Top-k and Clustering with Noisy Comparisons

SUSAN DAVIDSON and SANJEEV KHANNA, University of Pennsylvania
TOVA MILO, Tel Aviv University
SUDEEPA ROY, University of Washington

We study the problems of max/top-k and clustering when the comparison operations may be performed by oracles whose answer may be erroneous. Comparisons may either be of *type* or of *value*: given two data elements, the answer to a type comparison is "yes" if the elements have the same type and therefore belong to the same group (cluster); the answer to a value comparison orders the two data elements. We give efficient algorithms that are guaranteed to achieve correct results with high probability, analyze the cost of these algorithms in terms of the total number of comparisons (i.e., using a fixed-cost model), and show that they are essentially the best possible. We also show that fewer comparisons are needed when values and types are correlated, or when the error model is one in which the error decreases as the distance between the two elements in the sorted order increases. Finally, we examine another important class of cost functions, concave functions, which balances the number of rounds of interaction with the oracle with the number of questions asked of the oracle. Results of this article form an important first step in providing a formal basis for max/top-k and clustering queries in crowdsourcing applications, that is, when the oracle is implemented using the crowd. We explain what simplifying assumptions are made in the analysis, what results carry to a generalized crowdsourcing setting, and what extensions are required to support a full-fledged model.

Categories and Subject Descriptors: H.1.2 [**Information Systems**]: User/Machine Systems—*Human information processing*; H.2.m [**Database Management**]: Miscellaneous

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Top-k, clustering, algorithm, approximation, crowdsourcing

ACM Reference Format:
Susan Davidson, Sanjeev Khanna, Tova Milo, and Sudeepa Roy. 2014. Top-k and clustering with noisy comparisons. ACM Trans. Datab. Syst. 39, 4, Article 35 (December 2014), 39 pages.
DOI: http://dx.doi.org/10.1145/2684066

1. INTRODUCTION

Max/top-k and clustering/group-by are fundamental operations for database queries. The implementation of these operations involves *value* comparisons ("Is $a > b$?") for max/top-k, and *type* comparisons ("Are a and b of the same type?" e.g., belong to the same group) for clustering. When the data elements being compared are simple, for example, integers or strings, and when the clustering is based on the value of a field of a tuple,

---

This is an extended version of the paper titled "Using the Crowd for Top-k and Group-by Queries" that appeared in the International Conference on Database Theory (ICDT) 2013.
This work was supported in part by NSF grants IIS-0803524, IIS-0911036, IIS-1115188, IIS-1302212, European Research Council under the FP7, ERC grant MoDaS, agreement 291071, and by the Israel Ministry of Science.
Authors' addresses: S. Davidson and S. Khanna, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104; T. Milo, Department of Computer Science, Tel Aviv University, Israel 6997801; S. Roy (corresponding author), Department of Computer Science and Engineering, University of Washington, Seattle, WA; email: sudeepa@cs.washington.edu.

0362-5915/2014/12-ART35 $15.00

DOI: http://dx.doi.org/10.1145/2684066

as in SQL `group-by` queries, one may assume the comparisons are performed correctly. However, when the data elements are complex, such as an image, the comparisons are less straightforward and may therefore result in errors.

As an example, consider an application in which we have a database of unlabeled photos, `PhotoDB`, where each photo focuses on a single person (e.g., Alice standing in front of Niagara Falls or Bob in front of the Louvre, but not of Alice and Bob standing together). Using `PhotoDB`, we now wish to: (1) group (cluster) the photos by the person they represent; and (2) find the most recent photo of each person or, alternatively, the five most recent photos of each person (max/top-k). Using an SQL-like syntax and assuming that `PhotoDB` has a single attribute `photo`, this query could be represented as follows.

```
SELECT Most-recent(photo)
FROM PhotoDB
GROUP BY Person(photo)
```

Since the database does not have attributes representing the person in the photo or the date that the photo was taken, the user-defined function `Person` groups photos of the same person, and the aggregate function `Most-recent` selects the single photo that is the most recent one within the group.[1] The `Person` function, which recognizes when two photos are of the same person, could be possibly performed by photo processing software; however, it may result in errors, especially if the pictures are spread over a time span of 20 years during which the person ages from childhood to an adult. The `Most-recent` function could be calculated using the date that a photo was taken as captured by the camera, but again may not be completely trustworthy if the photos are contributed by several different people who have different (or no) date/time settings in their camera.

Crowdsourcing is also increasingly being used to implement user-defined functions such as `Person` and `Most-recent` [Franklin et al. 2011; Marcus et al. 2011b]. In particular, the crowd has been used to obtain high-quality labeled image or text datasets for clustering or classification solutions in computer vision, natural language processing, or social media analysis [Li and Perona 2005; André et al. 2014, 2012; Rashtchian et al. 2010; Fernández and Gómez 2008] (e.g., classifying photos as forests, mountains, streets, kitchen, office, etc.), or to create sessions of coherent research papers in a conference by the *community clustering* approach [André et al. 2013]. The crowd has also been used to find the top photos or captions that match a given concept, top candidates for a given position, or the best Facebook profile that matches a given person [Rastegari et al. 2011; Venetis et al. 2012; Polychronopoulos et al. 2013].

In our application, whether photo processing software, the crowd, or some combination of the two is used, the user-defined functions will result in errors. We therefore model such functions as *oracles* whose answers may be erroneous, and which are used by the system in one or more rounds of interaction; the system may ask multiple oracles the same question to reduce the error probability, and may decide which questions to ask in the next round based on the answers obtained in the previous one. Given two data elements, the answer to a *type* question is "yes" if the elements have the same type and therefore belong to the same group or cluster; the answer to a *value* question orders the two data elements. The assumption here is that there is an underlying ground truth but that the oracle may make mistakes, that is, may not correctly identify two pictures as being of the same person (type error) or of one picture of a person being

[1]This is an abuse of SQL `GROUP BY` notation but operates on a similar principle: an "equivalence" function is being used to group tuples, and a single value is calculated over the nongrouped attributes.

Table I.
Summary of our results for constant $\epsilon, \delta > 0$ and the dependence on ϵ, δ can be found in the referenced theorems.

Problem	Inputs	Assumptions	Results
Max/Top-k	n elements, k, error function f	f is strictly growing and (a lower bound on) f is known	Max: $n + o(n)$ (Theorem 4.2) Top-k: $n + o(nk) + O(k^2)$ (Corollary 4.8)
Clustering	n elements (number of clusters J is not needed)		Upper bound: $O(nJ \log n)$ Lower bound: $\Omega(nJ)$ (Theorem 5.1)
Clustering with correlated type and value	n elements, α (ref. Section 3.4)	(an upper bound on) α is known	Upper bound: $O(\alpha J + n \log(\alpha J))$ (Theorem 6.1)
Max with concave cost function	n elements		No error (also for top-k): $O(\log \log n)$-approximation Constant error: $O(\log n)$-approximation (Theorems 7.2 and 7.6)

more recent than another picture of this same person (value error). Our algorithms must therefore be designed to take this into account.

The standard approach to model this error is to assume some fixed probability of getting an incorrect answer. In particular, one can assume that each type or value comparison is answered correctly with a constant probability $> \frac{1}{2}$, which reflects the assumption that the oracle is always better than a random yes/no answer; we call this the *constant error model*. For value comparisons, we propose a more interesting error model motivated by human behavior which we call the *variable error model*. Here the error is related to how close the elements are in the ordering of interest. For example, if a value question involves two pictures of the same person, one from her high-school graduation and the other her retirement, it is easy for humans to decide which is the most recent. However, if the value question involves two pictures of the same person separated by a year or two, it is much harder to decide which is the most recent.

The oracle may also have an associated *cost* per question. For example, if crowdsourcing is used, this could be the cost per Human Interaction Task (HIT), as in Amazon Mechanical Turk; more generally, it could be the amortized cost of the software used (e.g., complex image processing software or a crowdsourcing platform). To model this cost, the standard approach is to assign a *fixed* cost per question, that is, asking one question has a cost of 1 and asking N questions has a cost of N, regardless of who/what the questions are asked of, or what type the questions. There are also other, more complex cost functions that are especially relevant for crowdsourcing, in particular *concave* cost functions in which asking many questions at once of the oracle is cheaper than asking the questions individually. In other words, the cost of asking N questions followed by M questions is more expensive than asking $N + M$ questions at once.

Using these error and cost models, we formalize the max/top-k and clustering problems when the comparison operations are performed by an oracle whose answers may be erroneous, and give efficient algorithms that are guaranteed to achieve the desired results with high probability. Our results are summarized as follows (also shown in Table I).

Max/top-k. Finding max/top-k under the constant error (and fixed cost) model has been thoroughly studied. Feige et al. [1994] show that, when each value question is

answered correctly with probability $\geq \frac{1}{2}+\epsilon$ for a constant ϵ, there is a simple algorithm to find the maximum with probability $\geq 1-\delta$ using $O(n \log \frac{1}{\delta})$ questions; they also show this bound is tight. We review this algorithm in Section 4 and go on to show that, using our novel *variable* error model, a much better upper bound on the cost can be obtained: Suppose the two elements being compared by a value question are Δ apart in the sorted order on values. Then the probability of error is $\leq \frac{1}{f(\Delta)}$ for a monotone nonnegative error function f with superconstant growth rate (called a *strictly growing function*; refer to Section 3.3), that is, the error in the answer decreases when the distance Δ between the elements increases. For max and top-k (with small values of k), we show that $n + o(n)$ value questions are sufficient to find the answers with high probability given *any* strictly monotone error function f (Theorem 4.2). Here $o(n)$ denotes a function of n that is strictly asymptotically smaller than the linear function.

Clustering. For the general clustering problem using the fixed-cost model, we give a lower bound of $\Omega(nJ)$, where J is the number of clusters. This bound holds even for: (1) randomized algorithms; (2) when the answers to the type questions are always correct; and (3) the value of J is known by the algorithm (Theorem 5.1). We show this bound is essentially the best possible by providing a simple algorithm that compares $O(nJ)$ pairs of elements by type questions; if the answers to the type questions are *erroneous*, the number of questions increases by a factor of $O(\log n)$ to output the clusters with high probability. Our algorithm does not require any a priori knowledge of J.

Correlated type and value. There are also scenarios where type and value questions are used together, such as in the example we having been using so far where we want to find max/top-k elements from each cluster of individual photos. A simple solution is to first find the clusters by running an algorithm for the group-by queries (type questions), and then find the top answers in each cluster by running an algorithm for top-k queries (value questions). However, this solution can be improved in cases where the types and values are *correlated*, either fully or partially, using the correlation to reduce the total number of questions asked. Full correlations trivially exist when the elements are assigned their types by partitioning the sorted order according to their values. If we restrict our `PhotoDB` example to the photos of a single person sorted according the dates they were taken (value), the photos can be partitioned into different groups like *young child* (0–9), *preteenager* (10–12), *teenager* (13–19), *twentysomething* (20–29), *thirtysomething* (30–39), and so on. As another example, suppose that we have a database of hotels in a city where the hotels are sorted according to their average price (value) and are assigned a price category (type) for example, *high*, *middle*, *low*, partitioning the sorted order. On the other hand, there can be partial correlation between values and types. If the hotels were to be clustered by districts or neighborhood as done in Web sites like hotels.com or booking.com, then there would be a high correlation between the district and the average price of a hotel. For instance, in most cities, downtown hotels are on average more expensive than hotels in the outskirts. Here a correlation exists between the neighborhood (type) of a hotel and its average price (value). A similar partial correlation is expected to exist when the type of a hotel in a given district is its quality (e.g., star ratings) and the value is its average price (which may depend on other factors like location).

We formalize the scenario when the types and values are correlated, and show that $O(n \log J)$ type and value questions in total (with additional logarithmic factors for erroneous answers) are sufficient when elements of the same type form contiguous blocks in the sorted order (called the *full correlation case*). In the *partial correlation case*, there are at most $\alpha - 1$ changes in type between any two elements of the same type in the sorted order by value. We show that in this case $O(\alpha J + n \log(\alpha J))$ questions

(with logarithmic factors) suffice (Theorem 6.1). We also discuss the problem of finding the max/top-k elements from each cluster.

Concave cost functions. Finally, we studied a general class of cost models based on *concave functions*. Nonnegative concave cost functions are important since they exhibit subadditive behavior: asking the oracle many questions at once is cheaper than asking the questions individually. Finding an optimal algorithm for an arbitrary cost function turns out to be nontrivial, even if there are no comparison errors, that is, the oracle returns the correct answer for each comparison. We therefore study approximation algorithms that return the exact answer by paying a total cost not much worse than optimal and prove that, for *any* monotone nonnegative concave function, there exists an $O(\log \log n)$-approximation algorithm to find the max when there are no comparison errors. We then discuss how these algorithms can be extended to find the top-k, and close by discussing how to extend the algorithms to the constant error model.

The primary motivation for this work is its application to crowdsourcing. In particular, the variable error model is motivated by how humans seem to behave, as in the scenario involving correlated type and value; the use of concave cost functions is also particularly relevant for crowdsourcing applications. The preceding results therefore form an important first step in providing a formal basis for max/top-k and clustering queries when the oracle is implemented using the crowd. However, we make several simplifying assumptions that are not completely realistic. For example, error is not constant among users as some will be better at performing particular tasks than others, and there may be spammers. There may also not be a total order on data elements, or the ordering may not be detectable when elements are extremely close, for example, if the time between two pictures of the same person is a matter of minutes or hours, no-one would be able to tell which is the most recent. Furthermore, when operations are grouped together in a single human task, the answers are no longer independent. There may also be a bound on the number of operations that a crowd member is willing to perform in a single human task no matter how high the reward, for instance, for lack of available time. In general, the crowd is very hard to model precisely and algorithmic results in this setting should be tested by experiments. Nevertheless, results in this article provide an important formal foundation that can be used as a basis for investigating the more general real-life model. For example, the lower bounds naturally carry over to the generalized model and can serve as a yardstick on what could be expected from the performance of such algorithms; and the principles employed in the upper bound algorithms can serve as basis for appropriate extensions in practical algorithms for generalized crowdsourcing models.

Roadmap. After reviewing related work in Section 2, we present the model and definitions that will be used throughout the article in Section 3. Results on max and top-k (*value* comparisons) for the variable error model are given in Section 4, while results on clustering (*type* comparisons) are given in Section 5. In Section 6, we consider the case when values and types are correlated, and give an improved clustering algorithm using both type and value questions. Results for concave cost functions are presented in Section 7. Finally, we conclude in Section 8 with a detailed discussion of the simplifying assumptions that we make, and directions for extensions and future work.

2. RELATED WORK

Crowdsourcing is a topic of recent interest to the database community and has been suggested as a method for data cleansing, data integration, entity resolution, schema expansion, and data analytics; see, for example, Guo et al. [2012], Selke et al. [2012], Wang et al. [2012], Baharad et al. [2011], and Liu et al. [2012]. Different crowdsourced databases like CrowdDB [Franklin et al. 2011], Deco [Parameswaran et al. 2011], Qurk

[Marcus et al. 2011a], and AskIt! [Boim et al. 2012] have been built. These systems help decide which questions should be asked of the crowd, taking into account various requirements such as latency, monetary cost, quality, etc.

The problem of finding max in the crowdsourced setting has been considered in Guo et al. [2012]. However, instead of finding the maximum element exactly, Guo et al. [2012] focus on the *judgment problem* (given a set of comparison results, which element has the maximum likelihood of being the maximum element) and the *next vote problem* (given a set of results, which future comparisons will be most effective). Finding the exact solution to these problems is shown to be hard, and efficient heuristics are proposed that work well in practice. Finding max in crowdsourced settings has also been considered in Venetis et al. [2012]. It provides efficient heuristics and evaluates them empirically; the results can be tuned using parameters like execution time, cost, and quality of the result. In a recent paper [Polychronopoulos et al. 2013], the authors consider the crowdsourced top-k problem in the setting where the unit of task (for time and cost) is not comparing two elements, but ranking any number of elements. The error model is captured by potentially different rankings of the same elements by different people. If many people disagree on the ranking of certain elements, more people are asked to rank these elements to resolve the conflict in future rounds. The more general sorting problem is considered in Marcus et al. [2011b], where the authors aim to minimize the cost of asking questions using the Qurk system by optimization techniques like batching and replacing pairwise comparisons by numerical ratings. Similarly, in CrowdDB [Franklin et al. 2011], the system can ask the crowd to check whether the values of two elements are equal as well as to rank or order a list of elements, and the results are stored for future queries. The focus of Marcus et al. [2011b] is mainly on handling different implementation aspects, in contrast to obtaining rigorous theoretical results, which is the goal of this article.

On the other hand, the problem of finding max/top-k elements in the presence of noisy comparison operators has been extensively studied in the theory community. Our work is closest to Feige et al. [1994] who consider the more standard *constant error model*, where the probability of getting a correct answer while comparing two elements is a constant $> \frac{1}{2}$ (i.e., the answer is better than a random answer). We compare our work with Feige et al. [1994] in detail in Section 4, and show that we obtain better bounds for max and for small values of k in top-k when the error function in the variable error model is strictly monotone. Other error models for noisy comparisons have also been considered in the literature (see the references in Venetis et al. [2012] and Guo et al. [2012]). For instance, Ajtai et al. [2009] assume that, if the values of two elements being compared differ by at least δ for some $\delta > 0$, then the comparison will be made correctly; when the two elements have values that are within δ, the outcome of the comparison is unpredictable. Indeed, 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.

A related problem studied in the machine learning community is that of *learning to rank in information retrieval* (see Liu [2009] for a survey). Ranking is a critical component in applications like search engines, collaborative filtering, question answering, online advertisement, etc., where the ranking function is responsible for matching the processed queries with the available results. In Burges et al. [2005], the authors investigate learning ranking using gradient descent methods and probabilistic cost functions. An active exploration strategy is proposed in Radlinski and Joachims [2007] that

helps ranking functions in search engines train faster from the available clickthrough data.

The clustering problem in the crowdsourced setting has been considered in Gomes et al. [2011]. However, the goal of Gomes et al. [2011] is to define cluster types instead of only classifying the elements. It also assumes that workers may have different clustering criteria and each worker only has a partial view of the data. Their paper proposes a Bayesian model of how the workers can approach clustering, which is in contrast to our model where we assume a fixed (but unknown) set of clusters partitioning the elements. On the other hand, Yi et al. [2012] consider the setting where, given the annotations obtained through crowdsourcing for a small subset of the objects, the objective is to cluster the entire collection of objects using their low-level features. Previous work has also considered filtering data items based on a set of properties that can be manually verified [Parameswaran et al. 2012], and also entity resolution using crowdsourcing [Wang et al. 2012]. In a recent work [Whang et al. 2013], the authors have considered the related entity resolution problem: given a number of elements, find those that refer to the same entity. Similar to our type questions, Whang et al. [2013] ask the crowd to check whether two elements represent the same entity. However, the focus of Whang et al. [2013] is to find the best set of questions to maximize the expected accuracy of the F_1-metric (harmonic mean of the precision and recall) against an unknown gold standard (pairs of elements from the same entity), assuming the answers are always correct. Another related problem, that of finding the centroid of a dataset, is studied in Heikinheimo and Ukkonen [2013]. To the best of our knowledge, our work is the first to formally study the problem of finding the exact clusters in the crowdsourced setting assuming a ground truth on the clusters, and to study the correlation between types and values.

Apart from the standard unit cost function where the cost is measured in terms of the number of comparisons, in this article we study the problem of finding the maximum element under *monotone nonnegative concave cost functions*. This class of functions exploits economy of scale using the subadditivity property and has been studied for many optimization problems in the literature (e.g., Fotakis and Tzamos [2013] and Guisewite and Pardalos [1991]).

3. PRELIMINARIES

In this section we present preliminary notions and formally define the problems that we study in this article.

There are n elements $x_1, \ldots, x_n$. Each element x_i is associated with a *type* (denoted by $\texttt{type}(x_i)$) and a *value* (denoted by $\texttt{val}(x_i)$). We denote by J the number of distinct types. Types induce a partition of the elements into J clusters, where each cluster contains elements of the same type. The clusters are *balanced* when the ratio of maximum and minimum cluster size is $O(1)$, that is, when each cluster has about n/J elements. The values of the elements are distinct and there is a total order on the values, that is, for any two elements x_i and x_j, either $\texttt{val}(x_i) > \texttt{val}(x_j)$ or $\texttt{val}(x_j) > \texttt{val}(x_i)$. For simplicity, we will use $x_i > x_j$ instead of $\texttt{val}(x_i) > \texttt{val}(x_j)$ when clear from the context. From now on, we will assume the elements are indexed in decreasing order of their values, that is, $x_1 > x_2 > \cdots > x_n$.

In the PhotoDB example mentioned in the Introduction, each photo is an element. The type of a photo is the person appearing in the photo, while the value of a photo is the age of the person in the photo (or the date when the photo was taken). The number of clusters J is the number of distinct people in the PhotoDB database. Since some people may appear in many more photos than others, the clusters are not necessarily balanced. Note that the name or age of the individuals may not be explicitly recorded in the photos, but that there is an underlying ground truth.

3.1. Top-*k* and Group-by Database Queries

If we consider the elements as tuples in a relational setting, then the types and values of the elements can be assumed to be two different hidden attributes of the tuples that are not explicitly mentioned in the database. In a simple *top-k query*, we assume that only elements of the same type (from the same cluster) are present. Here the goal is to find the maximum or the top-k elements having the highest values (the smallest elements can be found similarly). As an example, consider the scenario where a person accesses only her photos in the database and wants to find the most recent photo, or where someone wants to find the most recent photo of a place of interest given a number of such photos. That is, there is a preprocessing step that selects a set of photos of interest from `IndividualPhotoDB`; we will call this set `MyPhotoDB`. The value of a photo is the *date* when it was taken. This query can be expressed as follows, where *Most-recent* is a function that asks value queries to the comparison oracle (e.g., the crowd) to select the most recent photo (recall that the photos do not have an explicit date attribute).

```
SELECT Most-recent(photo)
FROM MyPhotoDB
```

On the other hand, the goal of a simple *group-by query* is to group together those elements having same types, that is, to find the clusters as mentioned before. An example group-by query that clusters the photos based on the individuals appearing in them and counts the size of each cluster is the following (recall that the photos do not have an explicit person attribute, and the grouping is implemented asking type queries to the comparison oracle to compare the persons in the photos).

```
SELECT count(*)
FROM PhotoDB
GROUP BY Person(photo)
```

Note that a simple top-k query uses only value queries, whereas a simple group-by query uses only type queries. However, top-k and group-by queries can be combined in the natural way when we want to find the top-k/maximum element from each group, as in the example database query given in the Introduction.

3.2. Questions Asked to the Comparison Oracle

As previously mentioned, a comparison oracle is used to compute the functions "*Most-recent(photo)*" or "`GROUP BY` *Person(photo)*" in the preceding queries. This is done by posing *questions* to the oracle that ask it to compare types or values of two elements, followed by some computation performed within the system; these calculations can be performed in rounds. In our model, the oracle can be asked either a *type question*, that is, given two elements x_i and x_j, whether $\texttt{type}(x_i) = \texttt{type}(x_j)$, or a *value question*, that is, given two elements x_i and x_j, whether $\texttt{val}(x_i) > \texttt{val}(x_j)$. Note it is possible to compare the values of two elements of different types, as we will do in Section 6. The answers to these questions are always "yes" or "no"; in particular, we cannot ask what is the type or value of a given element. This is motivated by crowdsourcing applications, since the crowd may not know the exact date when the photo was taken, who the person is in the photos they are shown, or where are the places of interest. From now on, we use "queries" to denote database group-by or top-k queries, and "questions" or "comparisons" to denote the type or value questions asked of the oracle.

Independence assumption. We assume that the answers of two different questions asked of the oracle are *mutually independent*. This simulates a crowdsourcing setting

in which the same person compares different pairs of elements (at least one of the elements differs) and answers them independently. For unit cost, we can assume the oracle models different crowd workers. Therefore, even if the same pair of elements is compared multiple times by the oracle to reduce the probability of error, we receive independent answers. On the other hand, for concave cost functions as discussed in Section 7, the oracle is asked a batch of questions at the same time. As long as each batch contains different comparisons (no two pairs have the same two elements), we assume the answers from the oracle are independent. Note that, in some settings, the answers from the same person may not be independent due to implicit biases, which we leave as future work.

3.3. Error Model

Although each element has a fixed type and a fixed value, we assume the comparison oracle may return incorrect answers. For example, due to differing skill levels or the amount of time and effort spent, the answers returned by the crowd may be erroneous. We first model the potential noisy answers by a simple and standard *error model for the questions*: both type and value questions are answered correctly with probability $\geq \frac{1}{2} + \epsilon$, where $0 < \epsilon \leq \frac{1}{2}$. For simplicity, we assume ϵ is the same for both type and value questions, but the results obtained are similar if they are different. This ensures that the answer returned by the oracle is always correct with higher probability than a random "yes" or "no" answer returned with probability $\frac{1}{2}$. We call this the *constant error model*.

For the max/top-k problem, we will see the effect of a more refined *variable error model* for value questions, where the probability of error decreases when two elements that are far apart in the total order on values are compared. For instance, given two photos of an individual, it is easier for the crowd to decide which one has been taken earlier if the time difference between the photos is 10 years rather one week. We formalize this concept as follows: A function $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ is *monotone* (respectively, *strictly monotone*) if, for all $n_1 \geq n_2$, $f(n_1) \geq f(n_2)$ (respectively, $f(n_1) > f(n_2)$).[2] In addition, f is said to have a *superconstant growth rate*[3] if $f(n) = \omega(1)$, that is, as $n \to \infty$, $f(n) \to \infty$. We call a function f *strictly growing* if f is strictly monotone and $f(n) = \omega(1)$[4]. In the variable error model, given two distinct elements x_i, x_j such that $x_i > x_j$, the probability of error

$$\Pr[x_j \text{ is returned as the larger element}] \leq \frac{1}{f(j-i)}, \tag{1}$$

where f is a strictly growing function, $f(1) \geq 2 + \epsilon'$, $\epsilon' > 0$ is a constant. The conditions $f(1) \geq 2 + \epsilon'$ ensure that, even if x_i and x_j are consecutive elements in the total order, that is, $j = i + 1$, the probability of error is $\leq \frac{1}{2} - \epsilon$ for some constant $\epsilon > 0$. In other words, the probability of making the right decision is also strictly greater than $\frac{1}{2}$. Note that, when $f(\Delta) = 2 + \epsilon'$ for all inputs Δ, the variable error model is the same as the constant error model for value questions.

[2] $\mathbb{N}, \mathbb{R}$, and $\mathbb{R}^{\geq 0}$, respectively, denote the set of natural numbers, the set of real numbers, and the set of nonnegative real numbers.

[3] We will frequently use the standard notation $O()$ and $\Omega()$ for asymptotic upper and lower bounds, and $o()$ and $\omega()$ for strict asymptotic upper and lower bounds, respectively [Cormen et al. 2009]. We will also use the notation $\tilde{O}()$ or $\tilde{\Theta}()$ that hides the associated logarithmic terms in an expression.

[4] Note that a function can be only monotone or only with superconstant growth rate, for instance, $f(n) = \sum_{j=1}^{n} \frac{1}{2^j}$ is strictly monotone but is not $\omega(1)$. On the other hand, if $f(n) = 2$ for $n \leq 10$ and $= n^2$ otherwise, then f is not strictly monotone but is $\omega(1)$.

The function f is called the *error function*, which we assume known. We discuss the problem of estimating f in Section 8 (which we leave as future work), but note that a lower bound on f is sufficient for our algorithms although a better lower bound on f will give a better upper bound on the number of comparisons. In a crowdsourcing application with human oracle, f can have different rates of growth depending on the dataset and skill level of the crowd. For instance, when the set of n photos spans a timeline of a few weeks, the probability of error in ordering them according to the age of people is likely to be high, even when the oldest and most recent photos are compared. However, if the n photos span a timeline of over 20 years, the probability of error is much smaller when the first and last photos are compared. In Section 4, we will study the effect of different error functions on the number of questions asked to the comparison oracle.

There is a natural "value-based" alternative to the "ranking-based" variable error model described previously, where the error probability is a monotone function of the difference in values of the two elements being compared, instead of the difference in their ranks in the sorted order. In this value-based variable error model, inequality (1) becomes

$$\Pr[x_j \text{ is returned as the larger element}] \leq \frac{1}{f(x_j - x_i)}.$$

The upper bounds given in this article for the ranking-based variable error model also hold for the value-based variable error model when $x_i \geq x_{i+1}+1$ for all $i \in [1, n-1]$. Then $\frac{1}{f(x_j-x_i)} \leq \frac{1}{f(j-i)}$, that is, the value-based error is bounded above by the ranking-based error.

3.4. Problem Statements

The problems studied in this article are as follows.

(i) *Max and top-k*. Here our goal is find the maximum and, in general, the top-k elements having the highest values in order to compute top-1/top-k functions in the queries. In other words, assuming without loss of generality that $x_1 > x_2 > \cdots > x_n$, we want to find x_1 (for max) or $x_1, \ldots, x_k$ (for top-k) only using value questions (the elements are assumed to have the same type).

(ii) *Clustering*. Here we want to find the J clusters using type questions grouping together elements having the same type. This allows us to find the groups in group-by queries.

(iii) *Clustering with correlated types and values*. The problem stated earlier uses only type questions to cluster the elements. However, sometimes the types and values of elements are highly correlated. For instance, consider the average price (value) of rooms in a hotel (elements) versus their quality (type). When we sort rooms in the same hotel according to their price, it is likely that rooms of similar quality (e.g., standard versus deluxe) will form a contiguous block in the sorted order on their value. This we call the *full correlation case*. On the other hand, consider the average price (value) of hotels in a district (elements) versus their quality (type). When we sort the hotels according to their average prices, the hotels of similar quality (star-ratings) are expected to form "almost" contiguous blocks in the sorted order (due to other factors like location). This we call the *partial correlation case*. In this problem, once again, we want to find the J clusters for group-by queries, however, we use both type and value questions in order to exploit any correlation between types and values.

The prior intuition is formalized as follows (see Figure 1 for an example). Suppose $x_1 > x_2 > \cdots > x_n$. There are at most α changes in types of elements in the sorted order

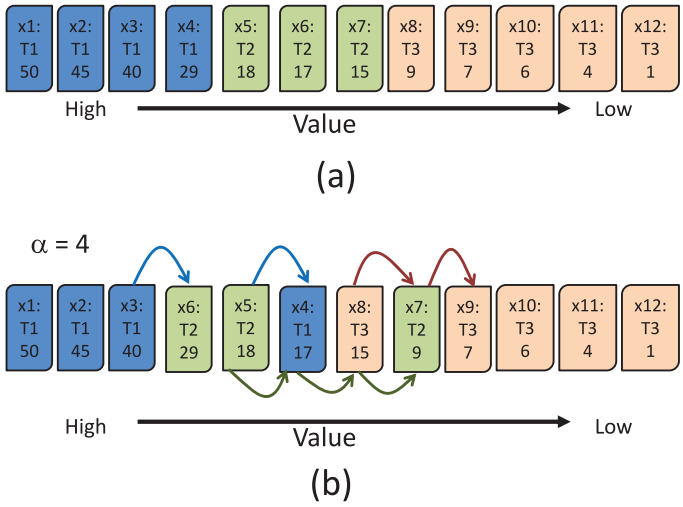

Fig. 1. Examples of (a) full correlation; (b) partial correlation with $\alpha = 4$ (there are at most $\alpha - 1 = 3$ changes between any two elements of the same type). Here $n = 12$ elements are identified with $x_1, \ldots, x_{12}$. There are three clusters T_1, T_2, T_3, that is, $J = 3$. The values of the elements are also shown, for example, $\texttt{val}(x_1) = 50$, $\texttt{val}(x_2) = 45$, etc.

between any two elements of the same type for some value of α. Formally, consider any two elements $x_i > x_j$ such that $\texttt{type}(x_i) = \texttt{type}(x_j)$. There exists a value[5] of α, $\alpha \in [1, n-1]$, such that between x_i and x_j there are at most $\alpha - 1$ elements x_ℓ, $i \leq \ell < j$, where $\texttt{type}(x_\ell) \neq \texttt{type}(x_{\ell+1})$. For instance, when $\texttt{type}(x_1) = \texttt{type}(x_n)$ and all other elements have distinct types, $\alpha = n - 1$. On the other hand, when $\alpha = 1$, we get the full correlation case. In general, when the value of α is small, the clusters are nearly sorted according to the values of the elements in them, and we get the partial correlation case. We will show that fewer type and value questions in total are needed compared to type questions needed in the standard clustering problem when the value of α is small. We also discuss how we can find top-k elements from each cluster using both type and value questions (for queries using both top-k/max function and `GROUP BY` clause).

Objective. Errors in the answer to value and type questions result in errors in the answer to a database query. Therefore, we seek to find solutions to the queries that are correct with high probability: given any constant $\delta > 0$, our goal is to find the exact max/top-k elements or the exact clusters with probability $1 - \delta$.

A dominant cost factor in crowdsourcing applications is the number of questions being asked. This is because answering may incur monetary cost, involves human effort, and may be slow. Therefore, we will provide upper and lower bounds on the cost to solve the aforesaid problems by counting the total number of type and value questions in the presence of comparison errors, so the bounds will depend on the error function f, ϵ, and δ. On the other hand, in Section 7, we will consider the class of concave cost functions to study max/top-k problems where the number of comparisons is not necessarily proportional to the total cost paid. Instead of minimizing the number of comparisons, our goal in this setting will be to minimize the total cost incurred by the algorithms. Note that there are other natural cost functions such as the *latency* or *number of rounds of questions* asked of the crowd, which we leave as future work.

[5]For positive integers a, b, where $a \leq b$, $[a, b]$ denotes the interval $a, a + 1, \ldots, b$.

All logs used in the article refer to logarithms with base 2 unless stated otherwise. Next we discuss the three problems mentioned before in Sections 4, 5, and 6, respectively.

4. MAX AND TOP-K

The problem of finding the maximum and top-k elements with faulty comparisons has been extensively studied in Feige et al. [1994] under the constant error model. When value comparisons are performed by the crowd (i.e., when the comparison oracle is a person), the probability of error is likely less when two elements far apart in the sorted order are compared, which can reduce the number of questions asked of the crowd. This motivates the study of max and top-k problems under the variable error model given by inequality (1) in the previous section. In this section we show that, when the error function f in inequality (1) is strictly monotone and $f = \omega(1)$, only $n + o(n)$ value questions are sufficient to find the max or the top-k elements for a small value of k (which is typically the case in practice) with high probability. We start with the algorithm for finding max in Section 4.1, and then use this algorithm as a building block to find the top-k elements in Section 4.2.

4.1. Finding Max

Suppose $x_1 > x_2 > \cdots > x_n$. Given $\delta > 0$, we want to find x_1 with probability $\geq 1-\delta$ using a small number of value questions. When the answers to value questions are correct, $n-1$ questions are necessary and sufficient to find x_1. On the other hand, in the constant error model where each value question is answered correctly with probability $\geq \frac{1}{2} + \epsilon$ for a constant ϵ, Feige et al. [1994] give a simple algorithm to find the maximum with probability $\geq 1 - \delta$ using $O(\frac{n}{\epsilon^2} \log \frac{1}{\delta})$ questions (we sketch the algorithm later). They also show that this bound is tight as stated in the following theorem.

THEOREM 4.1 [FEIGE ET AL. 1994]. *For all $\epsilon > 0, \delta \in (0, \frac{1}{2})$, $\Theta(\frac{n}{\epsilon^2} \log(\frac{1}{\delta}))$ value questions (comparisons) are both sufficient and necessary to find the maximum with probability $\geq 1 - \delta$ in the constant error model.*

Moreover, the proof of the lower bound shows a stronger result when $\delta + \epsilon \leq \frac{1}{2}$, that is, when a high probability of success is needed in spite of a high probability of error.

Observation 1 [*Feige et al. 1994*]. There exists a constant $c > 0$ such that at least $(1 + c)n$ comparisons are needed to compute the maximum with probability $1 - \delta$ for any δ, ϵ satisfying $\delta + \epsilon \leq \frac{1}{2}$.

In this section we show that a much better upper bound can be obtained in the variable error model that almost matches both the upper bound of $n - 1$ comparisons when the value questions are correctly answered, and the lower bound of $(1 + c)n$ stated in Observation 1. Recall the probability of error for value questions in the variable error model in terms of the error function f given in (1) in the previous section: the strictly growing error function f is such that $f(1) \geq 2 + \epsilon'$, and the probability of error for comparing two elements at distance Δ in the sorted order is $\leq \frac{1}{f(\Delta)} \leq \frac{1}{2} - \epsilon$, for some constants $\epsilon', \epsilon > 0$.

The following theorem shows that, for all strictly growing functions $f = \omega(1)$, $n+o(n)$ questions suffice (for constant δ) to find the maximum with high probability. Further, the number of questions improves to $n + O(\log \log n)$ when f is at least linear (i.e., $f(\Delta) = \Omega(\Delta)$) and to $n + O(1)$ when f is exponential ($f(\Delta) = 2^{\Delta}$).

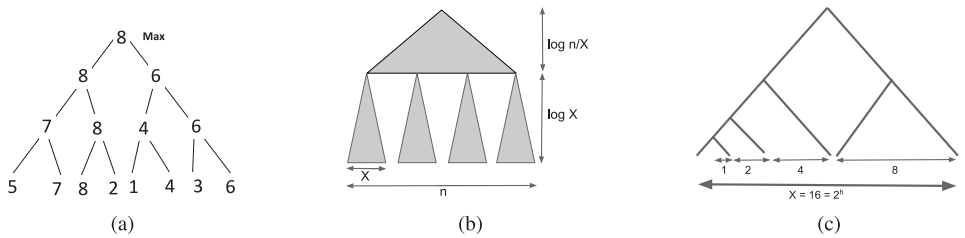

Fig. 2. (a) An example comparison tree; (b) upper and lower levels in Algorithm 1; (c) amplified single X-tree with X nodes and (lower) $\log X$ levels.

THEOREM 4.2. *For all strictly growing functions f and constant $\epsilon, \delta > 0$, $n + o(\frac{n}{\epsilon^2\delta}\log\frac{1}{\delta})$ value questions are sufficient to output the maximum element x_1 with probability $\geq 1-\delta$ in the variable error model.*

Further, if $f(\Delta) = \Omega(\Delta)$, then $n + O(\frac{\log\log n}{\epsilon^2\delta^2}\log\frac{1}{\delta})$ questions are sufficient. If $f(\Delta) = 2^{\Delta}$, then $n + O(\frac{1}{\epsilon^2}\log^2\frac{1}{\delta})$ questions are sufficient.

Next we present our algorithm and prove the bounds given in the theorem.

Our algorithm. Our algorithm uses the tournament approach using a *balanced comparison tree*[6]. Its leaves are grouped into $\frac{n}{2}$ pairs, the two elements in each pair are compared using value questions, and the winner (larger of the two elements) propagates to the level above. This is continued until only one element remains as the root of the tree that is declared as the maximum. Figure 2(a) shows an example comparison tree that uses $n-1$ comparisons at $n-1$ internal nodes assuming there is no comparison error.

In the presence of comparison errors, our key idea is to choose a random permutation of the elements $x_1, \ldots, x_n$ that appear as leaves in the tree. We divide the $\log n$ levels of the comparison tree into upper $\log\frac{n}{X}$ levels and lower $\log X$ levels (see Figure 2(b)). In the lower levels, only one value comparison is performed at each internal node. In the upper levels $L = \log X + 1$ to $\log n$, N_L comparisons are performed at each internal node, and a majority vote is taken to decide the larger element. Algorithm 1 presents our method; the parameter X will be calculated later depending on the nature of error function f.

ALGORITHM 1: Algorithm for finding the maximum element (X will depend on the error function f).

1 Choose a random permutation Π of the elements $x_1, \ldots, x_n$;
2 **for** *levels $L = 1$ to $\log n$ in the comparison tree* **do**
3 *leaves are in level 0, the root is in level $\log n$;*
4 If $L \leq \log X$ (lower $\log X$ levels), do one comparison at each internal node. Propagate the winners to the level above;
5 If $L > \log X$ (upper $\log\frac{n}{X}$ levels), do $N_L = (2(L - \log X) - 1) \times O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$ comparisons at each internal node. Take majority vote and propagate the winners to the level above;
6 **end**
7 **return** *The element at the root node of the comparison tree.*

[6]In general, a comparison tree can be any binary tree.

Analysis. The number of nodes at level L is $2^{\log n - L}$, for $L = 1$ to $\log n$. The total number of comparisons performed by the algorithm is

$$n - \frac{n}{X} + \sum_{L=\log X+1}^{\log n} N_L \times 2^{\log n - L} \leq n + \sum_{L=\log X+1}^{\log n} N_L \times 2^{\log n - L}.$$

We will show that the maximum element x_1 is returned with probability $\geq 1 - 6\delta$; to obtain the desired $1-\delta$ probability as stated in Theorem 4.2, the algorithm is run with $\delta' = \delta/6$.

We analyze the upper $\log \frac{n}{X}$ levels and the lower $\log X$ levels separately: (i) in the upper levels, we use the algorithm from Feige et al. [1994] that returns the maximum element with probability $\geq 1-\delta$; (ii) in the lower levels, we show that x_1 does not lose in any comparison with probability $\geq 1-5\delta$, even when only one comparison is performed at each internal node in the lower levels. Therefore, by union bound[7], the maximum element x_1 will be returned with probability $\geq 1 - 6\delta$. Moreover, the chosen value of X ensures that the number of comparisons in the upper levels is $o(n)$ (for constant ϵ, δ) for any strictly growing error function f.

Analysis of the upper levels. For the upper $\log \frac{n}{X}$ levels, we simply use the fact that each value question is answered correctly with probability $\geq \frac{1}{2} + \epsilon$, irrespective of the function f. Then we use the algorithm and bounds given in Feige et al. [1994] for the constant error model (see Theorem 4.1). We briefly sketch the algorithm for sake of completeness.

Consider the subtree consisting of the upper $\log \frac{n}{X}$ levels, which has $\frac{n}{X}$ nodes. Each internal node in levels $\ell = 1$ to $\log \frac{n}{X}$ uses $S_\ell = (2\ell - 1) \times O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ comparisons, and $N_L = S_{L - \log X}$. By a simple application of Chernoff bounds [Motwani and Raghavan 1995], it can be shown that the maximum element can be found with probability $\geq 1-\delta$. The total number of comparisons is $\sum_{\ell=1}^{\log \frac{n}{X}} (2\ell - 1) \times \frac{n}{X2^\ell} \times O(\frac{1}{\epsilon^2} \log(\frac{1}{\delta})) = O(\frac{n}{\epsilon^2 X}) \log(\frac{1}{\delta})$.

Therefore, given constant $\epsilon, \delta > 0$, to find the maximum element in the upper $\log \frac{n}{X}$ levels with probability $\geq 1 - \delta$,

$$\text{it suffices to ask } O\left(\frac{n}{\epsilon^2 X} \log \frac{1}{\delta}\right) \text{ value questions.} \tag{2}$$

Analysis of the lower levels. The expression in (2) bounds the number of comparisons in the upper $\log \frac{n}{X}$ levels; the number of comparisons in the lower $\log X$ levels is bounded by n. Next we show there exists a value of X such that $\frac{n}{X} = o(\frac{n}{\delta})$ for any strictly growing function f, and the maximum element does not lose in any comparison with probability $\geq 1 - 5\delta$ in the lower levels.

Algorithm 1 starts with a random permutation Π of the elements $x_1, \ldots, x_n$. Let us partition Π into block of size (at most) X of consecutive elements. Let us call the subtrees of the comparison tree in the bottom $\log X$ levels on each block of X elements an *X-tree* (the subtrees in Figure 2(b)); the number of X-trees is $\frac{n}{X}$. The algorithm performs only one comparison at each non-leaf node of each X-tree.

Consider the X-tree that contains the maximum element in Figure 2(c). Without loss of generality, assume that the leftmost leaf is the maximum element x_1. Consider the leftmost path of length h to the root of the X-tree. We will compute the probability that x_1 is never eliminated along this path.

[7]The union bound says that, for any (not necessarily independent) random events $X_1, \ldots, X_\ell$, $\Pr[X_1 \cup \cdots \cup X_\ell] \leq \sum_{i=1}^{\ell} \Pr[X_\ell]$.

The height of the X-tree is $h = \log X$. Let r_ℓ, $\ell \in [1, h]$, be a non-leaf node on the leftmost path of the X-tree. The right subtree of r_ℓ will have $2^{\ell-1}$ leaf nodes (see Figure 2(c)). Note that, if x_1 survives all the comparisons in levels 1 to $\ell - 1$, in the internal node r_ℓ, x_1 can only be compared with the nodes in the right subtree of r_ℓ. For parameters. Δ_ℓ, $\ell \in [1, h]$, to be decided later, we will bound the following probabilities.

(1) $\delta_\ell = \Pr[$ at least one leaf in the right subtree of r_ℓ corresponds to an element from the set $\{x_j : 2 < j \leq \Delta_\ell + 1\}]$.
(2) $p_\ell = \Pr[x_1$ loses the comparison at node r_ℓ, when none of the leaves in the right subtree of r_ℓ corresponds to an element from the set $\{x_j : 2 < j \leq \Delta_\ell + 1\}]$.

In particular, we prove the following proposition.

PROPOSITION 4.3. *There exist values of $h = \log X$ and Δ_ℓ, $\ell \in [1, h]$ such that $\sum_{\ell=1}^{h} \delta_\ell + \sum_{\ell=1}^{h} p_\ell \leq 5\delta$ and $\frac{n}{X} = o(\frac{n}{\delta})$ for any monotone error function f.*

It follows from Proposition 4.3 (using union bound) that the maximum element x_1 cannot lose any comparison in the lower levels. Next, we show how the values of h and Δ_ℓ, $\ell \in [1, h]$, are chosen.

Since Π is a random permutation, given a fixed position of x_1, any of the $n-1$ elements other than x_1 can appear in another given position in Π with probability $\frac{1}{n-1}$. Therefore, by union bound,

$$\delta_\ell \leq \Delta_\ell 2^{\ell-1}/(n-1). \tag{3}$$

On the other hand, if the right subtree of r_ℓ does not contain any element from $\{x_j : 2 < j \leq \Delta_\ell + 1\}$, then the minimum distance between the ranks of x_1 and the elements in the right subtree of r_ℓ is $(\Delta_\ell + 2) - 1 = \Delta_\ell + 1$. In this case

$$p_\ell \leq \frac{1}{f(\Delta_\ell + 1)} \leq \frac{1}{f(\Delta_\ell)}. \tag{4}$$

Inequality (4) follows from Eq. (1), since f is a monotone function. Given any $\delta > 0$, for all $\ell \in [1, h]$, we set

$$\frac{\Delta_\ell 2^{\ell-1}}{n-1} = \frac{(h-\ell+1)\delta}{2^{h-\ell}}. \tag{5}$$

The following lemma gives a bound on $\sum_{\ell=1}^{h} \delta_\ell$.

LEMMA 4.4. *If $\frac{\Delta_\ell 2^{\ell-1}}{n-1} = \frac{(h-\ell+1)\delta}{2^{h-\ell}}$, then $\sum_{\ell=1}^{h} \delta_\ell \leq 4\delta$.*

PROOF. Let $S = \sum_{\ell=1}^{h} \delta_\ell \leq \sum_{\ell=1}^{h} \frac{(h-\ell+1)\delta}{2^{h-\ell}}$ (from (3)). Then

$$\begin{aligned} S &= \delta + 2\delta/2^1 + 3\delta/2^2 + \cdots + h\delta/2^{h-1} \\ S/2 &= \delta/2^1 + 2\delta/2^2 + \cdots + (h-1)\delta/2^{h-1} + h\delta/2^h \\ \Rightarrow S/2 &= \delta + \delta/2^1 + \delta/2^2 + \cdots + \delta/2^{h-1} - h\delta/2^h \leq 2\delta \\ \Rightarrow S &\leq 4\delta. \quad \square \end{aligned}$$

The bound on $\sum_{\ell=1}^{h} p_\ell$ is obtained in two steps. First, in Lemma 4.5, we give an upper bound on $\sum_{\ell=1}^{h} p_\ell$ for any monotone function f in terms of $h = \log X$. Then, in Lemma 4.6, we show there exists a value of X such that $\sum_{\ell=1}^{h} p_\ell \leq \delta$ and $\frac{n}{X} = o(\frac{n}{\delta})$, which will complete the proof of Proposition 4.3.

LEMMA 4.5. $\sum_{\ell=1}^{h} p_\ell \leq \frac{h}{f\left(\frac{\delta n}{2^h}\right)}$.

PROOF. By inequality (4) and Eq. (5), $\Delta_\ell = \frac{(h-\ell+1)\delta(n-1)}{2^{h-1}}$, and $p_\ell \leq \frac{1}{f(\Delta_\ell)}$. Then $\sum_{\ell=1}^{h} p_\ell$
$\leq \sum_{\ell=1}^{h} \frac{1}{f(\Delta_\ell)}$
$= \sum_{\ell=1}^{h} \frac{1}{f\left(\frac{(h-\ell+1)\delta(n-1)}{2^{h-1}}\right)} = \sum_{\ell=1}^{h} \frac{1}{f\left(\frac{2(h-\ell+1)\delta(n-1)}{2^h}\right)}$
$\leq \sum_{\ell=1}^{h} \frac{1}{f\left(\frac{(h-\ell+1)\delta n}{2^h}\right)}$ (for $n \geq 2$, $2(n-1) \geq n$). Therefore,

$$\sum_{\ell=1}^{h} p_\ell \leq \sum_{q=1}^{h} \frac{1}{f\left(\frac{q\delta n}{2^h}\right)} \tag{6}$$

$$\leq \frac{h}{f\left(\frac{\delta n}{2^h}\right)} \quad \text{(since } f \text{ is monotone).} \quad \square \tag{7}$$

LEMMA 4.6. *Given any strictly growing function f and a constant $\delta > 0$, there exists an h and $n_0 \in \mathbb{N}$ such that, for all $n \geq n_0$, $\sum_{\ell=1}^{h} p_\ell \leq \delta$, and $\frac{n}{X} = \frac{n}{2^h} = o(\frac{n}{\delta})$.*

PROOF. Since f is strictly growing, f is strictly monotone and $f(\Delta) = \omega(1)$ (superconstant growth rate). We choose h as follows.

—$h = \log n - \log\log n - \log\frac{1}{\delta}$, if $f(\Delta) = \omega(\Delta)$.
Then $n/2^h = \frac{\log n}{\delta} = o(\frac{n}{\delta})$.
—$h = \log f(n^{1/4}) - \log\frac{1}{\delta}$, if $f(\Delta) = O(\Delta)$. Then $n/2^h = \frac{n}{\delta f(n^{1/4})} = \frac{n}{\delta\omega(1)} = o(\frac{n}{\delta})$.

If $f(\Delta) = \omega(\Delta)$, $\sum_{\ell=1}^{h} p_\ell \leq \frac{h}{f\left(\frac{\delta n}{2^h}\right)}$ (from Lemma 4.5)
$= \frac{\log n - \log\log n - \log\frac{1}{\delta}}{f\left(\frac{\delta n}{\frac{\delta n}{\log n}}\right)} \leq \frac{\log n}{f(\log n)} = \frac{\log n}{\omega(\log n)} = o(1) \leq \delta$.
When $\delta > 0$ is a constant, there exists an n_0 such that, for all $n \geq n_0$, the last step holds.
If $f(\Delta) = O(\Delta)$, $\sum_{\ell=1}^{h} p_\ell \leq \frac{h}{f\left(\frac{\delta n}{2^h}\right)}$ (from Lemma 4.5)
$= \frac{\log f(n^{1/4}) - \log\frac{1}{\delta}}{f\left(\frac{\delta n}{\delta f(n^{1/4})}\right)} \leq \frac{\log f(n^{1/4})}{f\left(\frac{n}{O(n^{1/4})}\right)} = \frac{\log f(n^{1/4})}{f(\Omega(n^{3/4}))} \leq \frac{\log f(n^{1/4})}{f(n^{1/4})}$ (for large enough n, since f is strictly monotone) $\leq \delta$.
For all constant $\delta > 0$, there exists n_0 such that, for all $n \geq n_0$, the last step holds. $\square$

Since $\frac{n}{X} = \frac{n}{2^h} = o(\frac{n}{\delta})$, by the expression in (2), the upper level uses $o(\frac{n}{\epsilon^2\delta}\log\frac{1}{\delta})$ comparisons in total. Combined with the total number of comparisons in the lower levels, which is $\leq n$, and summing up the bad probabilities in the upper and lower levels by the union bound (from the expression in (2) and Proposition 4.3), the maximum element is found with probability $\geq 1 - 6\delta$ with $n + o(\frac{n}{\epsilon^2\delta}\log\frac{1}{\delta})$ value questions.

The proof of Lemma 4.6 also shows that, when $f(\Delta) = \omega(\Delta)$, $n + O(\frac{\log n}{\delta}\log\frac{1}{\delta})$ value questions suffice. However, Lemma 4.7 shows that better bounds can be obtained when $f(\Delta) = \Omega(\Delta)$ or $f(\Delta) = 2^\Delta$, by a tighter analysis using inequality (6). The proof of the lemma appears in Appendix A.1.

LEMMA 4.7. *Given any* $\delta > 0$:

(1) *for exponential error function,* $\exists\, X$ *such that* $\frac{n}{X} = O(\log^2 \frac{1}{\delta})$ *and* $\sum_{\ell=1}^{h} p_\ell \le \delta$;
(2) *for linear error function,* $\exists\, X$ *such that* $\frac{n}{X} = O(\frac{\log\log n}{\delta^2})$ *and* $\sum_{\ell=1}^{h} p_\ell \le \delta$;
(3) *for logarithmic error function,* $\exists\, X$ *such that* $\frac{n}{X} = O(\frac{n^{\frac{1}{1+\delta}}}{\delta^{\frac{\delta}{\delta+1}}})$, *and* $\sum_{\ell=1}^{h} p_\ell \le \delta$.

Substituting the value of $\frac{n}{X}$ in the expression in (2), the better upper bounds for functions f such that $f(\Delta) = \Omega(\Delta)$ or $f(\Delta) = 2^{\Delta}$ in Theorem 4.2 can be obtained. This completes the proof of Theorem 4.2.

4.2. Finding Top-k

Suppose $x_1 > x_2 > \cdots > x_n$. Given an integer k, Feige et al. [1994] have given an algorithm to find the top-k elements $x_1, \ldots, x_k$ in the constant error model. This algorithm uses $O(n \log \frac{\min(k, n-k)}{\delta})$ comparisons to find the k-th largest element with probability $\ge 1 - \delta$. For simplicity, assume $k \le \frac{n}{2}$, that is, $\min(k, n-k) = k$.

In practice, for database top-k queries, the value of k is likely to be much smaller than the total number of elements n. When the value of k is small, a better bound on the number of value comparisons can be obtained using Theorem 4.2 in Section 4.1 and the algorithm given in Feige et al. [1994] with strictly growing error functions. In particular, we show the following corollary to Theorem 4.2 that solves the top-k problem with high probability.

COROLLARY 4.8. *For all strictly growing functions* f *and constant* $\epsilon, \delta > 0$, $n + o(\frac{nk}{\epsilon^2\delta} \log \frac{k}{\delta}) + O(\frac{k^2}{\epsilon^2\delta} \log \frac{k}{\delta})$ *value questions are sufficient to output all the top-k elements* $x_1, \ldots, x_k$ *with probability* $\ge 1 - \delta$.

Further, if $f(\Delta) = \Omega(\Delta)$, *then* $n + O(\frac{k \log\log n}{\epsilon^2\delta^3} \log \frac{k}{\delta}) + O(\frac{k^2}{\epsilon^2\delta} \log \frac{k}{\delta})$ *value questions are sufficient. If* $f(\Delta) = 2^{\Delta}$, *then* $n + O(\frac{k^2}{\epsilon^2\delta} \log \frac{k}{\delta})$ *value questions are sufficient.*

When $k = O(1)$ and $\delta > 0$ is constant, we once again get a bound of $n + o(n)$ even to find all the top-k elements with high probability, which exceeds n only by lower-order additive terms. Note that, even when the comparisons are exact, the linear-time recursive selection algorithm [Blum et al. 1973] requires cn comparisons for a constant $c > 1$ to find the k-th element (although it works for all values of k). The same guarantee of $n + o(n)$ can be obtained for any $k = o(\sqrt{n})$, when the error function $f(\Delta) = \Omega(\Delta)$ (the growth rate is at least linear). The algorithm in Feige et al. [1994] gives a better bound for other error functions and values of k.

As an aside we note that, for any fixed $\delta > 0$, when the answers to value questions have no errors and when $k = o(\sqrt{n})$, our techniques give a bound of $n + g(k, \delta)$ on the value comparisons for finding the top-k elements with probability $\ge 1 - \delta$. Here $g(k, \delta)$ is a polynomial function of k and $\frac{1}{\delta}$ independent of n (see Appendix A.2).

To conclude this section, we sketch how we can obtain Corollary 4.8 using Theorem 4.2; the details are given in Appendix A.3. Once again, we use a comparison tree and start with a random permutation of the elements in the leaves of this tree. We also divide the $\log n$ levels of the comparison tree into lower $\log X$ levels and upper $\log \frac{n}{X}$ levels. In the lower levels, we have $\frac{n}{X}$ X-trees. We show there is a value of X such that with high probability each of $x_1, \ldots, x_k$ appear in different X-trees so that they are the maximum elements in their respective X-trees. In all the X-trees we use Algorithm 1, and argue that all of $x_1, \ldots, x_k$ are the winners in their respective X-trees with high probability. Therefore, in the upper $\log \frac{n}{X}$ levels, $x_1, \ldots, x_k$ remain to be the top-k elements. In the upper levels, which have $\frac{n}{X}$ elements, we use the top-k algorithm from

Feige et al. [1994]. We show that the total number of questions asked of the oracle is given by the expressions in Corollary 4.8. We also argue that the total probability of error (the probability that exactly $x_1, \ldots, x_k$ is not returned) is bounded by δ so, with probability $\geq 1 - \delta$, we find the top-k elements.

5. CLUSTERING

In this section we study the clustering problem motivated by group-by queries. Recall that there are J distinct types, and the goal of the clustering problem is to find the J clusters, that is, those groups of elements having the same type. Instead of value questions used in the previous section for the max and top-k problems, here we will use type questions, that is, the oracle is asked to decide whether two elements have the same type, for instance, whether two photos capture the same person or place.

We prove the following theorem which gives a bound on the number of type questions that are necessary and sufficient to find the exact J clusters. In our algorithms, we do not assume J is known a priori—when a set of questions regarding a photo database is asked, the crowd (oracle) may not know the number of people participating in the database. However, our (tight) lower bounds hold even when the value of J is known. Recall that we assume the constant error model for type questions, that is, each type question is answered correctly with probability $\geq \frac{1}{2} + \epsilon$, for a constant $\epsilon > 0$.

THEOREM 5.1. *For all $\epsilon, \delta > 0$, to group n elements into J clusters with probability $\geq 1 - \delta$, $O(\frac{nJ}{\epsilon^2} \log \frac{n}{\delta})$ type questions in expectation are sufficient in the constant error model.*

On the other hand, $\Omega(nJ)$ type questions are necessary: (i) even if the algorithm is randomized; (ii) even when answers to all type questions are exact; and (iii) even when the value of J is known.

The basic idea of the clustering algorithm is to scan the list of elements iteratively. In each iteration, all remaining elements having the same type as the first element in the list are collected as a new cluster (along with the first element) and deleted from the list. Since there are J clusters, the list will be empty after J iterations (even without prior knowledge on the value of J) assuming no comparison errors. To handle comparison errors, each pair of elements is compared multiple times and a majority vote is taken; the pseudocode is given in Algorithm 2.

ALGORITHM 2: Algorithm for clustering

```
List the elements in an arbitrary order L;
Initialize a set for clusters P = ∅;
while L is not empty do
    Let y be the first remaining element in L;
    Create a new cluster C = {y};
    /* Find elements with the same type as y among the remaining elements in L */
    Scan L; for each remaining element x in L
        Compare the types of x, y by asking the type question type(x) = type(y)
        O(1/ε^2 (log n/δ)) times.
        If the majority of the answers are "yes"
            /* x, y are decided to have the same type */
            C = C ∪ {x};
            Remove x from L;
        /* otherwise do nothing */
    Add the cluster C to P.
end
return the clusters in P.
```

Proof of upper bound. Here we argue that Algorithm 2 finds the J clusters with high probability. With appropriate choices of constants, by the Chernoff bound, whether the elements x, y compared in step 8 have the same type is decided incorrectly with probability $\leq \frac{\delta}{n^3}$. Since $J \leq n$, $nJ \leq n^2$. By union bound, with probability $\geq 1 - \frac{\delta}{n}$, for all pairs of elements considered by the algorithm whether they have the same type is decided correctly. When the type comparisons are correct, it is easy to check that the correct clusters are returned in J iterations. This happens with probability $\geq 1 - \frac{\delta}{n}$.

Note that, in each iteration, at least the first remaining element from the list L is deleted, therefore the loop is run at most n times. However, as argued earlier, with probability $\geq 1 - \frac{\delta}{n}$, the number of iterations of the while loop is J (when the clusters are correctly returned), and with probability $\leq \frac{\delta}{n}$, the number of iterations is $\leq n$. Hence the expected number of iterations is $O(J)$. In each iteration, at most $O(\frac{n}{\epsilon^2} \log \frac{n}{\delta})$ type questions are asked, therefore, in expectation, the bound given in Theorem 5.1 follows.

Proof of lower bound. First we give the proof of lower bound for the deterministic algorithm, when there is no error in the answers to type questions (exact comparisons), and when the value of J is known. Then we prove the lower bound for randomized algorithms.

Lower bound for deterministic algorithms. Let $s_{\max}$ be the size of the maximum cluster in an instance, and $s_{\min}$ be the size of the minimum cluster. Recall that the instance is called balanced if $s_{\max}/s_{\min} = O(1)$. We will prove the lower bound of $\Omega(nJ)$ for deterministic algorithms even when the clusters are balanced.

Consider any deterministic algorithm A that solves the clustering problem. Let us number the clusters arbitrarily as $C_1, \ldots, C_J$. The adversary starts by assigning $2n/3J$ elements to each of the clusters $C_1, C_2, \ldots, C_J$ and reveals these elements for free to A (therefore, algorithm A knows the value of J). At this point, the number of unassigned elements is $n/3$. Let this be the set U. The adversary now plays an evasive game on this set U. An element $x \in U$ is active iff it has been compared with $\leq (J-1)/2$ elements. For an active element, whenever the algorithm A asks a question involving it, the answer is always "no". Once an element ceases to be active, it has at least $J-(J-1)/2 = (J+1)/2$ valid clusters among $C_1, \ldots, C_J$ to which it can still be assigned. We always assign it to a cluster with smallest number of elements, breaking ties arbitrarily. This ensures that no cluster C_i ever gets assigned more than $\frac{n/3}{(J+1)/2} < 2n/3J$ elements from U. So the minimum cluster size is $2n/3J$ (recall that all clusters initially had $2n/3J$ elements), and the maximum cluster size is $4n/3J$. The ratio is bounded by 2. The total work done is clearly $\Omega(nJ)$.

Lower bound for randomized algorithms. We next show that an $\Omega(nJ)$ lower bound holds for randomized algorithms as well, even when all type comparisons are exact. By Yao's min-max principle [Motwani and Raghavan 1995], it suffices to exhibit a distribution on input instances such that any deterministic algorithm needs $\Omega(nJ)$ comparisons in expectation with respect to this distribution.

Suppose the clusters are $C_1, \ldots, C_J$. For each element, we randomly choose $j \in [1, J]$ and assign it to cluster C_j. Let us call an element x to be *settled* if either the algorithm performs $J-1$ comparisons involving x or if the algorithm performs a type comparison between x and some element y whose result is a "yes". Note that, to cluster all n elements, each element must be settled. This is because if $\leq J-2$ comparisons are performed involving x and all of the comparisons return "no", there are still at least two clusters where x can go. Next we compute the expected number of comparisons needed to make an element settled by a "yes" answer.

Suppose ℓ comparisons have been performed involving x, all of which answered "no". Since the type of each element is chosen uniformly at random, under the previous assumptions, for any element x that has participated so far in ℓ type comparisons, each of which resulted in a "no", the probability that the next type comparison returns a "yes" is bounded by $\frac{1}{J-\ell}$. The probability that each of the first ℓ type comparisons of x returns a "no" is at least $\frac{J-\ell}{J}$. Thus, the expected number of type comparisons before an element gets a "yes" answer is at least $\sum_{\ell=1}^{J} \ell \times \frac{1}{J-\ell} \times \frac{J-\ell}{J} = \frac{J+1}{2}$. Therefore the expected number of comparisons for an element to get settled is $\Omega(J)$. Since every type comparison involves exactly two elements, it follows by linearity of expectation that the total number of type comparisons is $\Omega(nJ)$. We leave the exact bound for randomized algorithms for the balanced case as an open problem.

6. CLUSTERING WITH CORRELATED TYPES AND VALUES

In the previous section, we used only type questions for clustering that compares whether two elements have the same type. We showed that, to cluster n elements into J clusters, $\tilde{\Theta}(nJ)$ questions are necessary and sufficient. However, as mentioned in Section 3.4, types and values can be correlated in some scenarios and elements of the same type can form (almost) contiguous blocks in the sorted order according to the values (e.g., quality of hotels as types versus their prices as values). We formalized this idea assuming at most α changes in types between any two elements of the same type. In this section we will see that this bound improves to $\tilde{O}(n \log J)$ when α is small and both type and value questions are asked. Note that both value and type questions are answered correctly with probability $\geq \frac{1}{2} + \epsilon$, given a constant $\epsilon > 0$.

THEOREM 6.1. *Given any $\delta > 0$, it is sufficient to ask $O((n \log(\alpha J) + \alpha J)\frac{1}{\epsilon^2} \log \frac{n}{\delta})$ type and value questions in expectation to cluster n elements into J clusters with probability $\geq 1 - \delta$.*

As in the previous section, we do not assume that the value of J is known a priori, however, we assume the value of α (or an upper bound on α) is known. In this section we will also explain why the bound given in the preceding theorem is tight in a certain sense, and briefly discuss how top-k/maximum elements from each of the J clusters can be found with high probability using both type and value questions.

When $\alpha = 1$, we have the full correlation case where elements from the same type exactly form contiguous blocks in the sorted order on values. When the value of α is small, we have the partial correlation case. First we present Algorithm 3 for the full correlation case where we also assume the answers to type and value questions are exact (there is no comparison error) and analyze this algorithm. Then we discuss how erroneous answers to type and value questions can be handled and how the algorithm can be extended for general α.

6.1. Clustering for Full Correlation

Since the elements from the same cluster form contiguous blocks in the sorted order on the values, a simple algorithm will sort the elements using $O(n \log n)$ value questions, and then scan the sorted order to find elements from the same cluster by comparing consecutive elements using $O(n)$ type questions. Here we give an algorithm that improves the number of value and type questions from $O(n \log n)$ to $O(n \log J)$, where J is the number of clusters and typically much smaller than n.

Algorithm 3 identifies those elements at the boundaries of the clusters in the sorted order in $O(n \log J)$ time. This is achieved by iteratively partitioning the list of input

ALGORITHM 3: Algorithm for clustering for the full correlation case (of types and values)

```
List all elements in L in an arbitrary order;
Initialize link(y) = null for each element y;
Set repeat_loop = true;
while repeat_loop is true do
    Let s = |L|;
    Initially, the entire L forms a single interval;
    while |L| > s/2 do
        /* The total number of elements in L is not halved */ ;
        if each interval has exactly one element then
            repeat_loop = false; break;
        end
        else
            /* Divide each interval in half to form two smaller intervals */
            for each interval B with two or more elements do
                Find the median of the elements in B;
                Partition the elements in B in two halves comparing with the median using
                value questions;
                Each of these two halves forms a new interval, say B_1 and B_2. B_1 and B_2
                respectively contain elements greater and less than the median;
                for both B_i, i ∈ {1, 2} do
                    Check if B_i has at least two types (then it is called an active interval):
                    The first element y in B_i is compared with each of the other elements z
                    in B_i to check if there is a z such that type(y) ≠ type(z);
                    if B_i is active then
                        Do nothing;
                    end
                    else
                        /* All elements in B_i have the same type, keep only one, delete the
                        rest */
                        Choose an arbitrary element y from B_i;
                        For the other elements z ≠ y in the interval, set link(z) = y. Delete
                        z;
                    end
                end
            end
        end
    end
end
/* There may be successive blocks in the list L with single elements that are from the same
cluster. Group them by a linear scan */
while L is not empty do
    Let x be the first element in L. Scan L. Let y be the next element in L;
    while type(y) = type(x) and y is a valid element (i.e., the end of list is not reached) do
        Assign link(y) = x. Delete y from L. Let y be the next element in L;
    end
    Remove x from L. /* link(x) remains null */
end
return all elements y with their link link(y)
```

elements around their median $\log J$ times into blocks, and discarding all but one element from blocks that only contain elements from the same cluster.

Since the clusters do not have any name to identify them, Algorithm 3 forms the clusters as a forest of trees. For each cluster C, except one element in C, each element

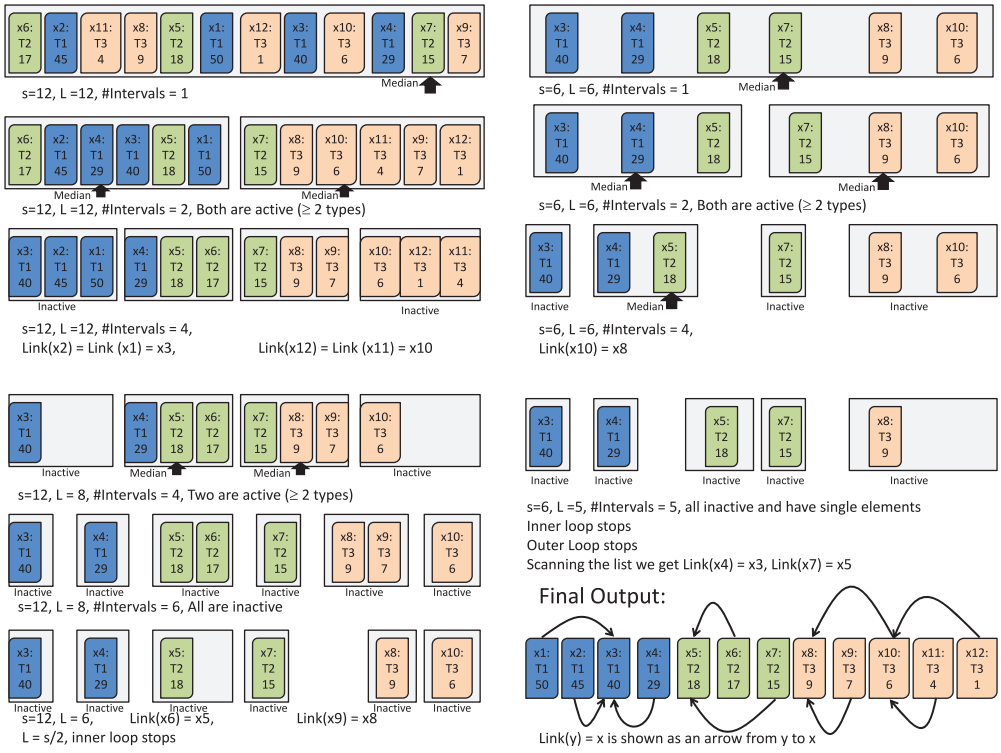

Fig. 3. An illustration of execution of Algorithm 3 on the example of full correlation in Figure 1: (a) First iteration of the outer loop; (b) the second (and last) iteration of the outer loop. The final output of the algorithm with `link` structure is also shown in (b).

y stores a pointer $\texttt{link}(y)$ that points to another element z in the same cluster (i.e., $\texttt{type}(y) = \texttt{type}(z)$). We argue that, when the algorithm finishes, the links form a tree for each cluster and there is exactly one element with *null* link that forms the root of this tree. Clearly, from this forest the clusters can be output in $O(n)$ time. The execution of Algorithm 3 on the example in Figure 1(a) is illustrated in Figure 3.

Analysis. We first analyze Algorithm 3 assuming the answers to all type and value questions are correct. Note that we assign $\texttt{link}(z) = y$ if and only if $\texttt{type}(y) = \texttt{type}(z)$, therefore we never set $\texttt{link}(z)$ incorrectly for any element z. Also, whenever an element z is deleted (step 26), another element y such that $\texttt{type}(y) = \texttt{type}(z)$ is retained, that is, we never delete all elements from a type. Further, as $\texttt{link}(z) = y$ is set, we delete z from the list so the link structure is always acyclic. Therefore, we argue that the algorithm returns exactly one element y from each type such that $\texttt{link}(y) = \mathit{null}$, then the links form a tree structure that proves its correctness.

For sake of analysis, consider the elements $x_1 > \cdots > x_n$ in sorted order. In the full correlation case, elements from the same type form contiguous blocks in the sorted order. We argue that the two while loops in the algorithm keep exactly one element from each such block.

The algorithm tries to identify these blocks by dividing the list of elements (presented in an arbitrary order L) into intervals. As long as there is one interval with more than two types, the interval is partitioned into two halves by the median, which also ensures

that all elements before (respcectively, after) the median are greater (respcectively, smaller) than the median. Therefore, by repeatedly finding the medians, we divide the list of elements into intervals such that all elements of any earlier interval are larger than all elements in any interval after. When the variable `repeat_loop` is set to FALSE, only one element from each block (i.e., from each cluster) is retained, and the others are in sorted order of their values in L. However, two consecutive elements in L can be from the same cluster. They are grouped together by a linear scan on L and only one element of the same cluster is retained. These remaining elements with *null* link are returned by the algorithm as root of the trees of the clusters.

Number of questions asked. The following lemma bounds on the total number of type and value questions.

LEMMA 6.2. *The total number of type and value questions used by the algorithm is* $O(n \log J)$ *assuming the answers to these questions are correct.*

PROOF. First we compute the total number of questions asked in each iteration of the inner while loop (step 7). Let us count unit cost for the repeated value and type questions in steps 15, 16, and 19. Consider the first `for` loop with original intervals in step 14. Let the number of intervals be b, and let the number of elements in these intervals be $n_1, \ldots, n_b$. Since the intervals are disjoint, $n_1 + \cdots + n_b \leq s$. In the j-th interval, the linear-time selection algorithm [Blum et al. 1973] can find the median using $O(n_j)$ value questions (step 15) and the partition can also be done using $O(n_j)$ value questions (step 16). This `for` loop further partitions the intervals into two disjoint intervals B_1, B_2. In the inner `for` loop (step 18), only one element from each interval is compared with the other elements using type questions, hence the total number of type questions in B_1, B_2 is $O(n_j)$. Therefore, the total number of value and type questions in the outer `for` loop (step 14) is $\sum_{i=1}^{b} O(n_j) = O(s)$.

Next we compute the number of iterations in the inner while loop. Consider the contiguous blocks of elements of the same type in the sorted order. Since there are J clusters, the number of blocks is J. Suppose the algorithm reduces the number of elements in P iterations of the inner while loop. Then the number of elementes at the start of the while loop s is divided into 2^P intervals. When $2^P = 4J$, at most J intervals may be active (two or more types). The active intervals have $\leq s/4$ elements in total. Each interval is of size $s/4J$ and one element is retained from each inactive interval, hence the number of remaining elements will be $s/4 + s/4J \leq s/2$. Therefore, the while loop will terminate in $P = O(\log J)$ iterations.

Now we compute the number of questions in the outer `while` loop in step 4. The inner while loop ensures that the problem size s is halved in each of its iterations. Hence the number of questions $Q(n)$ with input size n is captured by the following recurrence relation (counting unit cost in steps 16 and 19).

$$Q(n) = Q\left(\frac{n}{2}\right) + O(n \log J). \tag{8}$$

The solution of this recurrence relation is $O(n \log J)$. □

Handling erroneous answers to type and value questions. Here we discuss how erroneous answers to type and value questions can be handled using standard techniques. When type and value comparisons are correct, $cn \log J$ questions suffice for some constant c. Now consider the case when the comparisons are erroneous, but correct answers are returned with probability $\geq \frac{1}{2} + \epsilon$, for constant $\epsilon > 0$. In this case we repeat each type or value comparison performed by Algorithm 3 between two elements $O(\frac{1}{\epsilon^2} \log \frac{n}{\delta})$ times and take the majority vote (omitted in the algorithm for simplicity) to decide

whether they have the same type, or to order them according to their values. This adds the multiplicative $\frac{1}{\epsilon^2}\log\frac{n}{\delta}$ factor in the total number of questions asked. Moreover, we abort the algorithm after comparing cn^2 pairs of elements (note that we do not assume the value of J is known).

With appropriate choices of the constants, by the Chernoff bound, the answer to each type and value comparison between any two elements is correct with probability $\geq 1 - \frac{\delta}{cn^2}$. By union bound, the total bad probability in the $cn\log J \leq cn^2$ comparisons is bounded by δ. Hence, with probability $\geq 1-\delta$, all the comparisons are correct and the previous analysis holds. The expected number of questions asked by the algorithm is $O(n\log J) \times O(\frac{1}{\epsilon^2}\log\frac{n}{\delta})$, where $cn\log J$ comparisons are performed with probability $\geq 1-\delta$ and cn^2 comparisons are performed only with probability $\leq \delta$. This proves Theorem 6.1 for the full correlation case.

6.2. Extension to General α

For arbitrary α, there are at most α changes in types between any two elements of the same type. Again partition the elements in the sorted order $x_1 > \cdots > x_n$ into consecutive blocks of the same type. Algorithm 3 is run for the case of general α. As argued earlier, this will give one representative element from each block. However, now there may be more than one representative element from the same cluster, so we need to group them together. Further note that these representative elements will be sorted according to their values due to repeated partitioning using medians of active intervals.

To group elements of the same types, consider the list of remaining elements L returned by Algorithm 3. While L is not empty, select the first element y in L. For the next α elements z in L, check whether y and z have the same type. For all elements z with $\texttt{type}(y) = \texttt{type}(z)$, set $\texttt{link}(z) = y$. Delete these elements from S. Then repeat the procedure with the remaining elements in L (in order).

We already argued in the full correlation case that Algorithm 3 leaves one element from each consecutive block of the same type. In the additional step to group elements of the same types, the consecutive α elements in L are examined by type questions. At most α changes in types are present between the first and the last element of any cluster in the sorted order. This ensures that all elements of the same type as the first element y in S will be grouped together. This process is repeated until the list L is empty, which returns all clusters.

To count the number of type and value questions, note that the number of consecutive blocks for general α is $\leq \alpha J$, therefore Algorithm 3 asks $O(n\log(\alpha J))$ questions (similar to Lemma 6.2). The additional step to group elements of the same type needs J iterations. So $O(n\log(\alpha J) + \alpha J)$ comparisons suffice when the answers to type and value questions are exact. Errors in the answers can be handled by repeating each comparison $O(\frac{1}{\epsilon^2}\log\frac{n}{\delta})$ times and taking the majority vote, as described for the full correlation case.

6.3. Lower Bounds

Let us briefly discuss why the bounds given in Theorem 6.1 are tight up to logarithmic factors in a certain sense, even when there is no error in the comparisons. Recall that we proved $\Omega(nJ)$ lower bound for clustering in Section 5. Since α in the worst case is $\Omega(n)$ (no correlation between types and values), we cannot hope to get a better bound than $O(\alpha J)$ for all values of α. Further, there is also a lower bound of $\Omega(n\log n)$ which explains that we cannot get a better bound than $O(n\log(\alpha J))$ for all values of α and J. This lower bound follows from the *element distinctness problem*, that is, given n elements, check whether any two elements have the same value, which is known to

have a lower bound of $\Omega(n \log n)$ [Ben-Or 1983]. In the reduction, two elements belong to the same cluster if and only if they have the same value, and a cluster has ≥ 2 elements if and only if the elements are not distinct.

6.4. Max/Top-k from Each Cluster Using Constant and Variable Error Model

When both type and value questions are asked, a natural question to ask is to find top-k or the maximum element from each cluster (see the query example given in the Introduction). This can be achieved by combining our results on clustering and top-k: first find the clusters using Algorithms 2 or 3, and then find the top-k or max from each cluster using the algorithms in Section 4. Clearly, to guarantee that top-k elements are found from all clusters with probability $\geq 1 - \delta$ given $\delta > 0$, the value of δ in the max or top-k algorithm has to be replaced by $\delta' = \frac{\delta}{J}$ (when the clusters are constructed, the value of J is known). The maximum elements from each cluster can be found by a small modification of Algorithm 2: as each cluster is computed, in addition to type comparisons between two elements, also compare their values; retain the element with larger value. We can also use the algorithms from Section 4 for a monotone error function f to obtain better bounds on the number of questions asked[8].

7. MAX AND TOP-K FOR CONCAVE COST FUNCTIONS

So far we have used the fixed-cost model in which each question incurs unit cost, resulting in a total cost that is the number of comparisons performed. However, when the crowd is used as the oracle, other monotone cost functions may also be applicable. In this section, we study the class of nonnegative monotone concave functions as the cost function.

Definition 7.1. A function g is a nonnegative monotone concave function if: (i) for all N, $g(N) \geq 0$; (ii) for all $N_1 \geq N_2$, $g(N_1) \geq g(N_2)$; and (iii) for any N_1, N_2; and any $t \in [0, 1]$, we have $g(tN_1 + (1-t)N_2) \geq tg(N_1) + (1-t)g(N_2)$.

Nonnegative concave functions are interesting since they grow more slowly than linear cost functions and exhibit the *subadditive property*:

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

Therefore, if many questions are asked together, we pay less than asking the questions one by one. This is reasonable in a crowdsourced setting since, if we ask a person a number of questions at the same time, it is likely to require less effort than asking the questions one by one, requiring her to submit an answer before getting the next question. Such cost functions display an interesting 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 an arbitrary number of questions for some of the cost functions. As an extreme example, for finding the maximum element 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 in just one round for a cost of a; the system will simply return the element that did not lose in any of the comparisons as the maximum. However, this is not a reasonable cost function in the crowdsourced setting since a

[8]Let Δ and Δ' be the distance of two elements having same type in the entire sorted order, and in the sorted order restricted to the cluster containing them, respectively. Since $\Delta \geq \Delta'$, for monotone error function f, the probability of error in value comparisons $\frac{1}{f(\Delta)} \leq \frac{1}{f(\Delta')}$. Therefore, the same error function can be assumed even when the elements in respective clusters are compared.

person can be asked a large number of comparison questions and still will be paid the same. This is not a good strategy for a linear cost function.

Some other standard and more reasonable monotone concave functions are: (i) $g(N) = N^a$, for a constant a, $0 < a \leq 1$; (ii) $g(N) = \log N$ (or $\log^2 N$; and other polylogarithmic functions, or $\log \log N$, etc.); and (iii) $g(N) = 2^{\sqrt{\log N}}$ which grows between N^a for a constant a and polylogarithmic functions. Many optimization problems have been studied under concave cost functions, such as Fotakis and Tzamos [2013] and Guisewite and Pardalos [1991].

Interaction between the system and the oracle. As before, we are assuming there is a system that interacts with the oracle in rounds. In every round i, the system gives a set of pairs of elements S_i (called a batch of questions) to the oracle. The oracle is responsible for performing comparisons between the elements in each pair $(x, y) \in S_i$, and to return whether or not $x < y$. The system looks at the answers, performs some internal computation, decides the set of pairs S_{i+1} for the next round, and continues the process. 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 assume no limit on the maximum size of the batch of questions that can be asked of the oracle at once. In some crowdsourced scenarios, it may be useful to assume an upper bound 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. However, if the maximum batch size is a constant c, the simple tournament algorithm for finding max gives a constant factor approximation for any concave function g. The lower bound of any algorithm to find the max (with no error) is $\frac{(n-1)g(c)}{c}$: $n-1$ comparisons must be performed to find the max, and they cannot be grouped by batch size more than c. On the other hand, the tournament algorithm that compares two elements and proceeds with the winner will incur a cost of $(n-1) \times g(1)$, so we get a constant factor $\frac{cg(1)}{g(c)}$ approximation.

Optimization problem. Under the fixed-cost model, our goal was to minimize the total number of comparisons performed. However, using an arbitrary concave cost function g, the total number of comparisons may not be proportional to the total cost under g—recall the extreme example of $g(N) = a$. Hence, instead of minimizing the total number of comparisons, our goal is now to minimize the sum of the cost under g, where the sum is over all rounds and over all batches of questions asked of the oracle within each round. We will denote the cost of an optimal algorithm by *OPT*.

As we will see shortly, finding an optimal algorithm for an arbitrary concave cost function g is nontrivial even for the simple problem of finding max, and even if there are no comparison errors (i.e., the oracle returns the correct answer to all comparisons). Therefore, we will study approximation algorithms that return the exact answer by paying a total cost not much worse than *OPT*. An algorithm is a *$\mu(n)$-approximation algorithm* for some nondecreasing function μ if, for every input of size n, it can find the solution (e.g., the maximum element) with a cost $\leq \mu(n) \times OPT$.

In the unit cost model, for no comparison errors, we know that $n-1$ comparisons are necessary and sufficient to find the max from n elements, and hence the cost incurred is also $n-1$. However, any standard algorithm (e.g., the tournament algorithm) that uses $n-1$ comparisons uses multiple rounds, and eliminates candidates based on the results of comparisons from previous rounds. Hence the $n-1$ comparisons used by these algorithms do not translate into $g(n-1)$ cost according to our payment model. Nevertheless, $g(n-1)$ serves as a lower bound for *OPT*, since any such algorithm must

use $n-1$ comparisons to find the maximum and, since a nonnegative concave cost function is subadditive, the minimum cost for $n-1$ comparisons is at least $g(n-1)$.

Observation 2. For any nonnegative concave cost function g, $OPT \geq g(n-1)$, to find the maximum among n elements even if there are no comparison errors.

We now present two algorithms for finding max under the concave cost model with no comparison errors, and discuss an extension to top-k. We then discuss algorithms for max using the constant error model (see Section 3). For no comparison errors, the algorithms will give the comparisons to the oracle in batches to exploit the subadditive property of the concave cost function. However, for the constant error model we will repeat each comparison multiple times and take a majority vote, which will require us to give each copy of the same comparison to the oracle in different batches (that simulates comparison by different people) to ensure independence of the answers (see Section 7.2). The results for the constant error model directly apply to the variable error model, since the probability of comparison error for the variable error model is no more than that for the constant one. We leave the clustering problem for concave cost functions as future research[9].

7.1. No Comparison Error Model

We start by considering how the tournament algorithm for finding max, which uses $n-1$ comparisons in $\log n$ rounds (see Figure 2(a) in Section 4), behaves under different concave cost functions g. If the cost function g is polynomial, that is, $g(N) = N^a$ for a constant $0 < a \leq 1$, then the tournament algorithm gives a constant factor approximation. However, but for $g(N) = a$, for a constant $a > 0$, the ratio to OPT is $\Theta(\log n)$ (see Appendix A.4). In fact, for any arbitrary cost function g, the tournament algorithm gives a $\log n$ approximation. On the other hand, as mentioned earlier, for the constant cost function $g(N) = a$ we can do all $\binom{n}{2}$ comparisons in the same round and pay the same cost as $OPT = a$ (the cost incurred by the tournament algorithm and the one-round algorithm for different concave functions are further discussed in Appendix A.4). In this section we will study algorithms that give better than $\log n$ approximation for any concave cost function g.

THEOREM 7.2. *For any arbitrary monotone nonnegative concave function g, there exists an $O(\log \log n)$-approximation algorithm to find the max if there are no comparison errors.*

We prove this by giving two algorithms (Algorithms 4 and 5) that achieve the desired approximation. Algorithm 4 is simple and easy to implement, whereas the more complex Algorithm 5 is useful since it can be modified to an algorithm for finding top-k elements.

Apart from these two algorithms, a result by Valiant [1975] also gives an $O(\log \log n)$ approximation for max, although the algorithm is much more complex to implement than Algorithm 4. Valiant showed that, if $P = n$ processors are available (same as the total number of elements), then the maximum element can be found in $\log \log n + const$ rounds [Valiant 1975, Theorem 2]. In each round, at most $P = n$ comparisons are performed, and therefore the total cost incurred is $O(\log \log n)g(n) = O(\log \log n) \cdot OPT$.

Option 1. A simple $O(\log \log n)$-approximation algorithm for finding max can be constructed by adapting the tournament algorithm so that the depth of the tree is reduced from $\log n$ to $\log \log n$. To do so, it combines the results of many comparisons from the

[9]It can be verified that Algorithm 2 in Section 5 incurs a cost of $O(Jg(n))$, where J is the number of clusters. It will be interesting to find an optimal or good approximation algorithm for arbitrary J and g.

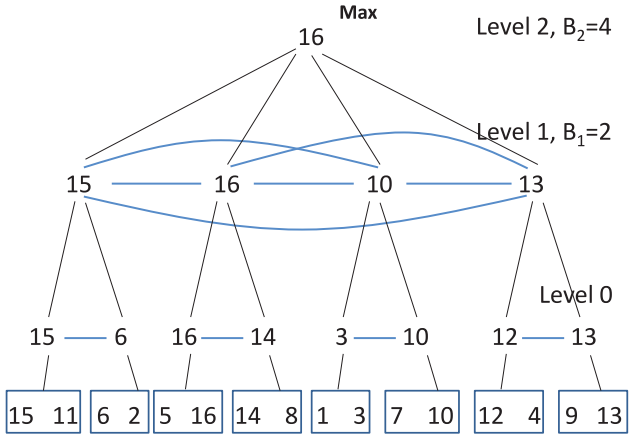

Fig. 4. An illustration of execution of Algorithm 4 First $n = 16$ elements are grouped into $\frac{n}{2} = 8$ pairs, and the larger elements of each pair form the leaves (Level 0) of the comparison tree. In Level 1, $B_1 = 2$ and pairwise comparisons are performed of each group of two elements. In Level 2, $B_2 = B_1^2 = 4$ and all $\binom{4}{2} = 6$ comparisons are performed in the single group of four elements.

ALGORITHM 4: A simple $O(\log \log n)$-approximation algorithm for finding the maximum element.

1 Group n elements into $\frac{n}{2}$ pairs, the leaves (level 0) of the comparison tree contain the larger element from each pair;
2 Let B_h denote the number of children of an internal node in the h-th level ($h \geq 1$). Set $B_1 = 2$ and $B_{h+1} = B_h^2$ for $h > 1$;
3 **for** $h = 1$ *to* $\log \log n$ **do**
4 Group elements from level $h - 1$ into blocks of size B_h;
5 Perform all pairs of comparisons in each of these blocks;
6 Propagate the maximum element from each block (which does not lose any comparison) to an internal node in level i;
7 **end**
8 **return** *the unique element at the root of the tree as the max.*

previous level (see Algorithm 4). We will describe the algorithm using a comparison tree, but the number of children of nodes will increase in higher levels ($h = 0$ denotes the leaves). Let B_h denote the number of children of a node in level h, $h \geq 1$, and N_h denote the total number of nodes in level h, $h \geq 0$. In the lowest level, $N_0 = \frac{n}{2}$. These $\frac{n}{2}$ elements are obtained by grouping n elements into pairs and collecting the larger element by a direct comparison from each pair. In non-leaf levels h, $h \geq 1$, B_h elements from level $h - 1$ are grouped together to form the children of each node, where $B_1 = 2$ and $B_{h+1} = B_h^2$. Further, the max of these B_h elements from level $h - 1$ is propagated to level h: this is achieved simply by doing all $\binom{B_h}{2}$ comparisons along the edges of a complete graph of size B_h, and then doing an offline computation to discard all but a unique element that wins all the comparisons. An example execution of the algorithm is given in Figure 4.

The following observation shows that the number of candidates for max decreases rapidly in each level.

Observation 3. For $h \geq 1$, $B_h = 2^{2^{h-1}}$, and $N_h = \frac{n}{2^{2^h}}$.

PROOF. We prove by induction. The base case holds for $h = 1$. Suppose the hypothesis holds up to level h. In the $h+1$-th level, $B_{h+1} = (B_h)^2 = 2^{2\times 2^{h-1}} = 2^{2^h}$, and $N_{h+1} = \frac{N_h}{B_{h+1}^2} = \frac{n}{2^{2^h}\cdot 2^{2^h}} = \frac{n}{2^{2^{h+1}}}$. □

The following lemma gives the required bound for Theorem 7.2. Note that, to prove the theorem, it suffices to show that the max can be found in $O(\log\log n)$ rounds, where in each round $\leq n$ comparisons are performed. Since g is subadditive, $g(n-1)+g(1) \geq g(n)$ and therefore $OPT \geq g(n-1) \geq g(n) - const$.

LEMMA 7.3. *The number of elements at the level $h = \log\log n$ is 1, and in each level at most n comparisons are performed.*

PROOF. From Observation 3, when $h = \log\log n$, then $N_h = 1$.

Consider any level $h \geq 1$. For each node in level h, $\leq (B_h)^2$ comparisons are performed. From Observation 3, the total number of comparisons is at most $N_h \times (B_h)^2 = \frac{n}{2^{2^h}} \times 2^{2^h} = n$. □

Option 2. The second algorithm is based on Pippenger [1987], who uses a fixed-cost model and gives upper bounds on the number of comparisons for finding max given a bound on the number of rounds. That paper proves the following theorem.

THEOREM 7.4 [PIPPENGER 1987, THEOREM 3]. *For every integer $r \geq 1$, there are explicitly constructed algorithms that select an element of prescribed rank from among n elements in r rounds using $X = O(n^{1+\frac{2^r-2}{3^r-1-2^r-2}}(\log n)^{2-\frac{2^r-1}{3^r-1-2^r-2}})$ comparisons.*

The algorithm presented is based on comparisons along the edges of an *a-expanding graph* in every round: An undirected graph is a-expanding if any two disjoint sets of vertices, each containing at least $a+1$ vertices, are joined by an edge. Based on the results of the comparisons in the first round, an upper bound can be derived on the number of candidates for the desired rank using the properties of an expanding graph. Then the algorithm recursively finds the element of the desired rank in $r-1$ rounds. Both the cost spent in the first round as well as the total cost spent recursively in the other $r-1$ rounds are shown to be $O(X)$.

If we set $r = O(\log\log n)$, the algorithm runs in $O(\log\log n)$ rounds. However, it can be verified that the total cost in the first round itself is $O(n\log^2 n)$. Therefore, we use a preprocessing step to reduce the number of candidates of max from n to $\frac{n}{\log^2 n}$ in $O(\log\log n)$ rounds. Then the number of rounds still remains $O(\log\log n)$, but the cost spent in each round reduces to $g(O(n)) = O(g(n))$, leading to Theorem 7.2. Algorithm 5 combines Pippenger's algorithm with the standard tournament one (using a balanced binary comparison tree) to find the max. The following lemma gives the required bound for Theorem 7.2.

ALGORITHM 5: An $O(\log\log n)$-approximation algorithm for finding the maximum element adapting Pippenger's algorithm.

1 Given n input elements, run tournament algorithm for $2\log\log n$ levels of a balanced binary comparison tree;
2 Given all the candidates for max retained by the tournament algorithm, run Pippenger's algorithm with the number of rounds $r = 2\log\log n$ to find the max;

LEMMA 7.5. *Algorithm 5 finds max in $O(\log\log n)$ rounds, where the cost incurred at each round is $O(g(n)) = O(OPT)$.*

PROOF. Clearly the first phase of the tournament algorithm runs for $O(\log\log n)$ rounds, and in each round $\leq g(n)$ cost is paid (see Figure 2(a)).

After $2\log\log n$ rounds of the tournament algorithm, the number of candidates for max is $N = \frac{n}{2^{2\log\log n}} = \frac{n}{\log^2 n}$. In the second phase where Pippenger's algorithm is used, the number of comparisons performed in any given round is bounded by the total number of comparisons given in Theorem 7.4:

$$O\left(N^{1+\frac{2^{r-2}}{3^{r-1}-2^{r-2}}}(\log N)^2\right) = O\left(N \cdot e^{\frac{\ln N}{(\ln N)^{2(\log_2^3-1)-1}}} \cdot (\ln n)^2\right)$$
$$= O(N \cdot (\ln n)^2) \quad \left(\text{since} 2\left(\log_2^3 - 1\right) - 1 > 1\right) = O(n).$$

Hence the second phase also has $O(\log\log n)$ rounds, and the cost paid at each round is $g(O(n)) = O(g(n)) = O(OPT)$ (for a concave function, $g(cn) \leq cg(n)$). □

Extension to top-k. We now describe a randomized algorithm based on Algorithm 5 that finds all top-k elements for any $k \in [1, n]$ with an expected cost of $O(\log\log n) \cdot OPT$. Note that it suffices to show that the k-th largest element x_k can be found with an expected cost of $O(\log\log n) \cdot OPT$. Then the top-k elements can be found by simply scanning the rest of the $n-1$ elements and finding those greater than x_k (we assumed no two elements have the same value). This additional pass requires $n-1$ comparisons, which can be done in a single round. Therefore this additional round requires cost $g(n-1)$. Furthermore, the optimal cost for finding top-k is also $\geq g(n-1)$, hence the total expected cost for finding all top-k elements is still $O(\log\log n) \cdot OPT$.

The second phase of Algorithm 5 that uses Pippenger's algorithm can select the element of any given rank with the same cost (Theorem 7.4). So it suffices to show that the first phase of Algorithm 5, that reduces the number of candidate elements from n to $N = \frac{n}{\log^2 n}$, can be modified so that the k-th element x_k is retained (although the rank of x_k among the N remaining elements can change from k to k').

We use the recursive procedure (although for a fewer number of rounds) from the standard *randomized-select* algorithm to reduce the number of candidate elements for x_k from n to $N = \frac{n}{\log^2 n}$ [Cormen et al. 2009]. The idea is to choose a pivot p at random (every element is chosen with probability $\frac{1}{n}$); partition the rest of the $n-1$ elements based on whether they are greater or less than the pivot p, which would reveal the rank k_p of the pivot (max has rank 1 and min has rank n). If $k_p = k$, return p as the k-th element; if $k_p > k$, recursively search for rank k among those elements greater than p; and if $k_p < k$, recursively search for rank $k - k_p$ among those elements smaller than p. In randomized-select, this recursive process is continued until the number of elements is exactly 1, which is the k-th largest element. However, we will continue this process only for $R = 16\log\log n$ rounds. Note that in every round $\leq n$ comparisons are performed, so the cost in each round is $\leq g(n)$.

Now we show that the number of candidates after R rounds is $\leq \frac{n}{\log^2 n}$ with high probability. We call a round $i \in [1, R]$ *good* if the rank $k_{p,i}$ of the pivot p_i in round i is $\in [\frac{n_i}{4}, \frac{3n_i}{4}]$, where n_i is the number of candidates in round i, otherwise the round is *bad*. Since the pivot is chosen at random from n_i elements, the rank of the pivot is between $[\frac{n_i}{4}, \frac{3n_i}{4}]$ with exactly probability $= \frac{1}{2}$. Hence the expected number of good rounds is $\frac{R}{2}$. Using Chernoff bounds [Motwani and Raghavan 1995], the number of good rounds is $\geq \frac{R}{4}$ with probability[10] $\geq 1 - e^{-\frac{R}{16}} = 1 - \frac{1}{\log n}$.

[10]Let $X_1, \ldots, X_N$ be independent binary random variables. Let $X = X_1 + \cdots + X_N$ and $\mu = E[X]$. Then, using Chernoff bounds, $\Pr[X \leq \mu(1-\epsilon)] \leq e^{-\frac{\mu\epsilon^2}{2}}$.

Whenever the rank of the pivot $k_{p,i} \in [\frac{n_i}{4}, \frac{3n_i}{4}]$, then the number of candidates decreases by at least a factor of $\frac{1}{4}$ in the following round. If there are $\geq \frac{R}{4}$ good rounds, then the number of candidates after R rounds is $\leq \frac{n}{4^{\frac{R}{4}}} \leq \frac{n}{2^{2\log\log n}} = \frac{n}{\log^2 n}$; this happens with probability $\geq 1 - \frac{1}{\log n}$. However, we ensure that the number of candidates always decreases to $\leq \frac{n}{\log^2 n}$ with an expected cost of $O(g(n)\log\log n)$. To achieve this, we employ this (standard) procedure: run the recursive process for $R = 16\log\log n$ rounds; if the number of candidates after R rounds is $> \frac{n}{\log^2 n}$ (this happens with probability $\leq \frac{1}{\log n}$), simply sort all the n elements and return the k-th element; this can be done with cost $\leq O(g(n)\log n)$ (e.g., by merge-sort, where the number of rounds in merge-where sort is $O(\log n)$, each round requires $O(n)$ comparisons which are done in a single, batch, incurring a cost of $O(g(n))$ in each round). The total expected cost is $\leq g(n)R + O(g(n)\log n) \times \frac{1}{\log n} = O(g(n)\log\log n)$, since $R = O(\log\log n)$. This completes the argument for top-k.

7.2. Constant Error Model

We now study concave cost functions for the constant error model where the probability of getting a wrong answer from the comparison oracle is $\leq \frac{1}{2} - \epsilon$ for a constant $\epsilon > 0$. Recall that Feige et al. [1994] show $O(n\log\frac{1}{\delta})$ comparisons are sufficient to output max with probability $\geq 1-\delta$ under the constant error model (Theorem 4.1). This gives a cost of $O(\log\frac{1}{\delta}) \times OPT$ for the linear cost function $g(N) = N$. The basic idea is to use the tournament algorithm with a binary comparison tree, but repeat each comparison multiple times and decide the larger element by a majority vote; the numbers of comparisons in different levels form a convergent series and the total number of comparisons remains linear in n.

There are two main issues that prohibit us from getting a similar approximation for an arbitrary concave cost function g: (1) as we argued earlier, now we need to make payments level by level, for every batch of questions asked; and (2) if each of N comparisons in a level is repeated M times, they must not be performed in the same batch so that M independent answers are obtained[11]. Therefore, although we perform MN comparisons in that level, we have to call the oracle M times with batches of size N and therefore we have to pay $Mg(N)$, which can be much larger than $g(MN)$ (for instance, when $g(N) = 1$). The following theorem gives a bound on the approximation factor for polynomial[12] and arbitrary concave cost functions g, using the algorithms already mentioned along with the standard technique of taking a majority vote. Whether a better approximation can be obtained is left for future research.

THEOREM 7.6. *Consider the constant error model where each comparison is correct with probability $\geq \frac{1}{2} + \epsilon$. Given $\delta \in (0,1)$, the max can be found with probability $\geq 1-\delta$:*

(1) with a cost of $O(\frac{1}{\epsilon^2}(\log\log n + \log\frac{1}{\delta})) \times OPT$ if $g(N) = N^a$ for a constant a, $0 < a \leq 1$;

(2) with a cost of $O(\frac{1}{\epsilon^2}(\log n + \log\log n \log\frac{\log\log n}{\delta})) \times OPT$ for any arbitrary nonnegative monotone concave cost function g.

[11]We assume the comparisons performed by the oracle in different batches are independent, even if the same pairs of elements are compared. We also assume that the comparisons given to the oracle in the same batch do not include the same pairs of elements more than once and are independent.

[12]The polynomial cost function $g(N) = N^a$, where $0 \leq a < 1$ is a constant, is a natural way of modeling economies of scale.

The next observation, which relates the number of repetitions in a majority vote with the probability of obtaining the correct result, directly follows from Chernoff bounds.

Observation 4. If a comparison is repeated M times and the winner is decided by majority vote, then the probability of error is $\leq e^{-M\frac{\epsilon^2}{1+\epsilon}} \leq e^{-\frac{M\epsilon^2}{2}}$.

Finally we prove Theorem 7.6. We again use the fact that $OPT \geq g(n-1) = g(n) - const$.

PROOF.

For $g(N) = N^a$ *for a constant* a, $0 < a \leq 1$. We use the tournament algorithm with a binary comparison tree and repeat each comparison $M = \frac{2}{\epsilon^2}(\ln\log n + \ln\frac{1}{\delta})$ times[13] in each level. At level h, $2^{\log n - h}$ internal nodes exist ($h = \log n$ contains the root and $h = 0$ contains the leaves). The total cost paid is $\sum_{h=1}^{\log n} Mg(2^{\log n-h}) = \sum_{\ell=1}^{\log n} Mg(2^{\ell-1}) = M\sum_{\ell=1}^{\log n} 2^{a(\ell-1)} = M\frac{2^{a(1+\log n)}-1}{2^a-1} = O(Mn^a) = O(M) \times OPT$. From $h = 1$ to $h = \log n$, the max is compared with $\log n$ elements in total at different levels of the tree. By Observation 4 and by the union bound, max wins all these comparisons with probability $\geq 1 - \frac{\log n}{e^{\frac{M\epsilon^2}{2}}} = 1 - \delta$ and therefore is returned at the root of the tree.

For arbitrary concave cost function g. We use Algorithm 4 from the previous section, but in level $h \geq 1$, each comparison is repeated $M_h = \frac{2}{\epsilon^2}(2^h + \ln\frac{\log\log n}{\delta})$ times. Recall that the number of distinct comparisons at each level is bounded by n (Lemma 7.3). Then the total cost across all levels is bounded by

$$\begin{aligned}\sum_{h=1}^{\log\log n} M_h \times g(n) &= g(n) \times \frac{2}{\epsilon^2} \times \sum_{h=1}^{\log\log n}\left(2^h + \ln\frac{\log\log n}{\delta}\right)\\ &\leq g(n) \times \frac{2}{\epsilon^2} \times \left(\log\log n \ln\frac{\log\log n}{\delta} + 2^{1+\log\log n}\right)\\ &= O\left(\frac{1}{\epsilon^2}\left(\log n + \log\log n \log\frac{\log\log n}{\delta}\right)\right) \times OPT.\end{aligned}$$

Next we argue that the max of n elements survives in all these comparisons. In level h, max is compared with B_h elements. Since $B_h = 2^{2^{h-1}}$, by Observation 4 and the union bound, the total error probability is

$$\sum_{h=1}^{\log\log n} \frac{B_h}{e^{\frac{M_h\cdot\epsilon^2}{2}}} = \sum_{h=1}^{\log\log n} \frac{B_h}{e^{2^h + \ln\frac{\log\log n}{\delta}}} \leq \sum_{h=1}^{\log\log n} \frac{2^{2^h}}{2^{2^h}+\ln\frac{\log\log n}{\delta}} \leq \log\log n \times \frac{\delta}{\log\log n} = \delta. \quad \square$$

8. DISCUSSION

In this article, we studied max/top-k and clustering problems in the presence of a noisy comparison oracle. These problems are motivated by top-k and group-by database queries in the crowdsourced setting, where the criteria used for grouping and ordering are difficult to evaluate by machines but much easier by the crowd (e.g., grouping photos by the individuals featured in them or finding their most recent photo). These queries are evaluated using the oracle to answer either type or value questions.

Two important factors must be modeled in crowdsourced algorithms: error (since the crowd may give erroneous answers) and cost (since users must be incentivized

[13]In this section, ln denotes natural logarithm and log denotes logarithm with base 2 unless mentioned otherwise.

to participate, or there may be cost associated with the crowdsourced platform). To measure error, we adopted the standard constant error model in which each type or value question is answered correctly by the oracle with a constant probability $> \frac{1}{2}$. We then introduced a more interesting variable error model for value questions, in which the error is related to how close the elements are in the ordering of interest. To measure cost, we adopted the standard fixed-cost model in which N questions have a cost of N, and then studied a class of cost models based on concave functions in which asking N questions followed by M questions (or N questions of one person and M questions of another person) is more expensive than asking $N + M$ questions at once of a single person (the subadditive property).

Using these error and cost models, we formalized the max/top-k and clustering problems and gave efficient algorithms guaranteed to achieve the desired results with high probability. For max/top-k queries under the fixed-cost model, we showed that fewer questions are needed in the variable error model compared to the constant error model. For the clustering (group-by) queries under the fixed-cost model, we showed that fewer questions are needed when there is a correlation between the types and values of the elements.

An important contribution of this article over Davidson et al. [2013] is the study of max/top-k in the context of concave cost functions. Since concave cost functions exhibit the subadditive property, there is an interesting tension between the number of rounds used in the algorithm and the total number of questions asked. Finding an optimal algorithm for an arbitrary cost function turns out to be nontrivial, even if there are no comparison errors. We therefore studied approximation algorithms that return the exact answer by paying a total cost not much worse that optimal. In particular, we proved that, for any monotone nonnegative concave function, there exists an $O(\log \log n)$-approximation algorithm to find the max when there are no comparison errors. We then extended one such algorithm to find the top-k. We closed by discussing how to extend the algorithms to the constant error model.

Future research. Our simple error and cost models were motivated by crowdsourcing applications, and allow us to do a thorough analysis to obtain formal bounds on the number of comparisons necessary to achieve correct results. As such, our results form the basis for studying more complex models suitable for real-life crowdsourced applications by providing lower bounds as well as algorithmic ideas. As the next step, several directions of applied and theoretical research can be pursued.

(1) *Experimental validation of our assumptions.* We assumed there exists an underlying and unknown total order on the values of the elements, and that any two elements can be correctly ordered by the comparison oracle with probability $> \frac{1}{2}$. For a human oracle, however, there may be cases where the values of two elements are almost indistinguishable, for instance, if two photos are taken a few minutes apart, it is essentially impossible for a human to correctly order them. Furthermore, we assumed that comparisons performed by the oracle in the same batch are independent, which may not be true for a human oracle. There may also be a bound on the number of comparisons that a crowd member may be willing to perform together, no matter how high the reward, such as for lack of available time, whereas our algorithms for the concave cost functions assume any number of comparisons can be given to the oracle in a single batch. The actual error, cost, and latency models for the crowd are hard to formalize and need to be substantiated with experiments.
(2) *Experimental evaluation of our algorithms.* The applicability of our solutions for practical crowdsourcing applications, like clustering a large image set or finding

top-k best-matched images for a concept, should be evaluated. For the fixed-cost model, we gave upper and lower bounds on the number of comparisons; however, one needs to evaluate how they translate to actual costs like money and query runtime, and how good the final results are for practical purposes. The total number of comparisons in the fixed-cost model remains independent of the actual set of workers that answer the questions; however, performance and efficiency may vary depending on whether a large or small group of crowd workers are employed, and when the (fixed) per-comparison reward is varied. Similarly, one can evaluate how different choices of concave cost functions affect the monetary and response-time costs.

(3) *Estimating additional parameters required by the algorithms.* For the constant error model, we assumed constant rate of error $\frac{1}{2} - \epsilon$, $\epsilon > 0$ is known. Similarly, in the variable error model, we assumed the error function f (or a lower bound f' on f that is also strictly growing) is known, as in Algorithm 1. Therefore as a preprocessing step, we need to estimate f for the crowd workers with a small number of additional comparisons. One simple potential approach that can be evaluated is to start with a set of elements ordered according to their values, give pairs of elements at distance Δ to the workers for different values of $\Delta \in [0, n-1]$, and plot the fraction of incorrect answers received against Δ. Estimating (constant) error rates for crowdsourcing and other applications using machine learning approaches has been studied in the literature [Dawid and Skene 1979; Karger et al. 2011; Li et al. 2013; Lease 2011; Ipeirotis 2011; Raykar et al. 2010] and may be useful for estimating the error function f. Similarly, Algorithm 3, when extended to handle the partial correlation case with arbitrary α, assumes the value of α (or an upper bound on its value) is known. Although an upper bound of $\alpha \leq n$ always holds, a better estimate of α can significantly reduce the number of comparisons. Achieving good estimates of these additional parameters by experiment may be of independent interest.

(4) *Hybrid approaches.* Our framework uses only humans to do the comparisons, whereas in some applications some of the (easier) comparisons can be done computationally, possibly using machine learning techniques, to reduce the time and cost incurred while interacting with the crowd (e.g., Marcus et al. [2011b]). Extending our algorithms to hybrid approaches will be an interesting direction to explore.

There are also several interesting open questions for theoretical research.

(5) *Other objective functions.* There are several other objective functions that one can study. One would be to reduce *latency*, which includes not only the number of rounds but size of the set of comparisons given to a user in one round; it could take a single person a long time to answer $O(n)$ comparisons. We may therefore want to limit the number of comparisons given to a single user in each round. Another objective function would include a budget (e.g., on the number of comparisons or the maximum cost that can be incurred with an arbitrary cost function) and minimize the probability of error.

(6) *Concave cost functions.* With regard to concave cost functions, one can study: (i) whether there is a stronger lower bound on OPT for max/top-k in the no-error and constant error models; (ii) finding optimal max/top-k algorithms for specific concave cost functions[14]; and (iii) exploring the clustering problem in the context of concave cost functions.

[14]In particular, when $g(n) = 2^{\sqrt{\log n}}$, is there an algorithm with cost $\leq c \cdot g(n)$ for a constant c, or is there a lower bound of $\omega(g(n))$ where the cost is strictly asymptotically higher than $g(n)$? We know that an algorithm with cost $\leq c \cdot g(n)$ exists for polynomial cost functions $g(n) = n^a$, where $0 < a \leq 1$ is a constant, that grow faster than $2^{\sqrt{\log n}}$ (tournament algorithm with binary comparison tree). Also such an algorithm with cost

APPENDIXES

A. OMITTED PROOFS

In this section we give the proofs omitted in the previous sections.

A.1. Proof of Lemma 4.7

PROOF. The particular choices of the exponential, linear, or logarithmic functions in the proof ensure that $f(1) \geq 2 + \epsilon'$ for a small constant $\epsilon' > 0$; the results also hold for any other choices of these functions (and functions having steeper growth rates).

Exponential error function. Suppose $f(\Delta) = 1 + 2^{\Delta}$. From (4) and (5), $p_\ell \leq \frac{1}{1+2^{\Delta_\ell}} \leq \frac{1}{2^{\Delta_\ell}}$ and $\Delta_\ell = \frac{(h-\ell+1)\delta(n-1)}{2^{h-1}}$. We set $X = 2^h = \frac{2\delta(n-1)}{\log(2/\delta)}$. With these choices of X, $\sum_{\ell=1}^{h} p_\ell \leq \sum_{\ell=1}^{h} \frac{1}{2^{\Delta_\ell}} \leq \sum_{q=1}^{h} \frac{1}{2^{\frac{q\delta n}{2^h}}}$ (from (6)) $\leq \sum_{q=1}^{\infty} \frac{1}{2^{\frac{q\delta n}{2^h}}} \leq \sum_{q=1}^{\infty} \frac{1}{(2^{\frac{\delta n}{2^h}})^q} \leq \frac{2}{2^{\frac{\delta(n-1)}{2^{h-1}}}} = \frac{2}{2^{\frac{\delta(n-1)}{\log(2/\delta)}}} = \frac{2}{2/\delta} = \delta$.

Therefore $\frac{n}{X} = \frac{n\log(2/\delta)}{2\delta(n-1)} \leq \frac{n\log(2/\delta)}{\delta n} = \log(2/\delta)$.

Linear error function. Suppose $f(\Delta) = \Delta + 2$. We set $X = 2^h = \frac{\delta^2 n}{\log\log n}$. With this choice of X, $\sum_{\ell=1}^{h} p_\ell \leq \sum_{\ell=1}^{h} \frac{1}{\Delta_\ell + 2} \leq \sum_{\ell=1}^{h} \frac{1}{\Delta_\ell} \leq \sum_{q=1}^{h} \frac{1}{\frac{q\delta n}{2^h}}$ (from (6)) $= \frac{2^h}{\delta n} \sum_{q=1}^{h} \frac{1}{q} \leq \frac{2^h \log h}{\delta n}$

$\leq \frac{\delta^2 n}{\log\log n} \times \frac{\log(\log \frac{\delta^2 n}{\log\log n})}{\delta n} \leq \frac{\delta^2 n}{\log\log n} \times \frac{\log\log n}{\delta n} = \delta$.

Therefore $\frac{n}{X} = n \times \frac{\log\log n}{\delta^2 n} = \frac{\log\log n}{\delta^2}$.

Logarithmic error function. Suppose $f(\Delta) = \log\Delta + 3$. We set $X = (2\delta n)^{\frac{\delta}{\delta+1}}$. With this choice of X, $\sum_{\ell=1}^{h} p_\ell \leq \sum_{\ell=1}^{h} \frac{1}{\log\Delta_\ell} \leq \frac{h}{\log(\frac{\delta n}{2^h})}$ (from (7)) $= \frac{\log(\delta n)^{\frac{\delta}{\delta+1}}}{\log\left(\frac{\delta n}{(\delta n)^{\frac{\delta}{\delta+1}}}\right)} \leq \frac{\frac{\delta}{\delta+1}\log(\delta n)}{\log\left(\frac{\delta n}{(\delta n)^{\frac{\delta}{\delta+1}}}\right)}$

$\leq \frac{\frac{\delta}{\delta+1}\log(\delta n)}{\log(\delta n)^{\frac{1}{\delta+1}}} = \delta$.

With this choice of X, $\frac{n}{X} = \frac{n}{(\delta n)^{\frac{\delta}{\delta+1}}} = \frac{n^{\frac{1}{1+\delta}}}{\delta^{\frac{\delta}{\delta+1}}}$. □

A.2. An Upper Bound of $n + O\left(\frac{k^2}{\delta}\right)$ for Exact Comparison

(From Section 4.2) Suppose the answers to the value comparisons are exact. Here we sketch how our techniques presented earlier can find all top-k elements with probability $\geq 1 - \delta$ given $\delta > 0$ using only $n + O(\frac{k^2}{\delta})$ comparisons when $k = o(\sqrt{n})$. As discussed previously, we choose $X = \frac{\delta n}{k^2}$. This ensures that, with probability $\geq 1 - \delta$, all top-k elements appear in different X-trees and therefore survive in the upper levels. In each X-tree, we perform $X - 1$ comparisons to perform the maximum element in it. Clearly, all $x_1, \ldots, x_k$ are chosen as maximum elements in their respective X-trees. The upper level has $\frac{k^2}{\delta}$ elements, and we run the linear-time selection algorithm [Blum et al. 1973] to find x_k. A linear pass on the upper level finds $x_1, \ldots, x_k$ that are larger than x_1. The total number of value comparisons performed is $\leq n + O(\frac{k^2}{\delta})$.

A.3. Proof of Corollary 4.8

Here we show that we obtain the exact top-k elements with probability $\geq 1 - 8\delta$. To obtain the bound of $1 - \delta$, we need to run our algorithm with $\delta' = \delta/8$. We assume $k = o(\sqrt{n})$ as discussed in Section 4.2, otherwise the algorithm in Feige et al. [1994] gives a better bound on the number of value comparisons.

$\leq c \cdot g(n)$ exists for the logarithmic cost function $g(n) = \log n$ that grows slower than $2^{\sqrt{\log n}}$ (all possible $\binom{n}{2}$ comparisons in a single round).

Once again, we start with a random permutation Π and the comparison tree is again divided into upper and lower levels. In the lower levels, n elements in Π are partitioned into $\frac{n}{X}$ number of X-trees. Instead of focusing on the single X-tree that contains the largest element x_1, we consider all those X-trees that contain the top-k elements $x_1, \ldots, x_k$. We show that, with probability $1 - 6\delta$: (A) each of $x_1, \ldots, x_k$ appear in different X-trees so are the maximum elements in their respective X-trees; and (B) none of these elements loses any comparison in its X-tree.

First we consider (A) and argue that no two elements in $x_1, \ldots, x_k$ appear in the same X-tree with probability $\geq 1 - \delta$ if $X \leq \frac{\delta n}{k^2}$. Since Π is a random permutation, any of the other $n - 1$ elements have equal probability of belonging to any fixed leaf of the X-tree containing x_i, for any $i \in [1, k]$. Therefore, by union bound, the probability that this X-tree contains another element from $x_1, \ldots, x_k$ is $\leq \frac{k(X-1)}{n-1} \leq \frac{kX}{n}$, which is $\leq \frac{\delta}{k}$ when $X \leq \frac{\delta n}{k^2}$. We choose $X = \frac{\delta n}{k^2}$. Applying union bound for all x_i, $i \in [1, k]$, with probability $\geq 1 - \delta$ no two top-k elements appear in the same X-tree. From now on, we will consider that the elements $x_1, \ldots, x_k$ belong to separate X-trees.

Now consider (B). Given (A), each x_i, $i \in [1, k]$ is the maximum element in its respective X-tree. We apply Algorithm 1 on all $\frac{n}{X}$ X-trees with $\delta' = \delta/k$, hence the total number of comparisons $= (\frac{n}{X}) \times X + o(X) \times O(\frac{1}{\delta'} \log \frac{1}{\delta'}) = n + o(n) \times O(\frac{k}{\delta} \log \frac{k}{\delta})$. By Theorem 4.2, all of $x_1, \ldots, x_k$ are decided as the maximum elements in their respective X-trees with probability $\geq 1 - 6\delta/k$. By union bound, all of them go to the upper levels with probability $\geq 1 - 6\delta$.

In the upper levels, we employ the algorithm given in Feige et al. [1994] to find the top-k elements that are still $x_1, \ldots, x_k$. Since these upper levels have $\frac{n}{X}$ elements, $O(\frac{n}{X} \log \frac{k}{\delta}) = O(\frac{k^2}{\delta} \log \frac{k}{\delta})$ comparisons suffice to find $x_1, \ldots, x_k$ with probability $1 - \delta$.

Combined with the number of comparisons in the lower levels and the bad probabilities from (A) and (B), with probability $\geq 1 - 8\delta$ the top-k elements are found with the stated number of comparisons. For error functions $f(\Delta) = \Omega(\Delta)$ or $f(\Delta) = 2^{\Delta}$, better bounds can be obtained by using Lemma 4.7 in the lower levels.

A.4. Tournament and One-Round Algorithm for Different Concave Functions

(From Section 7.1) Consider no comparison errors. Here we discuss two algorithms for finding max and compare them with an optimal algorithm for different concave cost function g: (i) the tournament algorithm discussed in Section 4 where, at each node of the comparison tree, only one comparison is performed, and in the ℓ-th level, $\ell \in [1, \log n]$, $2^{\ell-1}$ comparisons are performed in total; and (ii) the one-round algorithm, a trivial algorithm for finding max that performs all $\binom{n}{2}$ comparisons for all pairs in the same round and outputs the unique element that wins in all $n - 1$ comparisons. Recall the cost of the optimal algorithm $OPT \geq g(n - 1)$ (see Observation 2).

(1) For any arbitrary concave cost function g, the tournament algorithm incurs a cost of $\sum_{\ell=1}^{\log n} g(2^{\ell-1}) \leq \sum_{\ell=1}^{\log n} g(n-1) = \log n \cdot g(n-1) \leq \log n \cdot OPT$, hence gives an $\log n$-approximation for any arbitrary g.

(2) If $\mathbf{g(x)} = \mathbf{x^a}$, where $1 \geq a > 0$ is a constant, the tournament algorithm incurs a cost of $\sum_{\ell=1}^{\log n} g(2^{\ell-1}) = \sum_{\ell=1}^{\log n} 2^{a(\ell-1)} = \frac{2^{a(1+\log n)}-1}{2^a-1} \leq c' n^a$.
On the other hand, $OPT \geq (n-1)^a$, which gives a constant factor approximation.

(3) If $\mathbf{g(x)} = \mathbf{2^{\sqrt{\log x}}}$:
   (a) the tournament algorithm incurs a cost of $\sum_{\ell=1}^{\log n} g(2^{\ell-1}) = \sum_{\ell=1}^{\log n} 2^{\sqrt{\log 2^{(\ell-1)}}} = \sum_{\ell=1}^{\log n} 2^{\sqrt{\ell-1}} \leq \int_{\ell=1}^{\log n} 2^{\sqrt{\ell-1}} = \leq c'(\sqrt{\log n - 1}) 2^{\sqrt{\log n - 1}}$,
   whereas $OPT \geq 2^{\sqrt{\log(n-1)}}$, which gives an $O(\sqrt{\log n - 1})$-factor approximation;

(b) if we do all $\binom{n}{2}$ comparisons in a single round, the cost is $\leq g(n^2) = 2^{\sqrt{2\log n}}$, and the ratio to OPT is $2^{O(\sqrt{\log n})}$.

Here the tournament algorithm is better than the one-round algorithm.

(4) If $\mathbf{g}(\mathbf{x}) = \log \mathbf{x}$:
   (a) the tournament algorithm incurs a cost of $\sum_{\ell=1}^{\log n} g(2^{\ell-1}) = \sum_{\ell=1}^{\log n} \log 2^{(}\ell - 1) = \sum_{\ell=1}^{\log n}(\ell - 1) = \frac{\log n(\log n - 1)}{2}$,
   whereas $OPT \geq \log(n-1)$, which gives an $O(\log n)$-factor approximation;
   (b) but now doing all $\binom{n}{2}$ comparisons in a single round has a cost of $\leq g(n^2) = 2\log n$, which is a constant factor approximation.

Here the one-round algorithm is better than the tournament algorithm.

(5) If $\mathbf{g}(\mathbf{x}) = \mathbf{a}$, where $a > 0$ is a constant:
   (a) the tournament algorithm incurs a cost of $\sum_{\ell=1}^{\log n} g(2^{\ell-1}) = a \log n$, whereas $OPT = a$, which gives a $\Theta(\log n)$-factor approximation;
   (b) here the one-round algorithm is optimal.

ACKNOWLEDGMENTS

We thank Benoit Groz for very helpful comments, and the anonymous referees for their valuable feedback.

REFERENCES

Miklós Ajtai, Vitaly Feldman, Avinatan Hassidim, and Jelani Nelson. 2009. Sorting and selection with imprecise comparisons. In *Proceedings of the 36th International Colloquium on Automata, Languages, and Programming (ICALP'09)*. 37–48.

Paul André, Michael Bernstein, and Kurt Luther. 2012. Who gives a tweet? Evaluating microblog content value. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'12)*. ACM Press, New York, 471–474.

Paul André, Aniket Kittur, and Steven P. Dow. 2014. Crowd synthesis: Extracting categories and clusters from complex data. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW'14)*. 989–998.

Paul Andre, Haoqi Zhang, Juho Kim, Lydia B. Chilton, Steven P. Dow, and Robert C. Miller. 2013. Community clustering: Leveraging an academic crowd to form coherent conference sessions. In *Proceedings of the AAAI Conference on Human Computation and Crowdsourcing (HCOMP'13)*.

Eyal Baharad, Jacob Goldberger, Moshe Koppel, and Shmuel Nitzan. 2011. Distilling the wisdom of crowds: Weighted aggregation of decisions on multiple issues. *Auton. Agents Multi-Agent Syst.* 22, 1, 31–42.

Michael Ben-Or. 1983. Lower bounds for algebraic computation trees. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing (STOC'83)*. 80–86.

Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. 1973. Time bounds for selection. *J. Comput. Syst. Sci.* 7, 4, 448–461.

Rubi Boim, Ohad Greenshpan, Tova Milo, Slava Novgorodov, Neoklis Polyzotis, and Wang-Chiew Tan. 2012. Asking the right questions in crowd data sourcing. In *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE'12)*. 1261–1264.

Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd International Conference on Machine Learning (ICML'05)*. 89–96.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* 3rd Ed. The MIT Press.

Susan B. Davidson, Sanjeev Khanna, Tova Milo, and Sudeepa Roy. 2013. Using the crowd for top-k and group-by queries. In *Proceedings of the 16th International Conference on Database Theory (ICDT'13)*. 225–236.

Alexander Philip Dawid and Allan M. Skene. 1979. Maximum likelihood estimation of observer error-rates using the em algorithm. *J. Royal Statist. Soc. Appl. Statist.* 28, 1, 20–28.

Uriel Feige, Prabhakar Raghavan, David Peleg, and Eli Upfal. 1994. Computing with noisy information. *SIAM J. Comput.* 23, 5, 1001–1018.

Alberto Fernández and Sergio Gómez. 2008. Solving non-uniqueness in agglomerative hierarchical clustering using multidendrograms. *J. Classificat.* 25, 1, 43–65.

Dimitris Fotakis and Christos Tzamos. 2013. Strategy proof facility location for concave cost functions. In *Proceedings of the 14th ACM Conference on Electronic Commerce (EC'13).* 435–452.

Michael J. Franklin, Donald Kossmann, Tim Kraska, Sukriti Ramesh, and Reynold Xin. 2011. CrowdDB: Answering queries with crowdsourcing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11).* 61–72.

Ryan Gomes, Peter Welinder, Andreas Krause, and Pietro Perona. 2011. Crowdclustering. In *Proceedings of the Neural Information Processing Systems Conference (NIPS'11).* 558–566.

Geoffrey M. Guisewite and Panagote M. Pardalos. 1991. Algorithms for the single-source uncapacitated minimum concave-cost network flow problem. *J. Global Optim.* 1, 3, 245–265.

Stephen Guo, Aditya Parameswaran, and Hector Garcia-Molina. 2012. So who won? Dynamic max discovery with the crowd. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12).* 385–396.

Hannes Heikinheimo and Antti Ukkonen. 2013. The crowd-median algorithm. In *Proceedings of the AAAI Conference on Human Computation and Crowdsourcing (HCOMP'13).*

Panos Ipeirotis. 2011. Crowdsourcing using mechanical turk: Quality management and scalability. In *Proceedings of the 8th International Workshop on Information Integration on the Web in Conjunction with the Conference on World Wide Web (IIWeb/WWW'11).*

David R. Karger, Sewoong Oh, and Devavrat Shah. 2011. Iterative learning for reliable crowdsourcing systems. In *Proceedings of the Neural Information Processing Systems Conference (NIPS'11).*1953–1961.

Matthew Lease. 2011. On quality control and machine learning in crowdsourcing. In *Proceedings of the 3rd Human Computation Workshop (HCOMP'11).* AAAI, 97–102.

Fei-Fei Li and Pietro Perona. 2005. A bayesian hierarchical model for learning natural scene categories. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05).* Vol. 2. 524–531.

Hongwei Li, Bin Yu, and Dengyong Zhou. 2013. Error rate bounds in crowdsourcing models. http://arxiv.org/pdf/1307.2674.pdf.

Tie-Yan Liu. 2009. Learning to rank for information retrieval. *Found. Trends Inf. Retr.* 3, 3, 225–331.

Xuan Liu, Meiyu Lu, Beng Chin Ooi, Yanyan Shen, Sai Wu, and Meihui Zhang. 2012. CDAS: A crowdsourcing data analytics system. *Proc. VLDB Endow.* 5, 10, 1040–1051.

Adam Marcus, Michael S. Bernstein, Osama Badar, David R. Karger, Samuel Madden, and Robert C. Miller. 2011a. Twitinfo: Aggregating and visualizing microblogs for event exploration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'11).* 227–236.

Adam Marcus, Eugene Wu, David R. Karger, Samuel Madden, and Robert C. Miller. 2011b. Human-powered sorts and joins. *Proc. VLDB Endow.* 5, 1, 13–24.

Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms*. Cambridge University Press, New York.

Aditya Parameswaran, Hyunjung Park, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. 2011. Deco: Declarative crowdsourcing. http://ilpubs.stanford.edu:8090/1015/.

Aditya G. Parameswaran, Hector Garcia-Molina, Hyunjung Park, Neoklis Polyzotis, Aditya Ramesh, and Jennifer Widom. 2012. CrowdScreen: Algorithms for filtering data with humans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'12).* 361–372.

Nicholas Pippenger. 1987. Sorting and selecting in rounds. *SIAM J. Comput.* 16, 6, 1032–1038.

Vassilis Polychronopoulos, Luca De Alfaro, James Davis, Hector Garcia-Molina, and Neoklis Polyzotis. 2013. Human-powered top-k lists. In *Proceedings of the International Workshop on the Web and Databases (WebDB'13).* 25–30.

Filip Radlinski and Thorsten Joachims. 2007. Active exploration for learning rankings from clickthrough data. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'07).* ACM Press, New York, 570–579.

Cyrus Rashtchian, Peter Young, Micah Hodosh, and Julia Hockenmaier. 2010. Collecting image annotations using amazon's mechanical turk. In *Proceedings of the NAACL HLT Workshop on Creating Speech and Language Data with Amazon's Mechanical Turk (CSLDAMT'10).* Association for Computational Linguistics, 139–147.

Mohammad Rastegari, Chen Fang, and Lorenzo Torresani. 2011. Scalable object-class retrieval with approximate and top-k ranking. In *Proceedings of the International Conference on Computer Vision (ICCV'11).* 2659–2666.

Vikas C. Raykar, Shipeng Yu, Linda H. Zhao, Gerardo Hermosillo Valadez, Charles Florin, Luca Bogoni, and Linda Moy. 2010. Learning from crowds. *J. Mach. Learn. Res.* 11, 1297–1322.

Joachim Selke, Christoph Lofi, and Wolf-Tilo Balke. 2012. Pushing the boundaries of crowd-enabled databases with query-driven schema expansion. *Proc. VLDB Endow.* 5, 6, 538–549.

Leslie G. Valiant. 1975. Parallelism in comparison problems. *SIAM J. Comput.* 4, 3, 348–355.

Petros Venetis, Hector Garcia-Molina, Kerui Huang, and Neoklis Polyzotis. 2012. Max algorithms in crowdsourcing environments. In *Proceedings of the 21st International Conference on World Wide Web (WWW'12).* 989–998.

Jiannan Wang, Tim Kraska, Michael J. Franklin, and Jianhua Feng. 2012. CrowdER: Crowdsourcing entity resolution. *Proc. VLDB Endow.* 5, 11, 1483–1494.

Steven Euijong Whang, Peter Lofgren, and Hector Garcia-Molina. 2013. Question selection for crowd entity resolution. *Proc. VLDB Endow.* 6, 6, 349–360.

Jinfeng Yi, Rong Jin, Anil K. Jain, Shaili Jain, and Tianbao Yang. 2012. Semi-crowdsourced clustering: Generalizing crowd labeling by robust distance metric learning. In *Proceedings of the Neural Information Processing Systems Conference (NIPS'12).* 1781–1789.

Received October 2013; revised June 2014; accepted October 2014