Crowdsourced Databases: Query Processing with People

Adam Marcus, Eugene Wu, David R. Karger, Samuel Madden, Robert C. Miller
MIT CSAIL
{marcua, sirrice, karger, madden, rcm}@csail.mit.edu

ABSTRACT

Amazon's Mechanical Turk ("MTurk") service allows users to post short tasks ("HITs") that other users can receive a small amount of money for completing. Common tasks on the system include labelling a collection of images, combining two sets of images to identify people which appear in both, or extracting sentiment from a corpus of text snippets. Designing a workflow of various kinds of HITs for filtering, aggregating, sorting, and joining data sources together is common, and comes with a set of challenges in optimizing the cost per HIT, the overall time to task completion, and the accuracy of MTurk results. We propose Qurk, a novel query system for managing these workflows, allowing crowd-powered processing of relational databases. We describe a number of query execution and optimization challenges, and discuss some potential solutions.

1. INTRODUCTION

Amazon's Mechanical Turk service (`https://www.mturk.com/mturk/welcome`) ("MTurk") allows users to post short tasks ("HITs") that other users ("turkers") can receive a small amount of money for completing. A HIT creator specifies how much he or she will pay for a completed task. Example HITs involve compiling some information from the web, labeling the subject of an image, or comparing two documents. More complicated tasks, such as ranking a set of ten items or completing a survey are also possible. MTurk is used widely to perform data analysis tasks which are either easier to express to humans than to computers, or for which there aren't yet effective artificial intelligence algorithms.

Task prices vary from a few cents ($.01-$.02/HIT is a common price) to several dollars for completing a survey. MTurk has around 100,000-300,000 HITs posted at any time (`http://mturk-tracker.com/general/`). Novel uses of MTurk include matching earthquake survivor pictures with missing persons in Haiti (`http://app.beextra.org/mission/show/missionid/605/mode/do`), authoring a picture book (`http://bjoern.org/projects/catbook/`), and embedding turkers as editors into a word processing system [2].

From the point of view of a database system, crowd-powered tasks can be seen as operators—similar to user-defined functions—that are invoked as a part of query processing. For example, given a database storing a table of images, a user might want to query for images of flowers, generating a HIT per image to have turkers perform the filter task. Several challenes arise in processing such a workflow. First, each HIT can take minutes to return, requiring an asynchronous query executor. Second, in addition to considering time, a crowdworker-aware optimizer must consider monetary cost and result accuracy. Finally, the selectivity of operators can not be predicted *a priori*, requiring an adaptive approach to query processing.

In this paper, we propose Qurk, a crowdworker-aware database system which addresses these challenges. Qurk can issue HITs that extract, order, filter, and join complex datatypes, such as images and large text blobs. While we describe Qurk here using the language of MTurk (turkers and HITs), and our initial prototype runs on MTurk, we aim for Qurk to be crowd-platform-independent. Future versions of Qurk will compile tasks for different kinds of crowds with different interfaces and incentive systems. Qurk is a new system in active development; we describe our vision for the system, propose a high level sketch of a UDF-like syntax for executing these HITs, and explore a number of questions that arise in such a system, including:

- What is the HIT equivalent of various relational operations? For example, an equijoin HIT might require humans to identify equal items, whereas a HIT-based sort can use human comparators.
- How many HITs should be generated for a given task? For example, when sorting, one can generate a HIT that asks users to score many items, or to simply compare two. How can the system optimize each HIT?
- Given that a large database could result in millions of HITs, how should the system choose which HITs to issue? Given that only a fraction of items in a large database can have HITs generated for them, what is the proper notion of query correctness?
- How much to charge for a HIT? Higher prices can lead to faster answers. Can the system adaptively optimize the price of HITs based on user-specified response time and budget constraints?
- Who is the right crowdworker for a task? MTurk provides paid workers, but an enterprise might prefer expert workers, and an operation with a tight budget might look for non-monetary incentives.

2. MOTIVATING EXAMPLES

Here is a list of tasks that Qurk should be able to run:

CIDR 2011

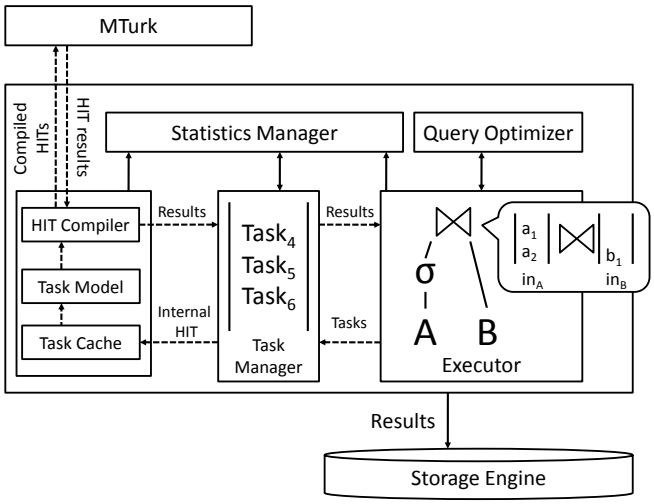

Figure 1: A Qurk system diagram.

- Given a database with a list of company names, find the CEO and contact number for each company (see `http://www.readwriteweb.com/enterprise/2010/01/crowdsourcing-for-business.php`).
- Given a set of photographs of people from a disaster, and pictures submitted by family members of lost individuals, perform a fuzzy join across both sets, using humans to determine equality.
- Given a collection of messages (e.g., from twitter), identify the sentiment of each (e.g., as either "positive" or "negative") [3].
- Given a list of products, rank them by "quality" by searching for reviews on Amazon.

These workflows have relatively small query plans, but in practice, plans see size increases as more filters and sorts appear. Additionally, even small plans benefit from Qurk's adaptive cost optimizations.

3. SYSTEM OVERVIEW

Qurk is architected to handle an atypical database workload. Workloads rarely encounter hundreds of thousands of tuples, and an individual operation, encoded in a HIT, takes several minutes. Components of the system operate asynchronously, and the results of almost all operations are saved to avoid re-running unnecessary steps. We now discuss the details of Qurk, which is depicted in Figure 1.

The **Query Executor** takes as input query plans from the query optimizer and executes the plan, possibly generating a set of tasks for humans to perform. Due to the latency in is processing HITs, the query operators communicate asynchronously through input queues, as in the Volcano system [4]. The example join operator maintains an input queue for each child operator, and creates tasks that are sent to the **Task Manager**.

The Task Manager maintains a global queue of tasks to perform that have been enqueued by all operators, and builds an internal representation of the HIT required to fulfill a task. The manager takes data which the **Statistics Manager** has collected to determine the number of turkers and the cost to charge per HIT, which can differ per operator. Additionally, the manager can collapse several tasks generated by operators into a single HIT. These optimizations collect several tuples from a single operator (e.g., collecting multiple tuples to sort) or from a set of operators (e.g., sending multiple filter operations over the same tuple to a single turker).

A HIT that is generated from the task manager first probes the **Task Cache** and **Task Model** to determine if the the result has been cached (if an identical HIT was executed already) or if a learning model has been primed with enough training data from other HITs to provide a sufficient result without calling out to a crowdworker. We discuss these optimizations further in Section 6.

If the HIT cannot be satisfied by the Task Cache or Task Model, then it is passed along to the **HIT Compiler**, which generates the HTML form that a turker will see when they accept the HIT, as well as other information required by the MTurk API. The compiled HIT is then passed to **MTurk**. Upon completion of a HIT, the Task Model and Task Cache are updated with the newly learned data, and the Task Manager enqueues the resulting data in the next operator of the query plan. Once results are emitted from the topmost operator, they are added to the database, which the user can check on periodically.

4. DATA MODEL AND QUERY LANGUAGE

Qurk's data model is close to the relational model, with a few twists. Two turkers may provide different responses to the same HIT, and results must often be verified for confidence across multiple turkers. In Qurk's data model, the result of multiple HIT answers is a multi-valued attribute. Qurk provides several convenience functions (e.g, `majorityVote`) to convert multi-valued attributes to usable single-valued fields.

We now propose a simple UDF-like approach for integrating SQL with crowdworker-based expressions. We use SQL since it should be familiar to a database audience, but we plan to study a variety of different interfaces in Qurk, some of which will have a more imperative flavor, or which will operate over files instead of database tables. We introduce our proposal through a series of examples.

Filters

In our first example, we look at a query that uses the crowd to find images of flowers in a table called `images`, with an image-valued field called `img`. The user writes a simple SQL query, and uses a UDF called `isFlower`, which is a single-valued function that takes an image as an argument.

```
SELECT * FROM images where isFlower(img)
```

Using the Qurk UDF language, the user defines `isFlower` as the following task:

```
TASK isFlower(Image img) RETURN Bool:
    TaskType: Question
    Text: ''Does this image: <img src='%s'>
          contain a flower?'',URLify(img)
    Response:  Choice(''YES'',''NO'')
```

In our language, UDFs specify the type signature for the `isFlower` function, as well as a set of parameters that define the job that is submitted. In systems like MTurk, a job looks like an HTML form that the turker must fill out. The

parameters above specify the type and structure of the form that the database will generate. The `TaskType` field specifies that this is a question the user has to answer, and the `Response` field specifies that the user will be given a choice (e.g., a radio button) with two possible values (YES, NO), which will be used to produce the return value of the function. The `Text` field shows the question that will be supplied to the turker; a simple substitution language is provided to parameterize the question. Additionally, a library of utility functions like `URLify` (which converts a database blob object into a URL stored on the web) are provided.

Crowd-provided Data

In this example we show how crowdworkers can supply data that is returned in the query answer. Here, the query is to find the CEO's name and phone number for a list of companies. The query would be:

```
SELECT companyName, findCEO(companyName).CEO,
       findCEO(companyName).Phone
FROM companies
```

Observe that the `findCEO` function is used twice, and that it returns a tuple as a result (rather than just a single value). In this case, the `findCEO` function would be memoized after its first application. We allow a given result to be used in several places, including different queries.

The task for the `findCEO` function follows:

```
TASK findCEO(String companyName)
RETURNS (String CEO,String Phone):
    TaskType:  Question
    Text: ``Find the CEO and the CEO's phone
          number for the company %s'', companyName
    Response: Form((``CEO'',String),
                   (``Phone'',String))
```

This task definition is similar to the previous one, except that the response is a tuple with two unconstrained fields.

Table-valued Ranking Functions

Suppose we want the turker to rank a list of products from their Amazon reviews. Our SQL query might be:

```
SELECT productID, productName FROM products
ORDER BY rankProducts(productName)
```

Here, the `rankProducts` function needs take in a list of products, rather than just a single product, so that the user can perform the comparison between products. The task definition is as follows:

```
TASK rankProducts(String[] prodNames) RETURNS String[]:
    TaskType:  Rank
    Text: ``Sort the following list of products
           by their user reviews on Amazon.com''
    List: prodNames
```

The syntax is similar to the previous tasks, except that the function takes an array-valued argument that contains values from multiple rows of the database. The `TaskType` is specified as `Rank`. Ranking tasks don't accept a specific `Response` field, but just provide a list of items for the user to order—in this case, that list is simply the provided array.

The Qurk system may call `rankProducts` multiple times for a given query, depending on the number of items in the table. Ranking tasks are often decomposed into smaller subtasks. To combine results from subtasks, users can be asked to provide a numeric score, or the system can provide an approximate ordering of results by ensuring that subtasks overlap by a few products.

Table-valued Join Operator

In the final example, we show how the crowd can join two tables. Suppose we have a **survivors** table of pictures of earthquake survivors, and a **missing** table with pictures submitted by family members. We want to find missing persons in the survivors table.

```
SELECT survivors.location, survivors.name
FROM survivors, missing
WHERE imgContains(survivors.image, missing.image)
```

The `imgContains` function needs takes two lists of images to join. The task definition is as follows:

```
TASK imgContains(Image[] survivors, Image[] missing)
RETURNS Bool:
    TaskType:  JoinPredicate
    Text: ``Drag a picture of any <b>Survivors</b>
           in the left column to their matching
           picture in the <b>Missing People</b>
           column to the right.''
    Response: DragColumns("Survivors", survivors,
                          "Missing People", missing)
```

Here, `imgContains` is of type `JoinPredicate`, and takes two table-valued arguments. The task is compiled into a HIT of type `DragColumns` which contains two columns labeled `Survivors` and `Missing People`. Turkers drag matching images from the left column to the right one to identify a match. The number of pictures in each column can change to facilitate multiple concurrent comparisons per HIT.

5. QUERY EXECUTION

As mentioned in Section 3, Qurk's components communicate asynchronously. Each query operator runs in its own thread and consumes tuples from its child in the query tree. For each tuple in its queue (or queues, in the case of joins), it generates a task to be completed by a turker. These tasks are enqueued for the task manager, which runs a thread for pricing HITS and collapsing multiple tasks into a single HIT under the instruction of the optimizer. Multiple HITs will be outstanding concurrently.

After a task is returned from a crowdworker, it is used to train a model for potentially replacing turker tasks and is cached for future reuse. A second thread in the manager then enqueues the result of the task into the parent operator's queue. Finally, if the completed task was generated by the root of the query tree, the manager inserts the result into a results table in the database, which the issuer of the Qurk query can return to at a later time to see the results.

6. OPTIMIZATIONS

A Qurk query can be annotated with several parameters that are used to guide its execution. The first, `maxCost`, specifies maximum amount of money the querier is willing to pay. A query might be infeasible if the number of HITs,

each priced at a penny (the cheapest unit of payment), exceed `maxCost`. `minConfidence` specifies the minimum number of turkers who must agree on an answer before it is used. `maxLatency` specifies the maximum amount of time that the user is willing to wait for a HIT to complete.

We plan to explore a cost model based on these parameters in the future. The parameters will help the optimizer identify the cost per HIT, since higher prices decrease the turker wait time. The parameters also affect the number of items batched into a table-valued HIT, such as placing 10 items to a `Rank` task, or 5 items in each column of a `JoinPredicate` task. Finally, the parameters dictate how many times each HIT can be verified by a subsequent turker.

We now discuss several potential optimizations.

Runtime Pricing: If it appears that one type of task takes longer to complete than others, Qurk can offer more money for it in order to get a faster result.

Input Sampling: For large tables that could lead to many HITs, Qurk attempts to sample the input tables to generate Qurk jobs that uniformly cover the input space. This is similar to issues that arise in online query processing [5].

Batch Predicates: When a query contains two filters on the same table, we can combine them into a single HIT, decreasing cost and time. For example, a query with predicates `imgColor(image) == 'Red'` and `imgLabel(image) == 'Car'` could have a single HIT in which turkers are presented with the `imgColor` and `imgLabel` tasks at the same time.

Operator Implementations: To assist with operator pipelining and provide users with results as early as possible, operators should be non-blocking when possible (e.g., symmetric hash joins). Additionally, several potential implementations of each operator are possible. For example, a `Rank` task might ask users to sort a batch of items relative to one-another, or it might ask users to assign an absolute score to each and sort items based on their scores.

Join Heuristics: The space of comparisons required for JoinPredicates can be reduced with a preprocessing step identifying necessary join conditions. For example, if `gender` and `race` must match on pictures of `survivors` and `missing` persons, a user could add a `NecessaryConditions` statement to the `imgContains` task in Section 4 with those two fields. Qurk would generate HITs for each image which instructs turkers to extract `gender` and `race` properties from pictures where possible. `JoinPredicate` HITs could then only compare items that are compatible, reducing the search space.

Task Result Cache: Once a HIT has run, its results might be relevant in the future. For example, if the `products` table has already been ranked in one query, and another query wishes to rank all red `products`, the result of the old HITs can be used for this. Additionally, as explored in TurKit [6], queries might crash midway, or might be iteratively developed. In such scenarios, a cache of completed HITs can improve response time and decrease costs.

Model Training: There are situations where an otherwise acceptable learning algorithm requires training data to be useful. For example, the `isFlower` task in Section 4 could potentially be replaced by an automated classifier with enough training data. If we consider HIT responses as ground truth data, we can incrementally train an algorithm and measure its performance relative to the labeled data set, and then automatically switch over to that algorithm when its performance becomes good enough.

7. RELATED WORK

Little et al. present TurKit [6], which introduces the notion of iterative crowd-powered tasks. TurKit supports a process in which a single task, such as sorting or editing text, might require multiple coordinated HITs, and offers a persistence layer that makes it simple to iteratively develop such tasks without incurring excessive HIT costs.

Several systems have been built on top of MTurk's API to facilitate higher-order tasks using HITs. Crowdflower [1] provides an API to make developing HITs and finding cheating turkers easier, and has abstracted the notion of HITs so that other systems, such as gaming platforms, can also be used to deliver HITs to willing participants.

Parameswaran et al. propose a declarative language for querying crowd workers and outline considerations for an uncertainty model over the results [7]. With Qurk, we instead focus on how to implement crowd-powered operators and design a system for executing queries with these operators. We then describe the optimizations for reducing the cost and time to execute these queries.

8. CONCLUSION

The MTurk ecosystem is growing daily, and a system for building and optimizing data processing workflows across crowdworkers presents many challenges with which the database community is familiar. We proposed Qurk, a system for writing SQL-like queries to manage complex turker workflows, and discussed several challenges and optimization opportunities for the future.

9. REFERENCES

[1] Crowdflower, July 2010. `http://crowdflower.com/`.

[2] M. S. Bernstein et al. Soylent: a word processor with a crowd inside. In *UIST '10: Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 313–322, New York, NY, USA, 2010.

[3] N. A. Diakopoulos and D. A. Shamma. Characterizing debate performance via aggregated twitter sentiment. In *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 1195–1198, New York, NY, USA, 2010. ACM.

[4] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[5] P. J. Haas et al. Selectivity and cost estimation for joins based on random sampling. *J. Comput. Syst. Sci.*, 52(3):550–569, 1996.

[6] G. Little et al. Turkit: human computation algorithms on mechanical turk. In *UIST '10: Proceedings of the 23nd annual ACM symposium on User interface software and technology*, pages 57–66, 2010.

[7] A. Parameswaran and N. Polyzotis. Answering queries using databases, humans and algorithms. Technical report, Stanford University.