

The Complexity of Evaluating Order Queries with the Crowd *

Benoit Groz, Tova Milo, Sudeepa Roy

▶ **To cite this version:**

Benoit Groz, Tova Milo, Sudeepa Roy. The Complexity of Evaluating Order Queries with the Crowd *. Bulletin of the Technical Committee on Data Engineering, 2016. hal-01803655

HAL Id: hal-01803655
https://hal.science/hal-01803655

Submitted on 31 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Complexity of Evaluating Order Queries with the Crowd*

Benoît Groz
Univ Paris-Sud
groz@lri.fr

Tova Milo
Tel Aviv University
milo@cs.tau.ac.il

Sudeepa Roy†
Duke University
sudeepa@cs.duke.edu

1 Introduction

One of the foremost challenges for information technology over the last few years has been to explore, understand, and extract useful information from large amounts of data. Some particular tasks such as annotating data or matching entities have been outsourced to human workers for many years. But the last few years have seen the rise of a new research field called *crowdsourcing* that aims at delegating a wide range of tasks to human workers, building formal frameworks, and improving the efficiency of these processes.

The database community has thus been suggesting algorithms to process traditional data manipulation operators with the crowd, such as joins or filtering. This is even more useful when comparing the underlying "tuples" is a subjective decision – *e.g.*, when they are photos, text, or simply noisy data with different variations and interpretations – and can presumably be done better and faster by humans than by machines.

The problems considered in this article aim to retrieve a subset of preferred items from a set of items by delegating pairwise comparison operations to the crowd. The most obvious example is finding the maximum of a set of items (called *max*). We also consider two natural generalizations of the max problem:

1. *Top-k:* computing the top-k items according to a given criterion (max = top-1), and
2. *Skyline:* computing all Pareto-optimal items when items can be ordered according to multiple criteria (max has a single criterion).

We assume that there is an underlying ground truth for all these problems, *i.e.*, the items have fixed values along all criteria that lead to well-defined solutions for these problems. However, these values are unknown and the only way to reach a solution is by asking the crowd to compare pairs of items. Toward this goal, we adopt a widespread and simple model for the crowd and present the prevalent theoretical ideas from the existing work in the literature that allow us to compute the preferred items in this setting. This simple model builds the foundation for a formal framework, allows us to do a rigorous analysis, and serves as the basis for understanding the more general and practical cases. We also outline the limitations of the current settings and algorithms, and identify some interesting research opportunities in this domain.

To illustrate the above problems, imagine some Mr. Smith has been provided with a voucher for a stay in a ski resort location of his choice from a set of locations. Mr. Smith may wish to choose the location with the

Copyright 0000 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work has been partially funded by the European Research Council under the FP7, ERC grant MoDaS, agreement 291071, the Israel Ministry of Science, and by the NSF award IIS-0911036.

†This work has been done while the author was at the University of Washington.

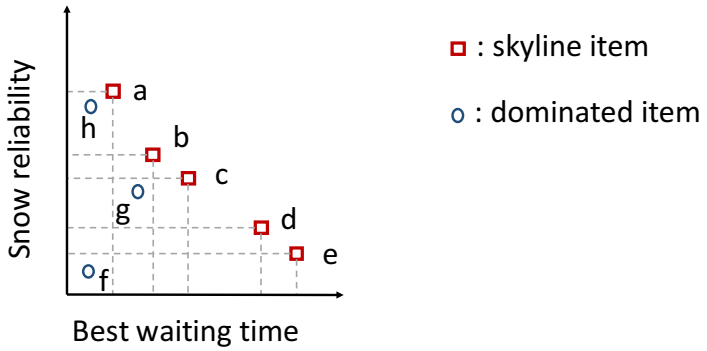


Figure 1: Example: Skyline of ski resorts

best possible crowd management (*i.e.*, where the average waiting time in the queue is the shortest). One possible way to reach a decision would be to ask people on the web about how the ski resorts compare to one another according to this criterion (pairwise comparisons). But to consider more options, he may also ask for the top-5 locations with respect to waiting time. And finally, if he cares about multiple criteria, like both the *waiting time* and the *snow reliability* of the ski resort at a given time, he may wish to remove from the list all locations that are beaten by another simultaneously on waiting time and snow reliability, and retain every location surviving this process for further examination (*e.g.*, to consider additional criteria like slopes, availability of cable cars and elevators, cleanliness, etc.). Figure **??** shows eight such fictive locations on these two criteria assuming that higher values are better for both criteria.

To achieve such goals, platforms like Amazon Mechanical Turk or CrowdFlower [40, 12, 39, 30, 11] help their customers connect to the crowd from all over the world through the Internet. The customers post *jobs* on these platforms, and the *workers* from the crowd solve multiple *tasks* for these jobs, and get paid for each task in exchange. For a given job (*e.g.*, sorting a list of items), the workers are given some tasks in rounds (*e.g.*, compare two given items). Once the answers to the tasks in the previous round are returned, some internal computation is done by the machine, the next set of tasks is decided, and given to the user in the next round. As the crowd is a more expensive resource compared to a machine, the *cost* of an algorithm in crowdsourcing setting is typically measured by the total cost paid to the workers while solving a job.

Workers gathered from the web, however, are prone to return erroneous answers. One major issue in crowdsourcing is therefore to mitigate those errors, typically through redundant tasks. In the absence of identified experts, the traditional "wisdom of the crowd" assumption in crowdsourcing means that the results aggregated from multiple workers will eventually converge toward the truth. Therefore, crowdsourcing algorithms endeavor to submit as few redundant tasks as possible to achieve a desired *accuracy* level, which bounds the probability of returning an incorrect result for the intended job. Sometimes, the *latency* of a crowdsourcing algorithm is also evaluated as a criterion in addition to cost and accuracy, which may include the number of rounds of tasks given to the workers, and also the workload in those rounds.

Obtaining precise cost, accuracy, and latency models for crowd workers by accurately estimating human behavior is a non-trivial task by itself. Different workers may have different levels of expertise and sincerity to solve a task, and may take different amounts of time in the process. Tasks can be assigned individually, or multiple tasks can be assigned together in a batch. The answers to various tasks returned by the same or different workers may be correlated. The customers may want to pay the same or different amounts of money for different tasks. In this article, however, we consider a simple and standard probabilistic setting to model the possible incorrect answers returned by the crowd, where the crowd is perceived as a *noisy comparison oracle*.

Crowd as a noisy comparison oracle. We assume that the items admit a correct, albeit implicit, solution

to each of the problems considered in this article (max, top-k, or skyline). We assume that values of the items (along single or multiple criteria) are not known explicitly and can only be recovered by asking the crowd to compare two items. In the simple and standard error model (called the *constant error model*), the crowd is considered as an oracle that always returns the correct answer with a constant probability $> \frac{1}{2}$ (say, $\frac{2}{3}$), *i.e.*, the oracle always performs better than a random answer. In other words, given two items x, y such that $x > y$, each *oracle call* $\mathcal{O}(x > y)$ will return $true$ with probability $\frac{2}{3}$ and $false$ with probability $\frac{1}{3}$. Further, the answers to two oracle calls are independent.

In this model, one can increase the confidence arbitrarily by repeating a query to the oracle and adopting majority vote. In practice, this means asking several people the same question. Every problem could be thus solved by repeating each oracle call a large number of times and adopting the majority vote. The challenge in this setting is to obtain more efficient algorithms by focusing on the most crucial questions to reduce redundancy on others. To control the error introduced by potentially incorrect oracle answers, all of our algorithms take as input some parameter $\delta < \frac{1}{2}$ called the *tolerance*. The algorithms will be designed with sufficient redundancy so that their final output is correct with probability at least $1 - \delta$. In the simple cost model, we will assume unit (*i.e.*, constant) cost per oracle call, and aim to optimize the total *number* of oracle calls; we will also discuss other interesting error and cost models in the following sections.

Roadmap. In Section 2, we discuss the max and top-k problems for noisy comparison oracles based on the work by Feige et al. [16] and Davidson et al. [13, 14]. In Section 3, we discuss the skyline problem for noisy comparisons based on the work by Groz and Milo [20]. We review the related work in Section 4 and conclude with future directions in Section 5.

2 Max and Top-k

First, we discuss the max and top-k problems in the setting with a noisy comparison oracle:

> **Input:** A set of items $S = \{x_1, \cdots, x_n\}$ in arbitrary order, where $x_1 > x_2 > \cdots > x_n$; tolerance δ; noisy comparison oracle $\mathcal{O}$; and $k \in [2, n]$ for top-k.
> **Objective:** (i) (**Max**) Return the maximum item (x_1) with error probability $\leq \delta$. (ii) (**Top-k**) Return $x_1, \cdots, x_k$ in order with error probability $\leq \delta$.

In Figure **??**, the location that maximizes the experience with waiting time is location e, and the top-k locations according to this ordering for $k = 5$ are e, d, c, b, g.

2.1 Constant and Variable Error Model, Unit Cost Function

For the unit cost function, each call to the oracle for comparing two items has the same constant cost. Therefore, it suffices to count the number of calls to the oracle. Further, in practice, the comparisons can be delegated to one or more crowd workers arbitrarily without affecting the total cost. Next, we discuss two error models for the unit cost function.

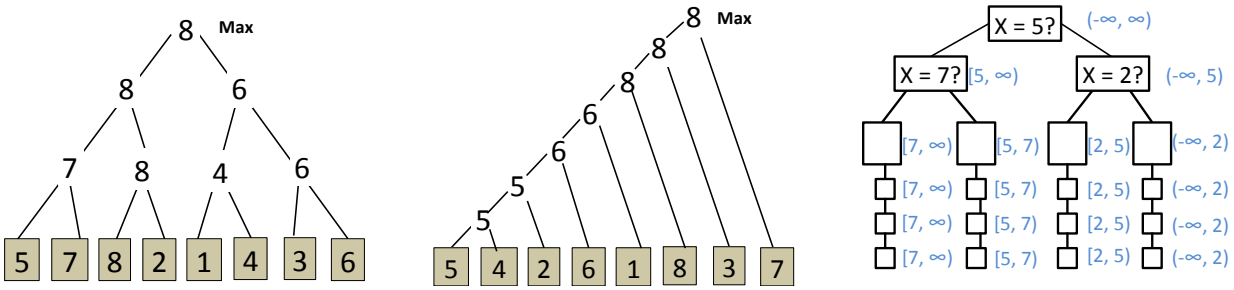
Constant Error Model

The *constant error model* is a standard error model for noisy comparison operations [16, 31, 19], where the probability of getting the correct answer from a noisy oracle is a constant $> \frac{1}{2}$, *i.e.*,

$$(\textbf{Constant error model}) \qquad \Pr[\mathcal{O}(x_i > x_j \mid i < j) = \mathit{false}] = p, \qquad \textit{where } p \textit{ is a constant} < \frac{1}{2}$$

Theorem 1: Feige et. al. [16]: The oracle complexity of top-k under the constant error model is $\Theta(n \log(m/\delta))$, where $m = \min(k, n-k)$. In particular, the oracle complexity of $\max$ is $\Theta(n \log(1/\delta))$.

Tournament Trees. *Tournament trees* (also called *comparison trees* in [14, 13]) are frequently used for the max (min) or selection problems. A tournament tree is a form of max-heap (or min-heap if the minimum is sought) which is a complete binary tree. Every leaf node represents an item and every internal node represents the winner (*e.g.*, the larger item) between two of its children as a result of their comparison. Figures 2a and 2b show two example tournament tree that returns the maximum of $n = 8$ items at the root of the tree, assuming there is no error in the comparisons. For noisy comparisons, at each internal node, multiple comparisons are performed and then the winner is decided, *e.g.*, by a majority vote.



(a) (b) (c)

Figure 2: Constant error model: (a, b) Two tournament trees for max, (c) Comparison tree from [16] after sorting inputs $(2, 5, 7)$.

Max: Consider the $\max$ problem first. In the absence of comparison errors, the max of n items can be found using $n-1$ comparisons along a tournament tree (Figures 2a and 2b). For constant error model, if the probability of error in each comparison is a constant p, it is not hard to see that repeating each comparison in the tournament tree $O(log\frac{n}{\delta})$ times and taking the majority vote to decide the winner will make each comparison right with probability $\leq \frac{\delta}{n}$, and therefore by union bound, the correct max at the root of the tree can be found with probability $\leq \delta$ with total $O(n \log \frac{n}{\delta})$ comparisons.

However, by a simple trick, the better bound of $O(n \log \frac{1}{\delta})$ in Theorem 1 can be obtained: (i) Do comparisons along a balanced tournament tree of height $\log n$; the leaves (in the lowest level) are at level 0, the root is at level $\log n$. (ii) At level $i \in [1, \log n]$, do $i \times O(\log \frac{1}{\delta})$ comparisons for each comparison node. Take majority vote to decide the winner for the higher level. (iii) Output the winner at the root as max. There are $2^{\log n - i} = \frac{n}{2^i}$ internal nodes at level i; therefore the total number of comparisons is $n \times \sum_{\ell=1}^{\log n} \frac{i}{2^i} O(\log \frac{1}{\delta}) = O(n \log \frac{1}{\delta})$. Similarly, by Chernoff bound and union bound[1], and with suitable choices of the constants, the total probability of error that the max item will lose at any of the $\log n$ level is bounded by $\leq \delta$.

Top-k: Next consider the top-k problem and outline the algorithms from [16]. By symmetry, we can assume that $k \leq \frac{n}{2}$, hence we need to show an upper bound of $O(n \log \frac{k}{\delta})$. We consider two cases, (a) when $k \leq \sqrt{n}$, and (b) when $k > \sqrt{n}$.

(a) $k \leq \sqrt{n}$: (i) Build a max-heap using the same algorithm as for max using a tournament tree. However, the error probability is $\frac{\delta}{2k}$, which ensures that the max-heap is consistent with respect to all top-k items with error probability $\leq \frac{\delta}{2}$. (ii) For k times, extract the maximum item from the root, and "reheapify" the max-heap.

[1]Note that the union bound is applied to only the path of the max from the leaves to the root, *i.e.*, to $\log n$ internal nodes and not to $n-1$ internal nodes.

For noiseless comparisons, each of the reheapying steps requires $\log n$ comparisons, *i.e.*, $k \log n$ comparisons in total. By repeating each of the comparisons $O(\log \frac{k \log n}{\delta})$ times, each of the top-k items can be extracted with error probability $\leq \frac{\delta}{2k}$, *i.e.*, with probability $\leq \frac{\delta}{2}$ in total. The total error probability is $\leq \delta$ and the top-k items are output in the sorted order. The number of comparisons in the first step to build the initial heap is $O(n \log \frac{k}{\delta})$. The number of comparisons in the second step to extract the top-k items and to reheapify is $O(k \log n \log(\frac{k \log n}{\delta}))$, which is $O(n \log \frac{k}{\delta})$ for $k \leq \sqrt{n}$.

(b) $k > \sqrt{n}$: We give a sketch of an $O(n \log \frac{n}{\delta})$ algorithm from [16] that can sort n items using $O(n \log \frac{n}{\delta})$ comparisons with probability of error $\leq \delta$, and therefore also solves the top-k problem for $k > \sqrt{n}$ with $O(n \log \frac{k}{\delta})$ comparisons and probability of error $\leq \delta$. The algorithm uses random walk on another form of comparison tree, which are extensions of binary search trees. In a binary search tree, the leaves $x_1 \geq \cdots \geq x_n$ are sorted in order; assuming $x_0 = \infty$ and $x_{n+1} = -\infty$, each leaf $x_i, i \in [1, n+1]$ represents the interval $[x_i, x_{i-1})$. Any internal node with leaves in its subtrees $x_h, x_{h+1}, \cdots, x_\ell$ represents the interval $[x_\ell, x_h)$. When an item x^* is searched on this tree, at each internal node with range $[x_\ell, x_h)$, it is compared with x_z, $z = \lceil \frac{\ell+h}{2} \rceil$. If $x^* \geq x_z$, it is sent to the left child with range $[x_z, x_h)$, otherwise is sent to the right child with range $[x_\ell, x_z)$.

In order to handle noisy comparisons, the comparison tree extends the binary search tree by attaching chains of length $m' = O(\log \frac{n}{\delta})$ to each leaf (in practice they can be implemented as counters from each leaf); each node in a chain has the same interval as that of the corresponding leaf. With noisy comparisons, when an item x^* is searched at a node u with range $[x_\ell, x_h)$ (u is either an internal node or at a chain), first it is checked whether x^* reached at the correct interval, *i.e.*, whether $x^* \geq x_\ell$ and $x^* < x_h$. If these two comparisons succeed, x^* is moved to the correct child of u (its unique child if u is in a chain). Otherwise, it is assumed that some previous step has been wrong, and therefore the search backtracks to the unique parent node of u. By modeling these steps as a Markov process by orienting all edges in the comparison tree toward the right location of x^*. We assume that (without loss of generality), from any node, the probability of transition along the (unique) outgoing edge is $\frac{2}{3}$, and along all the incoming edges together is $\frac{1}{3}$. The search is continued for $m = m_f + m_b < m'$ steps, where m_f, m_b denote the number of forward and backward transitions. When $m = O(\log \frac{n}{\delta})$, $m_f - m_b > \log n$ with high probability. So the final step reaches the correct chain with high probability. If the item does not already appear in the binary search tree, it is inserted. The tree is maintained as a balanced binary search tree by using standard procedures which do not require additional comparisons. By repeating this search followed by insertion procedure for all n items, we get the sorted order in the leaves with $O(n \log \frac{n}{\delta})$ comparisons in total with error probability $\leq \delta$. The matching lower bounds for max and top-k can be found in [16].

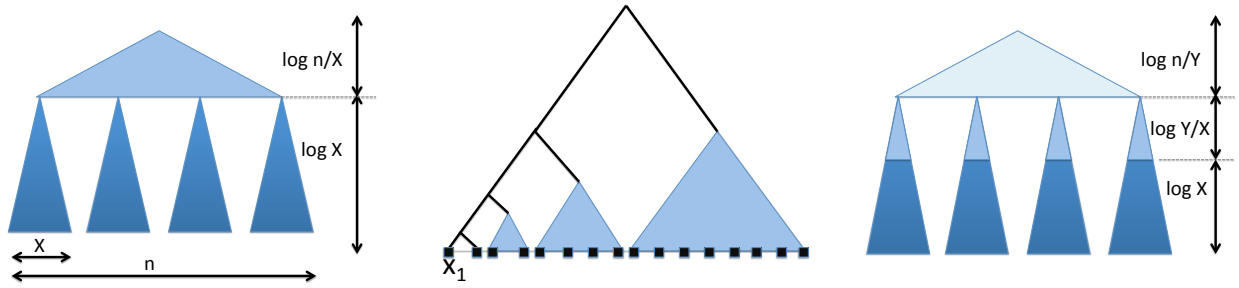
Variable Error Model

Sometimes the comparison error of the noisy oracle may vary with the two input items being considered. As an example, suppose S corresponds to a set of photos of a person, and the items x_i's correspond to the age of the person in the i-th photo. Intuitively, it is much easier to compare the ages in two photos if they are 20 years apart than if they are a few years apart. To model such scenarios Davidson et al. [14, 13] proposed a more general *variable error model*:

$$(\textbf{Variable error model}) \qquad \Pr[\mathcal{O}(x_i > x_j \mid i < j) = false] \leq \frac{1}{f(j-i)}$$

Here $f(x)$ is a *strictly growing error function* that grows with x (*i.e.*, $f = \omega(1)$, *e.g.*, $f(\Delta) = e^{\Delta}$, Δ, $\sqrt{\Delta}$, $\log \Delta$, $\log \log \Delta$, etc.), is strictly monotone ($f(X) > f(Y)$ for $X > Y$), and satisfies $f(1) > 2$ (the probability of error is always $< \frac{1}{2}$).

Theorem 2: Davidson et al. [14, 13]: For all strictly growing error functions f and constant $\delta > 0$, $n + o(n)$ oracle calls are sufficient to output the maximum item x_1 with error probability $\leq \delta$.



(a) (b) (c)

Figure 3: Variable error model: (a) Upper and lower levels for max, (b) Amplified single X-tree containing max (x_1), (c) Extension to top-k

Further, if $f(\Delta) \geq \Delta$, then $n + O(\log \log n)$ oracle calls are sufficient. If $f(\Delta) \geq 2^{\Delta}$, then $n + O(1)$ oracle calls are sufficient.

Max. The basic idea in [14, 13] was to divide a balanced tournament tree for max (*e.g.*, Figure 2a) into lower and upper levels (Figure 3a). The upper levels will only have $\frac{n}{X} = o(n) \times F(\delta)$ items, for a function F. Then we will apply the tournament algorithm from [16] described above to obtain the max with error probability $\leq \delta$ with $O(\frac{n}{X} \log \frac{1}{\delta}) = o(n)$ comparisons for constant δ, provided the max item x_1 survives in the lower levels.

In the lower levels, *only one* comparison is performed at each internal node, resulting in $< n$ comparisons. Surprisingly, assuming variable error model with any strictly growing error function f, the max item x_1 still survives at the lower levels with probability $\leq 2\delta$ for any constant $\delta > 0$. The leaves of the tournament tree are divided into blocks of length X; they form $\frac{n}{X}$ trees of height $\log X$ at the lower levels, called X*-trees*. We only need to focus on the X-tree containing the max item x_1 (Figure 3b), and ensure that x_1 does not lose in any of the $\log X$ comparisons in this X-tree. The key idea is to do a random permutation of the leaves of the tournament tree. This ensures that the probability of x_1 meeting a large item (say x_2 or x_3) in its first comparison is very small, and therefore using the variable error model, x_1 will have a high probability of winning the comparison. As x_1 progresses in the upper levels, it will have a higher chance of meeting larger items, however, with high probability, the larger items will still appear in the other X-trees, and therefore x_1 will eventually progress to the top of its X-tree despite only one comparison being performed at each node in the lower levels. In particular, we bound the probabilities that (i) P_1 = x_1 meets a large item within a range $x_2, \cdots, x_{\Delta_\ell + 1}$ at level ℓ for all $1 \leq \ell \leq \log X$, and (ii) P_2 = x_1 wins the comparison at level ℓ for all $1 \leq \ell \leq \log X$ provided x_1 does not meet $x_2, \cdots, x_{\Delta_\ell + 1}$ at level ℓ. We will show that there are choices of X, Δ_ℓ such that both $P_1, P_2 \leq \delta$ and $\frac{n}{X} = o(n)$. This will give an algorithm to find the max with error probability $\leq 3\delta$ using $n + o(n)$ queries to the oracle, which can easily be extended to an algorithm with tolerance δ by choosing constants appropriately.

Instead of showing the bounds in Theorem 2, we will show a weaker upper bound on the number of oracle calls of $n + O(n^{\frac{1}{2}+\mu})$ for max for error functions that are at least linear ($f(\Delta) \geq \Delta$), but will ensure a better tolerance level of $\frac{3\delta}{n^\mu}$, for all constant $\mu < \frac{1}{2}, \delta$. The arguments will be simpler to achieve these bounds, and we will be able to use these bounds to extend this algorithm for max to an algorithm for top-k under the variable error model. The proof of the bounds in Theorem 2 can be found in [14].

(i) First consider P_1. Since the leaves have a random permutation, the probability that any of $x_2, \cdots, x_{\Delta_\ell}$ belongs to the right subtree of an internal node at height ℓ in the path of the max item x_1, by union bounds, is $\leq \frac{\Delta_\ell 2^{\ell-1}}{n-1}$ (these right subtrees are highlighted in Figure 3b). Therefore, $P_1 \leq \sum_{\ell=1}^{\log X} \frac{\Delta_\ell 2^{\ell-1}}{n-1}$. (ii) Next consider P_2. Assuming x_1 does not meet $x_2, \cdots, x_{\Delta_\ell + 1}$ at level ℓ, x_1 loses the comparison at level ℓ is $\leq \frac{1}{f(\Delta_\ell)}$ (since f is monotone) $\leq \frac{1}{\Delta_\ell}$. Hence $P_2 \leq \sum_{\ell=1}^{\log X} \frac{1}{\Delta_\ell}$.

We need both P_1, P_2 to be small ($\leq \frac{\delta}{n^\mu}$ for constant $\mu < \frac{1}{2}$); further we need $\frac{n}{X} = o(n)$. Note that P_1 has

Δ_ℓ in the numerator and P_2 has Δ_ℓ in the denominator. Therefore, we need to choose Δ_ℓ carefully to meet our requirements. Here we choose $\Delta_\ell = \frac{2^\ell n^\mu}{\delta}$ and $X = \delta n^{\frac{1}{2}-\mu}$. When the required error probability is a constant δ, and when f has a high growth rate (*e.g.*, exponential functions), better bounds can be obtained. With these choices of Δ_ℓ and X, (i) $P_1 \leq \sum_{\ell=1}^{\log X} \frac{\Delta_\ell 2^{\ell-1}}{n-1} \leq \frac{n^\mu}{2\delta(n-1)} \sum_{\ell=1}^{\log X} 4^\ell = \frac{4n^\mu}{2\delta\times 3(n-1)}(4^{\log X}-1) \leq \frac{n^\mu X^2}{\delta n}$ (when $n \geq 3$) $= \frac{\delta^2 \times n^{1-2\mu}}{\delta n^{1-\mu}} = \frac{\delta}{n^\mu}$. (ii) $P_2 \leq \sum_{\ell=1}^{\log X} \frac{1}{\Delta_\ell} = \sum_{\ell=1}^{\log X} \frac{\delta}{n^\mu 2^\ell} \leq \frac{\delta}{n^\mu}$.

The number of nodes in the upper levels is $\frac{n}{X} = \frac{n}{\delta n^{\frac{1}{2}-\mu}} = \frac{n^{\frac{1}{2}+\mu}}{\delta}$, which is $o(n)$ when $\mu < \frac{1}{2}, \delta$ are constants. The number of oracle calls to obtain the max in the upper levels provided it did not lose in the lower levels with error probability $\leq \frac{\delta}{n^\mu}$ is $O(\frac{n}{X}\log(\frac{n^\mu}{\delta})) = O(n^{\frac{1}{2}+\mu}\log n) = o(n)$ when μ, δ are constants. Hence we compute the max at the root of the tournament tree with at most $n + O(n^{\frac{1}{2}+\mu}\log n) = n + o(n)$ oracle calls and error probability $\leq \frac{3\delta}{n^\mu}$, for all constant $\mu < \frac{1}{2}, \delta$.

Top-k. We can extend the above algorithm to obtain an algorithm for top-k under the variable error model when $k \leq n^\mu$, for constant $\mu < \frac{1}{2}$ and $f(\Delta) \geq \Delta$ with $n + o(n)$ oracle calls and error probability $\leq \delta$ for a constant $\delta > 0$. We again use two levels, upper levels (called *super-upper levels* to avoid confusion) with height $\log(\frac{n}{Y})$ nodes, and lower levels (called *super-lower levels*) with height $\log Y$. In the super-upper levels, the top-k algorithm from [16] is run with error probability $\leq \delta$. The super-lower levels are divided into multiple tournament trees, each with $Y = \frac{\delta n}{k^2}$ leaves. The items are again permuted randomly. First we claim that, when $k \leq n^\mu$ for $\mu < \frac{1}{2}$, then all the top-k items will appear in different tournament trees with error probability $\leq \delta$. This follows from union bound: the probability that any of the tournament trees with Y leaves containing one of the top-k items will contain another top-k item is $\leq \frac{Y(k-1)}{n-1} \leq \frac{kY}{n} = \frac{k\delta n}{nk^2} = \frac{\delta}{k}$. Once again, by union bound, all top-k items appear in different tournament trees with probability $\leq \delta$.

Now in the super-lower levels, every top-k item is the maximum item in its respective tournament tree, and we need to return each of them at the root of its tournament tree with error probability $\leq \frac{\delta}{k}$; then by union bound the error probability is $\leq \delta$ for the lower levels. We use the algorithm for max as described above. However, we need a more careful analysis as otherwise the bound would only guarantee the error probability in the individual tournament tree with Y nodes to be $\leq \frac{\delta}{Y^\mu}$ which may be much larger than $\frac{\delta}{k}$.

We divide each tournament tree in the super-lower levels again into upper and lower levels; the lower levels comprise X-trees (Figure 3c). Select $X = \delta Y^{\frac{1}{2}-\mu}$, but $\Delta_\ell = \frac{2^\ell n^\mu}{\delta}$. Since all n items are randomly permuted among the leaves of all tournament trees (instead of only permuting the Y items within each such tree), (i) Using similar calculations as in the case of max, $P_1 \leq \frac{n^\mu X^2}{\delta n} = \frac{X^2}{\delta n^{1-\mu}} = \frac{\delta^2 Y^{1-2\mu}}{\delta n^{1-\mu}} \leq \frac{\delta^2 n^{1-2\mu}}{\delta n^{1-\mu}} = \frac{\delta}{n^\mu}$ which is at most $\frac{\delta}{k}$ when $k \leq n^\mu$. (ii) Using previous calculations, $P_2 \leq \sum_{\ell=1}^{\log X} \frac{1}{\Delta_\ell} \leq \frac{\delta}{n^\mu} \leq \frac{\delta}{k}$.

In the upper levels of each of the tournament trees, that have only $\frac{Y}{X}$ items (Figure 3c), the max can be found with $O(\frac{Y}{X}\log\frac{k}{\delta})$ oracle calls with error probability $\leq \frac{\delta}{k}$. Hence the total error probability that each of top-k items wins in its respective tournament tree is $\leq 3\delta$. Here $X = \delta Y^{\frac{1}{2}-\mu}$. Hence, the total number of comparisons in the super-lower levels is $\frac{n}{Y} \times (Y + O(\frac{Y}{\delta Y^{\frac{1}{2}-\mu}}\log\frac{k}{\delta})) = n + O(\frac{n}{Y^{\frac{1}{2}-\mu}}\log k)$ for constant δ. Note that $Y^{\frac{1}{2}-\mu} = (\frac{\delta n}{k^2})^{\frac{1}{2}-\mu} \geq (\delta n^{1-2\mu})^{\frac{1}{2}-\mu} \geq n^\epsilon$ for a constant $\epsilon > 0$ (since $k \leq n^\mu$ and $\mu < \frac{1}{2}$ is a constant). Therefore, $O(\frac{n}{Y^{\frac{1}{2}-\mu}}\log k) = o(n)$.

In the upper levels, with $\frac{n}{Y}$ nodes, the number of oracle calls to achieve error probability $\leq \delta$ is $O(\frac{n}{Y}\log\frac{k}{\delta}) = O(\frac{k^2}{\delta}\log\frac{k}{\delta})$, which is $o(n)$ for $k < n^\mu$ and constant $\mu < \frac{1}{2}$. Hence with total error probability $\leq 4\delta$, we find all top-k items with $n + o(n)$ oracle calls, for all $k \leq n^\mu$, constant $\mu < \frac{1}{2}$, when the variable error function is at least linear. Further, the top-k items are returned in sorted order (since the top-k algorithms in [16] return them in sorted order).

Summary: Under the constant error model, to achieve the correct answers with tolerance δ, $O(n \log \frac{1}{\delta})$ oracle calls for max and $O(n \log \frac{k}{\delta})$ oracle calls for top-k, $k \leq \frac{n}{2}$, suffice respectively (and these bounds are tight) [16]. A better bound can be obtained under the variable error model, when the error in comparisons depend on the relative distance of the items in the sorted order. Then only $n +$ `smaller order terms` suffice for max for constant tolerance δ, and also for top-k for smaller values of k and when the error functions are at least linear [14, 13].

Discussion: The variable as well as the constant error models assume that, given two distinct items, the oracle will always return the correct answer with probability $> \frac{1}{2}$. However, this may not always hold in crowdsourcing. For instance, if two photos of a person are taken a few minutes (even a few days) apart, it will be almost impossible for any human being to compare their time by looking at them. However, both variable and constant error models cannot support such scenarios. Such an error model has been considered by Ajtai et. al. [4] who assume that, if the values of two items being compared differ by at least Δ for some $\Delta > 0$, then the comparison will be made correctly. When the two items have values that are within Δ, the outcome of the comparison is unpredictable. Of course, it may be impossible to compute the correct maximum under this model for certain inputs. However, the authors show that the maximum can be obtained within a 2Δ-additive error with $O(n^{3/2})$ comparisons and $\Omega(n^{4/3})$ comparisons are necessary (they also generalize these upper and lower bounds). In contrast, even under the constant error model, the correct maximum can be computed with high probability with $O(n)$ comparisons.

Open question: What are the upper and lower bounds on the oracle complexity and guarantee on the accuracy level if we combine the constant/variable error model with the model in [4], where the answers can be arbitrary if the two items being compared have very similar values?

2.2 Concave Cost Functions

We have used the fixed-cost model so far in which each question incurs unit cost, resulting in a total cost which is the number of comparisons performed. However, in some scenarios, many questions are asked together, *i.e.*, multiple tasks are grouped in *batches* before they are assigned to a worker[2]. In these cases, the total payment for all these questions in a batch can be less than the payment when the questions are asked one by one, since it is likely to require less effort from the worker (for instance, the worker does not need to submit an answer before getting the next question and potentially can choose the answers faster). Such cost functions can be modeled as non-negative monotone *concave functions*[14], *i.e.*, (**Concave functions**) *for all* $t \in [0, 1], N_1, N_2$,

$$g(tN_1 + (1-t)N_2) \geq tg(N_1) + (1-t)g(N_2).$$

Examples include $g(N) = N^a$ (for a constant a, $0 < a \leq 1$), $\log N, \log^2 N, \log \log N$, $2^{\sqrt{\log N}}$, etc. Non-negative concave functions are interesting since they exhibit the *sub-additive property*:

$$g(N_1) + g(N_2) \geq g(N_1 + N_2).$$

Concave cost functions display a tension between the number of rounds and the total number of questions asked in an algorithm since (1) it is better to ask many questions in the same batch of the oracle in the same round, rather than distributing them to many different batches and multiple rounds; but (2) the cost function is monotone, so we cannot ask too many questions as well. As an extreme (and non-realistic) example, for finding the maximum item under the constant concave cost function $g(N) = a$, $a > 0$, we could ask all $\binom{n}{2}$ comparisons of the oracle in the same batch and compute the maximum; however, this is not a good strategy for a linear cost function. We assume that the cost function g is known in advance, and the goal is to develop algorithms for arbitrary concave cost functions that aim to minimize the total incurred cost.

[2] *Batches* denote grouping comparison tasks before assigning them to a worker, while after each *round* the machine gets backs the answers from the workers in the batches of that round.

If the oracle receives a set with N comparisons in the same batch, we incur a cost of $g(N)$ for these N comparisons. We still assume that the answers to two different comparisons (where at least one item is different) are independent even in the same batch. However, we cannot ask the oracle to compare the same pair of items in the same batch when we need redundancy for erroneous answer (*i.e.*, the same worker cannot be asked to compare the same two items twice). In this case, multiple comparisons of the same pair of items must go to different batches, and therefore will incur additional cost. We also assume that there is no limit on the maximum size of the batch of questions that can be asked of the oracle at once. Unlike the fixed-cost model where it sufficed to minimize the total number of comparisons, now our goal is to optimize the total cost under g.

Algorithms. Assuming no comparison errors, [14] gives a simple $O(\log\log n)$-approximation algorithm[3] to find the max item using, once again, a tournament tree (a similar algorithm is given in [19] for finding max in $O(\log\log n)$ rounds with $O(n)$ comparisons in total). However, unlike the standard tournament tree, at any level h of the tree with B_h nodes, all possible $\binom{B_h}{2}$ comparisons are performed. In addition, the internal nodes at level h have 2^h children from the lower level $h-1$. It is shown that, the number of levels of the tree is $\log\log n$, and at any level of the tree, at most n comparisons are performed incurring cost $g(n)$, giving an algorithm with cost $O(g(n)\log\log n)$. Since the cost of the optimal algorithm has a lower bound $OPT \geq g(n-1) \geq g(n) - g(1)$. Assuming $g(1)$ to be a constant, this gives an $O(\log\log n)$-approximation algorithm. The same algorithm can be extended for the constant error model by repeating the comparisons multiple times in different batches of calls to the oracle[4]. Two other algorithms for finding max in $O(\log\log n)$ rounds are also mentioned in [14] that can be extended to give the same approximation: by Valiant [41] and by Pippenger [36]; however, these algorithms are more complex to implement for practical purposes. Nevertheless, the algorithm based on Pippenger's approach can be extended to a randomized algorithm for top-k under the concave cost function and no comparison error, with an expected cost of OPT $\times\, O(\log\log n)$.

Summary: Assuming no comparison error, we get an $O(\log\log n)$-approximation for max under any concave cost function g. For the constant error model, the same algorithm with repeated comparisons gives an $O(\log n)$-approximation for any concave cost function g, and $O(\log\log n)$-approximation for $g(n) = n^\alpha$, where $\alpha \in (0, 1]$ is a constant. Using standard and known techniques, an $O(\log\log n)$-approximation can be achieved for top-k and no error (however, not simple to implement for practical purposes).

Discussions: For unit cost function, assigning tasks to single or multiple workers amounts the same total cost, which is not the case for concave cost functions. In some practical crowd-sourced applications, it may be useful to assume an upper bound B on the batch size, as workers may not be able to answer questions in a very large batch without error due to fatigue or time constraint. Further, the independence assumption is even less likely to hold when multiple questions are asked of the oracle in the same batch. Finding a practical model when tasks are grouped into batches is a direction to explore.

Open questions: (1) Assuming no comparison errors, can we devise algorithms for concave cost functions that have better bounds than the algorithms in [14] (for max), and are simpler to implement (for top-k)? What can we infer about optimal algorithms? Can we have a better lower bounds for OPT than $g(n-1)$ for all or some concave cost functions g?
(2) The bounds for the constant error model have been obtained in [14] under concave cost functions simply by repeating individual comparisons by different workers and taking the majority vote. How can we obtain algorithms with better bounds for constant and variable error models under arbitrary concave cost functions?

[3] An algorithm is a $\mu(n)$-*approximation algorithm* for some non-decreasing function μ, if for every input of size n it can find the solution (*e.g.*, the maximum item) with a cost $\leq \mu(n) \times$ OPT.

[4] For constant δ, $O(\log\log n)$-approximation when $g(n) = n^\alpha$, α is a constant $\in (0, 1]$, and $O(\log n)$-approximation for arbitrary concave function g

3 Skyline queries

Skylines. Let S be a set of n items. We assume these items admit a full (but not necessarily strict) order $\leq_i$ along d dimensions $i \in \{1, \ldots, d\}$. We also write $v \preccurlyeq v'$ to denote that $v \leq_i v'$ for each $i \leq d$. When $v \preceq v'$ and there is some $i \leq d$ such that $v <_i v'$, we say that v' *dominates* v, which we denote by $v \prec v'$.

Given a set of d-dimensional items S, the *skyline* of S is the set of items that are not dominated (we assume that two items can not coincide):

$$\text{Sky}(S) = \{v \in S \mid \forall v' \in S \setminus \{v\}, \exists i \leq d.\ v >_i v'\}.$$

Input: A set of items S; tolerance δ; noisy comparison oracle $\mathcal{O}$. **Objective:** Return $\text{Sky}(S)$ with error probability δ.

In our example of Figure **??**, the skyline consists of locations $\{a, b, c, d, e\}$, whereas location h, for instance, is dominated by location a.

Algorithms

Skyline queries must identify the items that are not dominated. The classical skyline algorithms are usually based on one of two paradigms (possibly combined): divide and conquer, or sweeping line approaches. In the divide and conquer approach, input items are split around the median on one of the dimensions, then the skyline of both halves are computed. After the splitting, all items from the smaller half that are dominated by some larger item are pruned out. Finally the union of the two partial results is returned. One major stumbling block with this divide and conquer approach for skylines in presence of noise seems to be the issue of splitting items around the median; with noisy comparisons, identifying exactly the set of items which are larger than the median is as expensive as a full sort of the input [16]. Because of this it seems unlikely that the classical divide and conquer skyline algorithms [23, 24] can be adapted into efficient algorithms in presence of noisy comparisons, though more intricate schemes as proposed for parallel sort with noise [28] might still provide efficient algorithms with noisy comparisons.

In contrast, the sweeping line approach adapts easily to noisy comparisons. To compute iteratively the skyline, each iteration adds to the result the maximal item for lexicographic order (the maximum on $<_1$, if there are ties we take the maximum for $<_2$ among those, etc.) among the items that are not dominated. The algorithm stops after k iterations.

We first present efficient procedures to check if a given item is dominated. Given an item v and a set C of items, we say that C dominates v if there is $v' \in C$ such that $v \prec v'$, i.e., if $\bigvee_{v' \in C} \bigwedge_{j \leq d} v <_j v'$. The formula expressing domination is thus a composition of boolean functions involving basic comparisons, which can be computed in $O(d|C| \log \frac{1}{\delta})$ [20, 31]. Another approach to check dominance consists in sorting the set of items C along each dimension, then using binary search to obtain the relative position v within C on each dimension. The complexity of sorting C is $O(d|C| \log \frac{d|C|}{\delta})$. Once C is sorted, each item v can be inserted along each dimension with error probability δ/d in $O(\log \frac{d|C|}{\delta})$.

We next observe that classical algorithms for MAX with noisy comparisons (and similarly with boolean formulae) are what we call "trust-preserving". This means that these algorithms return in $O(n)$ the correct output with (at most) the same error probability as the input oracle. Our line-sweeping algorithms thus compute one skyline point per iteration, using any of the two dominance-checking procedures to compute the maximal item among those that are not dominated. The algorithm makes sure the probability of error per iteration is at most δ/k. The value of k is not known in advance but the error requirements are met by adapting a classical

trick in output sensitive algorithms [9] and proceeding with maximal error $\delta/2^i$ as soon as 2^{2^i} items have been discovered.

As a result, efficient upper bounds can be obtained for skyline computation:

Theorem 3: Groz and Milo [20]: Sky(S) can be computed with any of the following complexities:

1. $O(dn \log(dn/\delta))$
2. $O(dk^2 n \log(k/\delta))$
3. $O(dkn \log(dk/\delta))$

The first result corresponds to sorting all input items on every dimension. The second and third compute the skyline iteratively; each iteration adding the lexicographically maximal item among those that are not dominated. The second uses the boolean formula approach to check dominance whereas the third sorts the skyline item as they are discovered and then uses binary insertion to check dominance. The first and third results only count the number of calls to the comparison oracles executed, so that computational complexity may be higher: sorting the input provides all the ordering information necessary to compute the skyline, but a classical algorithm to compute the skyline (without noise) must still be applied before a skyline can be returned based on these orderings. The exact complexity of skylines in this model remains open since the only lower bound we are aware of is $\Omega(n \log(k/\delta) + dn)$. But even without noise an optimal bound of $\Theta(n \log k)$ is only known when d is assumed to be constant.

Rounds. The number of rounds required by those algorithms are respectively $O(\log n)$, $O(k \cdot \log\log n \cdot \log^*(d) \cdot \log^*(k))$, and $O(k \cdot \log\log n \cdot \log k)$. Two interesting patterns can be observed. First the number of rounds does not depend on δ: the additional oracle calls that must be performed to gain accuracy can always be processed in parallel. Second, none of the three algorithms proposed so far for skyline computation stands out in terms of rounds: each can outperform the others depending on input parameters. Last, we recall from [41] that $\log\log n$ rounds are required to compute the maximum of n items with $O(n)$ comparisons even when comparisons are accurate, therefore it is not surprising that all of our algorithms require at least $\log\log n$ rounds even for small values of k.

Summary: [20] presents several algorithms to compute skylines under the constant error model. The complexity of these algorithms is dn multiplied (depending on the algorithm) by k or k^2, and logarithms in d, n, or $1/\delta$. Each algorithm may outperform the others for some values of the input parameters both in terms of the cost and the number of rounds.

Discussion: Skylines are generally harder than sorting problems, since even in the case of two dimensions the comparisons required to check if the skyline contains all items necessarily provide a complete ordering of all items. Therefore one cannot hope to achieve drastic improvements in the constant factors as we showed for Max and Top-k, nevertheless one could reasonably hope that variable error models could benefit the algorithms in a sense that remains to be defined.

Open questions: (1) Can the bounds of the skyline problem be improved assuming other error models (*e.g.*, the variable error model)?
(2) How many comparisons are required to compute the skyline when neither d nor k are assumed to be constant? This question is already open without noise in comparisons [10]. To what extend do noisy comparisons raise the complexity?
(3) Multiple variants of skyline problems such as layers of skylines or approximated skylines have been considered in standard data models without noise. It could be interesting to investigate how such problems can be tackled by the Crowd. The approximation of skylines, in particular, can be justified by the observation that the skyline cardinality tends to increase sharply with the number of criteria considered, on random instances.

4 Related work

In the past few years, crowdsourcing has emerged as a topic of interest in the database and other research communities in Computer Science. Here we mention some of the relevant work in crowd sourcing, and computation with noisy operations in general.

Crowdsourcing database/data mining operators. Different crowd-sourced databases like CrowdDB [17], Deco [32], Qurk [29], AskIt![6] help users decide which questions should be asked of the crowd, taking into account various requirements such as latency, monetary cost, quality, etc. On the other hand, several papers focus on a specific operator, like joins [44], group-by or clustering[18, 46], entity resolution [43, 45], filters [34, 33], frequent patterns identification [5], centroid [22], and investigate techniques to optimize the execution of this operator in a crowdsourcing environment.

Max, top-k, and ranking. Multiple approaches have been proposed to sort or search the maximal items using the crowd. The comparative merits of comparisons and ratings have thus been evaluated experimentally in the Qurk system [29], as well as the effect of implementation parameters such as batch size. Ranking features were similarly proposed in the CrowdDB system [17]. Heuristic to compute the maximal item with the Crowd have been introduced in [42]. In their model, the Crowd must return the largest item in the batch, and the probability of returning each item depends on its rank in the batch. This line of work thus also allows to model distance-dependent error, though not in a comparison framework. A classical issue in voting theory is rank aggregation: given a set of ranking, one must compute the "best" consensus ranking. This is the approach for the *judgment problem* studied in [21] (given a set of comparison results, which item has the maximum likelihood of being the maximum item) and the next vote problem (given a set of results, which future comparisons will be most effective). They propose hardness results and heuristics to compute the most likely maximum. Their model assumes a comparison oracle with constant error probability, and the input consists in a matrix summarizing the results of oracle queries that have already been performed. Beyond the computation of the most likely maximum they also consider the problem of best using a fixed budget of additional questions that should be issued to complement the vote matrix. In [37], the authors consider the crowdsourced top-k problem when the unit of task (for time and cost) is not comparing two items, but ranking any number of items. The error model is captured by potentially different rankings of the same items by different people. If many people disagree on the ranking of certain items, more people are asked to rank these items to resolve the conflict in future rounds. A related problem studied in the machine learning community is that of learning to rank in information retrieval (*e.g.*, [25, 8, 38]). Our noisy comparison model with constant error probability appears with some refinements in [16, 19, 31, 28], while other models have also been considered for ranking problems in [35] (the total number of erroneous answers in bounded) and [4] (comparisons are arbitrary if two items are too close).

Skyline. While Pareto-optimality has received ample attention in machine learning, algorithm [23, 10] and traditional database [7] communities, fewer uncertain data models have been considered so far for skylines in the database community. One popular model considers inputs given by a (discrete) probability distribution on the location of items in space [2]. Heuristic approaches have been considered over incomplete or imprecise data [26, 27]. Beside [20], the parallel computation of skylines has only been considered in the absence of noise [1].

Number of query rounds and beyond worst case. The number of rounds required by fault-tolerant sorting and searching algorithms has been a topical issue for a long time [16], in parts driven by parallelism in circuits. Efficient Sorting [28] and Maxima [19] algorithms with respectively $\log n$ and $\log \log n$ rounds have thus been proposed, matching the theoretical lower bounds. Whether in our error model or others, worst-case complexity may appear too conservative a measure to derive efficient crowdsourcing algorithms, so it may be interesting to look at average complexity or instance-optimality, which have been considered for top-k [15] and skyline [3] problems in other models.

5 Discussions and Future Work

The crowd has become a valuable resource today to answer queries on which the computers lack sufficient expertise. This article gives an overview of known theoretical bounds under different cost and error models for max/top-k and skyline queries. These results assume a simple probabilistic model for the crowd as a noisy oracle performing pairwise comparisons between elements. This simple model for the crowd lays the foundation for several interesting future research directions for practical purposes:

- The actual error, cost, and latency models for the crowd, as mentioned earlier, are hard to formalize and need to be substantiated with real experiments with crowd that model their behavior in different scenarios.
- None of the algorithms in this article considers the actual number of crowd workers who are involved in the tasks, as long as there are sufficient number of crowd workers for redundant comparisons. One can study the dependency on the cost. accuracy, and latency of the algorithms for different numbers of crowd workers involved, and variations of the independence assumptions and uniformity of error models across all workers.
- Even assuming the model is a reasonable approximation of the crowd's behavior, the algorithms must know the parameters (tolerance δ, error function f) precisely. Estimating those parameters accurately with low cost may require additional experimental study.
- The relevance of asymptotic estimations provides some hindsight on the cost one may expect to achieve, but may not accurately capture the cost for practical purposes where either more comparisons may be needed or fewer comparisons may suffice for reasonable results. Similarly, some easier comparisons can be performed by machines to save on cost and time. A rigorous analysis of human behaviors with experiments on crowd sourcing platforms, and adapting the models and algorithms accordingly, will be an important and challenging research direction.

References

[1] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012.

[2] P. Afshani, P. K. Agarwal, L. Arge, K. G. Larsen, and J. M. Phillips. (approximate) uncertain skylines. *Theory Comput. Syst.*, 52(3):342–366, 2013.

[3] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. In *FOCS*, pages 129–138, 2009.

[4] M. Ajtai, V. Feldman, A. Hassidim, and J. Nelson. Sorting and selection with imprecise comparisons. In *ICALP (1)*, pages 37–48, 2009.

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

[6] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, and W.-C. Tan. Asking the right questions in crowd data sourcing. *ICDE*, 0:1261–1264, 2012.

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

[8] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *ICML*, ICML '05, pages 89–96, 2005.

[9] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4):361–368, 1996.

[10] T. M. Chan and P. Lee. On constant factors in comparison-based geometric algorithms and data structures. In *SOCG*, page 40, 2014.

[11] ClickWorker. In `http://www.clickworker.com`.

[12] CrowdFlower. In `http://www.crowdflower.com`.

[13] 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.

[14] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Top-k and clustering with noisy comparisons. *ACM Trans. Database Syst. (TODS)*, 39(4):35:1–35:39, 2014.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[16] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, Oct. 1994.

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

[18] R. Gomes, P. Welinder, A. Krause, and P. Perona. Crowdclustering. In *NIPS*, pages 558–566, 2011.

[19] N. Goyal and M. Saks. Rounds vs. queries tradeoff in noisy computation. *Theory of Computing*, 6(1):113–134, 2010.

[20] B. Groz and T. Milo. Skyline queries with noisy comparisons. In *PODS*, pages 185–198, 2015.

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

[22] H. Heikinheimo and A. Ukkonen. The crowd-median algorithm. In *HCOMP*, 2013.

[23] D. G. Kirkpatrick and R. Seidel. Output-size sensitive algorithms for finding maximal vectors. In *SOCG*, pages 89–96, 1985.

[24] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[25] T.-Y. Liu. Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, Mar. 2009.

[26] C. Lofi, K. E. Maarry, and W.-T. Balke. Skyline queries in crowd-enabled databases. In *EDBT*, pages 465–476, 2013.

[27] C. Lofi, K. E. Maarry, and W.-T. Balke. Skyline queries over incomplete data - error models for focused crowd-sourcing. In *ER*, pages 298–312, 2013.

[28] Y. Ma. An $o(n \log n)$-size fault-tolerant sorting network (extended abstract). In *STOC*, pages 266–275, 1996.

[29] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller. Twitinfo: aggregating and visualizing microblogs for event exploration. In *CHI*, pages 227–236, 2011.

[30] MicroWorkers. In https://microworkers.com.

[31] I. Newman. Computing in fault tolerant broadcast networks and noisy decision trees. *Random Struct. Algorithms*, 34(4):478–501, 2009.

[32] A. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: Declarative crowdsourcing. Technical report, Stanford University.

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

[34] 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.

[35] A. Pelc. Searching games with errors - fifty years of coping with liars. *Theor. Comput. Sci.*, 270(1-2):71–109, 2002.

[36] N. Pippenger. Sorting and selecting in rounds. *SIAM J. Comput.*, 16(6):1032–1038, Dec. 1987.

[37] V. Polychronopoulos, L. de Alfaro, J. Davis, H. Garcia-Molina, and N. Polyzotis. Human-powered top-k lists. In *WebDB*, pages 25–30, 2013.

[38] F. Radlinski and T. Joachims. Active exploration for learning rankings from clickthrough data. In *SIGKDD*, KDD '07, pages 570–579, New York, NY, USA, 2007. ACM.

[39] SamaSource. In http://www.samasource.org.

[40] A. M. Turk. In https://www.mturk.com/.

[41] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4(3):348–355, 1975.

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

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

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

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

[46] J. Yi, R. Jin, A. K. Jain, S. Jain, and T. Yang. Semi-crowdsourced clustering: Generalizing crowd labeling by robust distance metric learning. In *NIPS*, pages 1781–1789, 2012.