Demonstration of Qurk: A Query Processor for Human Operators

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

ABSTRACT

Crowdsourcing technologies such as Amazon's Mechanical Turk ("MTurk") service have exploded in popularity in recent years. These services are increasingly used for complex human-reliant data processing tasks, such as labelling a collection of images, combining two sets of images to identify people that appear in both, or extracting sentiment from a corpus of text snippets. There are several challenges in designing a workflow that filters, aggregates, sorts and joins human-generated data sources. Currently, crowdsourcing-based workflows are hand-built, resulting in increasingly complex programs. Additionally, developers must hand-optimize tradeoffs among monetary cost, accuracy, and time to completion of results. These challenges are well-suited to a declarative query interface that allows developers to describe their worflow at a high level and automatically optimizes workflow and tuning parameters. In this demonstration, we will present Qurk, a novel query system that allows human-based processing for relational databases. The audience will interact with the system to build queries and monitor their progress. The audience will also see Qurk from an MTurk user's perspective, and complete several tasks to better understand how a query is processed.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Databases

Keywords

Database, Mechanical Turk, Human Computation

1. INTRODUCTION

Crowdsourcing platforms such as Amazon's Mechanical Turk service[1] ("MTurk") allow users to post short tasks ("HITs") that other users ("turkers") can complete for a small amount of money. 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. These platforms are used to perform data analysis tasks that are either easier to express to humans than to computers, or for which there are not yet effective artificial intelligence algorithms.

Task prices vary from a few cents ($.01-$.03/HIT is a common price) to several dollars for completing a survey. Mechanical Turk has around 100,000-300,000 HITs posted at any time[2]. Novel uses include matching earthquake survivor pictures with missing persons in Haiti[3], authoring a picture book[4], and using turkers as editors in a word processor [1].

Qurk is a database system that integrates MTurk-style tasks as first-class operators. Several challenges arise in developing such a system. First, the naïve implementations of traditional operators with human guidance results in too many tasks (e.g., joins as cross products) that result in extraordinary monetary cost. Second, operator implementations must have redundancy built-in, as individual turker results are often inaccurate. Third, query execution must be asynchronous because each HIT may take several minutes to generate results. Finally, the difficulty and selectivity of tasks can not be predicted *a priori*, requiring an adaptive approach to query processing.

Several systems provide a programming layer on top of MTurk [4][5]. While they ease the development process of writing HIT-based systems and introduce performance optimizations, they take a procedural approach to workflow development. CrowdDB [2] offers a SQL interface to human computation. Qurk differs from CrowdDB in its data model for handling multiple results from turkers as well as its focus on operator implementations and optimizations. Parameswaran and Polyzotis [6] propose some database-oriented optimizations which could benefit Qurk.

In this demonstration, we present Qurk from the views of a Qurk user, the query optimizer, and an MTurk user. Audience members will be able to issue queries that generate HITs to extract, order, filter, and join complex datatypes, such as images and text blobs. Additionally, a system dashboard will display optimizer statistics and illustrate how HIT results flow through the query plan. Finally, the audience can participate in query processing by answering HITs. This interaction will highlight the use cases for human-powered query processing, as well as the process, constraints, and optimizations involved in effectively processing HIT-based queries in Qurk.

[1] `https://www.mturk.com/mturk/welcome`

[2] `http://mturk-tracker.com/general/`

[3] `http://app.beextra.org/mission/show/missionid/605/mode/do`

[4] `http://bjoern.org/projects/catbook/`

[5] `http://www.crowdflower.com`

SIGMOD'11, June 12–16, 2011, Athens, Greece.
ACM 978-1-4503-0661-4/11/06.

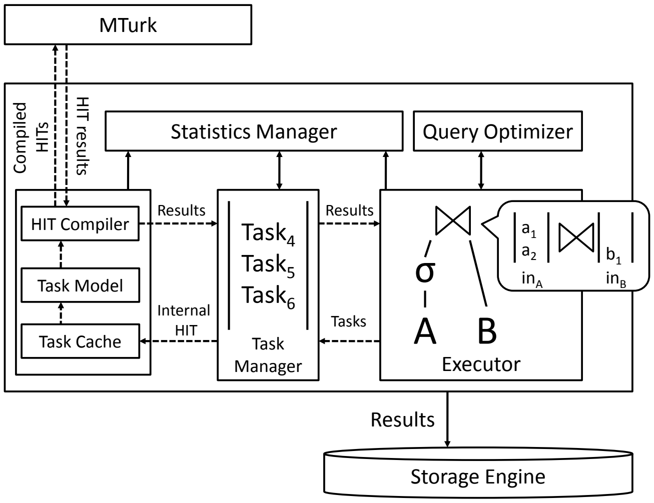


Figure 1: A system diagram of Qurk.

2. SYSTEM OVERVIEW

Qurk is architected to handle an atypical database workload. Human computation workloads rarely approach hundreds of thousands of tuples, but an individual operation on a tuple, encoded in a HIT, can take 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 Optimizer** compiles the query into a query plan and adaptively optimizes it during query execution. Query selectivities for HIT-based operators are not known *a priori* and user metrics may change mid-query. Additionally, the optimization function must take into account monetary cost, the number turkers to assign to each HIT, and the overall query performance.

The **Query Executor** takes as input query plans from the query optimizer, executes the plan, and generates a set of tasks for humans to perform. There are two key differences from traditional executors. First, due to the latency in processing HITs, the query operators communicate asynchronously through input queues, as in the Volcano system [3]. The join operator in Figure 1 contains two input queues from each child operator, and creates tasks that are sent to the **Task Manager**. Second, in contrast to the pull based iterator model, results are automatically emitted from the top-most operator and inserted into a results table. The user can periodically poll the table for new result tuples.

The Task Manager maintains a global queue of tasks that have been enqueued by all operators, and builds an internal representation of the HIT required to fulfill a task. The manager takes data from the **Statistics Manager** to determine the number of HITs, HIT assignments, and the cost of each task, each of which can differ across operators. As an optimization, the manager can batch several tasks into a single HIT. The task manager can feed batches of tuples to a single operator (e.g., collecting multiple tuples to sort). It can also generate HITs from a set of operators (e.g., grouping multiple filter operations over the same tuple).

The **HIT Compiler** generates the HTML form that a turker will fill out when they accept the HIT (along with MTurk-specific information), and sends it to **MTurk**. The result is passed to the Task Manager, which enqueues the result in the next operator of the plan. As an optimization, Qurk caches results in the **Task Cache**. If Qurk is aware of a learning model for the task, it trains this model with HIT results with the hope of eventually reducing monetary costs through automation (**Task Model**). Once results are emitted from the topmost operator, they are stored in the database, which the user can check on periodically.

3. DATA MODEL AND QUERY LANGUAGE

Qurk's data model is close to the relational model, with a key difference: two turkers may provide different responses to the same HIT. The current method to resolve this is to run a HIT multiple times in order to improve result confidence. It is difficult to quantify the uncertainty of a HIT based on a small sample of results. In our current implementation, we don't incorporate an uncertainty model. Instead, Qurk returns multiple answers to a HIT in a list, which can be reduced using user-defined aggregates.

We use a SQL-based query language with lightweight UDFs to give turkers instructions on completing HITs. We introduce the language using two examples.

MTurk-Provided Data

In this example we show how MTurk can be used to supply data that is returned in the query answer. Query 1 finds the CEO's name and phone number for a list of companies.

Query 1

```
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. In this case, the `findCEO` function would only be run on MTurk once per company. We cache a given result to be used in several places (even possibly in different queries).

Task 1

```
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))
```

The MTurk task for the `findCEO` function is in Task 1. In our language, UDFs specify the type signature for the `findCEO` function, as well as a set of parameters that control the MTurk job that is submitted. On the MTurk website, a job is an HTML form that the turker fills out. The `TaskType` field specifies that this is a question the user must answer. The `Response` field specifies that the user will provide two strings as free-text inputs that will be used to produce the return value of the function. The `Text` field shows the question that will be supplied to the turker. We provide a simple substitution language to parameterize the question.

Table-valued Join Operator

Query 2 uses MTurk to join two tables. Suppose we have a **celebrities** table with pictures of celebrities, and a **spot-**

tedstars table with submitted celebrity pictures. We want to identify each submitted celebrity.

Query 2

```
SELECT celebrities.name, spottedstars.id
FROM celebrities, spottedstars
WHERE samePerson(celebrities.image, spottedstars.image)
```

Task 2

```
TASK samePerson(Image[] celebs, Image[] spotted)
RETURNS BOOL:
    TaskType:  JoinPredicate
    Text: ‘‘Drag a picture of any <b>Celebrity</b>
            in the left column to their matching
            picture in the <b>Spotted Star</b>
            column to the right.’’
    Response: JoinColumns("Celebrity", celebs,
                          "Spotted Star", spotted)
```

The `samePerson` function takes two lists of images to join. The task definition is in Task 2. Here, `samePerson` is of type `JoinPredicate`, and takes two table-valued arguments. The task is compiled into a HIT of type `JoinColumns` which contains two columns labeled `Celebrity` and `Spotted Star`. Turkers select matching images from the left and right columns to identify a match. The number of pictures in each column can change to facilitate multiple comparisons per HIT.

Qurk also facilitates human-powered filter, rank, and group by operators. For more details, see [5].

4. DEMONSTRATION OVERVIEW

In this demonstration, we present an end-to-end prototype of the Qurk system and exhibit key features via two interactive interfaces. The first is a dashboard that shows the status of running queries as well as optimization metrics. The second asks the audience to solve HITs using an interface similar to MTurk. The core demonstration will focus on two long-running queries—a query that extends the schema of a companies table (*Query 1*) and a query that joins two tables of images (*Query 2*).

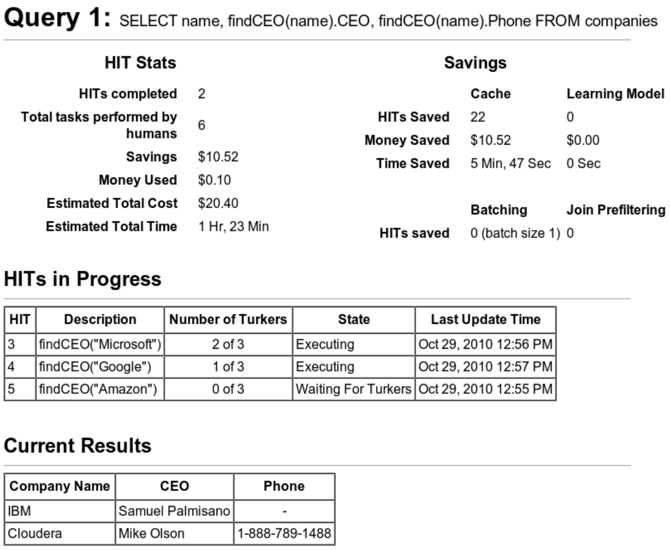
4.1 Query Status Dashboard

The Query Status Dashboard in Figure 2 provides a window into the system internals and will give the audience a sense of the time, budget, and optimization considerations that go into executing a Qurk query. Audience members will be able to view the dashboards of currently running queries as well as queries they have built.

There are several important features provided by the dashboard. The dashboard displays the current budget and estimates for for total query cost. The interface also describes the benefits gained from two optimizations: caching of previously executed UDFs on a tuple, and the use of classifiers in place of humans for various HITs. Additionally, the user can explore how different join interfaces, filtering-based reduction in cross-product size, and techniques like batching described in [5] affect accuracy, cost, and latency.

4.2 Task Completion Interface

To better understand the kinds of HITs that Qurk generates, audience members will be able to complete HITs for Query 1 and Query 2 using the Task Completion Interface.



Query 1: SELECT name, findCEO(name).CEO, findCEO(name).Phone FROM companies

HIT Stats

HITs completed	2
Total tasks performed by humans	6
Savings	$10.52
Money Used	$0.10
Estimated Total Cost	$20.40
Estimated Total Time	1 Hr, 23 Min

Savings

	Cache	Learning Model
HITs Saved	22	0
Money Saved	$10.52	$0.00
Time Saved	5 Min, 47 Sec	0 Sec
	Batching	Join Prefiltering
HITs saved	0 (batch size 1)	0

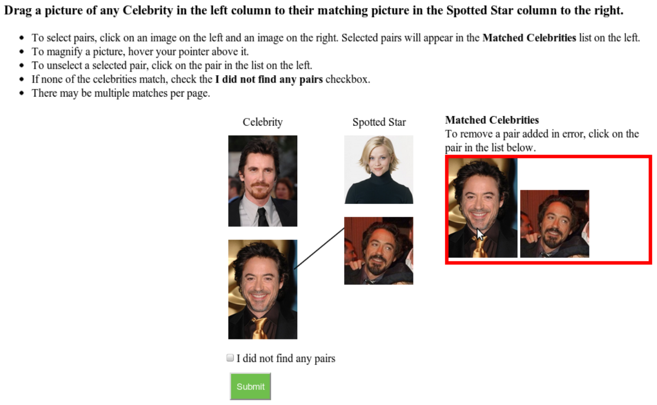
HITs in Progress

HIT	Description	Number of Turkers	State	Last Update Time
3	findCEO("Microsoft")	2 of 3	Executing	Oct 29, 2010 12:56 PM
4	findCEO("Google")	1 of 3	Executing	Oct 29, 2010 12:57 PM
5	findCEO("Amazon")	0 of 3	Waiting For Turkers	Oct 29, 2010 12:55 PM

Current Results

Company Name	CEO	Phone
IBM	Samuel Palmisano	-
Cloudera	Mike Olson	1-888-789-1488

Figure 2: The Qurk Query Status Dashboard.



Drag a picture of any Celebrity in the left column to their matching picture in the Spotted Star column to the right.

- To select pairs, click on an image on the left and an image on the right. Selected pairs will appear in the **Matched Celebrities** list on the left.
- To magnify a picture, hover your pointer above it.
- To unselect a selected pair, click on the pair in the list on the left.
- If none of the celebrities match, check the **I did not find any pairs** checkbox.
- There may be multiple matches per page.

Celebrity

Spotted Star

Matched Celebrities
To remove a pair added in error, click on the pair in the list below.

□ I did not find any pairs

Figure 3: A join task can take several forms as a HIT. In this example, turkers are asked to selet matching pictures in each column.

Figure 3 shows the two-column join interface for implementing joins in Query 2.

This portion of the demonstration will ensure that the audience's experience is *live*. As more audience members interact with the demonstration, the query workflows they contribute to will advance, and this progress will be visible in the Query Status Dashboard.

5. REFERENCES

[1] M. S. Bernstein et al. Soylent: a word processor with a crowd inside. In *UIST 2010*.

[2] M. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD 2011*.

[3] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*

[4] G. Little et al. TurKit: human computation algorithms on mechanical turk. In *UIST 2010*.

[5] A. Marcus, E. Wu, et al. Crowdsourced databases: Query processing with people. In *CIDR 2011*.

[6] A. Parameswaran and N. Polyzotis. Answering queries using humans, algorithms and databases. In *CIDR 2011*.