Answering Queries using Humans, Algorithms and Databases

Aditya Parameswaran
Stanford University
adityagp@cs.stanford.edu

Neoklis Polyzotis
Univ. of California at Santa Cruz
alkis@cs.ucsc.edu

ABSTRACT

For some problems, human assistance is needed in addition to automated (algorithmic) computation. In sharp contrast to existing data management approaches, where human input is either ad-hoc or is never used, we describe the design of the first declarative language involving human-computable functions, standard relational operators, as well as algorithmic computation. We consider the challenges involved in optimizing queries posed in this language, in particular, the tradeoffs between uncertainty, cost and performance, as well as combination of human and algorithmic evidence. We believe that the vision laid out in this paper can act as a roadmap for a new area of data management research where human computation is routinely used in data analytics.

Keywords

Human Computation, Crowdsourcing, Declarative Queries, Query Optimization, Uncertain Databases

1. INTRODUCTION

Large-scale human computation (or, crowd-sourcing) services, such as Mechanical Turk [3], oDesk [5] and SamaSource [6], harness human intelligence in order to solve tasks that are relatively simple for humans–identifying and recognizing concepts in images, language or speech, ranking, summarization and labeling [10], to name a few–but are notoriously difficult for algorithms. By carefully orchestrating the evaluation of several simple tasks, enterprises have also started to realize the power of human computation for more complex types of analysis. For instance, Freebase.com has collected over 2 million human responses for tasks related to data mining, cleansing and curation [24]. Several startups, such as CrowdFlower [1], uTest [7] and Microtask [4], have developed a business model on task management for enterprises. In addition, the research community has developed libraries that provide primitives to create and manage human computation tasks and thus enable programmable access to crowd sourcing services [25, 20, 26].

These recent programmatic methods are clearly important and useful, but we believe that they cannot scale to the increasing complexity of enterprise applications. Essentially, these procedural methods suffer from the same problems that plagued data management applications before declarative query languages. The onus falls on the programmer/developer to manage which tasks are created and evaluated, deal with discrepancies in the received answers, combine these answers with existing databases, and so on. Instead, we propose a declarative approach: the programmer specifies what must be accomplished through human computation, and the system transparently optimizes and manages the details of the evaluation.

Complex Task Example. We will use an example inspired by the popular restaurant aggregator service Yelp in order to illustrate the spirit of our approach. Assume that Yelp wishes to display a few images for each restaurant entry, picked from a big database of user-supplied images. The selected images must not be dark, must not be copyrighted, and they must either display food served in the restaurant or display the exterior of the restaurant with its name in view.

Yelp's developer can write a procedural application that accesses each image in the database, checks on its copyright restrictions (perhaps using another stored database), and then applies the remaining predicates using a crowd-sourcing service like Mechanical Turk. The application may use a toolkit like Turkit [25] in order to interface with Mechanical Turk. In addition to using the Mechanical Turk, the application may use computer vision algorithms in order to determine whether an image is dark or not, to augment human computation with algorithmic computation of lower latency. Of course, the algorithms may not be able to handle accurately all images, and hence the developer must make a judicious choice between the two options for different images. He/she must also orchestrate the composition of these simple tasks and determine the order in which they are evaluated and on which images. The developer also has to ensure that Mechanical Turk tasks are priced appropriately and that the total budget remains within limits. Furthermore, he/she has to deal with discrepancies in the answers returned by human workers–not all people may agree on what is a dark image–and to determine whether the uncertainty in the answers is acceptable.

Overall, this simple example illustrates the large complexity in developing even simple applications. It is straightforward to see that this programming model is unsustainable for complex tasks that need to combine information from databases, human computation and algorithms.

The Declarative Approach. Our proposed approach views the crowd sourcing service (CrSS for short) as another database where facts are computed by human processors. By promoting the CrSS to a first-class citizen on the same level as extensional data, it is possible to write a declarative query that seamlessly combines information from both. In our example, the developer has to only specify that the in-database images will be filtered with predicates

5th Biennial Conference on Innovative Data Systems Research (CIDR '11) January 9–12, 2011, Asilomar, California, USA.

that are computed using the Mechanical Turk or algorithms. The system becomes responsible for optimizing the order in which images are processed, the order in which tasks are scheduled, whether tasks are handled by algorithms or a CrSS, the pricing of the latter tasks, and the seamless transfer of information between the database system and the external services. Moreover, it provides built-in mechanisms to handle uncertainty, so that the developer can explicitly control the quality of the query results. Using the declarative approach, we can facilitate the development of complex applications that combine knowledge from human computation, algorithmic computation, and data.

Existing database technology can serve as the starting point to realize this novel paradigm, but we believe that new research is required in order to fully support it. Specifically, the declarative access of crowd sourcing services has a unique combination of features: the query processor must optimize both for performance and monetary cost (tasks posted to the CrSS cost money); the obtained data is inherently uncertain, since humans make mistakes or have different interpretations of the same task; the resulting uncertainty cannot be handled by existing techniques; and, the latency of human- and algorithmic-based computation is significant and also unpredictable. This combination is sufficiently complex to require a redesign of the query processor, creating fertile ground for new and innovative research in data management.

The goal of this paper is to identify this research opportunity, perform an initial demarkation of the problem space, and to propose some concrete research directions.

2. CROWD-SOURCING SERVICES: BASICS

A CrSS allows users to post unit *tasks* for humans to solve. A task comprises two main components, namely, a description and a method for providing an *answer*. For instance, consider the task of examining an image in order to determine whether it depicts a clean location. The description may be formed by the image along with the text "Is this location clean?", and the method for providing an answer can be simply a Yes/No radio button. The CrSS is responsible for posting the task and returning the answers of the human workers back to the application.

An examination of popular crowd-sourcing services reveals a wide variety of posted tasks. The degree of difficulty varies widely as well, from simple tasks that require a few minutes of work, to elaborate tasks that may take several hours to complete. Such complex tasks typically involve content creation, e.g., write an article on a topic or review a website. In this paper, we focus on simple tasks for two reasons: they can form the building blocks of more complex tasks, and due to their short duration they are more likely to be picked up by human workers. Extending our ideas to complex creative tasks is an interesting direction for future work.

There are three more components to a unit task: (a) the monetary reward for providing the answer, (b) the number of workers who must solve the task independently before it is deemed completed (in order to reduce the possibility of mistakes), and (c) any criteria that a worker must fulfill in order to attempt the task. Several previous studies have examined the effect of these components on the quality of the obtained answers, and have developed guidelines for their configuration [28, 21, 22]. In this paper, we adopt these guidelines and do not consider these components further.

It is important to note that, depending on the particular CrSS, there may be significant latency in obtaining answers to posted tasks. However, we can expect this latency to be reduced in subsequent years, as crowd-sourcing services gather a larger population of human workers and provide better interfaces for matching tasks to interested workers. In the same direction, several startup companies are aiming to create a more efficient marketplace for matching requesters to workers. (It is interesting to note that there has been a recent surge in crowd-sourcing startups, with more than 20 companies established in the past 5 years.)

3. DECLARATIVE QUERIES OVER HUMANS, ALGORITHMS AND DATA

The crux of our proposal is to employ a declarative query language in order to describe complex tasks over human processors, data and algorithms. In this section, we describe more concretely the features of this language and lay the groundwork for defining its semantics and evaluation methods in subsequent sections.

Query Model. We illustrate the query model using a Datalog-like formalism. We stress that this is done for notational convenience and not because we rely on Datalog's features. (It is unlikely that we will encounter recursive tasks in practice.)

Let us consider the following example complex task: *Find photos that are large, in jpeg format, and depict a clean beach or a city that is neither dangerous nor dirty.* The task can be encoded as the following declarative query in our formalism:

```
travel(I):= rJpeg(I), hClean(I), hBeach(I), aLarge(I)
travel(I):= rJpeg(I), hClean(I), haCity(I,Y), rSafe(Y)
```

Let us explain each construct in more detail, starting with the first rule and examining the predicates left-to-right.

- We assume an extensional database table, named `rJpeg`, that contains jpeg images. Hence, `rJpeg(I)` becomes true for every binding of `I` to an input image. (We overload `I` to represent both the image ID as well as the image itself, in order to simplify notation. The use should be clear from the context.)
- The predicate `hClean` is termed an h-predicate, as it is evaluated using a CrSS. The idea is that, for a given binding to variable `I`, `hClean(I)` is evaluated by posting a task to the CrSS–"Does this image depict a clean location?"–and retrieving the answer. We discuss the instantiation of such tasks later.
- Similarly, there is an h-predicate `hBeach(I)` for checking whether the image depicts a beach, which is evaluated with separate tasks.
- Finally, the predicate `aLarge` is termed an a-predicate and is evaluated by invoking an algorithm with the binding for `I` as an input argument. For instance, the algorithm may examine the image file to compute the number of pixels.

The second rule employs the same first two predicates and an additional extensional predicate `rSafe` which is a table of known safe cities. An interesting feature here is `haCity` which is a *hybrid* ha-predicate–it can be evaluated by a human worker or by invoking a specific algorithm. In the case of a human worker, the task will show the image bound to variable `I` and will ask for the name of the depicted city. This value will be returned in variable `Y`. An algorithmic evaluation will take `I` as input and generate `Y` as output.

We assume that the application developer provides templates in order to map a h-predicate to unit tasks, as described in §2. While unit task design is an extremely important issue, we expect to leverage work from the HCI and Questionnaire Design communities to design effective user interfaces corresponding for each unit task. The choice of the unit tasks posed for a given h-predicate can have an effect on both cost and latency, as we discuss in §6.4. To simplify presentation, we assume a fixed mapping from h-predicates to unit tasks for the remainder of the paper.

Consider the example of `haCity(I, Y)`. In this case, the template may have an image placeholder populated with the binding for `I`, a message "Which city (if any) is depicted in this image?", and a text box to write the name of the city. The value of the text box becomes the binding for `Y`. Similar templates are provided for a-predicates. Clearly, the provided templates also provide bound/free annotations for the variables in Datalog predicates. Returning to the example of `haCity(I,Y)`, it is meaningful to bind `X` and let the humans compute `Y`, while the other way around does not make much sense. These annotations clearly constrain the order in which predicates can be evaluated.

Semantics for hQuery. How do we define the results of an hQuery? This would be straightforward if we were dealing with a regular Datalog query: we can define what constitutes a correct output tuple, and then define the semantics of the query as the complete set of correct output tuples. However, neither of these definitions are obvious in our context.

The definition of a *correct* output tuple is not straightforward, since discrepancies arise naturally in the context of h- and a-predicates. Let us consider again the example of travel images. Different human workers may have different notions of cleanliness, hence giving different answers for the same `hClean(I)` predicate. It may also be possible to make mistakes, e.g., answer `hCity(I,Y)` with the wrong city for a given image. To complicate matters further, discrepancies may be correlated, e.g., if a human worker answers negatively for `hClean(I)` when `I` is bound to a beach image, then he/she may be more likely to answer positively when `I` is bound to a city image. Overall, it may be possible to both assert and negate `travel(I)` depending on which human workers or algorithms are used to evaluate the predicates.

Clearly, it is necessary to define a "consolidation" mechanism for discrepancies in order to define correctness. A simple mechanism might be majority voting – have several human workers (or algorithms) tackle the same question and then take the most frequent answer. We may even employ a more elaborate scheme where (potentially conflicting) answers are assigned probabilities, and correct answers are defined based on a probability threshold. In the coming sections, we consider several possibilities of varying complexity and scope. For the time being, we assume that some mechanism is employed to define what constitutes a correct output tuple.

The second complication is that it may not be desirable to obtain all output tuples. In some cases, the application may impose a budget on the total cost of CrSS tasks, which limits the number of h-predicates that can be evaluated and hence the number of output tuples. In other cases, it may be sufficient to obtain any k output tuples, for some fixed k, in order to avoid the high latency and monetary cost of computing all output tuples. We thus arrive at three different ways that the application can obtain the output of an hQuery:

1. Return all correct output tuples while minimizing the number of h-predicate evaluations and total time
2. Return any k correct output tuples while minimizing the number of h-predicate evaluations and total time, for fixed k.
3. Maximize the number of correct output tuples and minimize total time for a maximum of m h-predicate evaluations, for fixed m.

These definitions makes the simplifying assumption that all tasks are priced equally. This is a reasonable compromise, since applications are likely to use well established guidelines for pricing unit tasks [21, 28]. However, the problem of dynamic pricing is still interesting and we consider as a possible extension of our framework.

Other Examples. We now present three other examples to illustrate various aspects of our declarative query model.

Example 1: Consider the query: *Find all images of people at the scene of the crime who have a known criminal record.* This query can be written as:

```
names(A, I):= rCriminal(Y, A), rCand(I), haSim(I,Y)
```

The EDB table `rCriminal` contains images of known criminals as well as their names. The candidate table `rCand` contains images of people witnessed at the scene of a crime. The hybrid predicate `haSim` tries to evaluate if the person in the candidate table is also present in the criminal table by performing a fuzzy image match. This corresponds to the unit task " Do these two images depict the same person?". Alternatively, a face recognition algorithm may be used to determine whether the two images are of the same person.

Note also that our unit tasks need not have a one-one relationship with the human predicates. We consider this dimension of query optimization in more detail in §6.4. For instance, we may post k images from the candidate table in one column as well as k images from the criminal table in the second column and pose the task description "Match images from the first column to images from the second column that are of the same person." In this case, the relationship is one-many.

The results of a similarity join query such as the one given above can be used to derive a result for entity resolution, clustering or projection (with duplicate elimination).

Example 2: Subsequently, in results of the query above, we may wish to find the best image corresponding to each potential criminal `A`. This query can be posed as follows:

```
topImg(A, hBest(<I>)):= names(A, I)
```

In this query, we use an aggregation (i.e., group by) function on the `A`, and across all images for a given `A`, output the one that is the best (as evaluated by humans). Thus, the aggregation function applied on all the images for a given `A` is the h-predicate `hBest`. Note that in this case, we may need to post several tasks to ascertain which image is the best for a given name `A`. Thus the relationship is many-one between the posted tasks and the h-predicate.

Example 3: Sorting is another primitive that we would like to use in our queries. For instance, we may wish to sort the results corresponding to each `A` in the previous query instead of returning a single best image. Sorting can be expressed using additional syntactic machinery, as is done in prior work in datalog. The results of the sort are stored in a successor intensional database `succ(I, I')`, i.e., `I` appears immediately before `I'` in a sorted order. We can apply a selection condition to the results of a sorting operation to obtain the top-k results, for a given k.

Sorting and top-k are examples of second-order predicates, discussed in more detail in §6.3.

4. FIRST STEP: CERTAIN ANSWERS

Having defined the query language at a logical level, we consider next the design of a query processor to evaluate such queries.

Our starting assumption is that human processors and algorithms always answer questions precisely, i.e., no mistakes or discrepancies are observed. In other words, there is no need to ask a given question to more than one human or algorithm. The adopted assumption simplifies the semantics of the query language, as a query answer is any tuple that is derived through the query's rules based on the answers of humans and algorithms and the extensional database.

The certain-answers assumption is clearly unrealistic in several scenarios, but, as we discuss later, even this simplified case raises

interesting technical challenges for the design of a query processor. We lift this assumption in §5, when we discuss the evaluation of queries under various models of uncertainty.

Design Space. As we argued in §1, a conventional optimize-then-execute design is not appropriate for our problem, since it is very difficult to model the cost and distribution of answers from human processors and algorithms. We are thus led naturally to an adaptive query processing scheme, where any optimization is based on run-time observations.

What should the engine optimize for? Performance is clearly important, since in all cases we wish to minimize total running time. At the same time, it is equally important to optimize carefully the questions issued to the CrSS, since we also wish to minimize the total number of questions or we may even have a bound on the total number of questions. Thirdly, we may also want to maximize the number of correct output tuples when the number of questions is fixed. Essentially, we have a two-criteria optimization problem that involves two out of time, monetary cost and number of correct output tuples. This is a unique property of our problem statement that separates it from previous works in adaptive query processing.

Consider the case when we wish to retrieve all correct tuples. Selecting which questions to ask is a non-trivial problem and is in itself a very interesting direction for future work in this area. We revisit this point at the end of section where we outline some initial research directions. Based on our own work in this area [30], we can point out an interesting trade-off: we may employ a computationally expensive strategy, which adds to the total completion time of the query, in order to carefully select where to ask the fewest possible questions, or we may choose a cheaper strategy at the expense of asking more questions than necessary. It is very difficult to collapse the two dimensions in a single measure, and hence two different question-selection strategies may simply be incomparable. However, note that interesting strategies will lie along a skyline of low computational overhead and low number of questions.

Query Processing Engine. Based on the previous principles, we envision a query processing engine that operates in successive stages. In each stage, the engine performs computation solely on predicates of a specific type, i.e., intensional predicates, h-predicates, or a-predicates. For instance, the engine may first compute some intermediate results by using relational processing solely on extensional data, then issue some questions to the CrSS, then compute more intermediate results using relational operators, then evaluate a-predicates, continue with CrSS questions, and so on. (Since obtaining results for h-predicates may involve high latency, they will be evaluated asynchronously.) Hence, each stage generates more results by building on the available results of previous stages.

The stage-based approach separates the evaluation of predicates of different types and hence isolates the optimization decisions at two points: inter- and intra-stage. Inter-stage optimization determines which type of stage to run next and with what resources. For instance, consider a query that aims to maximize the number of answers under a limited number of questions. An intuitive optimization strategy may favor stages of relational and algorithmic processing, in order to generate query answers for "free", and will invoke CrSS stages infrequently and with a small number of questions each time. The choice of relational vs algorithmic stages may be driven by run-time statistics on the rate of generated results, since the aim is to maximize the number of query answers.

Intra-stage optimization depends on the type of the stage. For a purely relational stage, we can reuse any of the existing adaptive query processing techniques that optimize for performance. For a CrSS or an algorithmic stage, we need to develop strategies to select what questions to ask, as described earlier. These strategies can optimizer eithe for performance, in the case of algorithmic stages, or for a combination of performance and number of issued questions, in the case of CrSS stages.

What questions should be asked? Intuitively, we wish to ask questions whose answers provide the most "information gain", but the latter is non-trivial to quantify. However, we can define some useful heuristics that approximate this notion of gain.

- It is useful to prioritize questions based on the number of affected rules. For instance, we may prefer to ask about `hClean(`α`)` compared to `hClean(`β`)` if the former will enable the processing of more query rules.
- If we are interested in producing all the query answers, then it is prudent to ask questions that will reduce the volume of data to be processed in subsequent stages. Hence, selective questions are preferred.
- If, instead, for a fixed budget, we are interested in maximizing the number of query answers, then we want to ask questions whose answers are likely to lead to more results being generated in subsequent stages. Hence, non-selective questions are preferred.

The last two heuristics require some estimation of selectivity, which can be done in an obvious way based on observations on past questions. An interesting twist is to also consider data similarity in order to obtain more accurate estimates. For instance, consider again the choice between `hClean(`α`)` and `hClean(`β`)`, and assume that we are interested in non-selective questions (i.e., maximizing the number of query answers). Given a suitable data similarity function among image objects, we can examine which of α or β bears a higher resemblance to images for which past `hClean` questions have returned positive answers.

Overall, selecting the right questions is a non-trivial optimization problem in the development of an effective query processor. In our previous work, we developed several such strategies for specific classes of human-computable predicates that occur commonly in larger tasks [30]. Generalizing these strategies, and enhancing them by taking into account data similarity and the answers to previous questions, is part of our ongoing work in this area.

5. THE REALITY: UNCERTAINTY

So far, we have assumed that correct answers are provided by humans to unit tasks posted on the CrSS. However, this is hardly the case in practice, and we need to deal with incorrect and biased answers in most cases. In this section, we consider ways and means of dealing with and resolving this uncertainty.

We describe the sources of uncertainty in §5.1 and how to modify the ways to use hQuery in §3 to incorporate uncertainty. We describe three candidate methods of dealing with uncertainty, with increasing sophistication, in §5.3, §5.4 and §5.5.

For the purposes of this section, we assume that we do not use negation in hQuery. Even with this assumption, the issues that come up when dealing with uncertainty and bias are non-trivial.

5.1 Sources of Uncertainty

In the previous section, the only source of uncertainty is when not all certain answers are obtained. Additional tuples may or may not be present in the query result that is returned to the user — as a result of this uncertainty, the resulting answer could be one of several possible worlds [8], where the worlds contain all the certain answer tuples obtained, and may or may not contain the remaining answer tuples.

Also, in §4, we assumed that each human is omniscient; and

answers every task correctly. It is therefore sufficient to ask a single human to evaluate any h-predicate on a given tuple via the CrSS. However, the reality is very different.

Even for a unit task (i.e., a single question), there can be uncertainty in three ways: (a) Humans may make mistakes, e.g., Count the number of objects in an Image. (b) Humans have subjective views, e.g., Is the location depicted in this image clean? (c) Humans may provide spam/malicious responses. In this section, we consider (a) and (b), and we consider (c) in §6.2. For the purposes of this section, we assume that humans do not maliciously provide incorrect answers.

When there are multiple h-predicates, operating on multiple tuples/objects, there could be further sources of uncertainty or bias: (a) Answers that humans give to various h-predicates for the same tuple/object may be correlated. For instance, if a human thinks that an image is that of a city, then his answer to `hCity` would be (negatively) correlated with that of `hBeach`. (b) Answers that humans give to different tuples for the same h-predicate may be correlated. (c) Number of workers attempting each question may be very different, as a result, we may have several YES/NO answers on some questions for a given predicate but may not have them on other questions for the same predicate. (d) Number of people attempting different h-predicates may be different.

Thus, since the provided answers may be biased, incomplete, unnormalized or uncertain, we need to develop the means to reason when a h-predicate is true (on a specific tuple) given the answers of human workers for the same predicate and other predicates. Uncertainty also arises when dealing with ha-predicates or a-predicates. To handle this case, we assume that algorithms will provide an estimate on the reliability of the generated answers, which will allow us to handle answers from the CrSS and algorithms in a unified framework.

5.2 Incorporating Uncertainty

Since we have uncertainty as to whether or not an actual tuple is part of the query result, we can resort to using probabilities to quantify our belief as to whether or not a tuple is correct. This uncertainty could be in two forms, the probability of presence or absence of a tuple (defined before in uncertain databases [36, 12, 9] as *maybe* annotations), or the probability of each of a set of values for an attribute or group of attributes in a tuple (defined in uncertain databases as *alternative* annotations).

As a natural extension of *certain answers*, we propose the use of *probability thresholds*: For each way of using hQuery as defined in §3, we are given an additional parameter τ, representing the probability threshold. In other words, only tuples whose probability of being present in the query result is $\geq \tau$, are correct and should be returned to the user.

For instance, for problem 1, instead of requiring only certain answers to be present in the query result as in the previous section, we would like all tuples whose confidence is $\geq \tau$ to be present in the query result. Notice that when $\tau = 1$, this is equivalent to enforing all and only certain answers to be present in the query result, thus our technique of incorporating uncertainty into the ways of using hQuery forms a natural generalization of the ways of using hQuery in §3 when answers are uncertain.

Thus the main challenge that we will deal with in subsequent sections is how we compute confidences of tuples derived from h-ha- and a-predicates, and how we combine these confidences to compute those of tuples that should be in the query result.

5.3 Majority Voting: The easy way out

For the first method of computing probabilities, we use the simple strategy of majority voting. This scheme assumes that a fixed (large) number of identical, but independent workers attempt each question. In addition, no worker attempts more than one question. Thus, there are very few biases that come into play in this setting.

We compute the resulting answer for a h-predicate on a given tuple as simply the majority of the answers of the humans. Indeed, there is an implicit belief that the majority would be a good proxy for the correct answer. It is easy to see that this scheme essentially eliminates all probabilities, by simply awarding the most 'likely' alternative a probability of 1, while all other alternatives are awarded a probability of 0. Subsequently, we can produce results as we do for certain answers in §4.

Thus, in this scheme, it is not possible to get fine-grained probabilities for various tuples. As a result, there could be errors. For instance, we may miss out on certain tuples whose probabilities may be greater than τ, but were generated in conjunction with an alternative binding for a human-computed tuple that was not the one that had the majority. Secondly, when faced with alternatives that have nearly the same number of votes, or when there are many alternatives with the same number of votes, it is not clear how we can meaningful the majority answer is. One option is to simply ask more humans on the fly.

5.4 Probabilistic Approach with IID Workers

Here, we assume that for each question, workers are drawn independently and identically distributed (IID) from an a-priori distribution. For instance, there is an inherent distribution for the question `hBeach(Image3)`, which could be, for instance, Yes - 0.6, No - 0.4, and each human attempting the corresponding question would sample randomly from this inherent distribution. Of course, this distribution is not given to us, and needs to be inferred based on the answers of the humans to the question, based on maximum likelihood (i.e., each alternative has a probability equal to the fraction of humans that select it as an answer), or some other a-posteriori inference approach [11]. For hybrid predicates, we need to combine two distributions (one from algorithmic evaluation, and one from human computation) to give one unified distribution over alternatives.

This distribution is the same as distributions over alternatives in uncertain databases [36]. We assume, as in the previous scheme, that a fixed large number of people attempt each question. Thus, the probabilities are normalized and there are no additional biases.

Subsequently, for each unevaluated h-predicate on a given tuple, we assume that the probability is 0 for all alternatives. Once predicates get evaluated either by humans or by algorithms, the probabilities for various alternatives get updated. We can use standard confidence computation (as in uncertain databases) in order to compute confidence of each result tuple. These probabilities would have to be recomputed every time a human predicate is evaluated.

However, if there is no negation, then the confidence of each output tuple can only increase with additional h-predicate evaluation. We can therefore use this property to avoid evaluating additional human predicates that derive the same output tuple (that already has a confidence $\geq \tau$), and we can return output tuples as and when they become available. Of course, for those output tuples whose confidence is $< \tau$, it might still be the case that with additional human predicate evaluation, they become part of the query result.

5.5 Approach Using Item-Response Theory

In this scheme, we utilize all forms of uncertainty considered in §5.1. This situation has parallels with questionnaire analysis, when a questionnaire containing several questions is attempted by

many humans. Each human may choose to answer some or all of the questions. As in our case, there may be correlations between answers to various questions for each human, as well as uncertainty in the answers for even a single question.

Questionnaires are usually analyzed using Item Response Theory (IRT) [2]. Item Response Theory is the science of inferences and scoring for tests, questionnaires and examinations. In our case, the inferences may be especially tricky given that only a handful of humans attempt each question.

6. EXTENSIONS

We now describe some interesting extensions to the basic model.

6.1 Costing

An important aspect of posting tasks to the CrSS is to decide how to price each task. So far, we have assumed that we price each task equally, as a result of which the number of tasks posted is a measure of total amount spent. However, we may be able to speed up query processing by intelligent pricing of tasks.

More specifically, it is clear that we can reduce latency by assigning higher prices. Additionally, for tasks that help us obtain query tuples easier, it would be helpful to price them higher (for instance, if there are two queries deriving output tuples, one with a single h-predicate, and the other with 4 h-predicates, it would be beneficial to price the single h-predicate higher, since that would yield query tuples faster). Also, for tasks that are harder to attempt, pricing should be higher, otherwise humans may prefer to attempt other tasks earlier. Additionally, if we have a deadline for task completion, it would be wise to use a low price early in the evaluation process, and to increase the price closer to the completion deadline.

However, to maximize the number of questions under a fixed budget, we may want to replicate each question fewer times and issue a low price, at the cost of getting delayed, uncertain answers.

6.2 Spam

As in any setting, we need to safeguard our CrSS from adversarial attacks. These attacks could be similar to link farms (where a clique of nodes in a network gets a high pagerank due to high connectivity amongst themselves [17]), as well as algorithms mimicking humans (in other words, spam).

There are two ways of beating spam (each with limitations): (a) Use majority voting, penalizing any human who does not produce the majority answer. However, this method is susceptible to groups of algorithms mimicking humans colluding together to answer questions arbitrarily in the same fashion, as in a link farm. (b) Have a "test" question, with any human who gets it incorrect not being able to attempt the actual question. This test is akin to CAPTCHAs [34] that are essentially turing tests to discern algorithms from humans. If the ratio of test to actual questions is high, then the cost to use the CrSS may be too high. On the other hand, if the ratio of test to actual questions is low, then the humans may manually solve the test question, and then arbitrarily guess for the rest of the questions.

6.3 Second Order Predicates

Instead of h-predicates, one could also use human-computable algorithms, where the algorithms take in a question budget, and a latency requirement, select several questions to post to the CrSS and return an aggregated answer. These human-computable algorithms could be of various kinds, e.g., sort ten images, pick the odd one out, or cluster search results. For instance, the human-computable algorithm for sorting would be optimized to ask much fewer than the $\binom{n}{2}$ comparisons required. Design of a human computable algorithm to select an optimal set of questions to ask humans to solve a graph search problem has been examined in prior work [30].

6.4 Mapping to Unit Tasks

The choice of unit tasks corresponding to an h-predicate can affect the amount of time taken to obtain a result from the CrSS, the cost, as well as the inherent uncertainty in the answer. It is therefore an important dimension for query optimization. For instance, for a sorting predicate, we may use pairwise comparisons as our unit tasks (i.e., which of these two items is better?), or multi-way comparisons (i.e., what is the correct ranking for the following k items?). While pairwise comparisons may yield more accurate results, multi-way comparisons may result in answers being obtained faster, and at lesser cost. On the other hand, sorting by rating each item (i.e., how would you rate this item on a scale of 1-10?) would involve fewer tasks being posed to the CrSS, but may result in much higher uncertainty. Thus, in addition to deciding which questions to ask, the query optimizer needs to decide how to ask them to balance cost, uncertainty as well as latency.

6.5 Scheduling

In this section, we briefly discuss issues in the design of a query engine processing several hQuery queries in a batch. Firstly, we would want to schedule tasks at times based on availability of capable workers, e.g., if the query is to identify cities in Europe, it is pragmatic to schedule those tasks when it is daytime in Europe. Additionally, we may want to schedule and price tasks based on priority order of completion, i.e., price important tasks higher and post them earlier. As in standard joint query optimization, it would be beneficial to share computation between queries in the workload. In particular, h- and a-predicates that have been evaluated do not need to be reevaluated, and can be stored in an extensional database for reuse while processing other queries.

7. RELATED WORK

A number of recent works have considered the design of libraries to access crowd sourcing services such as Mechanical Turk [3]. Little et. al. [25, 26] suggested iterative and parallel tasks as design patterns in order to build more complex applications over Mechanical Turk. Heymann et. al. [20] describes the design of a programming environment built on top of Mechanical Turk that contains modular and reusable components.

Kochhar et. al. [24] describes how Freebase has been using human judgement for various data cleansing tasks. Crowd-sourcing services are also used to provide input to active learning algorithms [33]. A few works have also explored other aspects of crowd-sourcing, such as designing good tasks [22], pricing tasks [28, 21], the occurrence of spam and bias [23, 27] and so on. The survey by Doan et. al. [16] provides a good overview of the general area of human involvement or collaboration and how it may be used to solve problems.

We argued earlier about the necessity of adaptive query processing due to the difficulty of obtaining accurate cost models for h- and a-predicates. We hope to leverage the extensive previous work on this topic in the context of relational database systems [15]. Additionally, the issues of uncertainty arising in human computable predicates are more intricate than the ones found in existing uncertain databases [36, 9, 12], however some of the confidence computation techniques therein can be reused in our context.

There has been some work in the past on optimizing query plans that contain expensive predicates [14, 19], however, in our case, we also need to deal with pricing as well as uncertainty. The work

on online aggregation [18] allows users to specify on the fly which data is important and needs relational processing. However, in our case, humans perform the processing.

Additionally, from an application standpoint, there have been several papers considering the use of human experts in various tasks such as data integration and exploration [31, 35, 32, 29] and information extraction [13]. Our previous work examined the optimization of graph search tasks that harness human processors via a crowd-sourcing service [30].

8. CONCLUSIONS

With the vision of being able to process and answer queries on data more effectively, we proposed, in this paper, the use of a CrSS in addition to standard relational database operators. We presented hQuery, a declarative query model operating over human-computable predicates, databases as well as external algorithms, and discussed the challenges involved in optimizing hQuery, in particular, the presence of uncertainty derived from the answers of humans, as well as tradeoffs between number of certain (or probabilistic) answers, time allocated, and monetary cost. In addition, due to not-so-well-understood operators, unknown latencies, accuracy and selectivity and lack of cardinality estimates, query processing can become especially hairy. We defined several research problems, with increasing levels of complexity, for the case of perfect answers, as well as when answers are uncertain. This work also raises some research issues on how humans interpret and answer questions. We believe that the research problems outlined herein, and the general area of optimizing humans as well as databases, form an interesting and important area for future database research.

9. REFERENCES

[1] CrowdFlower. *http://crowdflower.com*.

[2] Item Response Theory. *http://en.wikipedia.org/wiki/Item_response_theory*.

[3] Mechanical Turk. *http://mturk.com*.

[4] Microtask. *http://microtask.com*.

[5] ODesk. *http://odesk.com*.

[6] Samasource. *http://samasource.com*.

[7] UTest. *http://utest.com*.

[8] Serge Abiteboul, Paris Kanellakis, and Gosta Grahne. On the representation and querying of sets of possible worlds. *SIGMOD Rec.*, 16(3):34–48, 1987.

[9] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing Incomplete Information with Probabilistic World-Set Decompositions. In *Proc. of ICDE*, 2007.

[10] Jeff Barr and Luis Felipe Cabrera. Ai gets a brain. *Queue*, 4(4):24–29, 2006.

[11] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1st ed. 2006. corr. 2nd printing edition, October 2007.

[12] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. MYSTIQ: a system for finding more answers by using probabilities. In *Proc. of ACM SIGMOD*, 2005.

[13] Xiaoyong Chai, Ba Q. Vuong, Anhai Doan, and Jeffrey F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *SIGMOD '09*.

[14] Surajit Chaudhuri and Kyuseok Shim. Query optimization in the presence of foreign functions. In *VLDB*, 1993.

[15] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[16] AnHai Doan, Raghu Ramakrishnan, and Alon Y. Halevy. Mass Collaboration Systems on the World-Wide Web. Technical report, Communications of the ACM (To Appear).

[17] Ye Du, Yaoyun Shi, and Xin Zhao. Using spam farm to boost pagerank. In *AIRWeb '07*, New York, NY, USA, 2007.

[18] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *SIGMOD Conference*, 1997.

[19] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD Conference*, pages 267–276, 1993.

[20] Paul Heymann and Hector Garcia-Molina. Human processing. Technical report, Stanford University, July 2010.

[21] John Horton and Lydia Chilton. The labor economics of paid crowdsourcing. *CoRR*, abs/1001.0627, 2010.

[22] Eric Huang, Haoqi Zhang, David C. Parkes, Krzysztof Z. Gajos, and Yiling Chen. Toward automatic task design: a progress report. In *HCOMP '10*, New York, NY, USA, 2010.

[23] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on amazon mechanical turk. In *HCOMP '10*, New York, NY, USA, 2010.

[24] Shailesh Kochhar, Stefano Mazzocchi, and Praveen Paritosh. The anatomy of a large-scale human computation engine. In *HCOMP '10*, New York, NY, USA, 2010.

[25] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *HCOMP '09*, New York, NY, USA, 2009.

[26] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Exploring iterative and parallel human computation processes. In *HCOMP '10*, New York, NY, USA, 2010.

[27] M. Motoyama et. al. Re: Captchas understanding captcha-solving services in an economic context. In *USENIX Security Symposium '10*.

[28] Winter Mason and Duncan J. Watts. Financial incentives and the "performance of crowds". In *HCOMP '09*, New York, NY, USA, 2009.

[29] Robert McCann, Warren Shen, and AnHai Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE '08*.

[30] Aditya Parameswaran, Anish Das Sarma, Hector Garcia-Molina, Neoklis Polyzotis, and Jennifer Widom. Human-assisted graph search: It's okay to ask questions. Technical report, Stanford University.

[31] S. Jeffery et. al. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD '08*.

[32] S. Sarawagi et. al. Interactive deduplication using active learning. In *KDD '02*.

[33] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[34] Luis von Ahn and Laura Dabbish. Designing games with a purpose. *Commun. ACM*, 51(8), 2008.

[35] W. Wu et. al. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD '04*.

[36] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Proc. of CIDR*, 2005.