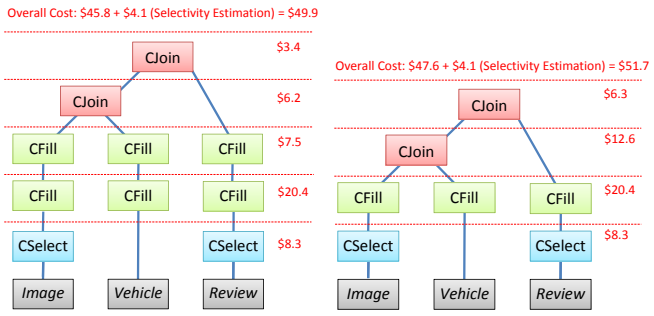
TABLE 4
Crowdsourcing tasks published on AMT for latency evaluation (# of assignment per task = 3).

Name	Task Num	Name	Task Num
CSELECT-B1	222	CFILL-B4	200
CSELECT-B2	24	CJOIN-B1	138
CFILL-B1	156	CJOIN-B2	280
CFILL-B2	112	CJOIN-B3	76
CFILL-B3	63	CJOIN-B4	141

Figures 10(a) shows the percentage of completed tasks over the crowdsourcing time. We have the following observations from the figure. First, a bulk of the answers arrive within a short period of time. For example, all the crowdsourcing jobs have completed more than 75% tasks within 60 minutes, although they have different numbers of tasks and correspond to different operators. Second, a small number of answers arrive much later, e.g., some "outstanding" answers do not arrive until 200 minutes after publishing the HITs. This phenomenon may be explained by the fact that the number of workers on the tasks will change over the crowdsourcing time. In the beginning, many workers are attracted by a newly created crowdsourcing job and they can work in parallel to complete a large portion of tasks. For example, as observed from Figure 10(b), for almost all crowdsourcing jobs, over 60% of the distinct workers arrive within 60 minutes. However, with the increase of the crowdsourcing time, fewer and fewer new workers come to answer the tasks. This may be attributed to ranking mechanism of crowdsourcing jobs on AMT: with the increase of crowdsourcing time, our jobs may be "overwhelmed" by the newly published jobs and become less likely to be presented to the workers.

7.2.2 Real Crowdsourcing Query Evaluation

To better evaluate our approach, we execute the complex query CQ_2 on the AMT platform. We first employ the sampling-based approach to estimate the selectivity. We randomly sample 20 tuples in each relation, and publish CFILL tasks on the unknown attributes of the sampled tuples. We use the same CFILL price function as mentioned above, and that results in $4.1 for the sampling. Next, we discuss two query plans produced by our optimization techniques under different budgets. Under budget $50, a plan with height 5 (as shown in Figure 11(a)) is generated, which uses one phase for CSELECT, two phases for CFILL and two phases for CJOIN. The overall cost of this plan is $49.9 ($4.1 for selectivity estimation and $45.8 for plan execution), and the overall latency is 545 minutes. In addition, when given more



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

Fig. 11. Example query plans for real crowdsourcing.

budget, i.e., $60, another plan with lower latency (as shown in Figure 11(b)) is generated, which has one phase for CSELECT, one phase for CFILL and two phases for CJOIN. The second plan has a lower latency 379 minutes but a larger monetary cost $51.7.

8 Related Work

Recently a large body of work has been proposed to perform important database operations powered by the intelligence of crowd, including selection [14], [18], join [12], [21], sort/rank [12], [6], [19] and count [11]. Meanwhile, a series of crowdsourcing systems have been designed to provide a declarative query interface to the crowd, such as CrowdDB [3], Qurk [13] and Deco [15]. Most of these works only focus on optimizing the monetary cost of some specific operations. In contrast, CROWDOP handles three fundamental operations (i.e., CSELECT, CJOIN and CFILL) and incorporates the cost-latency tradeoff into its optimization objective. Our latency model is similar to the one in CrowdFind [18]. Nevertheless, CrowdFind aims to find skylines of cost and latency for select operators only, while our work focuses more on optimizing general queries (with more fundamental operators) with minimal latency under a budget constraint. Another important metric in crowdsourcing applications is *accuracy*, which has been intensively studied in [17], [14], [10], [4].

Query optimization in relational databases is a well-studied problem [7]. Some of their techniques can be applied to the crowdsourcing scenario, such as pushing down the select predicates and utilizing selectivity to determine the select/join order. However, some inherent properties of crowdsourcing makes its query optimization a new and challenging problem. For instance, monetary cost is quite different from computation cost in RDBs, and latency, which is an important criteria in crowdsourcing, is not a serious problem in RDBs. In addition, many indexing schemes are exploited by RDBs to facilitate its query processing, while few of them can be used in crowdsourcing.

9 Conclusion and Future Work

In this paper, we propose a cost-based query optimization that considers the cost-latency tradeoff and supports multiple crowdsourcing operators. We develop efficient and effective optimization algorithms for select, join and complex queries. Our experiments on both simulated and real crowd demonstrate the effectiveness of our query optimizer and validate our cost model and latency model. In the future we would like to study how to incorporate correlations between select/join conditions into the optimizer for complex queries, and we also plan to extend CROWDOP to support more advanced SQL operators, such as sorting and aggregation.

Acknowledgement

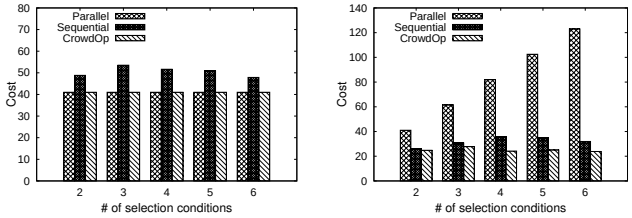
The work in this paper was in part supported by a Singapore Ministry of Education AcRF Tier 1 Grant No. R-252-000-513-112. Meihui Zhang was supported by SUTD Start-up Research Grant under Project No. SRG ISTD 2014 084. Stanley Kok is supported by a Ministry of Education AcRF Tier 1 grant, and by the National Research Foundation, Prime Ministers Office, Singapore under its IDM Futures Funding Initiative and administered by the Interactive and Digital Media Programme Office.

References

[1] 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.
[2] J. Fan, M. Lu, B. C. Ooi, W.-C. Tan, and M. Zhang. A hybrid machine-crowdsourcing system for matching web tables. In *ICDE Conference*, 2014.
[3] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD Conference*, pages 61–72, 2011.
[4] J. Gao, X. Liu, B. C. Ooi, H. Wang, and G. Chen. An online cost sensitive decision-making method in crowdsourcing systems. In *SIGMOD Conference*, pages 217–228, 2013.
[5] Y. Gao and A. G. Parameswaran. Finish them!: Pricing algorithms for human computation. *PVLDB*, 7(14):1965–1976, 2014.
[6] S. Guo, A. G. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD Conference*, pages 385–396, 2012.
[7] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD Conference*, pages 267–276, 1993.
[8] C.-J. Ho, S. Jabbari, and J. W. Vaughan. Adaptive task assignment for crowdsourced classification. In *ICML (1)*, pages 534–542, 2013.
[9] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is np-complete. *Inf. Process. Lett.*, 5(1):15–17, 1976.
[10] 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.
[11] A. Marcus, D. R. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. *PVLDB*, 6(2):109–120, 2012.
[12] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.
[13] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, pages 211–214, 2011.
[14] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD Conference*, pages 361–372, 2012.
[15] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: declarative crowdsourcing. In *CIKM*, pages 1203–1212, 2012.
[16] H. Park and J. Widom. Query optimization over crowdsourced data. *PVLDB*, 6(10):781–792, 2013.
[17] V. C. Raykar, S. Yu, L. H. Zhao, G. H. Valadez, C. Florin, L. Bogoni, and L. Moy. Learning from crowds. *Journal of Machine Learning Research*, 11:1297–1322, 2010.
[18] A. D. Sharma, A. Parameswaran, H. Garcia-Molina, and A. Halevy. Crowd-powered find algorithms. In *ICDE Conference*, 2014.
[19] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, pages 989–998, 2012.
[20] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.
[21] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD Conference*, pages 229–240, 2013.
[22] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.

Appendix

Proof of Lemma 1: We can prove the lemma by showing that *any swap* of conditions across groups in $\mathcal{G}^*$ would increase the overall cost. Suppose that we have a group set $\mathcal{G}^*$ satisfying the lemma. We will then show that *any swap* of conditions across groups in $\mathcal{G}^*$ would increase the cost. Consider arbitrary two conditions C_i



(a) Price function b. (b) Price function wx.

Fig. 12. Evaluation on additional price functions.

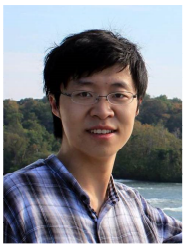
and C_j ($s(C_i) < s(C_j)$) respectively in groups G_p and G_q. Now, if we swap C_i and C_j, we examine how the groups are affected in the following two cases: 1) The estimated cost of group G_l with $l \leq p$ or $l > q$ remains unchanged, since G_l's input is not affected; 2) The estimated cost of group G_l with $p < l \leq q$ increases, as the estimated size of input $|R| \cdot \prod_{k=1}^{l-1} \prod_{C \in G_k} s(C)$ becomes larger due to the swap of C_i and C_j. Overall, the cost of group set $\mathcal{G}^*$ increases after the swap. Moreover, it is not hard to prove that any other group set must be reached from $\mathcal{G}^*$ by a sequence of such swapping operations. Hence we prove the lemma.

Proof of Lemma 2: To prove the optimal subproblem property, we need to show that: given $G^*(i, L)$ is optimal, any of its subproblem, say $G^*(k+1, L-1)$ must be optimal. We can prove the property by contradiction. Suppose that $G^*(k+1, L-1)$ is not optimal, i.e., there is another plan $G'(k+1, L-1)$ having lower cost. Then, there must be $G'(i, L) = \texttt{cost}(i, k) + G'(k+1, L-1)$ having lower cost than $G^*(i, L)$, which is contradict to that $G^*(i, L)$ is optimal. Hence, we prove the lemma.

Proof of Lemma 3: We prove the lemma by a reduction from a variant of the exact covering problem EC3 $(X, \mathcal{F})$ which consists of a set X and a family $\mathcal{F}$ of subsets of X where $\forall F \in \mathcal{F}$ contains exactly 3 elements, and finds an exact cover $\mathcal{F}^* \subseteq \mathcal{F}$ of X such that $\forall F_i, F_j \in \mathcal{F}^*$, $F_i \cap F_j = \emptyset$ and $\bigcup_{F_i \in \mathcal{F}^*} F_i = X$. The EC3 problem has been proved as NP-complete [9]. For any instance of EC3 $(X, \mathcal{F})$, we construct an instance of our partition tree optimization problem. Let pricing functions satisfy $u_{\texttt{J}} = u_{\texttt{F}}$. We first construct input tuple sets $\mathcal{T}_1 = \mathcal{T}_2 = X \cup \{a, b, c\}$ where $\{a, b, c\}$ are three elements not in X. Then, for each subset $F_i \in \mathcal{F}$, we construct a binary CFILL attribute A_i such that the elements in $\mathcal{T}_1$ satisfying $A_i = 1$ are exactly the ones in F_i. The same is true for $\mathcal{T}_2$. Under this construction, the optimal tree must have a path satisfying the following properties: 1) The path consists of a series of tree nodes respectively containing the attributes $A_{i_1}, A_{i_2}, \ldots A_{i_k}$, and 2) The node corresponding to A_{i_j} partitions exactly 6 tuples (3 from $\mathcal{T}_1$ and 3 from $\mathcal{T}_2$) to the subtree corresponding to $A_{i_j} = 1$, and puts the other tuples ($A_{i_j} = 0$) to the next attribute $A_{i_{j+1}}$. And the tuples in the leaf corresponding to $A_{i_1} = A_{i_2} = \ldots = A_{i_k} = 0$ are $\{a, a, b, b, c, c\}$. Thus, the optimal tree essentially embodies a solution of the EC3 problem. Therefore, we prove the lemma.

Additional Experiments: We also examined performance of our optimization approach under other price functions. We consider the following two special price functions: 1) function b with fixed charge for tasks, and 2) function wx without base charge b.

Figure 12 shows the result of cost minimization on selection queries with varying numbers of selection conditions. Given fixed charge b, the optimal strategy is packing all the selection conditions in one single CSELECT (i.e., the Parallel), as more conditions would not incur extra cost. On the other hand, given varying charge wx, the better strategy is to examine one selection condition in each phase to reduce as many tuples fed into CSELECT operators as possible. Our optimization approach CROWDOP can adapt to both price functions and achieve the lowest cost.



Ju Fan received the BEng degree in computer science from Beijing University of Technology, China in 2007 and the PhD degree in computer science from Tsinghua University, China in 2012. He is currently a research fellow in the School of Computing, National University of Singapore. His research interest includes crowdsourcing-powered data analytics, spatial-textual data processing, and database usability.

Meihui Zhang received the BEng degree in computer science from Harbin Institute of Technology, China in 2008 and the PhD degree in computer science from National University of Singapore in 2013. She is currently an assistant professor at the Singapore University of Technology and Design. Her research interest includes crowdsourcing-powered data analytics, massive data integration and spatio-temporal databases.

Stanley Kok received the BSc degree in computer science from Brown University in 1999, the MSc degree in computer science from University of Washington in 2005, and the PhD degree in computer science from University of Washington in 2010. He is currently an assistant professor at the Singapore University of Technology and Design. His research interests lie in the fields of machine learning and artificial intelligence. Other topics of interest are: statistical relational learning, probabilistic graphical models, natural language processing, reasoning under uncertainty, social network analysis, computational biology, data mining, text mining, and Web mining.

Meiyu Lu received the BEng degree in computer science from Harbin Institute of Technology, China in 2008 and the PhD degree in computer science from National University of Singapore in 2013. Her research interest includes crowdsourcing-powered data analytics, database exploration and schema extraction.

Beng Chin Ooi received the BSc (First Class Honors) and PhD degrees from Monash University, Australia, in 1985 and 1989, respectively. He is a professor of computer science at the School of Computing, National University of Singapore. His research interests include database system architectures, performance issues, indexing techniques and query processing, in the context of multimedia, spatio-temporal, distributed, parallel, in-memory, P2P, and Cloud database systems and applications. He has served as a PC member for a number of international conferences (including SIGMOD, VLDB, ICDE, WWW, EDBT, DASFAA, GIS, KDD, CIKM, and SSD). He was an editor of VLDB Journal and IEEE Transactions on Knowledge and Data Engineering, Editor-in-Chief of IEEE Transactions on Knowledge and Data Engineering (TKDE)(2009-2012), and a co-chair of the ACM SIGMOD Jim Gray Best Thesis Award committee. He is serving as a trustee board member and president of VLDB Endowment.