CrowdSky: Skyline Computation with Crowdsourcing

Jongwuk Lee
Hankuk University of Foreign Studies, Republic of Korea
julee@hufs.ac.kr

Dongwon Lee
The Pennsylvania State University, PA, USA
dongwon@psu.edu

Sang-Wook Kim
Hanyang University, Republic of Korea
wook@hanyang.ac.kr

ABSTRACT

In this paper, we propose a crowdsourcing-based approach to solving skyline queries with incomplete data. Our main idea is to leverage crowds to infer the pair-wise preferences between tuples when the values of tuples in some attributes are unknown. Specifically, our proposed solution considers three key factors used in existing crowd-enabled algorithms: (1) minimizing a *monetary cost* in identifying a crowdsourced skyline by using a *dominating set*, (2) reducing the number of rounds for *latency* by parallelizing the questions asked to crowds, and (3) improving the *accuracy* of a crowdsourced skyline by dynamically assigning the number of crowd workers per question. We evaluate our solution over both simulated and real crowdsourcing using the Amazon Mechanical Turk. Compared to a sort-based baseline method, our solution significantly minimizes the monetary cost, and reduces the number of rounds up to two orders of magnitude. In addition, our dynamic majority voting method shows higher accuracy than both static majority voting method and the existing solution using unary questions.

1. INTRODUCTION

In recent years, crowdsourcing has become a new paradigm for implementing human computation. Many extensions to existing DBMS techniques have been proposed to employ crowdsourcing so that the problems difficult for machines but easier for humans can be solved better than ever, *e.g.*, CrowdDB [5], Qurk [13], Deco [19], AskIt! [1], and CyLog/Crowd4U [16]. Motivated by their success, ones attempt to leverage crowdsourcing into existing micro-level query operations. For instance, they are selection [6, 18, 21], join [14, 24, 25, 26], group-by [4], max [8, 22, 23], and sort [14].

In this paper, we study a crowdsourcing-based approach to solving *skyline queries* with incomplete data. The skyline queries have gained considerable attention for assisting multi-criteria decision making applications [2, 9, 17]. Given two tuples s and t, it is said that s *dominates* t if the values of s are no worse than those of t over all attributes and the values of s are better than those of t over any attribute. Given a set $\mathcal{R}$ of tuples, the *skyline* is a set of tuples that are not dominated by any other tuples in $\mathcal{R}$. To illustrate this, we consider the following motivating example.

EXAMPLE 1 (SKYLINE QUERY) Suppose that Alice wants to find the skyline movies of her preference, *i.e.*, most popular and most romantic movies, released in 2010-2015.

```
SELECT * FROM movie_db
WHERE year >= 2010 and year <= 2015
SKYLINE OF box_office MAX, romantic MAX
```

The popularity can be estimated by the `box_office` (*i.e.*, the number of movie audiences) attribute. However, the `movie_db` table does not record how romantic a movie is. That is, in the `romantic` attribute, the values of tuples are all unknown (or missing). To address this problem, we utilize crowdsourcing that can effectively infer missing values on the subjective attribute. Specifically, we ask crowds which movie is more *romantic* with respect to two movies, and get a relative preference of movies by using a *pair-wise question*. By repeatedly asking such pair-wise questions and aggregating their answers, we can fill missing values of tuples, and then find a skyline. Note that our setting above is an extreme case (*i.e.*, all values of tuples are missing in the `romantic` attribute). When some values of tuples are missing, we can apply our proposed solution to only the tuples with missing values. □

A *crowd-enabled skyline query* is defined as a skyline query with incomplete data in which crowds are used to infer the missing preferences between tuples [12]. Although existing work [12] addresses crowd-enabled skyline queries, it assumes a fixed budget and computes a *probabilistic* skyline. In contrast, our goal is minimize the number of pair-wise questions to crowds in identifying a *complete* skyline. In addition, [12] is based on *unary* questions to assess missing values of tuples. Because it is difficult to obtain correct answers for unary questions, the skyline result in [12] can be inaccurate. In our empirical study, it is observed that the pair-wise questions achieve higher accuracy than the unary questions in existing work [12].

In order to address skyline queries with crowdsourcing, we deal with three key factors: (1) how to minimize rewards paid to crowds (*i.e.*, *monetary cost*), (2) how to reduce the delay of crowd-enabled computation (*i.e.*, *latency*), and (3) how to improve the quality of the skyline using the answers obtained from crowds (*i.e.*, *accuracy*). When asking questions to crowds, we have to pay certain monetary rewards. Assuming that a fixed amount of a reward per question is paid, the monetary cost is proportional to the number of questions asked to crowds. In order to measure the latency, we need to estimate the running time to obtain answers from crowds in a *round*. Assuming that each round has a fixed amount of time [25], the latency is proportional to the number of rounds needed for asking all questions.

We then propose a new solution that addresses crowd-enabled skyline queries for each factor. First, in order to minimize the monetary cost, it is essential to remove unnecessary questions while

Series ISSN: 2367-2005

10.5441/002/edbt.2016.14

computing a crowdsourced skyline. Toward this goal, we make use of two relationships between tuples, *dominance* and *incomparability*. In particular, we adopt a *dominating set* to remove unnecessary questions in identifying a skyline. That is, given a tuple t, the dominating set $DS(t)$ is a set of tuples that dominate t. Using $DS(t)$, we can skip the questions for tuples with the *incomparability* relationship, and selectively ask the questions for tuples with the *dominance* relationships. We also prune unnecessary questions by using the *transitivity* of dominance relationships in $DS(t)$.

Second, we address how to reduce the number of rounds by asking multiple *independent* questions in a round. A set of questions are said to be independent if the answers of each question do not affect the other questions in the set. We show that independent questions can be asked in a round, yielding the *parallelization* of asking questions. Based on this observation, we develop two parallelization methods that significantly reduce the number of rounds without influencing other factors.

Lastly, we explain how to improve the accuracy of a crowdsourced skyline. When a single crowd worker is assigned per question, some answers can be erroneous as workers can make mistakes. To alleviate this problem, we assign multiple workers per question and decide the final answer by using *majority voting* [6, 11, 15, 18]. As the simplest method, we can *equally* assign the same number of workers for each question. Because it neglects the characteristics of skyline queries, however, we develop a *dynamic* assignment strategy in which the number of workers can be assigned differently depending on the *importance* of questions. The proposed dynamic assignment can improve the accuracy of the crowdsourced skyline without incurring additional monetary cost.

To summarize, our main contributions are as follows:

- We formulate the problem of crowd-enabled skyline queries with three key factors used in crowd-enabled algorithms.
- To minimize monetary cost, we propose a crowd-enabled skyline algorithm with three pruning methods on top of the notion of a dominating set.
- To reduce the number of rounds, we develop two parallelization methods based on the notion of dominating sets and skyline layers.
- To improve the accuracy of the crowdsourced skyline, we design a dynamic majority voting that assigns the number of workers depending on the importance of questions.
- We validate the effectiveness of our proposed algorithm in both simulated and real-life crowdsourcing using the Amazon Mechanical Turk. In terms of accuracy, we also compare our proposed solution against existing work [12].

The remainder of this paper is organized as follows. In Section 2, we explain the concept of crowd-enabled algorithms and formulate the skyline query with crowdsourcing. In Section 3, we first propose an algorithm to minimize monetary cost using several pruning methods. In Section 4, we develop two algorithms to minimize the number of rounds by parallelizing multiple questions in a round. In Section 5, we design a dynamic majority voting that improves a static majority voting by considering the importance of questions. In Section 6, we empirically compare our proposed algorithm with the baseline and existing crowdsourced skyline algorithms with unary questions. In Section 7, we review our related work. In Section 8, we summarize and conclude our work.

2. PRELIMINARIES

In this section, we first present the concept of crowd-enabled algorithms (Section 2.1), and then explain the basic notion of skyline queries (Section 2.2). We lastly formulate the problem of skyline queries with crowdsourcing, where questions are asked to crowds to acquire the relative preferences between tuples with missing values (Section 2.3).

2.1 Crowd-Enabled Algorithms

The crowd-enabled algorithms first require us to design the format of *micro-tasks* asked to crowds. Because the micro-tasks are typically represented as *questions*, we use both terms, micro-tasks and questions, interchangeably. Let π denote a *latent scoring function* with which crowds assess the missing value of a tuple. The argument of π can differ depending on formats of questions.

Specifically, the micro-tasks are classified into *quantitative* and *qualitative* formats [14]. First, the quantitative format asks crowds to determine an *absolute* (normalized) preference. This can be abstracted as a *unary function* $\pi(t)$ for a tuple t, *e.g.*, rating from 1 to 7 for the size of a given square. Let n denote the number of tuples with missing values. While the unary function is effective for minimizing the number of questions in determining the total order of tuples, *i.e.*, n questions, the accuracy of answers can be low. That is, because crowds usually have no *global* knowledge for missing values of tuples, it is difficult for them to return correct answers.

Second, the qualitative format allows crowds to judge the *relative* preference between two (or more) tuples. As the simplest format, it can be modeled as a *binary function* $\pi(s, t)$ to compare two tuples s and t, *e.g.*, selecting one with a larger size between two squares. (Possibly, it can be extended to an m-ary format.) Compared to the quantitative format, because it only requires relative preference between two tuples, more questions are usually asked. In the worst case, $\binom{n}{2}$ questions are needed for obtaining a total order of n tuples. Meanwhile, because the crowd can answer binary questions correctly without global knowledge for missing values, the accuracy of answers in the qualitative format is higher than that in the quantitative format.

In this paper, we adopt binary function $\pi(s, t)$ for questions in order to obtain more accurate answers. That is, we use a *pair-wise question* (s, t) with *ternary* answers, where it is *symmetric*, *i.e.*, $(s, t) = (t, s)$. Given (s, t), the crowd chooses a *more* preferred one (*i.e.*, either s or t) or the third option, indicating that the two tuples are *equally* preferred. For example, the following questions are asked to crowds: “which square between the two is larger?” or “who is a more valuable baseball player?"

We next explain three key factors used in existing crowd-enabled algorithms, *e.g.*, [4, 6, 18, 20, 22, 25]. Using the key factors, we formulate the problem of crowd-enabled skyline queries (Section 2.3).

(1) **Monetary cost**: Unlike existing machine-only algorithms, crowd-enabled algorithms compensate rewards to crowds. Assuming that a fixed amount of a reward per question is paid, the monetary cost is proportional to the number of questions asked to crowds. For monetary cost, there are two optimization directions: (1) minimizing the number of questions asked for obtaining a complete query result and (2) selecting the most important questions for a given budget. In this paper, we focus on minimizing the total number of questions during executing a skyline query with crowdsourcing.

(2) **Latency**: Since each question can take different time to finish, it is non-trivial to design an effective model for estimating latency. As an alternative way, we assume that a fixed amount of time is assigned per question. Because multiple questions can be performed in parallel, we use the number of *rounds* (or *iterations*.) to measure

the latency [25]. Specifically, there are two strategies in latency: (1) *one-shot strategy* that generates all questions at once, and (2) *adaptive strategy* that asks questions in an interactive manner. Because the one-shot strategy needs only one round, it is much faster than the adaptive strategy. Meanwhile, the adaptive strategy can identify unnecessary questions by using the answers of questions asked at the previous rounds, and thus reduces the total number of questions. In this paper, we leverage the adaptive strategy and discuss how to minimize the latency for a given question set.

(3) **Accuracy**: Because crowds can make mistakes in answering questions, the accuracy of a query result can be imperfect. There are two models to improve the accuracy: *query-independent* and *query-dependent* models. Existing work [6, 11, 15, 18] proposed various query-independent methods to improve the accuracy for a single question by considering the proficiency of workers and the difficulty of questions. Although improving the accuracy of a query result in a micro-level manner, they do not consider the *importance* of questions depending on an inherent property of a query type in a macro-level way, *e.g.*, [4, 20]. In this paper, we aim to develop a query-dependent method by distinguishing the importance of questions for a skyline query.

2.2 Skyline Queries with Missing Data

Let $\mathcal{A}$ denote a finite set of d attributes, $\mathcal{A} = \{A_1, \ldots, A_d\}$, in which the domain D_i of A_i is a set of positive numbers and a *missing value*, denoted by $\Box$, *i.e.*, $D_i := \mathbb{R}^+ \cup \{\Box\}$. A base dataset $\mathcal{R}$ is an instance of database relations, *i.e.*, $\mathcal{R} \subseteq D_1 \times \ldots \times D_d$. Each tuple $t \in \mathcal{R}$ is represented by $t = (t_1, \ldots, t_d)$ such that $t_i \in D_i$ for $i = 1, \ldots, d$. In this paper, we use s, t, u, and v to point out arbitrary tuples.

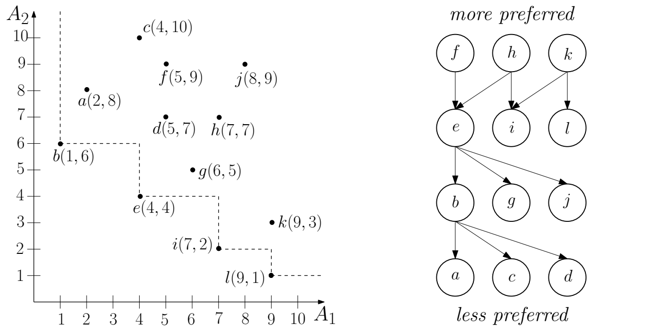
The attribute set $\mathcal{A}$ is divided into two attribute subsets. First, $\mathcal{A}^K$ is a set of attributes in which the values of tuples are *known*. The preference of values over $\mathcal{A}^K$ can be represented by a total order. Second, $\mathcal{A}^C$ is a subset of attributes in $\mathcal{A}$, where the values of tuples are *missing*. We call $\mathcal{A}^C$ *crowd attributes*. Two attribute sets are *disjoint*, *i.e.*, $\mathcal{A}^K \cap \mathcal{A}^C = \emptyset$, $\mathcal{A}^K \cup \mathcal{A}^C = \mathcal{A}$. We assume that all values of tuples in $\mathcal{A}^C$ are missing, *i.e.*, *hand-off crowdsourcing* [7]. That is, for any tuple $t \in \mathcal{R}$, $\forall A_j \in \mathcal{A}^C : t_j = \Box$ holds. This implies that all preferences between tuples in A^C should be assessed by crowds. After the missing preferences between tuples in $\mathcal{A}^C$ are judged by crowds, it can be represented by a *partial* order of tuples. When a subset of tuples in $\mathcal{R}$ only has missing values in many real applications, some known values of tuples can be represented by a pre-defined partial order. Therefore, we can extend our proposed techniques for real-life scenarios.

We next define fundamental notions used in skyline literatures [2, 9, 17]. In this paper, we assume that smaller values over $\mathcal{A}^K$ are more preferred. Given $s, t \in \mathcal{R}$, a *strict preference* $s <_i t$ is defined if s is preferred over t in A_i. Let $s \lesssim_i t$ define a *weak preference* if $s <_i t$ or $s =_i t$ in A_i. If no preference between s and t is inferred in $A_i \in \mathcal{A}^C$, an *indifferent preference* $s \perp_i t$ is defined.

Definition 1 (Dominance) Given $s, t \in \mathcal{R}$, *s dominates t* in $\mathcal{A}$, denoted by $s \prec_{\mathcal{A}} t$, if $\forall A_i \in \mathcal{A} : s \lesssim_i t$ and $\exists A_j \in \mathcal{A} : s <_j t$. If s does not dominate t in $\mathcal{A}$, it is denoted as $s \nprec_{\mathcal{A}} t$.

Definition 2 (Incomparability) Given $s, t \in \mathcal{R}$, they are *incomparable* in $\mathcal{A}$, denoted by $s \prec\succ_{\mathcal{A}} t$, if (i) $s \nprec_{\mathcal{A}} t$ and $t \nprec_{\mathcal{A}} s$ or (ii) $\exists A_j \in \mathcal{A}^C : s \perp_i t$.

Definition 3 (Skyline) Given a set $\mathcal{R}$ of tuples, a *skyline* is a set of tuples that are not dominated by any other tuples in $\mathcal{A}$, *i.e.*, $\textsc{Sky}_{\mathcal{A}}(\mathcal{R}) = \{t \in \mathcal{R} | \forall s \in \mathcal{R} : s \nprec_{\mathcal{A}} t\}$.



(a) $\mathcal{R}$ in $\mathcal{A}^K = \{A_1, A_2\}$ (b) Preferences in $\mathcal{A}^C = \{A_3\}$

Figure 1: A toy dataset for a crowdsourced skyline query with $\mathcal{A} = \{A_1, A_2, A_3\}$

These notions can be applied to a subset of $\mathcal{A}$. That is, $\mathcal{A}$ in $\prec_{\mathcal{A}}$, $\nprec_{\mathcal{A}}$, and $\prec\succ_{\mathcal{A}}$ can be replaced with $\mathcal{A}^K$ or $\mathcal{A}^C$.

2.3 Problem Formulation

The earlier work [12] maximized the accuracy of the crowdsourced skyline at a fixed budget. When tuples have missing values, [12] adopts crowds to assess missing values of tuples with a unary question. In contrast to [12], in order to achieve more reliable skyline results, our goal is to minimize monetary cost while computing crowd-enabled skyline queries in which binary questions are used for obtaining the preferences between tuples. (In Section 7, we explain the differences between ours and [12].)

When the crowds evaluate missing values of tuples, a skyline result can be iteratively updated. Initially, since the preferences of tuples over $\mathcal{A}^C$ are undefined, all tuples are incomparable to each other, and by default in the skyline. Let $Q(t) = \{(s,t) | s \in \mathcal{R} \setminus \{t\}\}$ be a set of all possible pair-wise questions for t. After a question set $Q'(t) \subseteq Q(t)$ is answered by crowds, s can exist such that $s \prec_{\mathcal{A}} t$. At this point, t becomes a non-skyline tuple, and is removed from the initial skyline. If the status of t is not changed regardless of remaining questions for t, it is called a *complete* tuple. That is, once t is determined as a complete tuple, additional questions for t are *unnecessary*.

Definition 4 (Complete tuple) After a set of questions $Q'(t) \subseteq Q(t)$ is answered, a tuple t is said to be *complete* if (i) $\exists s \in \mathcal{R} : s \prec_{\mathcal{A}} t$, (*i.e.*, a *complete non-skyline tuple*) or (ii) $\forall s \in \mathcal{R} : s \nprec_{\mathcal{A}} t$ holds regardless of the answers of remaining questions $Q(t) - Q'(t)$ for t, (*i.e.*, a *complete skyline tuple*).

Example 2 (Crowd-enabled skyline query) Given a set $\mathcal{R}$ of 12 tuples, Figure 1 depicts a toy dataset for a crowd-enabled skyline query such that $\mathcal{A}^K = \{A_1, A_2\}$ and $\mathcal{A}^C = \{A_3\}$. Note that all values of $\mathcal{R}$ in A^3 are missing. Since $\{b, e, i, l\}$ (a dashed line in Figure 1a) are not dominated by any other tuples in $\mathcal{A}^K$, they are always in the skyline in $\mathcal{A}$ regardless of preferences of tuples in $\mathcal{A}^C$. Therefore, they are complete skyline tuples. Because the other tuples can be non-skyline tuples depending on the results of questions on $\mathcal{A}^C$, however, they are regarded to be incomplete tuples. Suppose that the preferences of tuples in $\mathcal{A}^C$ are depicted in Figure 1(b), where an edge $s \rightarrow t$ indicates that s is preferred over t in $\mathcal{A}^C$ and transitivity between edges holds. (In Section 3.3, we discuss how to build a preference tree in $\mathcal{A}^C$.) Since $b \prec_{\mathcal{A}^K} a$ and $b \prec_{\mathcal{A}^C} a$, a becomes a non-skyline tuple. By using the preference relationships in Figure 1(b), all tuples become complete tuples, and the skyline is finally identified as $\{b, e, i, l, k, f, h\}$. □

In this paper, we consider three key factors, *i.e.*, monetary cost, latency, accuracy, in computing crowd-enabled skyline queries. We first aim to minimize the number of questions in identifying a complete skyline. Then, we further optimize the other factors for a given question set. Although this problem formulation does not identify a solution that optimizes three factors simultaneously, it can be a more practical setting as done in [24, 25]. Formally:

PROBLEM 1 (CROWDSOURCED SKYLINE) Let $\mathcal{P}$ be a set of possible *execution plans* for computing a complete skyline. Our problem is to identify an execution plan $P_{opt} \in \mathcal{P}$ that minimizes monetary cost $C(P, \mathcal{R})$ for questions in $\mathcal{A}^C$.

$$P_{opt} = \operatorname*{argmin}_{P \in \mathcal{P}} C(P, \mathcal{R})$$

Note that P_{opt} can also be used for optimizing the other factors such as latency and accuracy. ■

Assuming that a single question is asked at each round, execution plan P can be represented by a sequence set $\mathcal{Q}$ of questions, *e.g.*, $\mathcal{Q} = \langle (a, b), \ldots, (k, l) \rangle$. Let $|\mathcal{Q}|$ denote the number of questions in $\mathcal{Q}$. When the monetary cost per question is equal, $C(P, \mathcal{R})$ becomes proportional to $|\mathcal{Q}|$. Therefore, we minimize the number of questions $|\mathcal{Q}|$ in identifying a complete skyline tuple.

In the following sections, we first assume that the answers of crowds are always correct. Based on this assumption, we propose a crowd-enabled skyline algorithm to minimize $|\mathcal{Q}|$ (Section 3). When multiple questions are asked in parallel, we also develop how to reduce the number of rounds for $\mathcal{Q}$ (Section 4). By relaxing such an assumption, we lastly discuss how to improve the accuracy of a complete skyline while both the number of questions and the latency are kept (Section 5).

3. MINIMIZING MONETARY COST

As a baseline method, we may ask all possible pair-wise questions between tuples, *i.e.*, $\binom{n}{2}$ to obtain the total order of tuples. However, because it is too exhaustive, we modify existing sorting algorithms such as *tournament sort* and *bitonic sort* [3] with crowdsourcing. Specifically, the pair-wise comparisons in existing sorting-based algorithms [3] can be replaced by binary questions, and the total order of tuples in $\mathcal{A}^C$ can be used to identify a crowdsourced skyline. While the sorting-based method is effective for obtaining all missing preferences of tuples, it can incur unnecessary questions in computing the crowd-enabled skyline.

To address this problem, it is observed that the *dominance* and *incomparability* relationships of tuples in $\mathcal{A}^K$ can be used to reduce unnecessary questions. Based on this observation, we adopt the notion of a *dominating set* to remove unnecessary questions. In the following sections, we also develop several pruning methods on top of the dominance set to remove additional questions.

For a simpler presentation, when $\mathcal{A}^C$ has multiple attributes, *i.e.*, $m = |\mathcal{A}^C| > 1$, we suppose that m questions for (s, t) are asked at once, *i.e.*, $\forall A_j \in \mathcal{A}^C : (s_j, t_j)$. Because m questions can be asked simultaneously, we simply use (s, t) to denote m questions for (s, t), when context is clear.

In addition, we suppose that the values of tuples in $\mathcal{A}^K$ are distinct, *i.e.*, for any tuples $s, t \in \mathcal{R}$, $\exists \mathcal{A}_i \in \mathcal{A}^K : s_i \neq t_i$ holds. As a degenerate case, we separately handle a tuple set with the same values in $\mathcal{A}^K$. Because we cannot exploit our pruning methods for the tuple set, it is performed as a pre-processing step. That is, given (s, t) such that $\forall A_j \in \mathcal{A}^K : s_i = t_i$, we remove a non-skyline tuple by identifying the dominance relationship between two tuples in $\mathcal{A}^C$. (This degenerate case is described in lines 1–3 in Algorithm 1.) After performing the pre-processing, we can safely make use of our pruning methods without loss of correctness.

Table 1: Dominating sets and question sets for the toy dataset in Figure 1(a)

t	$DS(t)$
a	$\{b\}$
c	$\{a, b, e\}$
d	$\{b, e\}$
f	$\{a, b, d, e\}$
g	$\{e\}$
h	$\{b, d, e, g, i\}$
j	$\{a, b, d, e, f, g, h, i\}$
k	$\{i, l\}$

(a) Dominating sets

t	$Q(t)$
a	$\{(a, b)\}$
c	$\{(c, a), (c, b), (c, e)\}$
d	$\{(d, b), (d, e)\}$
f	$\{(f, a), (f, b), (f, d), (f, e)\}$
g	$\{(g, e)\}$
h	$\{(h, b), (h, d), (h, e),$ $(h, g), (h, i)\}$
j	$\{(j, a), (j, b), (j, d), (j, e),$ $(j, f), (j, g), (j, h), (j, i)\}$
k	$\{(k, i), (k, l)\}$

(b) Question sets

3.1 Using a Dominating Set

We first exploit the incomparability relationship of tuples to bypass unnecessary questions. For instance, when two tuples $a = (2, 8, \square)$ and $d = (5, 7, \square)$ in Figure 1(a) are incomparable in $\mathcal{A}^K = \{A_1, A_2\}$, they also become incomparable in $\mathcal{A}$ regardless of the answer of (a, d) in $\mathcal{A}^C = \{A_3\}$. That is, we only need to compare a question (s, t) in $\mathcal{A}^C$ by asking a question (s, t) if s and t are *not incomparable* in $\mathcal{A}^K$ by using the property of sharing incomparability [10].

Based on this property, we adopt a *dominating set* $DS(t)$ for a tuple $t \in \mathcal{R}$ as a set of candidates that can affect the dominance relationship of t in $\mathcal{A}$. That is, the dominance set ensures that only the questions $Q(t) = \{(s, t) | s \in DS(t)\}$ are enough to generate the dominance relationship between s and t in $\mathcal{A}$.

DEFINITION 5 (DOMINATING SET) A *dominating set* $DS(t)$ of a tuple $t \in \mathcal{R}$ is a set of tuples that dominate t in $\mathcal{A}^K$, *i.e.*, $DS(t) = \{s \in \mathcal{R} | \forall s \in \mathcal{R} \setminus \{t\} : s \prec_{\mathcal{A}^K} t\}$.

LEMMA 1 Given $t \in \mathcal{R}$ and $s \notin DS(t)$, (s, t) is unnecessary in $Q(t)$.

PROOF. We prove this by contradiction. Assume that a question (s, t) exists that $s \notin DS(t)$ dominates t in $\mathcal{A}$. By Definition 5, $s \notin DS(t)$ does not dominate t in $\mathcal{A}^K$. Because $\mathcal{A}^K \subset \mathcal{A}$, s cannot dominate t in $\mathcal{A}$ regardless of asking (s, t). That is, we do not have to ask a question (s, t) to determine whether t is complete. This contradicts that asking question (s, t) is necessary for t to be complete.

EXAMPLE 3 (DOMINATING SET) Continue to use the toy dataset in Figure 1(a). Table 1(a) illustrates dominating sets for each tuple. The questions generated by dominating sets are shown in Table 1(b). As a result, the total number of questions (*i.e.*, 26 questions) is calculated as $\sum_{t \in \mathcal{R}} |DS(t)|$, where $|DS(t)|$ is the number of tuples in $DS(t)$. □

3.2 Pruning Questions for Non-skylines in $\mathcal{A}$

While sequentially generating (s, t) such that $s \in DS(t)$, t can be determined as a complete tuple (by Definition 4). For *all* of the questions, if t is preferred to s, t becomes a *complete skyline tuple*. On the other hand, if t is not preferred to s for *any* question, t becomes a *complete non-skyline tuple* and remaining questions in $Q(t)$ can be skipped. Once $s \prec_{\mathcal{A}} t$ is determined, it can also be used for removing additional unnecessary questions for any

Table 2: Sorted dominating sets and question sets after removing a (/), g (\), and d (X), respectively

t	$DS(t)$
a	$\{b\}$
g	$\{e\}$
d	$\{b, e\}$
k	$\{i, l\}$
c	$\{a, b, e\}$
f	$\{a, b, d, e\}$
h	$\{b, d, e, g, i\}$
j	$\{a, b, d, e, f, g, h, i\}$

(a) Sorted dominating sets

t	$Q(t)$
a	$\{(a, b)\}$
g	$\{(g, e)\}$
d	$\{(d, b), (d, e)\}$
k	$\{(k, i), (k, l)\}$
c	$\{\cancel{(c, a)}, (c, b), (c, e)\}$
f	$\{\cancel{(f, a)}, (f, b), \xcancel{(f, d)}, (f, e)\}$
h	$\{(h, b), \xcancel{(h, d)}, (h, e),$ $\bcancel{(h, g)}, (h, i)\}$
j	$\{\cancel{(j, a)}, (j, b), \xcancel{(j, d)}, (j, e),$ $(j, f), \bcancel{(j, g)}, (j, h), (j, i)\}$

(b) Question sets

other tuple. That is, once a tuple t is determined as a complete non-skyline tuple, we can safely skip to ask all questions for t as unnecessary ones.

LEMMA 2 If $s \prec_{\mathcal{A}} u$ and $u \in DS(t)$ hold, then $s \in DS(t)$ also holds.

PROOF. By definition of $DS(t)$, $u \in DS(t) \Leftrightarrow u \prec_{\mathcal{A}^K} t$, and $\mathcal{A}$ is divided to two subsets $\mathcal{A}^K$ and $\mathcal{A}^C$. When $s \prec_{\mathcal{A}} u$, there are three cases:

1. $(s \prec_{\mathcal{A}^K} u) \wedge (s \prec_{\mathcal{A}^C} u)$: since $s \prec_{\mathcal{A}^K} u$ and $u \prec_{\mathcal{A}^K} t$ hold, $s \prec_{\mathcal{A}^K} t$ also holds by transitivity.

2. $(s \prec_{\mathcal{A}^K} u) \wedge (s =_{\mathcal{A}^C} u)$: since $s \prec_{\mathcal{A}^K} u$ and $u \prec_{\mathcal{A}^K} t$ hold, $s \prec_{\mathcal{A}^K} t$ also holds by transitivity.

3. $(s =_{\mathcal{A}^K} u) \wedge (s \prec_{\mathcal{A}^C} u)$: since $s =_{\mathcal{A}^K} u$ and $u \prec_{\mathcal{A}^K} t$ hold, $s \prec_{\mathcal{A}^K} t$ also holds.

In all cases, $s \prec_{\mathcal{A}^K} t$ always holds. By definition of $DS(u)$, $s \in DS(t)$ also holds.

COROLLARY 1 For a tuple $t \in \mathcal{R}$, if $s \prec_{\mathcal{A}} u$ and $u \in DS(t)$ hold, asking (t, u) is unnecessary in $Q(t)$.

PROOF When asking (s, t), there exist two possible cases:

1. $s \prec_{\mathcal{A}^C} t$ or $s =_{\mathcal{A}^C} t$: By Lemma 2, $s \in DT(t)$ holds. Because $s \prec_{\mathcal{A}^K} t$, $s \prec_{\mathcal{A}} t$ holds. In that case, t becomes a complete non-skyline tuples regardless of asking (t, u).

2. $t \prec_{\mathcal{A}^C} s$: Because $s \prec_{\mathcal{A}} u$, $u \not\prec_{\mathcal{A}^C} s$ holds. In that case, because $u \not\prec_{\mathcal{A}^C} t$ always holds.

In both cases, asking (t, u) is unnecessary by asking (s, t). ■

In order to remove unnecessary questions for non-skyline tuples by Corollary 1, it is essential to identify complete non-skyline tuples as many as possible. That is, it is *optimal* if $Q(t)$ is generated after all tuples in $DS(t)$ become complete. Toward this goal, we decide the ordering of evaluating tuples in an *iterative* manner. We first identify $\textsc{Sky}_{\mathcal{A}^K}(\mathcal{R})$ as complete tuples. Then, for a tuple $t \in \mathcal{R}$, we generate $Q(t)$ with the following steps: (1) If any tuple in $s \in DS(t)$ is a complete non-skyline tuple, s is removed from $DS(t)$; (2) If all tuples in $DS(t)$ are complete, $Q(t)$ is generated from $DS(t)$; (3) $Q(t)$ is asked until t is determined as a complete tuple; (4) The complete tuple set is updated by appending t. We repeat this process until all tuples become complete.

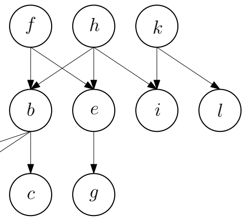


Figure 2: Preference tree $\mathcal{T}$ in A_3 until evaluating h

We now propose an efficient method that performs the iterative strategy. Given two tuples s and t, $Q(s)$ has to precede $Q(t)$ if $s \in DS(t)$. It is found that this condition is always satisfied if tuples are evaluated by the ascending order of the size of dominating sets. That is, if $s \in DS(t)$, then $|DS(s)| < |DS(t)|$ holds, implying that the size of the dominating set increases monotonically if the dominance relationship between tuples holds. That is, given $s, t \in \mathcal{R}$, $Q(s)$ such that $s \in DS(t)$ is first asked before generating $Q(t)$. Formally:

LEMMA 3 Given $s, t \in \mathcal{R}$, if $|DS(t)| \geq |DS(s)|$, then $t \not\prec_{\mathcal{A}^K} s$ holds.

PROOF We prove this by contradiction. Assume that there are two tuples s and t such that $t \prec_{\mathcal{A}^K} s$ and $|DS(t)| \geq |DS(s)|$. Because $t \prec_{\mathcal{A}^K} s$, two conditions hold: (1) $t \in DS(s)$ and (2) $DS(t) \subseteq DS(s)$. By combining them, $|DS(t)| < |DS(s)|$ holds, which contradicts the fact that $|DS(t)| \geq |DS(s)|$. ■

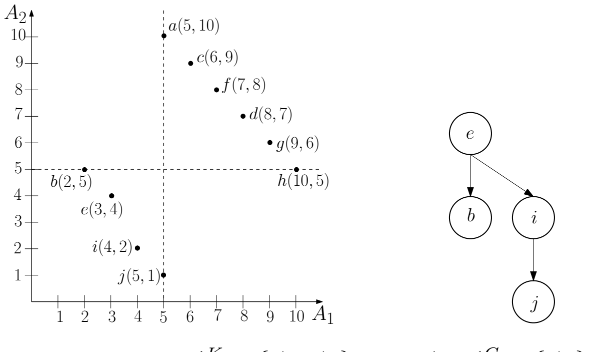
EXAMPLE 4 (SORTING DOMINATING SETS) We continue to use the toy dataset used in Example 3. Table 2(a) shows the dominating sets sorted by $|DS(t)|$ in Table 1(a). According to the ordering of tuples in Table 2(a), the questions are sequentially generated from a to j. Assuming that $\{a, g, d\}$ have been determined as non-skyline tuples, Table 2(b) illustrates questions for each tuple. When (a, b) is asked and a is identified as a non-skyline tuple (*i.e.*, $b \prec_{\mathcal{A}} a$), a is removed from $DS(c)$, $DS(f)$, and $DS(j)$. Similarly, after (g, e) is asked, if g is a non-skyline tuple, g is removed from $DS(h)$ and $DS(j)$. As a result, it only generates 18 questions by pruning 8 questions, *i.e.*, $\{(c, a)\ (f, a), (j, a)\}$, $\{(h, g), (j, g)\}$, and $\{(f, d), (h, d), (j, d)\}$ for a, g, and d, respectively. □

3.3 Pruning Questions for Non-skylines in $\mathcal{A}^C$

When generating $Q(t)$ from $DS(t)$, we can further reduce $DS(t)$ to $\textsc{Sky}_{\mathcal{A}^C}(DS(t))$. For instance, after asking $\{(f, b), (f, e)\}$ for f, f has been determined as a complete skyline tuple, *i.e.*, $f \prec_{\mathcal{A}^C} b$ and $f \prec_{\mathcal{A}^C} e$. In that case, the dominance relationship for f can be used for pruning questions for other tuples. For tuple j, $Q(j) = \{(j, b), (j, e), (j, f), (j, h), (j, i)\}$ for j is asked in Table 2(b). Because $f \prec_{\mathcal{A}^C} \{b, e\}$, it is better to ask (j, f) instead of asking (j, b) and (j, e). If $j \prec_{\mathcal{A}^C} f$ is obtained, we can infer $j \prec_{\mathcal{A}^C} b$ and $j \prec_{\mathcal{A}^C} e$ by transitivity. Based on this property, we can skip two questions (j, b) and (j, e) for $Q(j)$.

LEMMA 4 For each tuple $s \in \textsc{Sky}_{\mathcal{A}^C}(DS(t))$, if $t \prec_{\mathcal{A}^C} s$ holds, then $t \prec_{\mathcal{A}^C} u$ such that $u \in (DS(t) - \textsc{Sky}_{\mathcal{A}^C}(DS(t)))$ holds.

PROOF. By Definition 3, there exists that $s \in \textsc{Sky}_{\mathcal{A}^C}(DS(t))$ dominates $u \in (DS(t) - \textsc{Sky}_{\mathcal{A}^C}(DS(t)))$ in $\mathcal{A}^C$. As a result, if $\forall s \in \textsc{Sky}_{\mathcal{A}^C}(DS(t)) : t \prec_{\mathcal{A}^C} s$ holds, $t \prec_{\mathcal{A}^C} u$ holds by transitivity.

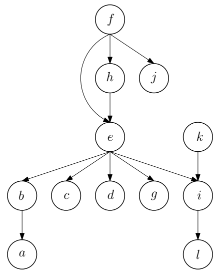


(a) Known values in $\mathcal{A}^K = \{A_1, A_2\}$ (b) $\mathcal{T}$ in $\mathcal{A}^C = \{A_3\}$

Figure 3: An anti-correlated toy dataset for the crowd-enabled skyline query with $\mathcal{A} = \{A_1, A_2, A_3\}$

t	$P(t)$	$Q(t)$	$DS(t)$
a	-	$\{(a, b)\}$	$\{b\}$
g	-	$\{(g, e)\}$	$\{e\}$
d	$\{(b, e)\}$	$\{(d, e)\}$	$\{b, e\}$
k	$\{(i, l)\}$	$\{(k, i)\}$	$\{i, l\}$
c	-	$\{(c, e)\}$	$\{e\}$
f	-	$\{(f, e)\}$	$\{e\}$
h	$\{(e, i)\}$	$\{(h, e)\}$	$\{e, i\}$
j	$\{(f, h)\}$	$\{(j, f)\}$	$\{f, h\}$

(a) Questions asked per tuple



(b) $\mathcal{T}$ in A_3

Figure 4: Overall procedure of probing dominating sets

COROLLARY 2 Given $t \notin \text{SKY}_{\mathcal{A}^C}(DS(u))$, (t, u) is unnecessary in $Q(u)$.

PROOF For each $s \in \text{SKY}_{\mathcal{A}^C}(DS(u))$, if asking (s, u), there are two cases:

1. $s \prec_{\mathcal{A}^C} u$ or $s =_{\mathcal{A}^C} u$: Because $s \prec_{\mathcal{A}^K} u$, $s \prec_{\mathcal{A}} u$ holds, and (t, u) is not needed.

2. $u \prec_{\mathcal{A}^C} s$: Because $s \prec_{\mathcal{A}^C} t$, u exists that $u \prec_{\mathcal{A}^C} t$ by Lemma 4.

In both cases, (t, u) is not needed by asking (s, u). ■

We now adopt a *preference tree* $\mathcal{T}$ that visualizes the preferences of tuples in $\mathcal{A}^C$ in order to compute $\text{SKY}_{\mathcal{A}^C}(DS(t))$ efficiently. Each tuple $t \in \mathcal{R}$ is represented by a node in $\mathcal{T}$. If $s \prec_{\mathcal{A}^C} t$ holds, an edge $s \rightarrow t$ exists. If there exists a path from s to t connected with multiple edges, then $s \prec_{\mathcal{A}^C} t$ also holds by transitivity. If $s \prec\succ_{\mathcal{A}^C} t$, there is no edge between s and t. After asking each question, $\mathcal{T}$ is iteratively updated, and is used for identifying the dominance relationships by checking the path between tuples.

EXAMPLE 5 (USING A PREFERENCE TREE) Continuing from Example 4, Figure 2 depicts a snapshot of $\mathcal{T}$ after evaluating h in A_3. After removing non-skyline tuples, $DS(j)$ is $\{b, e, f, h, i\}$. By checking the dominance relationships between tuples in $DS(j)$, $f \prec_{\mathcal{A}^C} b$, $f \prec_{\mathcal{A}^C} e$, and $h \prec_{\mathcal{A}^C} i$ can be found in $\mathcal{T}$. Therefore, $\text{SKY}_{\mathcal{A}^C}(DS(j))$ is identified as $\{f, h\}$, and only $Q(j) = \{(j, f), (j, h)\}$ is asked in Table 2(b). □

3.4 Probing Dominating Sets

In general, our pruning methods of Corollaries 1 and 2 are more effective if: (1) many tuples are determined as complete non-skyline tuples, and (2) many dominance relationships between tuples are inferred in $\mathcal{A}^C$. However, we observe that the pruning methods may not work well if many non-skyline tuples in $\mathcal{A}^K$ become skyline tuples in $\mathcal{A}$, *e.g.*, anti-correlated distribution. Figure 3(a) illustrates a dataset with 10 tuples in $\mathcal{A}^K = \{A_1, A_2\}$. This can be partitioned to two subsets $\{b, e, i, j\}$ and $\{a, c, f, d, g, h\}$, *i.e.*, the former is skyline tuples in $\mathcal{A}^K$, while the latter is non-skyline tuples in $\mathcal{A}^K$. When all non-skyline tuples become skyline tuples, our pruning methods cannot contribute to reduce the dominating sets of $\{a, c, f, d, g, h\}$. Thus, we have to ask a total of 24 (= 4×6) questions to crowds.

To overcome this problem, we progressively probe $DS(t)$ for t to minimize $\text{SKY}_{\mathcal{A}^C}(DS(t))$. Specifically, in asking questions for $\{b, e, i, j\}$, the dominance relationship can be used for reducing the dominance sets of $\{a, c, f, d, g, h\}$. For instance, suppose that the dominance relationships for $\{b, e, i, j\}$ in Figure 3(b) are updated after asking $\{(b, e), (i, j), (e, i)\}$. Since e dominates $\{b, i, j\}$ in $\mathcal{A}^C$, each single question for each tuple in $\{a, c, f, d, g, h\}$ are generated, *i.e.*, $\{(e, a), (e, c), (e, f), (e, d), (e, g), (e, h)\}$. As a result, our probing method for $DS(t)$ enables us to only ask 9 (= 3 + 6) questions.

We now discuss how to generate questions for probing $DS(t)$. Given $DS(t)$, the number of possible questions is $\binom{|DS(t)|}{2}$. Because all possible ordering in probing $DS(t)$ is exponential and the dominance relationships of tuples in $DS(t)$ are unpredictable, it is non-trivial to decide the right ordering of probing $DS(t)$. As one of feasible solutions, we propose a *greedy* method using the *frequency* of dominating tuples. Let $P(t)$ denote a set of all possible questions $\binom{|DS(t)|}{2}$ for probing $DS(t)$. For each question (u, v) in $P(t)$, $freq(u, v)$ is the number of tuples that are dominated by both u and v, *i.e.*, $freq(u, v) = |\{x \in \mathcal{R} | u \prec_{\mathcal{A}^K} x \wedge v \prec_{\mathcal{A}^K} x\}|$. As $freq(u, v)$ is higher, we suppose that the pruning power of (u, v) gets stronger. We choose the question with the highest frequency in $P(t)$, and remove the questions for less preferred tuples from $P(t)$. This process repeats until no questions exist in $P(t)$.

EXAMPLE 6 (PROBING DOMINATING SETS) The tuples are sorted by the size of dominating sets as shown in Table 2(a). For each tuple $t \in \mathcal{R}$, $DS(t)$ is first pruned by using the two pruning methods in Corollaries 1 and 2. We then probe $DS(t)$ before generating $Q(t)$. Figure 4(a) shows the questions asked per tuple. Before asking $Q(d)$, $P(d) = \{(b, e)\}$ is first probed. When $e \prec_{\mathcal{A}^C} b$ has been decided, b no longer needs to be compared, and thus the questions for b such as (c, b), (f, b), (h, b), and (j, b) can be removed for c, f, h, and j, respectively. Figure 4(b) depicts a preference tree $\mathcal{T}$ in A_3 after all tuples become complete. By probing dominating sets, we only ask 12 questions for identifying the final crowd-enabled skyline $\{b, e, i, l, k, f, h\}$. □

3.5 Algorithm Description

We present the pseudocode of our proposed algorithm, named CrowdSky (Algorithm 1). Overall, CrowdSky works as the combination of machine and crowds. That is, once the machine iteratively updates data structures and generates new questions, crowds return the answers. Specifically: (1) As a degenerate case, it first

Algorithm 1: CrowdSky($\mathcal{R}$)

```
1  foreach $(s,t) \in \mathcal{R} \times \mathcal{R}$ do
2      if $\forall A_j \in \mathcal{A}^K : s_i = t_i$ then
3          Ask $(s,t)$ to crowds and remove a less preferred tuple from $\mathcal{R}$
4  Initialize a preference tree $\mathcal{T}$ in $\mathcal{A}^C$
5  For each tuple $t \in \mathcal{R}$, compute $DS(t)$
6  $\text{SKY}_{\mathcal{A}}(\mathcal{R}) \leftarrow \{\}$ // Initialize a skyline in $\mathcal{A}$
   // P1: Early pruning for non-skylines in $\mathcal{A}$
7  Sort $\mathcal{R}$ by ascending order of $|DS(t)|$
8  for $t \in \mathcal{R}$ do
       // P2: Pruning non-skylines in $\mathcal{A}^C$
9      $DS(t) \leftarrow \text{SKY}_{\mathcal{A}^C}(DS(t))$
       // P3: Probing into $DS(t)$
10     $P(t) \leftarrow \{(u,v) | u,v \in DS(t), u \neq v\}$
11     Sort $P(t)$ by ascending order of $freq(u,v)$
12     for $(u,v) \in P(t)$ do
13         Ask $(u,v)$ to crowds, and update $\mathcal{T}$ with $(u,v)$
14         if $u \prec_{\mathcal{A}^C} v$ in $\mathcal{T}$ then
15             For $x \in DS(t)$, remove $(v,x)$ from $P(t)$
16             $DS(t) \leftarrow DS(t) - \{v\}$
17         else if $v \prec_{\mathcal{A}^C} u$ in $\mathcal{T}$ then
18             For $x \in DS(t)$, remove $(u,x)$ from $P(t)$
19             $DS(t) \leftarrow DS(t) - \{u\}$
       // Generating $Q(t)$ from $DS(t)$
20     $Q(t) \leftarrow \{(s,t) | s \in DS(t)\}$
21     for $(s,t) \in Q(t)$ do
22         Ask $(s,t)$ to crowds, and update $\mathcal{T}$ with $(s,t)$
23         if $s \prec_{\mathcal{A}^C} t$ in $\mathcal{T}$ then
24             break // $t$ is a non-skyline tuple
25     if $\forall s \in DS(t) : s \not\prec_{\mathcal{A}^C} t$ then
26         $\text{SKY}_{\mathcal{A}}(\mathcal{R}) \leftarrow \text{SKY}_{\mathcal{A}}(\mathcal{R}) \cup \{t\}$
27 return $\text{SKY}_{\mathcal{A}}(\mathcal{R})$
```

checks tuples with the same values in $\mathcal{A}^K$, and prunes less preferred tuples in $\mathcal{A}^C$ (lines 1-3). (2) It then builds dominating sets, and sorts them by $|DS(t)|$, incurring $O(n^2)$ in machine part (lines 6-7). (3) For each tuple $t \in \mathcal{R}$, $DS(t)$ is updated by $\text{SKY}_{\mathcal{A}^C}(DS(t))$ to remove non-skyline tuples (line 9). (4) It then generates all possible questions $P(t)$ to probe $DS(t)$ by $freq(u,v)$ (lines 10-11). By asking questions in $P(t)$, $\mathcal{T}$ is updated. In addition $DS(t)$ and $P(t)$ are updated (lines 12-19). (5) After that, $Q(t)$ is generated from $DS(t)$ (lines 20-24). (6) Finally, if t is not dominated by any other tuples in $DS(t)$, t is appended to $\text{SKY}_{\mathcal{A}}(\mathcal{R})$ (lines 25-26).

THEOREM 1 (COMPLETENESS OF CROWDSKY) CrowdSky guarantees that all tuples in $\mathcal{R}$ are complete.

PROOF We prove this by contradiction. Assume that t exists that is not determined as a complete tuple. This means a question (t,u) exists for t to be complete. Because the questions in CrowdSky are pruned by Corollaries 1 and 2, if (t,u) is not asked, a tuple v exists such that $v \prec_{\mathcal{A}^C} u$. In that case, (t,u) is unnecessary to check if t is complete, which is a contradiction. ■

As a result, if crowds always return correct answers, CrowdSky can identity all complete skyline tuples and the result is correct.

4. REDUCING LATENCY

In this section, we discuss how to reduce the number of rounds. Although existing work [25] addresses a parallelization method for asking multiple questions, it is based on a different problem for entity resolution, thereby being inapplicable to our problem. We develop two parallelization methods that are suitable for computing a crowdsourced skyline. Specifically, we first propose a *partitioning method* using dominating sets (Section 4.1), and then improve it using *skyline layers* (Section 4.2).

4.1 Parallelization with Dominating Sets

Given two questions (s,t) and (u,v), if asking (s,t) is not related to prune (u,v) and vice versa, it is said that they are *independent*. Meanwhile, if (s,t) can be pruned by asking (u,v), (s,t) is said to be *dependent* on (u,v). In order to remove unnecessary questions using our pruning methods in CrowdSky, the dependency of questions can happen as follows:

1. *Dominance relationship in* $\mathcal{A}^K$ (C1): Given two tuples s and t such that $s \in DS(t)$, $Q(s)$ needs to be asked before generating $Q(t)$ by Corollary 1 (line 7 in CrowdSky). In other words, when s is determined as a non-skyline tuple, (s,t) is unnecessary in $Q(t)$.

2. *Overlap between* $DS(s)$ *and* $DS(t)$ (C2): We suppose that $DS(s)$ and $DS(t)$ share a common tuple u. When probing $DS(s)$ using the third pruning methods, u can be removed from $\text{SKY}_{\mathcal{A}^C}(DS(t))$ (line 9 in CrowdSky), and (u,t) can be unnecessary in $Q(t)$ by Corollary 2.

3. *Questions for* $DS(t)$ (C3): When $Q(t)$ is sequentially generated from $DS(t)$, t can be a non-skyline tuple (lines 20-24 in CrowdSky), and remaining questions in $Q(t)$ are no longer needed.

Based on these observations, our idea of parallelizing questions in CrowdSky is to identify independent questions as many as possible at each round. Specifically, we first develop a *partitioning method* using dominating sets. First, $\mathcal{R}$ is partitioned into several subsets of tuples with the same sizes of dominating sets. Given $s,t \in \mathcal{R}$, if $|DS(s)| = |DS(t)|$, s and t cannot dominate each other (by Lemma 3). The questions for partitioned tuple sets can be asked together by avoiding dependent questions by (C1). We then check if dominating sets of tuples are disjoint. In that case, probing dominating sets can be parallelized without (C2). Lastly, because (C3) does not make parallelized questions, $Q(t)$ is sequentially generated from $DS(t)$.

EXAMPLE 7 (PARTITIONING METHOD) After the dominating sets are first computed, $\mathcal{R}$ is partitioned into $\{\{a,g\}, \{d,k\}, \{c\}, \{f\}, \{h\}, \{j\}\}$ with the same sizes of dominating sets. For each partitioned set, it then checks if dominating sets are disjoint. Given $\{a,g\}$ and $\{d,k\}$, because $DS(a) \cap DS(g)$ and $DS(d) \cap DS(k)$ are disjoint as illustrated in Figure 4(a), the questions for $\{a,g\}$ and $\{d,k\}$ can be asked together. For $\{a,g\}$, $\{(a,b),(g,e)\}$ is asked in a round. For $\{d,k\}$, $\{(b,e),(i,l)\}$ (in $P(d)$ and $P(k)$) and $\{(d,e),(k,i)\}$ (in $Q(d)$ and $Q(k)$) are asked in 2 rounds. Because $\{c,f,h,j\}$ is partitioned separately, 6 questions are asked in 6 rounds. As a result, our partitioning method generates 12 questions during 9 rounds by saving 3 rounds. □

4.2 Parallelization with Skyline Layers

Although our partitioning method reduces the number of rounds, the degree of parallelization is rather limited by keeping all dependencies of questions. To alleviate this problem, we adopt *skyline layers* that effectively visualize the dominance relationships between tuples in $\mathcal{A}^K$. Figure 5 depicts skyline layers for the toy dataset in Figure 1(a). To build skyline layers, skylines are

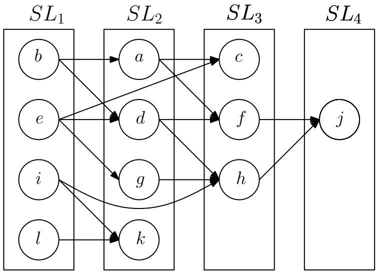


Figure 5: Skyline layers for the toy dataset in Figure 1(a)

computed in an iterative manner. Initially, SL_1 is computed by $\textsc{Sky}_{\mathcal{A}^K}(\mathcal{R})$. Then, the i-th layer is computed by $\textsc{Sky}_{\mathcal{A}^K}(\mathcal{R} - \bigcup_{j=1}^{i-1} SL_j)$ in an iterative manner.

DEFINITION 6 (SKYLINE LAYER) The i-th *skyline layer* SL_i is a set of skyline tuples in $\mathcal{R} - \bigcup_{j=1}^{i-1} SL_j$, *i.e.*, $SL_i = \{t \in \mathcal{R} - \bigcup_{j=1}^{i-1} SL_j | \forall s \in \mathcal{R} - \bigcup_{j=1}^{i-1} SL_j : s \not\prec_{\mathcal{A}^K} t\}$.

After building all layers, the dominance relationship of tuples in $\mathcal{A}^K$ is constructed. Similar to a *dominating graph* [27], the dominance relationship can be represented by a directed edge between two tuples across different layers. In particular, our skyline layers permit the dominance relationship between tuples in any two layers, *e.g.*, $e \to c$ and $i \to h$ in Figure 5. Note that the dominance relationship via transitivity can be inferred from multiple edges.

We now explain how to make use of skyline layers to reduce the number of rounds. Let $c(t)$ denote a set of tuples that directly point to t, *i.e.*, $c(t) = \{s \in \mathcal{R} | s \to t\}$. For each tuple $t \in \mathcal{R}$, we check if all tuples in $c(t)$ are complete. If so, the questions for t are asked *independently* of those of other tuples. When all tuples in $c(t)$ are determined as complete tuples, it implies that all tuples in $DS(t)$ are also complete. Because this method can effectively check the dominance relationships of tuples in (C1), we can significantly improve the magnitude of parallelization. Meanwhile, it is observed that (C2) is a main bottleneck for parallelizing questions. We thus violate the dependency of questions in (C2), by generating additional questions for (C2). (Our empirical study shows that the number of additional questions is negligible.)

Algorithm 2 describes the procedure of parallelizing questions using skyline layers. Let $\mathcal{C}$ be a set of complete tuples in $\mathcal{R}$. (1) $\mathcal{C}$ is initialized as SL_1 (line 4). (2) For each tuple $t \in \mathcal{R}$, if all tuples in $c(t)$ are complete, the questions for t are asked in parallel. (lines 5-7). (3) After asking questions for t, $\textsc{Sky}_{\mathcal{A}}(\mathcal{R})$ and $\mathcal{C}$ are updated (lines 8-10). This process continues until every tuple in $\mathcal{R}$ has been determined as a complete tuple. Because this method is based on the same pruning methods used in CrowdSky, our proposed parallelization methods can assure the correctness of crowd-enabled skyline computation.

EXAMPLE 8 (PARALLELIZATION WITH SKYLINE LAYERS) Given the skyline layers in Figure 5, Table 3 depicts the procedure of parallelization using skyline layers. First, $\mathcal{C}$ is initialized as SL_1, *i.e.*, $\mathcal{C} = \{b, e, i, l\}$. Because $c(a) = \{b\}$, $c(g) = \{e\}$, $c(d) = \{b, e\}$ and $c(k) = \{i, l\}$ are all complete tuples, $\{(a, b), (g, e), (b, e), (i, l)\}$ is asked in parallel (round 1), and $\mathcal{C} = \{b, e, i, l, a, g\}$ is updated (underlines in Table 3). After that, because $c(c) = \{a, e\}$ is complete, $\{(d, e), (k, i)\}$ is asked with $\{(c, e)\}$, and $\mathcal{C} = \{b, e, i, l, a, g, d, k, c\}$ is updated (round 2). Because $c(f) = \{a, d\}$ and $c(h) = \{d, g, i\}$ are complete, the questions for f and h are asked (rounds 3-4), and $\mathcal{C} = \{b, e, i, l, a, g, d, k, c, f, h\}$ is updated. Lastly, when $c(j) = \{f, h\}$ is complete, the questions for j are asked (rounds 5-6). As a result, our parallelization method generates 12 questions during 6 rounds. □

Algorithm 2: ParallelSL($\mathcal{R}$)

```
1  Execute lines 1-5 in Algorithm 1
2  $\textsc{Sky}_{\mathcal{A}}(\mathcal{R}) \leftarrow \{\}$ // Initialize a skyline in $\mathcal{A}$
3  Compute $SL_1, \ldots, SL_k$ for $\mathcal{R}$
4  $\mathcal{C} \leftarrow SL_1$ // Initialize a complete tuple set
5  for $t \in \mathcal{R}$ do in parallel
6      if $c(t) \subseteq \mathcal{C}$ then
7          For $t$, execute lines 9-24 in Algorithm 1
8          if $\forall s \in DS(t) : s \not\prec_{\mathcal{A}} t$ then
9              $\textsc{Sky}_{\mathcal{A}}(\mathcal{R}) \leftarrow \textsc{Sky}_{\mathcal{A}}(\mathcal{R}) \cup \{t\}$
10         $\mathcal{C} \leftarrow \mathcal{C} \cup \{t\}$ // Update $\mathcal{C}$
11 return $\textsc{Sky}_{\mathcal{A}}(\mathcal{R})$
```

Table 3: Procedure of asking questions using skyline layers in Figure 4(a)

t	$c(t)$	1	2	3	4	5	6
a	$\{b\}$	$\underline{(a, b)}$					
g	$\{e\}$	$\underline{(g, e)}$					
d	$\{b, e\}$	(b, e)	$\underline{(d, e)}$				
k	$\{i, l\}$	(i, l)	$\underline{(k, i)}$				
c	$\{a, e\}$		$\underline{(c, e)}$				
f	$\{a, d\}$			$\underline{(f, e)}$			
h	$\{d, g, i\}$			(e, i)	$\underline{(h, e)}$		
j	$\{f, h\}$					(f, h)	$\underline{(j, f)}$

5. IMPROVING ACCURACY OF ANSWERS

Because the answers of crowds are often erroneous, how to improve the accuracy is a central issue in crowdsourcing. As discussed in [24, 25], it was treated as a research problem orthogonal to the problem of minimizing the number of questions. Existing work [6, 11, 18] developed how to improve the accuracy for each question using *query-independent* methods.

As the simplest method, *majority voting* is used by assigning multiple workers per question. Let ω denote the number of workers per question, and p denote the probability that each worker's answer is correct. The probability that question (u, v) is correct can be modeled as the binomial distribution.

$$P(u, v) = \sum_{i=\lceil \frac{\omega}{2} \rceil}^{\omega} \binom{\omega}{i} p^i (1-p)^{\omega - i},$$

where ω is an odd integer. This method can improve the accuracy per question, but does not consider the *importance* of questions in computing a skyline. This method is called *static voting*. (In our experiments, $\omega = 5$ by default.)

As the *query-dependent* method, we develop a heuristic method to reflect the importance of questions. When computing the skyline, it is observed that the questions with many dominance relationships in $\mathcal{A}^K$ are more influential in identifying a more accurate preference tree $\mathcal{T}$ in $\mathcal{A}^C$. Based on this observation, we propose to use the *frequency* of questions $freq(u, v)$ for quantifying the importance of (u, v), *i.e.*, $freq(u, v) = |\{x \in \mathcal{R} | u \prec_{\mathcal{A}^K} x \wedge v \prec_{\mathcal{A}^K} x\}|$. That is, as $freq(u, v)$ gets larger, (u, v) becomes more important. Given question (u, v), the different numbers of workers can be assigned, depending on $freq(u, v)$. We refer to this method as *dynamic voting*. Note that the dynamic voting can help prevent the propagation of false dominance relationships.

Table 4: Parameter settings over synthetic datasets

parameters	value	default
cardinality n	2K, 4K, 6K, 8K, 10K	4K
# of known attributes $\lvert\mathcal{A}^K\rvert$	2, 3, 4, 5	4
# of crowd attributes $\lvert\mathcal{A}^C\rvert$	1, 2, 3	1
data distribution	IND, ANT	

For instance, based on the idea of the dynamic voting, one may dynamically assign the number of workers using two parameters α and β ($\alpha < \beta$ and $\alpha, \beta \geq 0$) as follows. Given $freq(u, v)$, we choose the number of workers ω' with the following inequalities.

$$\omega' = \begin{cases} \omega - 2, & if\ freq(u,v) < \alpha \\ \omega, & if\ \alpha \leq freq(u,v) < \beta \\ \omega + 2, & if\ freq(u,v) \geq \beta \end{cases}$$

By considering the importance of questions, our dynamic voting can improve the accuracy of the skyline result compared to the static voting. Note that it can be easily extended for multiple categories with three or more cases.

6. EXPERIMENTS

In this section, we first evaluate our proposed algorithm over synthetic datasets with extensive parameter settings (Section 6.1). We show the performance of our proposed algorithm in terms of three key factors. We then evaluate our algorithm using the Amazon Mechanical Turk over real-life datasets (Section 6.2).

6.1 Evaluation in Synthetic Datasets

We first evaluate our proposed algorithm over synthetic datasets. Because real-life datasets are limited for evaluating extensive settings, we adopted benchmark datasets [2] that are widely used in skyline evaluation. In particular, we used two data distributions, *independent* (IND) and *anti-correlated* (ANT) in [2]. All attribute values were randomly generated from real values in [0, 1]. The values on crowd attributes were only used for obtaining the answers of crowds for simulated questions. Table 4 summarizes parameter settings over synthetic datasets.

We validate our proposed algorithm for three key factors: monetary cost, latency, and accuracy (as discussed in Section 2.1). Because the running time in crowdsourcing part is much slower than generating questions in machine part, we focus on evaluating the number of rounds for latency. All experiments were conducted in Windows 7 running on Intel Core i7 950 3.07 GHz CPU with 16GB RAM. All the algorithms were implemented in C++. The average values of 10 runs are reported for all experiments.

Monetary cost. When the fixed amount of a reward per question is paid, the monetary cost is proportional to the number of questions asked to crowds. We thus use the number of questions to measure the monetary cost. As one of the sorting algorithms, *tournament sort* is used as a baseline, denoted by Baseline. When the number of rounds is not limited, tournament sort can produce the *total order* of tuples with the minimum number of questions. An existing work [12] studied crowd-enabled skyline queries, but focused on selecting the most influential unary questions for a restricted budget. Because the optimization methods in [12] are not effective for reducing questions in our problem setting, the direct comparison between CrowdSky and [12] is not fair to [12]. As such, we simulate the unary questions in [12] for comparing the accuracy, and mainly focus on scrutinizing CrowdSky to quantify the advantage of optimization methods. Specifically, it is divided into four phases: DSet (Section 3.1), P1 (Section 3.2), P2 (Section 3.3), and P3 (Section 3.4).

Figures 6(a) and 7(a) depict the number of questions over varying cardinality. Note that DSet is basically used for all pruning methods. It is clear that P1+P2+P3 minimizes the number of questions over all parameter settings, *e.g.*, reducing the number of questions 10 times more than that of Baseline over independent distribution. In particular, several interesting observations are found in Figures 6(a) and 7(a): (1) While DSet produces fewer questions than Baseline in independent distribution, it is reversed in anti-correlated distribution. This is because the number of skyline tuples increases exponentially with the cardinality over anti-correlated distribution. (2) Both P1 and P2 can contribute to reducing unnecessary questions. Notably, P2 (using the transitivity of tuples in $\mathcal{A}^C$) is fairly effective over anti-correlated distribution. (3) As expected, P3 (probing dominating sets) reduces a more number of unnecessary questions over anti-correlated distribution.

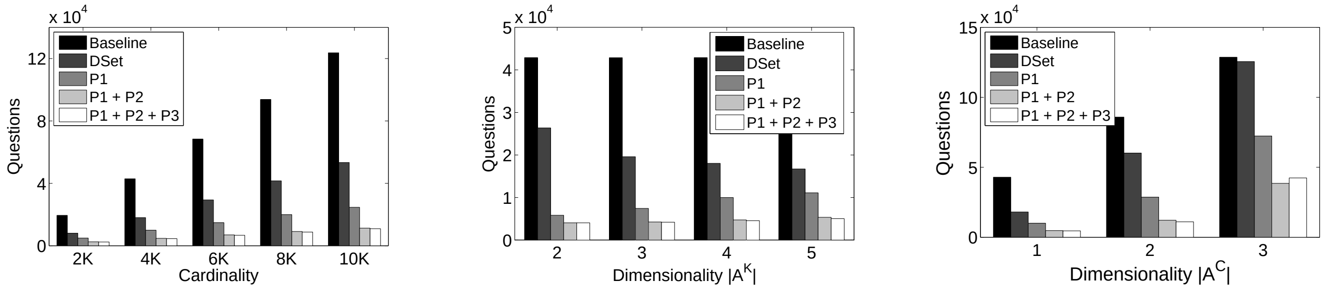
Figures 6(b) and 7(b) report the number of questions over varying $|\mathcal{A}^K|$. While Baseline shows a constant performance regardless of $|\mathcal{A}^K|$, our pruning methods reduce the number of questions as $|\mathcal{A}^K|$ increases. This is because the size of dominating sets tends to decrease with $|\mathcal{A}^K|$. We also found two key observations: (1) P1+P2+P3 minimizes the number of questions over all parameter settings. (2) When $|\mathcal{A}^K|$ is low, our pruning methods significantly reduce unnecessary questions, and are much more effective over anti-correlated distribution than independent distribution. When $|\mathcal{A}^K| = 2$, P1+P2+P3 reduces the number of questions in DSet by two orders of magnitude.

Figures 6(c) and 7(c) report the number of questions over varying $|\mathcal{A}^C|$. When $|\mathcal{A}^C| > 1$, suppose that all methods simply generate $|\mathcal{A}^C|$ questions in $\mathcal{A}^C$. (It is possible to use a round-robin strategy for multiple crowd attributes to reduce unnecessary questions as they become incomparable in $\mathcal{A}^C$, but it is not applied to our evaluation.) (1) As $|\mathcal{A}^C|$ increases, the number of questions increases for all methods. (2) Interestingly, when $|\mathcal{A}^C| = 2$, P3 becomes marginal. As $\mathcal{A}^C$ increases, P3 becomes less effective for generating dominance relationships in $\mathcal{A}^C$. This implies that probing dominating sets mainly incurs the questions for tuples with incomparable relationships. In high dimensionality in $\mathcal{A}^C$, we have to consider to use P3 in CrowdSky.

Latency. In order to measure latency, we used the number of rounds in performing an algorithm. We compared the following four algorithms: (1) Baseline is tournament sort; (2) Serial asks a single question in a round; (3) ParallelDSet is our parallelization algorithm using dominating sets (Section 4.1); (4) ParallelSL is our parallelization algorithm using skyline layers (Section 4.2).

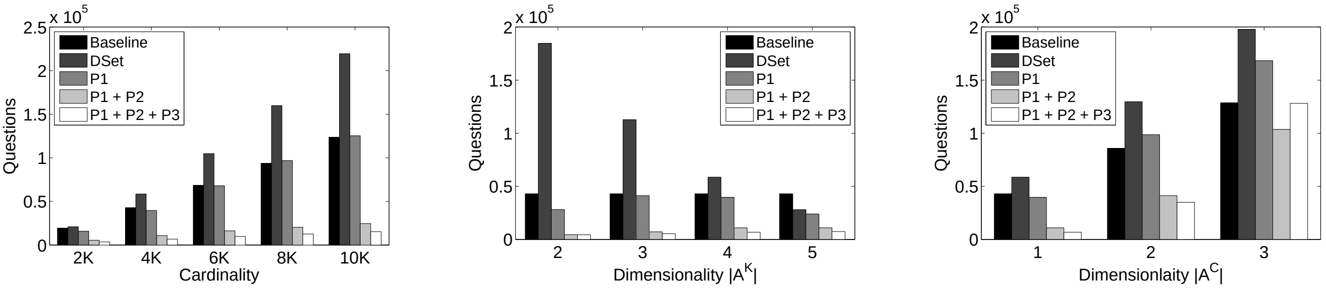
Figure 8 reports the number of rounds over varying cardinality. Note that the *y*-axis is log-scaled. The gap between Serial and ParallelDSet widens by one order of magnitude as cardinality increases. In addition, ParallelSL outperforms ParallelDSet by two orders of magnitude, *e.g.*, ParallelSL only needs about 20-30 rounds in both distributions. Although we did not report the number of questions for parallelization, it is found that ParallelDSet generates the same number of questions for Serial, and ParallelSL generates approximately 10% more questions than ParallelDSet, by violating the dependency of questions in (C2).

Figure 9 reports the number of rounds over varying dimensionality $|\mathcal{A}^K|$. Interestingly, while Serial incurs more rounds with higher $|\mathcal{A}^K|$, ParallelDSet and ParallelSL decrease the number of rounds with $|\mathcal{A}^K|$. This implies that the degree of parallelization becomes higher as $|\mathcal{A}^K|$ increases. As consistently observed in Figure 8, ParallelSL significantly outperforms ParallelDSet by two orders of magnitude in both distributions.



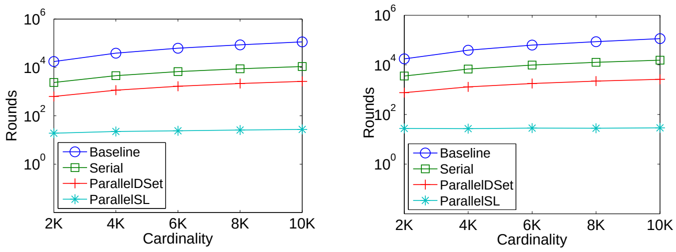
(a) varying cardinality (b) varying $|\mathcal{A}^K|$ (c) varying $|\mathcal{A}^C|$

Figure 6: Comparisons on the number of questions over independent distribution



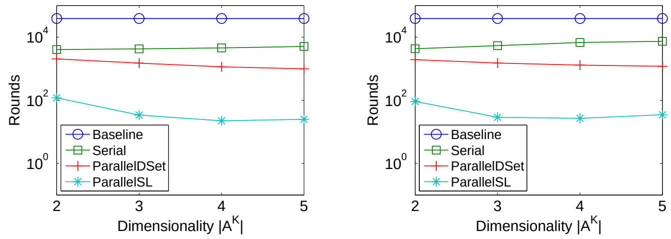
(a) varying cardinality (b) varying $|\mathcal{A}^K|$ (c) varying $|\mathcal{A}^C|$

Figure 7: Comparisons on the number of questions over anti-correlated distribution



(a) IND (b) ANT

Figure 8: Comparisons on the number of rounds over varying cardinality



(a) IND (b) ANT

Figure 9: Comparisons on the number of rounds over varying dimensionality of known attributes

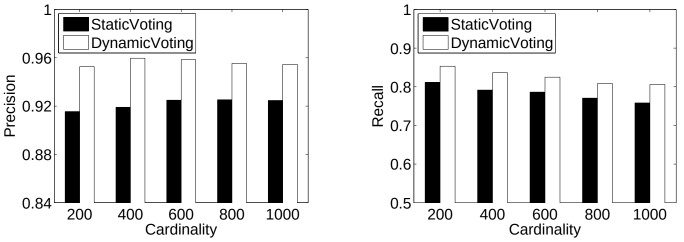
Accuracy. The crowdsourced skyline is defined as $\textsc{Sky}_{\mathcal{A}}(\mathcal{R})$. To measure the accuracy of skyline results, we only use a set of newly retrieved skyline tuples by crowdsourcing, *i.e.*, $\textsc{Sky}_{\mathcal{A}}(\mathcal{R}) - \textsc{Sky}_{\mathcal{A}^K}(\mathcal{R})$, with two metrics, *precision* and *recall*, which are widely used in Information Retrieval.

We first compared two assignment methods, StaticVoting and DynamicVoting in CrowdSky (as in Section 5). By default, we set $\omega = 5$ and $p = 0.8$. While StaticVoting equally assigns ω per question, DynamicVoting assigns $\omega + 2$, ω, $\omega - 2$ depending on the frequency of questions. For fair comparisons, we assigned the same total number of workers in both methods. The assignment of the number of workers in DynamicVoting is tuned as: the initial 30% questions are assigned to $\omega + 2$, and the last 30% questions are assigned to $\omega - 2$. This implies that the initial questions in DynamicVoting are more important than other questions.

Figure 10 reports the accuracy of two voting methods over varying cardinality. It is clear that DynamicVoting shows higher accuracy than StaticVoting for both metrics. Note that DynamicVoting improves the overall accuracy by assigning more workers to more important questions and by reducing the propagation for false positives/negatives. In addition, the precision is higher than the recall in all parameter settings. While most of the skyline tuples are decided correctly, some correct skyline tuples are determined as non-skyline tuples. This is because our methods focus on asking questions for skyline candidates and do not perform an additional validation for non-skyline tuples.

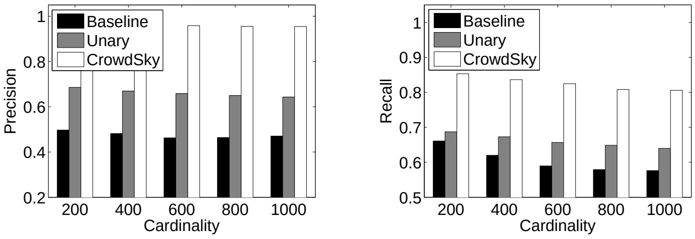
We also compared the following three algorithms: (1) Baseline generates the total order of tuples in $\mathcal{A}^K$ by performing the tournament sort, (2) Unary generates the total order of tuples by asking unary questions (as done in [12]), and (3) CrowdSky adopts DynamicVoting. To simulate the unary questions in [12], we randomly select a value from the normal distribution of actual value in $\mathcal{A}^C$. In this setting, it is found that Unary is more accurate for obtaining the total order of tuples than Baseline, *i.e.*, it is more favorable for Unary.

Figure 11 reports the accuracy of comparing CrowdSky and the two existing methods. Interestingly, even though Baseline generates more numbers of questions than CrowdSky, Baseline is



(a) Precision (b) Recall

Figure 10: Accuracy comparisons of two voting methods in CrowdSky over independent distribution



(a) Precision (b) Recall

Figure 11: Accuracy comparisons of CrowdSky with existing methods (*i.e.*, Unary as in [12]) over independent distribution

worse than CrowdSky. Because more questions in Baseline incur wrong answers, the total order of tuples in Baseline is less effective for identifying a correct skyline. In contrast, CrowdSky generates questions more selectively for skyline candidates. While Unary is better than Baseline, it is worse than CrowdSky. Because the pruning methods in [12] do not work well in our problem setting, Unary is less effective for identifying correct skyline tuples.

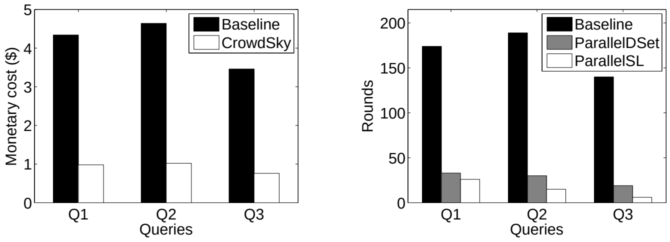
6.2 Evaluation in Real-life Datasets

We used three real-life datasets to validate our proposed algorithms: (1) **Rectangles**, adopted from [14], contains 50 images whose sizes are $\{(30+3i)\times(40+5i)|i\in[0,50)\}$ and are randomly rotated. (2) **IMDb Movies** includes 50 popular movies released in 2000-2012 (`http://www.imdb.com/`). (3) **MLB players** includes 40 baseball pitchers played in 2013 (`http://espn.go.com/mlb/stats/`). For these datasets, we used small-scale datasets in order to manage the monetary cost in real-life experiments, and executed the following crowd-enabled skyline queries:

Q1: Find the skyline using rectangle data with $\mathcal{A}^K$ = {`width`, `height`} and $\mathcal{A}^C$ = {`area`}. The larger values in $\mathcal{A}^K$ and $\mathcal{A}^C$ are more preferred. As the ground truth for crowd attribute `area` can be obtained using `width` and `height`, the accuracy of the crowdsourced skyline can be measured.

Q2: Find the skyline using movie data with $\mathcal{A}^K$ = {`box_office`, `release_year`} and $\mathcal{A}^C$ = {`rating`}. The larger values in $\mathcal{A}^K$ and $\mathcal{A}^C$ are more preferred. Since IMDb shows aggregated rating scores of movies, we compare the crowdsourced skyline (using preferences culled from crowds) against the IMDb rating-based skyline.

Q3: Find the skyline using MLB player data with $\mathcal{A}^K$ = {`wins`, `strikes_outs`, `ERA`} and $\mathcal{A}^C$ = {`valuable`}. The larger values are more preferred, except for `ERA`. The crowd attribute `valuable` is the preference of crowds on how *valuable* each pitcher is. For this query, we indirectly compare the crowdsourced skyline against the candidates of the “Cy Young" award, given to the best pitchers annually.



(a) Monetary cost (b) # of rounds

Figure 12: Comparisons of three queries over real-life datasets

The Amazon Mechanical Turk (AMT), a well-known crowdsourcing platform, was used for asking questions to crowds in real-life datasets. The budget per question was set to $0.02, and 5 workers were assigned for each question, *i.e.*, $\omega = 5$. Let r denote the total number of rounds, and $|\mathcal{Q}_i|$ denote the number of questions at the i-th round. The total monetary cost is thus calculated as: $0.02\times5\times\sum_{i=1}^{r}\lceil\frac{|\mathcal{Q}_i|}{5}\rceil$, where 5 questions are issued at each task. To filter out spam workers, we only permitted *Masters* workers who are qualified as the most reliable workers in AMT.

Monetary cost. Figure 12(a) compares the monetary cost between Baseline and CrowdSky. Note that CrowdSky saves the cost of Baseline by 3-4 times. (Because the cardinality of real-life datasets is smaller than that of synthetic datasets, the performance gap between Baseline and CrowdSky is reduced.) For *Q1* and *Q2*, while Baseline needs more than 200 questions, CrowdSky generates about 50 questions, where most questions are used for validating non-skyline tuples.

Latency. Figure 12(b) compares the latency of three algorithms: Baseline, ParallelDSet, and ParallelSL. For $Q1$, $Q2$, and $Q3$, the average working time per HIT was 22 secs, 49 secs, and 1 min 33 secs, respectively, implying that $Q3$ is the most difficult task. While Baseline incurs more than 140 rounds over all queries, ParallelDSet and ParallelSL only generate less than 30 rounds. In addition, ParallelSL generates 50% less rounds than ParallelDSet, without increasing the cost for all queries. For *Q3*, while Baseline and ParallelDSet need 140 and 19 rounds, ParallelSL computes the skyline with only 6 rounds.

Accuracy. For *Q1*, CrowdSky identifies the same skyline as the ground truth, yielding *Precision* = 1.0 and *Recall* = 1.0. For *Q2*, the crowdsourced skyline includes 5 movies such as {Avatar, The Avengers, Inception, The Lord of Rings: The Fellowship of the Ring, The Dark Knight Rises}. Except for the existing skyline {Avatar, The Avengers} in $\mathcal{A}^K$, we found that the average rating of three skyline movies is very high (*i.e.*, 8.7 out of 10.0) in IMDb, indicating that crowds were able to find decent skyline movies. For *Q3*, the skyline includes four players such as {Clayton Kershaw, Bartolo Colon, Yu Darvish, Max Scherzer} who were Cy Young award candidates, representing the best pitchers, in 2013. In particular, “Clayton Kershaw” and “Max Scherzer” were the winners of the Cy Young award in 2013. As such, we claim that the crowdsourced skyline be reasonably accurate. Based on the results, we argue that CrowdSky yields high accuracy while keeping the monetary cost and the latency low.

7. RELATED WORK

Skyline queries have been actively studied for assisting multi-criteria decision making applications. Pioneered by [2], skyline queries are used for various data settings such as distributed and stream environments. Tuples are represented by incomplete and probabilistic values, and data types vary from partially-ordered and categorical attributes. Existing work focused on developing efficient skyline computation with pre-defined preferences. In contrast, we aimed to collect missing preferences from crowds.

In recent years, in data management community, there have been active investigations toward crowd-enabled algorithms. Some of recent highlights include the following data operations with crowdsourcing embedded: selection [6, 18, 21], max [8, 22, 23], sorting [14], top-*k* [4], top-*k* set [20], join [14, 24, 25, 26], and group by [4]. Based on these advancements, our work combines the skyline queries with crowdsourcing.

In particular, our crowd-enabled skyline query is related to [12], which is the first work to address the problem of skyline queries with a crowdsourcing idea. However, our work has clear differences from [12] as follows:

- *Problem formulation*: While [12] used crowds to improve the accuracy of incomplete skyline queries, our work addressed a complete skyline by collecting all missing preferences in crowd attributes.
- *Optimization direction*: While [12] mainly aimed at maximizing the accuracy of skyline results, we considered three factors (monetary cost, latency, and accuracy) together.
- *Formats of questions*: Since [12] used the quantitative questions, it is inapplicable for crowd attributes with a large range. On the contrary, since our work is based on the qualitative questions, it is easier for crowds to evaluate tuples in crowd attributes without any constraints.

Because the optimization methods in [12] are not effective for reducing questions in our problem setting, the direct comparison between CrowdSky and [12] would have been unfair to [12]. As such, in Section 6, we have simulated the unary questions in [12] for comparing the accuracy in our setting, and indirectly demonstrated the superiority of CrowdSky with a strong evidence (*i.e.*, Figure 11).

8. CONCLUSIONS

In this paper, we have studied the problem of computing skyline queries with crowdsourcing. Specifically, we dealt with three key factors such as monetary cost, latency, and accuracy. Our proposed algorithm first aimed to minimize the number of questions with several pruning methods on top of the notion of a dominating set. We then developed an algorithm to minimize the number of rounds using skyline layers. We lastly improved the accuracy of a crowdsourced skyline using dynamic majority voting. Our experimental results showed that our proposed algorithm optimizes the three key factors effectively over synthetic and real-life datasets.

Acknowledgements

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10054151 and No. 2015R1C1A1A01055442).

9. REFERENCES

[1] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, and W. C. Tan. Asking the right questions in crowd data sourcing. In *ICDE*, pages 1261–1264, 2012.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[4] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *ICDT*, 2013.

[5] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[6] J. Gao, X. Liu, B. C. Ooi, H. Wang, and G. Chen. An online cost sensitive decision-making method in crowdsourcing systems. In *SIGMOD*, pages 217–228, 2013.

[7] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. W. Shavlik, and X. Zhu. Corleone: hands-off crowdsourcing for entity matching. In *SIGMOD Conference*, pages 601–612, 2014.

[8] S. Guo, A. G. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD*, pages 385–396, 2012.

[9] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[10] J. Lee and S. won Hwang. BSkyTree: scalable skyline computation using a balanced pivot selection. In *EDBT*, pages 195–206, 2010.

[11] 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.

[12] C. Lofi, K. E. Maarry, and W.-T. Balke. Skyline queries in crowd-enabled databases. In *EDBT*, 2013.

[13] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Demonstration of Qurk: a query processor for humanoperators. In *SIGMOD*, pages 1315–1318, 2011.

[14] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1):13–24, 2011.

[15] L. Mo, R. Cheng, B. Kao, X. S. Yang, C. Ren, S. Lei, D. W. Cheung, and E. Lo. Optimizing plurality for human intelligence tasks. In *CIKM*, pages 1929–1938, 2013.

[16] A. Morishima, N. Shinagawa, T. Mitsuishi, H. Aoki, and S. Fukusumi. CyLog/Crowd4U: A declarative platform for complex data-centric crowdsourcing. *PVLDB*, 5(12):1918–1921, 2012.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, pages 467–478, 2003.

[18] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD*, pages 361–372, 2012.

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

[20] V. Polychronopoulos, L. de Alfaro, J. Davis, H. Garcia-Molina, and N. Polyzotis. Human-powered top-k lists. In *WebDB*, 2013.

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

[22] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, pages 989–998, 2012.

[23] V. Verroios, P. Lofgren, and H. Garcia-Molina. tdp: An optimal-latency budget allocation strategy for crowdsourced MAXIMUM operations. In *SIGMOD*, pages 1047–1062, 2015.

[24] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. CrowdER: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[25] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, pages 229–240, 2013.

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

[27] L. Zou and L. Chen. Dominant graph: An efficient indexing structure to answer top-k queries. In *ICDE*, pages 536–545, 2008.