CrowdDB: Answering Queries with Crowdsourcing

Michael J. Franklin
AMPLab, UC Berkeley
franklin@cs.berkeley.edu

Donald Kossmann
Systems Group, ETH Zurich
donaldk@inf.ethz.ch

Tim Kraska
AMPLab, UC Berkeley
kraska@cs.berkeley.edu

Sukriti Ramesh
Systems Group, ETH Zurich
ramess@student.ethz.ch

Reynold Xin
AMPLab, UC Berkeley
rxin@cs.berkeley.edu

ABSTRACT

Some queries cannot be answered by machines only. Processing such queries requires human input for providing information that is missing from the database, for performing computationally difficult functions, and for matching, ranking, or aggregating results based on fuzzy criteria. CrowdDB uses human input via crowdsourcing to process queries that neither database systems nor search engines can adequately answer. It uses SQL both as a language for posing complex queries and as a way to model data. While CrowdDB leverages many aspects of traditional database systems, there are also important differences. Conceptually, a major change is that the traditional closed-world assumption for query processing does not hold for human input. From an implementation perspective, human-oriented query operators are needed to solicit, integrate and cleanse crowdsourced data. Furthermore, performance and cost depend on a number of new factors including worker affinity, training, fatigue, motivation and location. We describe the design of CrowdDB, report on an initial set of experiments using Amazon Mechanical Turk, and outline important avenues for future work in the development of crowdsourced query processing systems.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

General Terms

Human Factors, Languages, Design, Performance

1. INTRODUCTION

Relational database systems have achieved widespread adoption, not only in the business environments for which they were originally envisioned, but also for many other types of structured data, such as personal, social, and even scientific information. Still, as data creation and use become increasingly democratized through web, mobile and other technologies, the limitations of the technology are becoming more apparent. RDBMSs make several key assumptions about the correctness, completeness and unambiguity of the data they store. When these assumptions fail to hold, relational systems will return incorrect or incomplete answers to user questions, if they return any answers at all.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

1.1 Power to the People

One obvious situation where existing systems produce wrong answers is when they are missing information required for answering the question. For example, the query:

```
SELECT market_capitalization FROM company
WHERE name = "I.B.M.";
```

will return an empty answer if the company table instance in the database at that time does not contain a record for "I.B.M.". Of course, in reality, there are many reasons why such a record may be missing. For example, a data entry mistake may have omitted the I.B.M. record or the record may have been inadvertently deleted. Another possibility is that the record was entered incorrectly, say, as "I.B.N."

Traditional systems can erroneously return an empty result even when no errors are made. For example, if the record was entered correctly, but the name used was "International Business Machines" rather than "I.B.M." This latter "entity resolution" problem is not due to an error but is simply an artifact of having multiple ways to refer to the same real-world entity.

There are two fundamental problems at work here. First, relational database systems are based on the "Closed World Assumption": information that is not in the database is considered to be false or non-existent. Second, relational databases are extremely literal. They expect that data has been properly cleaned and validated before entry and do not natively tolerate inconsistencies in data or queries.

As another example, consider a query to find the best among a collection of images to use in a motivational slide show:

```
SELECT image FROM picture
WHERE topic = "Business Success"
ORDER BY relevance LIMIT 1;
```

In this case, unless the relevance of pictures to specific topics has been previously obtained and stored, there is simply no good way to ask this question of a standard RDBMS. The issue here is one of judgement: one cannot reliably answer the question simply by applying relational operators on the database.

All of the above queries, however, while unanswerable by today's relational database systems, could easily be answered by people, especially people who have access to the Internet.

1.2 CrowdDB

Recently, there has been substantial interest (e.g. [4, 6, 28]) in harnessing Human Computation for solving problems that are impossible or too expensive to answer correctly using computers. Microtask crowdsourcing platforms such as Amazon's Mechanical Turk (AMT)[3] provide the infrastructure, connectivity and payment mechanisms that enable hundreds of thousands of people to perform paid work on the Internet. AMT is used for many tasks that are easier for people than computers, from simple tasks such as labeling or segmenting images and tagging content to more complex tasks such as translating or even editing text.

It is natural, therefore, to explore how to leverage such human resources to extend the capabilities of database systems so that they can answer queries such as those posed above. The approach we take in CrowdDB is to exploit the extensibility of the iterator-based query processing paradigm to add crowd functionality into a DBMS. CrowdDB extends a traditonal query engine with a small number of operators that solicit human input by generating and submiting work requests to a microtask crowdsourcing platform.

As query processing elements, people and computers differ in fundamental ways. Obviously, the types of tasks at which each excel are quite different. Also, people exhibit wider variability, due to schedules such as nights, weekends and holidays, and the vast differences in expertise, diligence and temperament among people. Finally, as we discuss in the following sections, in a crowdsourcing system, the relationship between job requesters and workers extends beyond any one interaction and evolves over time. Care must be taken to properly maintain that relationship.

Crowdsourced operators must take into consideration the talents and limitations of human workers. For example, while it has been shown that crowds can be "programmed" to execute classical algorithms such as Quicksort on well-defined criteria [19], such a use of the crowd is neither performant nor cost-efficient.

Consider the example queries described in Section 1.1. Correctly answering these queries depends on two main capabilities of human computation compared to traditional query processing:

Finding new data - recall that a key limitation of relational technology stems from the Closed World Assumption. People, on the other hand, aided by tools such as search engines and reference sources, are quite capable of finding information that they do not have readily at hand.

Comparing data - people are skilled at making comparisons that are difficult or impossible to encode in a computer algorithm. For example, it is easy for a person to tell, if given the right context, if "I.B.M." and "International Business Machines" are names for the same entity. Likewise, people can easily compare items such as images along many dimensions, such as how well the image represents a particular concept.

Thus, for CrowdDB, we develop crowd-based implementations of query operators for finding and comparing data, while relying on the traditional relational query operators to do the heavy lifting for bulk data manipulation and processing. We also introduce some minimal extensions to the SQL data definition and query languages to enable the generation of queries that involve human computation.

1.3 Paper Overview

Our approach has four main benefits. First, by using SQL we create a declarative interface to the crowd. We strive to maintain SQL semantics so that developers are presented with a known computational model. Second, CrowdDB provides Physical Data Independence for the crowd. That is, application developers can write SQL queries without having to focus on which operations will be done in the database and which will be done by the crowd. Existing SQL queries can be run on CrowdDB, and in many cases will return more complete and correct answers than if run on a traditional DBMS. Third, as we discuss in subsequent sections, user interface design is a key factor in enabling questions to be answered by people. Our approach leverages schema information to support the automatic generation of effective user interfaces for crowdsourced tasks. Finally, because of its declarative programming environment and operator-based approach, CrowdDB presents the opportunity to implement cost-based optimizations to improve query cost, time, and accuracy — a facility that is lacking from most crowdsourcing platforms today.

Of course, there are many new challenges that must be addressed in the design of a crowd-enabled DBMS. The main ones stem from the fundamental differences in the way that people work compared to computers and from the inherent ambiguity in many of the human tasks, which often involve natural language understanding and matters of opinion. There are also technical challenges that arise due to the use of a specific crowdsourcing platform such as AMT. Such challenges include determining the most efficient organization of tasks submitted to the platform, quality control, incentives, payment mechanisms, etc.

In the following we explain these challenges and present initial solutions that address them. Our main contributions are:

- We propose simple SQL schema and query extensions that enable the integration of crowdsourced data and processing.
- We present the design of CrowdDB including new crowdsourced query operators and plan generation techniques that combine crowdsourced and traditional query operators.
- We describe methods for automatically generating effective user interfaces for crowdsourced tasks.
- We present the results of microbenchmarks of the performance of individual crowdsourced query operators on the AMT platform, and demonstrate that CrowdDB is indeed able to answer queries that traditional DBMSs cannot.

The remainder of this paper is organized as follows: Section 2 presents background on crowdsourcing and the AMT platform. Section 3 gives an overview of CrowdDB, followed by the CrowdSQL extension to SQL in Section 4, the user interface generation in Section 5, and query processing in Section 6. In Section 7 we present experimental results. Section 8 reviews related work. Section 9 presents conclusions and research challenges.

2. CROWDSOURCING

A crowdsourcing platform creates a marketplace on which requesters offer tasks and workers accept and work on the tasks. We chose to build CrowdDB using one of the leading platforms, Amazon Mechanical Turk (AMT). AMT provides an API for requesting and managing work, which enables us to directly connect it to the CrowdDB query processor. In this discussion, we focus on this particular platform and its interfaces.

AMT supports so-called microtasks. Microtasks usually do not require any special training and typically take no longer than one minute to complete; although in extreme cases, tasks can require up to one hour to finish [15]. In AMT, as part of specifying a task, a requester defines a price/reward (minimum $0.01) that the worker receives if the task is completed satisfactorily. In AMT currently, workers from anywhere in the world can participate but requesters must be holders of a US credit card. Amazon does not publish current statistics about the marketplace, but it contained over 200,000

workers (referred to as turkers) in 2006 [2], and by all estimates, the marketplace has grown dramatically since then [24].

2.1 Mechanical Turk Basics

AMT has established its own terminology. There are slight differences in terminology used by requesters and workers. For clarity in this paper, we use the requesters' terminology. Key terms are:

- **HIT**: A Human Intelligent Task, or HIT, is the smallest entity of work a worker can accept to do. HITs contain one or more jobs. For example, tagging 5 pictures could be one HIT. Note that "job" is not an official term used in AMT, but we use it in this paper where necessary.
- **Assignment**: Every HIT can be replicated into multiple assignments. AMT ensures that any particular worker processes at most a single assignment for each HIT, enabling the requester to obtain answers to the same HIT from multiple workers. Odd numbers of assignments per HIT enable majority voting for quality assurance, so it is typical to have 3 or 5 assignments per HIT. Requesters pay workers for each assignment completed satisfactorily.
- **HIT Group**: AMT automatically groups similar HITs together into HIT Groups based on the requester, the title of the HIT, the description, and the reward. For example, a HIT Group could contain 50 HITs, each HIT asking the worker to classify several pictures. As we discuss below, workers typically choose which work to perform based on HIT Groups.

The basic AMT workflow is as follows: A requester packages the jobs comprising his or her information needs into HITs, determines the number of assignments required for each HIT and posts the HITs. Requesters can optionally specify requirements that workers must meet in order to be able to accept the HIT. AMT Groups compatible HITs into HIT Groups and posts them so that they are searchable by workers. A worker accepts and processes assignments. Requesters then collect all the completed assignments for their HITs and apply whatever quality control methods they deem necessary.

Furthermore, requesters approve or reject each assignment completed by a worker: Approval is given at the discretion of the requester. Assignments are automatically deemed approved if not rejected within a time specified in the HIT configuration. For each approved assignment the requester pays the worker the pre-defined reward, an optional bonus, and a commission to Amazon.

Workers access AMT through their web browsers and deal with two kinds of user interfaces. One is the main AMT interface, which enables workers to search for HIT Groups, list the HITs in a HIT Group, and to accept assignments. The second interface is provided by the requester of the HIT and is used by the worker to actually complete the HIT's assignments. A good user interface can greatly improve result quality and worker productivity.

In AMT, requesters and workers have visible IDs so relationships and reputations can be established over time. For example, workers will often lean towards accepting assignments from requesters who are known to provide clearly-defined jobs with good user interfaces and who are known for having good payment practices. Information about requesters and HIT Groups are shared among workers via on-line forums.

2.2 Mechanical Turk APIs

A requester can automate his or her workflow of publishing HITs, etc. by using AMT's web service or REST APIs. The relevant (to CrowdDB) interfaces are:

- *createHIT(title, description, question, keywords, reward, duration, maxAssignments, lifetime)* $\rightarrow$ *HitID*: Calling this method creates a new HIT on the AMT marketplace. The *createHIT* method returns a *HitID* to the requester that is used to identify the HIT for all further communication. The *title, description,* and *reward* and other fields are used by AMT to combine HITs into HIT Groups. The *question* parameter encapsulates the user interface that workers use to process the HIT, including HTML pages. The *duration* parameter indicates how long the worker has to complete an assignment after accepting it. The *lifetime* attribute indicates an amount of time after which the HIT will no longer be available for workers to accept. Requesters can also constrain the set of workers that are allowed to process the HIT. CrowdDB, however, does not currently use this capability so the details of this and several other parameters are omitted for brevity.
- *getAssignmentsForHIT(HitID)* $\rightarrow$ *list(asnId, workerId , answer):* This method returns the results of all assignments of a HIT that have been provided by workers (at most, *maxAssignments* answers as specified when the requester created the HIT). Each answer of an assignment is given an *asnID* which is used by the requester to approve or reject that assignment (described next).
- *approveAssignment(asnID) / rejectAssignment(asnID)*: Approval triggers the payment of the reward to the worker and the commission to Amazon.
- *forceExpireHIT(HitID):* Expires a HIT immediately. Assignments that have already been accepted may be completed.

2.3 CrowdDB Design Considerations

The crowd can be seen as a set of specialized processors. Humans are good at certain tasks (e.g., image recognition) and relatively bad at others (e.g., number crunching). Likewise, machines are good and bad at certain tasks and it seems that people's and machines' capabilities complement each other nicely. It is this synergy that provides the opportunity to build a hybrid system such as CrowdDB. However, while the metaphor of humans as computing resources is powerful, it is important to understand when this metaphor works, and when it breaks down. In this section we highlight some key issues that have influenced the design of CrowdDB.

Performance and Variability: Obviously, people and machines differ greatly in the speed at which they work, the cost of that work, and the quality of the work they produce. More fundamentally, people show tremendous variability both from one individual to another and over time for a particular individual. Malicious behavior and "spamming" are also concerns. For CrowdDB these differences have implications for query planning, fault tolerance and answer quality.

Task Design and Ambiguity: Crowd-sourced tasks often have inherent ambiguities due to natural language and subjective requirements. The crowd requires a graphical user interface with human-readable instructions. Unlike programming language commands, such instructions can often be interpreted in different ways. Also, the layout and design of the interface can have a direct effect on the speed and accuracy with which people complete the tasks. The challenges here are that tasks must be carefully designed with the workers in mind and that quality-control mechanisms must be developed, even in cases where it is not possible to determine the absolute correctness of an answer.

Affinity and Learning: Unlike CPUs, which are largely fungible, crowd workers develop relationships with requesters and skills for certain HIT types. They learn over time how to optimize their revenue. Workers are hesitant to take on tasks for requesters who do not have a track record of providing well-defined tasks and pay-

ing appropriately and it is not uncommon for workers to specialize on specific HIT types (e.g., classifying pictures) or to favor HITs from certain requesters [16]. This behavior requires the CrowdDB design to take a longer-term view on task and worker community development.

Relatively Small Worker Pool: Despite the large and growing number of crowdsourcing workers, our experience, and that of others [18] is that the pool of workers available to work for any one requester is surprisingly small. This stems from a number of factors, some having to do with the specific design of the AMT web site, and others having to do with the affinity and learning issues discussed previously. For CrowdDB, this impacts design issues around parallelism and throughput.

Open vs. Closed World: Finally, as mentioned in the Introduction, a key difference between traditional data processing and crowd processing is that in the latter, there is an effectively unbounded amount of data available. Any one query operator could conceivably return an unlimited number of answers. This has profound implications for query planning, query execution costs and answer quality.

Having summarized the crowdsourcing platform, we now turn to the design and implementation of CrowdDB.

3. OVERVIEW OF CrowdDB

Figure 1 shows the general architecture of CrowdDB. An application issues requests using CrowdSQL, a moderate extension of standard SQL. Application programmers can build their applications in the traditional way; the complexities of dealing with the crowd are encapsulated by CrowdDB. CrowdDB answers queries using data stored in local tables when possible, and invokes the crowd otherwise. Results obtained from the crowd can be stored in the database for future use.

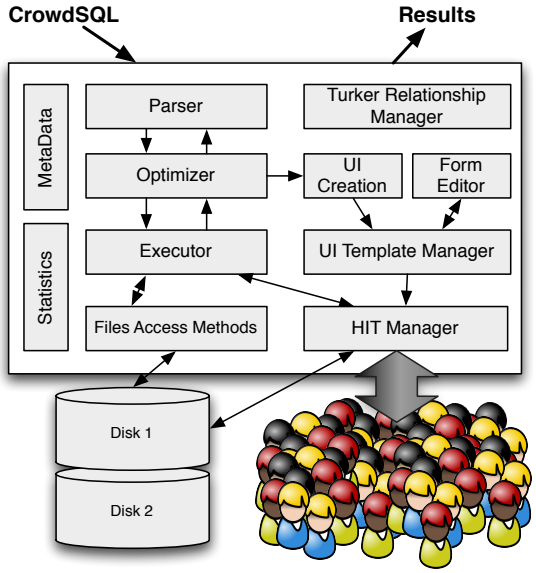

Figure 1: CrowdDB Architecture

As shown on the left side of the figure, CrowdDB incorporates the traditional query compilation, optimization and execution components. These components are extended to cope with human-generated input as described in subsequent sections. On the right side of the figure are new components that interact with the crowdsourcing platform. We briefly describe these from the top down.

Turker Relationship Management: As described in Section 2.3, the requester/worker relationship evolves over time and care must be taken to cultivate and maintain an effective worker pool. This module facilitates the most important duties of a requester, such as approving/rejecting assignments, paying and granting bonuses, etc. This way, CrowdDB helps to build communities for requesters which in the long run pays back in improved result quality, better response times, and lower cost.

User Interface Management: HITs require user interfaces and human-readable instructions. CrowdSQL extends SQL's data definition language to enable application developers to annotate tables with information that helps in user interface creation. At runtime, CrowdDB then automatically generates user interfaces for HITs based on these annotations as well as the standard type definitions and constraints that appear in the schema. Programmers can also create specific UI forms to override the standard UI generation in complex or exceptional cases.

The user interfaces generated for HITs are constrained where possible so that workers return only a pre-specified number of answers, and are designed to reduce errors by favoring simple tasks and limiting choices (i.e., adding dropdown lists or check boxes).

HIT Manager: This component manages the interactions between CrowdDB and the crowdsourcing platform (AMT in this case). It makes the API calls to post HITs, assess their status, and obtain the results of assignments. The HITs to be posted are determined by the query compiler and optimizer. The HIT manager also interacts with the storage engine in order to obtain values to preload into the HIT user interfaces and to store the results obtained from the crowd into the database.

The following sections explain the most important building blocks of CrowdDB in more detail: CrowdSQL, user interface generation, and query processing.

4. CrowdSQL

This section presents CrowdSQL, a SQL extension that supports crowdsourcing. We designed CrowdSQL to be a minimal extension to SQL to support use cases that involve missing data and subjective comparisons. This approach allows application programmers to write (Crowd) SQL code in the same way as they do for traditional databases. In most cases, developers are not aware that their code involves crowdsourcing. While this design results in a powerful language (CrowdSQL is a superset of SQL) with sound semantics, there are a number of practical considerations that such a powerful language entails. We discuss these issues and other extensions at the end of this section.

4.1 Incomplete Data

4.1.1 SQL DDL Extensions

Incomplete data can occur in two flavors: First, specific attributes of tuples could be crowdsourced. Second, entire tuples could be crowdsourced. We capture both cases by adding a special keyword, *CROWD*, to the SQL DDL, as shown in the following two examples.

EXAMPLE 1 (Crowdsourced Column) In the following *Department* table, the *url* is marked as crowdsourced. Such a model would be appropriate, say, if new departments are generated automatically (e.g., as part of an application program) and the *url* is often not provided but is likely available elsewhere.

```
CREATE TABLE Department (
  university STRING,
  name STRING,
  url  CROWD STRING,
```

```
  phone  STRING,
  PRIMARY KEY (university, name) );
```

EXAMPLE 2 (Crowdsourced Table) This example models a *Professor* table as a *crowdsourced table*. Such a crowdsourced table is appropriate if it is expected that the database typically only captures a subset of the professors of a department. In other words, CrowdDB will expect that additional professors may exist and possibly crowdsource more professors if required for processing specific queries.

```
CREATE CROWD TABLE Professor (
  name STRING PRIMARY KEY,
  email STRING UNIQUE,
  university STRING,
  department STRING,
  FOREIGN KEY (university, department)
     REF Department(university, name) );
```

CrowdDB allows any column and any table to be marked with the CROWD keyword. CrowdDB does not impose any limitations with regard to SQL types and integrity constraints; for instance, referential integrity constraints can be defined between two CROWD tables, two regular tables, and between a regular and a CROWD table in any direction. There is one exception: CROWD tables (e.g., *Professor*) must have a primary key so that CrowdDB can infer if two workers input the same new tuple (Section 4.3). Furthermore, CROWD columns and tables can be used in any SQL query and SQL DML statement, as shown in the following subsections.

4.1.2 SQL DML Semantics

In order to represent values in crowdsourced columns that have not yet been obtained, CrowdDB introduces a new value to each SQL type, referred to as *CNULL*. *CNULL* is the CROWD equivalent of the *NULL* value in standard SQL. *CNULL* indicates that a value should be crowdsourced when it is first used.

CNULL values are generated as a side-effect of `INSERT` statements. (The semantics of `DELETE` and `UPDATE` statements are unchanged in CrowdDB.) *CNULL* is the *default* value of any CROWD column. For instance,

```
INSERT INTO Department(university, name)
VALUES ("UC Berkeley", "EECS");
```

would create a new tuple. The *phone* field of the new department is initialized to *NULL*, as specified in standard SQL. The *url* field of the new department is initialized to *CNULL*, according to the special CrowdSQL semantics for CROWD columns. Both fields could be set later by an `UPDATE` statement. However, the *url* field could also be crowdsourced as a side-effect of queries, as described in the next subsection.

It is also possible to specify the value of a CROWD column as part of an `INSERT` statement; in this case, the value would not be crowdsourced (unless it is explicitly set to *CNULL* as part of an `UPDATE` statement); e.g.,

```
INSERT INTO Department(university, name, url)
VALUES ("ETH Zurich", "CS", "inf.ethz.ch");
```

It is also possible to initialize or set the value of a CROWD column to *NULL*; in such a case, crowdsourcing would not take effect. Furthermore, it is possible to explicitly initialize or set (as part of an `UPDATE` statement) the value of a CROWD column to *CNULL*.

CNULL is also important for CROWD tables, where *all* columns are implicitly crowdsourced. If an `INSERT` statement is executed on a CROWD table, then all non-key attributes would be initialized to *CNULL*, if not specified otherwise as part of the `INSERT` statement. However, the key of a CROWD table must be specified for the `INSERT` statement as the key is never allowed to contain *CNULL*. This rule is equivalent to the rule that keys must not be *NULL* enforced by most standard SQL database systems. However, new tuples of CROWD tables are typically crowdsourced entirely, as described in the next subsection.

In summary, CrowdSQL allows any kind of `INSERT`, `UPDATE`, and `DELETE` statements; all these statements can be applied to CROWD columns and tables. Furthermore, CrowdSQL supports all standard SQL types (including *NULL* values). A special *CNULL* value indicates data in CROWD columns that should be crowdsourced when needed as part of processing a query.

4.1.3 Query Semantics

CrowdDB supports any kind of SQL query on CROWD tables and columns; for instance, joins and sub-selects between two CROWD tables are allowed. Furthermore, the results of these queries are as expected according to the (standard) SQL semantics.

What makes CrowdSQL special is that it incorporates crowdsourced data as part of processing SQL queries. Specifically, CrowdDB asks the crowd to instantiate *CNULL* values if they are required to evaluate predicates of a query or if they are part of a query result. Furthermore, CrowdDB asks the crowd for new tuples of CROWD tables if such tuples are required to produce a query result. Crowdsourcing as a side-effect of query processing is best explained with the help of examples:

```
SELECT url FROM Department
WHERE name = "Math";
```

This query asks for the *url* of the Math department. If the *url* is *CNULL*, then the semantics of CrowdSQL require that the value be crowdsourced. The following query asks for all professors with Berkeley email addresses in the Math department:

```
SELECT * FROM Professor
WHERE email LIKE "%berkeley%" AND dept = "Math";
```

This query involves crowdsourcing the E-Mail address and department of all known professors (if their current value is *CNULL*) in order to evaluate the `WHERE` clause. Furthermore, processing this query involves asking the crowd for possibly additional Math professors that have not been crowdsourced yet.

CrowdSQL specifies that tables are updated as a side-effect of crowdsourcing. In the first query example above, for instance, the *url* column would be implicitly updated with the crowdsourced URL. Likewise, missing values in the *email* column would be implicitly populated and new professors would be implicitly inserted as a side-effect of processing the second query. Logically, these updates and inserts are carried out as part of the same transaction as the query and they are executed *before* the query; i.e., their effects are visible during query processing.

4.2 Subjective Comparisons

Recall that beyond finding missing data, the other main use of crowdsourcing in CrowdDB is subjective comparisons. In order to support this functionality, CrowdDB has two new built in functions: `CROWDEQUAL` and `CROWDORDER`.

`CROWDEQUAL` takes two parameters (an *lvalue* and an *rvalue*) and asks the crowd to decide whether the two values are equal. As syntactic sugar, we use the $\sim=$ symbol and an infix notation, as shown in the following example:

EXAMPLE 3 To ask for all "CS" departments, the following query could be posed. Here, the query writer asks the crowd to do entity resolution with the possibly different names given for Computer Science in the database:

```
SELECT profile FROM department
WHERE name ~= "CS";
```

CROWDORDER is used whenever the help of the crowd is needed to rank or order results. Again, this function is best illustrated with an example.

EXAMPLE 4 The following CrowdSQL query asks for a ranking of pictures with regard to how well these pictures depict the Golden Gate Bridge.

```
CREATE TABLE picture (
  p IMAGE,
  subject STRING
);
SELECT p FROM picture
WHERE subject = "Golden Gate Bridge"
ORDER BY CROWDORDER(p,
"Which picture visualizes better %subject");
```

As with missing data, CrowdDB stores the results of CROWDEQUAL and CROWDORDER calls so that the crowd is only asked once for each comparison. This *caching* is equivalent to the caching of expensive functions in traditional SQL databases, e.g., [13]. This cache can also be invalidated; the details of the syntax and API to control this cache are straightforward and omitted for brevity.

4.3 CrowdSQL in Practice

The current version of CrowdSQL was designed to be a minimal extension to SQL and address some of the use cases that involve incomplete data and subjective comparisons. This version was used in the first implementation of CrowdDB which was used as a platform for the experiments reported in Section 7. While this minimal extension is sound and powerful, there are a number of considerations that limit the usage of CrowdSQL in practice.

The first issue is that CrowdSQL changes the closed-world to an open-world assumption. Thus cost and response time of queries can be unbounded in CrowdSQL. If a query involves a CROWD table, then it is unclear how many tuples need to be crowdsourced in order to fully process the query. For queries that involve regular tables with CROWD columns or subjective comparisons, the amount of information that needs to be crowdsourced is bounded by the number of tuples and *CNULL* values stored in the database; nevertheless, the cost can be prohibitive. In order to be practical, CrowdSQL should provide a way to define a *budget* for a query. Depending on the use case, an application developer should be able to constrain the cost, the response time, and/or the result quality. For instance, in an emergency situation, response time may be critical whereas result quality should be given priority if a strategic decision of an organization depends on a query. In the current version of CrowdDB, we provide only one way to specify a budget — using a LIMIT clause [7]. The LIMIT clause constrains the number of tuples that are returned as a result of a query. This way, the LIMIT clause implicitly constrains the cost and response time of a query. A LIMIT clause (or similar syntax) is supported by most relational database systems and this clause has the same semantics in CrowdSQL as in (traditional) SQL. We used this mechanism because it was available and did not require any extensions to SQL syntax or semantics. It should be clear, however, that more sophisticated and explicit mechanisms to specify *budgets* are needed.

A second extension that we plan for the next version of CrowdSQL is *lineage*. In CrowdDB, data can be either crowdsourced as a side-effect of query processing or entered using SQL DML statements. In both cases, application developers may wish to query the lineage in order to take actions. For instance, it may become known that a certain worker is a spammer; in such a situation, it may be appropriate to reset all crowdsourced values to *CNULL* that were derived from input from that worker. Also, it may be important to know when data was entered in order to determine whether the data is outdated (i.e., time-to-live); the age of data may be relevant for both crowdsourced and machine-generated data.

A third practical issue involves the cleansing of crowdsourced data; in particular, entity resolution of crowdsourced data. As mentioned in Section 4.1, all CROWD tables must have a primary key. The current version of CrowdDB uses the primary key values in order to detect duplicate entries of CROWD tables. This approach works well if it can be assumed that two independent workers will enter exactly the same literals. In many cases, this assumption is not realistic. As part of future work, we plan to extend the DDL of CrowdSQL so that application developers can specify that data cleaning should be carried out on crowdsourced data (e.g., entity resolution). Of course, this feature comes at a cost so that developers should have control to use this feature depending on their budget.

5. USER INTERFACE GENERATION

Recall that a key to success in crowdsourcing is the provision of effective user interfaces. In this section we describe how CrowdDB automatically generates user interfaces for crowdsourcing incomplete information and subjective comparisons. As shown in Figure 1, the generation of user interfaces is a two-step process in CrowdDB. At compile-time, CrowdDB creates *templates* to crowdsource missing information from all CROWD tables and all regular tables which have CROWD columns. These user interfaces are HTML (and JavaScript) forms that are generated based on the CROWD annotations in the schema and optional free-text annotations of columns and tables that can also be found in the schema [21]. Furthermore, these templates can be edited by application developers to provide additional custom instructions. These templates are instantiated at runtime in order to provide a user interface for a concrete tuple or a set of tuples. The remainder of this section gives examples of user interfaces generated for the tables and subjective comparisons described in Section 4.

5.1 Basic Interfaces

Figure 2a shows an example user interface generated for crowdsourcing the *URL* of the "EECS" department of "UC Berkeley" (Example 1). The title of the HTML is the name of the Table (i.e., *Department* in this example) and the instructions ask the worker to input the missing information. In general, user interface templates are instantiated by copying the known field values from a tuple into the HTML form (e.g., "EECS" and "UC Berkeley" in this example). Furthermore, all fields of the tuple that have *CNULL* values become input fields of the HTML form (i.e., *URL* in this example).

In addition to HTML code, CrowdDB generates JavaScript code in order to check for the correct types of input provided by the workers. For instance, if the CrowdSQL involves a CHECK constraint that limits the domain of a crowdsourced attribute (e.g., "EUR", "USD", etc. for currencies), then a *select box* is generated that allows the worker to choose among the legal values for that attribute.

A possible optimization that is not shown in Figure 2a is to *batch*

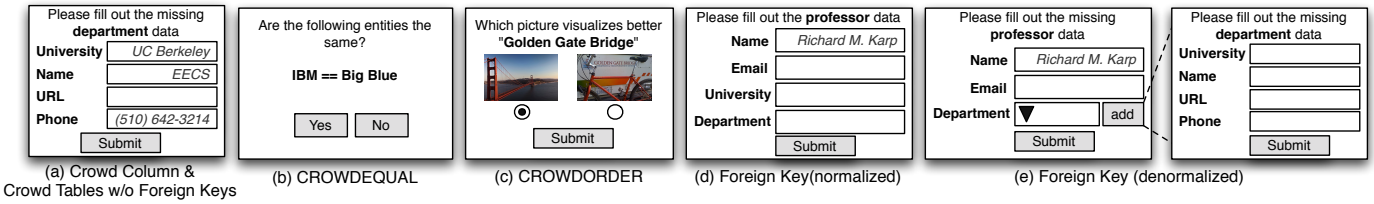

Figure 2: Example User Interfaces Generated by CrowdDB

several tuples. For instance, if the *URL* of several departments needs to be crowdsourced, then a single worker could be asked to provide the *URL* information of a set of department (e.g., EECS, Physics and Chemistry of UC Berkeley). The assumption is that it is cheaper to input two pieces of information of the same kind as part of a single form than twice as part of separate forms. CrowdDB supports this batching functionality.

Another optimization is *prefetching* of attributes for the same tuple. For instance, if both the *department* and *email* of a professor are unknown, but only the *email* of that professor is required to process a query, the question is whether the *department* should also be crowdsourced. The trade-offs are straight-forward: Prefetching may hurt the cost and response time of the current query, but it may help reduce the cost and response time of future queries. The current version of CrowdDB carries out prefetching, thereby assuming that the incremental cost of prefetching is lower than the return on the prefetching investment.

CrowdDB also creates user interfaces for the `CROWDEQUAL` and `CROWDORDER` functions. Figures 2b and 2c show example interfaces for comparing and ordering tuples. Both types of interfaces can also be batched as described earlier. However, for ordering tuples, our current implementation only supports binary comparisons. Multi-way comparisons remain future work.

5.2 Multi-Relation Interfaces

Crowdsourcing relations with foreign keys require special considerations. In the simplest case, the foreign key references a non-crowdsourced table. In this case, the generated user interface shows a drop-down box containing all the possible foriegn key values, or alternatively if the domain is too large for a drop-down box an Ajax-based "suggest" function is provided.

In contrast, if the referenced table is a CROWD table, the set of all possible foriegn key values is not known, and additional tuples of the referenced table may need to be crowdsourced. For the moment, let us assume that the *department* table is indeed a CROWD table (deviating from its definition in Example 1). In this case, a worker may wish to specify that a professor is member of a *new* department; i.e., a department that has not yet been obtained from the crowd.

CrowdDB supports two types of user interfaces for such situations. The first type is the *normalized* interface shown in Figure 2d. This interface requires the worker to enter values for the foreign key attributes, but does not allow him or her to enter any other fields of the referenced tuple. Here, as above, the "suggest" function can help to avoid entity resolution problems.

The second type is a *denormalized* interface, which allows a worker to enter missing values for non-key fields of the referenced table as part of the same task. To this end, the user interface provides an *add* button (in addition, the the drop down menu of *known* departments). Clicking on this *add* button brings up a *pop-up* window or new browser tab (depending on the browser configuration) that allows to enter the information of the new department as visualized in Figure 2e. How the optimizer chooses the interface for a particular query is explained in the next section.

Note that Figures 2d and 2e show the two types of user interfaces generated in the case where additional information is being crowdsourced for existing professor tuples. Similar user interfaces can be generated to crowdsource entirely *new* professor tuples. Such interfaces differ from these examples in the following ways: First, the key attribute(s) (i.e., the *name* of a professor in this case) becomes an input field. In order to restrict the set of possible new tuples, CrowdDB allows presetting the value of non-key attributes using the where clause, e.g. only considering professors from UC Berkeley. Second, JavaScript logic is generated that shows the names of professors that have already been recorded while typing in the name of a (potentially new) professor. Finally, an additional button is provided to allow workers to report if they cannot find any new professor records.

6. QUERY PROCESSING

Since CrowdDB is SQL-based, query plan generation and execution follows a largely traditional approach. In particular, as shown in Figure 1, CrowdDB has a parser, optimizer, and runtime system and these components play the same role in CrowdDB as in a traditional database system. The main differences are that the CrowdDB parser has been extended to parse CrowdSQL, CrowdDB has additional operators that effect crowdsourcing (in addition to the traditional relational algebra operators found in a traditional database system), and the CrowdDB optimizer includes special heuristics to generate plans with these additional Crowd operators. This section sketches these differences and discusses an example query plan.

6.1 Crowd Operators

CrowdDB implements all operators of the relational algebra, just like any traditional database system (e.g., join, group-by, index scans, etc.). In addition, CrowdDB implements a set of Crowd operators that encapsulate the instantiation of user interface templates at runtime and the collection of results obtained from crowdsourcing. This way, the implementation of traditional operators need not be changed. The next subsection describes how query plans are generated with such Crowd operators. This subsection describes the set of Crowd operators currently implemented in CrowdDB.

The basic functionality of all Crowd operators is the same. They are all initialized with a user interface template and the standard HIT parameters that are used for crowdsourcing as part of a particular Crowd operator. At runtime, they consume a set of tuples, e.g., *Professors*. Depending on the Crowd operator, crowdsourcing can be used to source missing values of a tuple (e.g., the *email* of a *Professor*) or to source new tuples. In both cases, each tuple represents one *job* (using the terminology from Section 2.1). Several tuples can be batched into a single *HIT* as described in Section 5.1; e.g., a user interface instance could be used to crowdsource the *email* of say, three, *Professors*. Furthermore, Crowd operators create *HIT Groups*. For instance, if the *email* of 300 professors needs to be sourced and each HIT involves three professors, then one HIT Group of 100 HITs could be generated. As shown in Section 7, when using AMT, the size of HIT Groups and how they are posted must be carefully tuned.

In addition to the creation of HITs and HIT Groups, each Crowd operator consumes results returned by the crowd and carries out

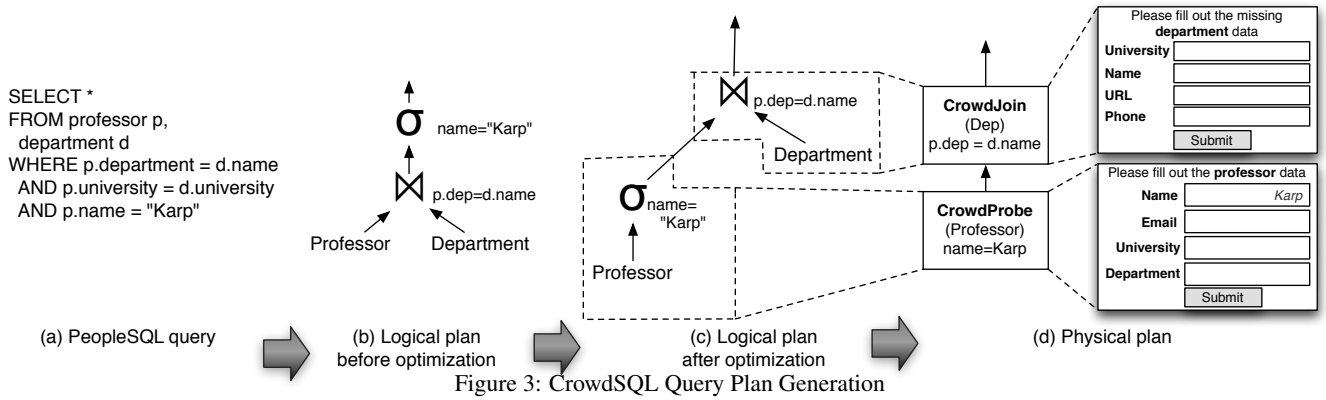

Figure 3: CrowdSQL Query Plan Generation

quality control. In the current CrowdDB prototype, quality control is carried out by a majority vote on the input provided by different workers for the same HIT. The number of workers assigned to each HIT is controlled by an *Assignments* parameter (Section 2.1). The initial number of *Assignments* is currently a static parameter of CrowdDB. As mentioned in Section 4.3, this parameter should be set by the CrowdDB optimizer based on budget constraints set via CrowdSQL.

The current version of CrowdDB has three Crowd operators:

- **CrowdProbe**: This operator crowdsources missing information of CROWD columns (i.e., *CNULL* values) and new tuples. It uses interfaces such as those shown in Figures 2a, 2d and 2e. The operator enforces quality control by selecting the majority answer for every attribute as the final value. That is, given the answers for a single tuple (i.e., entity with the same key), the majority of turkers have to enter the same value to make it the final value of the tuple. If no majority exists, more workers are asked until the majority agrees or a pre-set maximum of answers are collected. In the latter case, the final value is randomly selected from the values most workers had in common.

  In the case of newly created tuples it can happen that all workers enter tuples with different primary keys, making finding a majority impossible. In this case, the operator re-posts the tasks by leaving all non-confirmed attributes empty except the ones comprising the primary key. This allows CrowdDB to obtain more answers for every key in order to form a majority quorum.
- **CrowdJoin**: This operator implements an index nested-loop join over two tables, at least one of which is crowdsourced. For each tuple of the outer relation, this operator creates one or more HITs in order to crowdsource new tuples from the inner relation that matches the tuple of the outer relation. Correspondingly, the inner relation must be a CROWD table and the user interface to crowdsource new tuples from the inner relation is instantiated with the join column values of the tuple from the outer relation according to the join predicates. The quality control technique is the same as for CrowdProbe.
- **CrowdCompare**: This operator implements the *CROWDEQUAL* and *CROWDORDER* functions described in Section 4.2. It instantiates user interfaces such as those shown in Figures 2c and 2d. Note that CrowdCompare is typically used inside another traditional operator, such as sorting or predicate evaluation. For example, an operator that implements quick-sort might use CrowdCompare to perform the required binary comparisons. Quality control is based on the simple majority vote.

6.2 Physical Plan Generation

Figure 3 presents an end-to-end example that shows how CrowdDB creates a query plan for a simple CrowdSQL query. A query is first parsed; the result is a *logical plan*, as shown in Figure 3b. This *logical plan* is then optimized using traditional and crowd-specific optimizations. Figure 3c shows the optimized logical plan for this example. In this example, only predicate push-down was applied, a well-known traditional optimization technique. Some crowd-specific optimization heuristics used in CrowdDB are described in the next subsection. Finally, the *logical* plan is translated into a *physical plan* which can be executed by the CrowdDB runtime system. As part of this step, Crowd operators and traditional operators of the relational algebra are instantiated. In the example of Figure 3, the query is executed by a CrowdProbe operator in order to crowdsource missing information from the *Professor* table and a CrowdJoin operator in order to crowdsource missing information from the *Department* table. (In this example, it is assumed that the *Department* is a CROWD table; otherwise, the CrowdJoin operator would not be applicable.)

6.3 Heuristics

The current CrowdDB compiler is based on a simple rule-based optimizer. The optimizer implements several essential query rewriting rules such as predicate push-down, stopafter push-down [7], join-ordering and determining if the plan is bounded [5]. The last optimization deals with the open-world assumption by ensuring that the amount of data requested from the crowd is bounded. Thus, the heuristic first annotates the query plan with the cardinality predictions between the operators. Afterwards, the heuristic tries to re-order the operators to minimize the requests against the crowd and warns the user at compile-time if the number of requests cannot be bounded.

Furthermore, we also created a set of crowd-sourcing rules in order to set the basic crowdsourcing parameters (e.g., price, batching-size), select the user interface (e.g., normalized vs. denormalized) and several other simple cost-saving techniques. For example, a delete on a crowd-sourced table does not try to receive all tuples satisfying the expression in the delete statement before deleting them. Instead the optimizer rewrites the query to only look into existing tuples.

Nevertheless, in contrast to a cost-based optimizer, a rule-based optimizer is not able to exhaustively explore all parameters and thus, often produces a sub-optimal result. A cost-based optimizer for CrowdDB, which must also consider the changing conditions on AMT, remains future work.

7. EXPERIMENTS AND RESULTS

This section presents results from experiments run with CrowdDB and AMT. We ran over 25,000 HITs on AMT during October 2010, varying parameters such as price, jobs per HIT and time of day. We measured the response time and quality of the answers provided by the workers. Here, we report on Micro-benchmarks (Section 7.1), that use simple jobs involving finding new data or making subjective comparisons. The goals of these experiments are to ex-

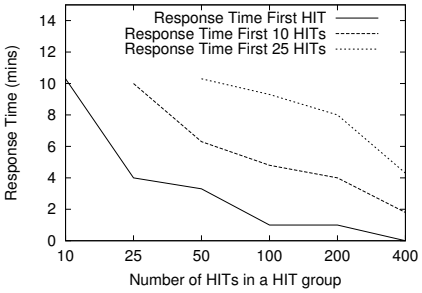

Figure 4: Response Time (min): Vary Hit Group *(1 Asgn/HIT, 1 cent Reward)*

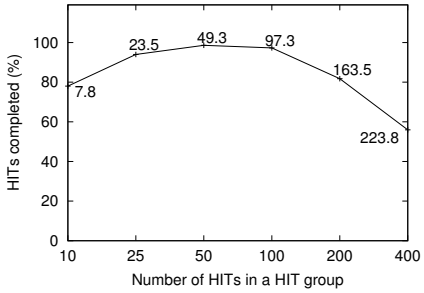

Figure 5: Completion (%): Vary Hit Group *(1 Asgn/HIT, 1 cent Reward)*

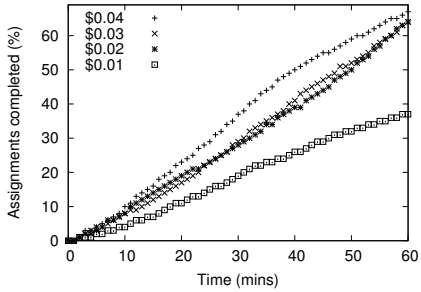

Figure 6: Completion (%): Vary Reward *(100 HITs/Group, 5 Asgn/HIT)*

amine the behavior of workers for the types of tasks required by CrowdDB, as well as to obtain insight that could be used in the development of cost models for query optimization. We also describe the results of experiments that demonstrate the execution of more complex queries (Section 7.2).

It is important to note that the results presented in this section are highly dependent on the properties of the AMT platform at a particular point in time (October 2010). AMT and other crowdsourcing platforms are evolving rapidly and it is quite likely that response time and quality properties will differ significantly between different platforms and even different versions of the same platform. Furthermore, our results are highly-dependent on the specific tasks we sent to the crowd. Nevertheless, we believe that a number of interesting and important observations can be made even from these initial experiments. We describe these below.

7.1 Micro Benchmarks

We begin by describing the results of experiments with simple tasks requiring workers to find and fill in missing data for a table with two crowdsourced columns:

```
CREATE TABLE businesses (
  name VARCHAR PRIMARY KEY,
  phone_number CROWD VARCHAR(32),
  address CROWD VARCHAR(256)
);
```

We populated this table with the names of 3607 businesses (restaurants, hotels, and shopping malls) in 40 USA cities. We studied the sourcing of the *phone_number* and *address* columns using the following query:

```
SELECT phone_number, address FROM businesses;
```

Since workers and skills are not evenly distributed across timezones, the time of day can impact both response time and answer quality [24]. While we ran tasks at many different times, in this paper we report only on experiments that were carried out between 0600 and 1500 PST, which reduces the variance in worker availability [14]. We repeated all experiments four times and provide the average values. Unless stated otherwise, groups of 100 HITs were posted and each HIT involved five assignments. By default the reward for each job was 1 cent and each HIT asked for the address and phone number of one business (i.e., 1 Job per HIT).

7.1.1 Experiment 1: Response Time, Vary HIT Groups

As mentioned in Section 2, AMT automatically groups HITs of the same kind into a HIT Group. In the first set of experiments we examined the response time of assignments as a function of the number of HITs in a HIT Group. For this experiment we varied the number of HITs per HIT Group from 10 to 400 and fixed the number of assignments per HIT to 1 (i.e., no replication). Figures 4 and 5 show the results of this experiment.

Figure 4 shows the time to completion of 1, 10, and 25 HITs as the HIT Group size is varied. The results show that response times decrease dramatically as the size of the HIT Groups is increased. For example, for a HIT Group of 25 HITs, the first result was obtained after approximately four minutes, while for a HIT Group containing 400 HITs, the first result was obtained in seconds.

Figure 5 provides a different view on the results of this experiment. In this case, we show the percentage of the HITs in a HIT Group completed within 30 minutes, for HIT Groups of different sizes. The points are annotated with the absolute number of HITs completed within 30 minutes. This graph shows that there is a tradeoff between throughput (in terms of HITs completed per unit time) and completion percent. That is, while the best throughput obtained was for the largest Group size (223.8 out of a Group of 400 HITs), the highest completion rates were obtained with Groups of 50 or 100 HITs. The implication is that performance for simple tasks can vary widely even when only a single parameter is changed. Thus, more work is required for understanding how to set the HIT Group size and other parameters — a prerequisite for the development of a true cost-based query optimizer for CrowdDB or any crowdsourced query answering system.

7.1.2 Experiment 2: Responsiveness, Vary Reward

While Experiment 1 showed that even a fairly technical parameter such as HIT Group size can greatly impact performance, there are other parameters that one would perhaps more obviously expect to influence the performance of a crowdbased system. High on this list would be the magnitude of the reward given to workers. Here, we report on a set of experiments that examine how the response time varies as a function of the reward. In these experiments, 100 HITs were posted per HIT Group and each HIT contained five assignments. The expectation is that the higher the reward for a HIT, the faster workers will engage into processing that HIT.

Figure 6 shows the fraction of HITs (i.e., all five assignments) that were completed as a function of time. The figure shows the results for HITs with a reward of (from the bottom up) 1, 2, 3, and 4 cents per assignment. Obviously, for all rewards, the longer we wait the more HITs are completed. The results show that for this particular task, paying 4 cents gives the best performance, while paying 1 cent gives the worst. Interestingly, there is very little difference between paying 2 cents and 3 cents per assignment.

Figure 7 shows the fraction of HITs that received at least one assignment as a function of time and reward. Comparing Figures 6 and 7, two observations can be made. First, within sixty minutes almost all HITs received at least one answer if a reward of 2, 3, or 4 cents was given (Figure 7), whereas only about 65% of the HITs

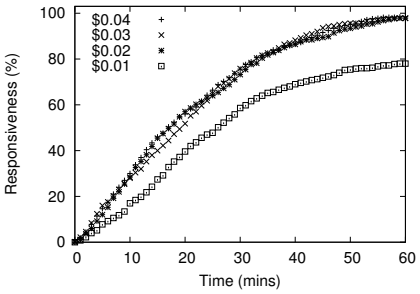

Figure 7: Completion (%): Vary Reward *(100 HITs/Group, 5 Asgn/HIT)*

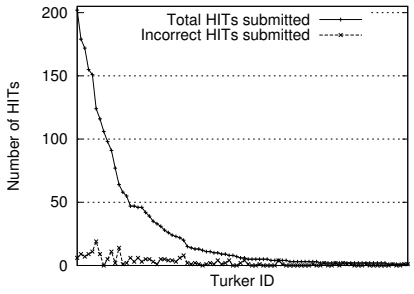

Figure 8: HITs/Quality by Worker *(Any HITs/Group, 5 Asgn/HIT, Any Reward)*

Non Uniform Name	Query Result	Votes
Bayerische Motoren Werke	BMW	3
International Business Machines	IBM	2
Company of Gillette	P&G	2
Big Blue	IBM	2

Figure 9: Entity Resolution on Company Names

were fully completed even for the highest reward of 4 cents (Figure 6). Second, if a single response to a HIT is acceptable, there is little incentive to pay more than 2 cents in this case.

As in Experiment 1, the results here show that parameters must be set carefully. In this case, of course, the parameter has a direct effect on the monetary cost of obtaining the data from the crowd, so setting it correctly for a given situation will be of significant importance, and tradeoffs between cost and response time will need to be exposed to users of the system.

7.1.3 Experiment 3: Worker Affinity and Quality

The previous experiments focused on the response time and throughput of the crowd. Here, we turn our attention to two other issues: distribution of work among workers and answer quality. In this experiment, every HIT had five assignments. We carried out majority votes among the five answers for phone numbers and computed the *ground truth* from this majority vote. Any answer that deviated from this ground truth (modulo formatting differences) was counted as an error.

Figure 8 shows the results. For each worker, Figure 8 shows the number of HITs computed by that worker and the number of errors made by that worker. In Figure 8, the workers are plotted along the x-axis in decreasing order of the number of HITs they processed. As can be seen, the distribution of workers performing HITs is highly skewed; this result confirms previous studies that have shown that requesters acquire communities of workers who specialize in processing their requests [15]. It seems that we were able to build a community of about a dozen fans.

We expected that one benefit of having such a community would be increased accuracy as workers gained experience with the task. However, we did not observe such an effect in our experiments. The lower curve in Figure 8 shows the number of errors made by each worker. In absolute numbers, heavy hitters, of course, have more errors. But in terms of error rate (Incorrect HITs / Total HITs, not shown) they were no more accurate than other workers, at least for the relatively simple task we examined here.

Note that the results shown in Figure 8 include HITs from the full reward range of 1 to 4 cents. In this range the reward had no noticeable impact on the error rate of the workers. We also observed that in our experiments the error rates did not depend on the HIT Group size.

7.2 Complex Queries

We now describe three experiments using CrowdSQL queries that cannot be asked of traditional database systems.

7.2.1 Entity Resolution on Companies

For this experiment, we used a simple company schema with two attributes, *company name* and *headquarter address*, and populated it with the Fortune 100 companies. We ran the following query to test entity resolution in CrowdDB:

```
SELECT name FROM company WHERE
  name~="[a non-uniform name of the company]"
```

In Figure 9 we show results for four different instances of this query. Each HIT involved comparing ten company names with one of the four "non-uniform names". Furthermore, each HIT had three assignments and the reward was 1 cent per HIT. Figure 9 shows that in all four cases, majority vote produced the correct answer. However, in only one of the cases (for BMW) was the vote unanimous. Note that the total time to complete the 40 HITs (i.e., 120 assignments) for these four queries was 39 minutes.

7.2.2 Ordering Pictures

For this experiment, we ran the query of Example 4 of Section 4; i.e., we asked for a ranking of pictures in different subject areas. Overall, we tested 30 subject areas having eight pictures for each subject area. Each HIT involved the comparison of four pairs of pictures. These rankings were conducted with 210 HITs, each with three assignments. It took 68 minutes to complete the experiment.

Figure 10 shows the ranking of the Top 8 pictures for the "Golden Gate" subject area. More concretely, Figure 10 shows the picture, the number of workers that voted for that picture, the ranking of that picture according to the majority vote, vs. the ranking according to a group of six experts who frequently visit the Golden Gate Bridge. In this case the CrowdDB results obtained via AMT match well with the ranks given by the experts.

7.2.3 Joining Professors and Departments

The last experiment we present compares the performance of two alternative plans for a join query. We populated the schema of Example 1 and 2 with 25 professors and 8 departments and asked the following SQL query:

```
SELECT p.name, p.email, d.name, d.phone
FROM   Professor p, Department d
WHERE  p.department = d.name AND
       p.university = d.university AND
       p.name = "[name of a professor]"
```

The first plan we executed for this query was the plan shown in Figure 3d (Section 6). That is, we first ask for the *Professor* information and the department the professor is associated with. Then in a second step, we ask for the remaining information (i.e., phone number) of the departments. The second plan was a *denormalized* variant that asked for the *Professor* and *Department* information in a single step, thereby creating only one type of HIT and the user interface of Figure 2e. For both plans, each HIT involved one job

(a) 15, 1, 1 (b) 15, 1, 2 (c) 14, 3, 4 (d) 13, 4, 5

(e) 10, 5, 6 (f) 9, 6, 3 (g) 4, 7, 7 (h) 4, 7, 8

Figure 10: Pictures of the Golden Gate Bridge [1] ordered by workers. The tuples in the sub-captions is in the following format: {the number of votes by the workers for this picture, rank of the picture ordered by the workers (based on votes), rank of the picture ordered by experts}.

(e.g., an entry form for a single professor) and three assignments. The reward per HIT was 1 cent.

The two plans were similar in execution time and cost but differed significantly in result quality. The first plan took 206 minutes and cost $0.99, while the second plan took 173 minutes and cost $0.75. In terms of quality, the second plan produced wrong results for all department telephone numbers. A close look at the data revealed that workers unanimously submitted the professors' phone numbers instead of the departments'. In contrast, only a single telephone number was incorrect when using the first plan.

7.3 Observations

In total, our experiments involved 25,817 assignments processed by 718 different workers. Overall, they confirm the basic hypothesis of our work: it is possible to employ human input via crowdsourcing to process queries that traditional systems cannot. The experiments also, however, pointed out some of the challenges that remain to be addressed, particularly in terms of understanding and controlling the factors that impact response time, cost and result quality. In this section, we outline several additional lessons we learned during our experimentation.

First, we observe that unlike computational resources, crowd resources involve long-term memory that can impact performance. Requesters can track workers' past performance and workers can track and discuss the past behavior of requesters. We found, for example, that to keep workers happy, it is wise to be less strict in approving HITs. In one case, when we rejected HITs that failed our quality controls, a worker posted the following on TurkOpticon [27]: "Of the 299 HITs I completed, 11 of them were rejected...I have attempted to contact the requester and will update...Until then be very wary of doing any work for this requester...". In another case, a bug in CrowdDB triggered false alarms in security warnings for browsers on the workers' computers, and within hours concerns about our jobs appeared on Turker Nation [26].

A second lesson is that user interface design and precise instructions matter. For our first experiments with AMT, we posted HITs with handcrafted user interfaces. In one HIT, we gave a list of company names and asked the workers to check the ones that matched "IBM". To our surprise, the quality of the answers was very low. The problem was that in some cases there were no good matches. We expected workers to simply not check any boxes in such cases, but this did not occur. Adding a checkbox for "None of the above" improved the quality dramatically.

Our experimental results, along with these anecdotes, demonstrate the complexity and new challenges that arise for crowdsourced query processing. They also show, we believe, the need to automate such decisions to shield query and application writers from these complexities and to enable robust behavior in the presence of environmental changes. Such a need should be familiar to database systems designers — it is essentially a requirement for a new form of *data independence*.

8. RELATED WORK

We discuss related work in two primary areas: database systems and the emerging crowdsourcing community.

CrowdDB leverages traditional techniques for relational query processing wherever possible. Furthermore, we found that some less-standard techniques developed for other scenarios were useful for crowdsourcing as well. For example, crowdsourced comparisons such *CROWDEQUAL*, can be seen as special instances of expensive predicates [13]. Top N optimizations [7] are a useful tool for dealing with the open-world nature of crowdsourcing, particularly when combined with operation-limiting techniques such as those developed for cloud scalability in [5]. Looking forward, we anticipate that the volatility of crowd performance will require the use of a range of adaptive query processing techniques [12].

An important feature of CrowdDB is the (semi-) automatic generation of user interfaces from meta-data (i.e., SQL schemas). As mentioned in Section 5, products such as Oracle Forms and MS Access have taken a similar approach. Model-driven architecture frameworks (MDA) enable similar functionality [17]. Finally, there are useful analogies between CrowdDB and federated database systems (e.g., [8, 11]): CrowdDB can be considered to be a mediator, the crowd can be seen as a data sources and the generated user interfaces can be regarded as wrappers for those sources.

As stated in the introduction, crowdsourcing has been gathering increasing attention in the research community. A recent survey of the area can be found in [10]. There have been a number of studies that analyze the behavior of microtask platforms such as Mechanical Turk. Ipeirotis, for instance, analyzed the AMT marketplace, gathering statistics about HITs, requesters, rewards, HIT completion rates, etc. [15]. Furthermore, the demographics of workers (e.g., age, gender, education, etc.) on AMT have been studied [24].

Systems making early attempts to automatically control quality and optimize response time include CrowdSearch [28] and Soylent [6]. TurKit is a set of tools that enables programming iterative algorithms over the crowd [19]. This toolkit has been used, for example, to decipher unreadable hand-writing. Finally, Usher is a research project that studies the design of data entry forms [9].

Independently of our work on CrowdDB, several groups in the

database research community have begun to explore the use of crowdsourcing in relational query processing. Two papers on the topic appeared in the recent CIDR 2011 conference [20, 23]. As with our work, both of these efforts suggest the use of a declarative query language in order to process queries involving human input. [20] proposes the use of user-defined functions in SQL to specify the user interfaces and the questions that should be crowdsourced as part of query processing. [23] suggests the use of Datalog for this purpose. However, neither of these early papers present details of the system design or experimental results. As these projects mature, it will be interesting to compare the design decisions made and implications with regard to usability and optimizability. Also related is recent work in the database community on graph searching using crowdsourcing [22]. Finally, there has also been work on defining workflow systems that involve machines and humans. Dustdar et al. describe in a recent vision paper how BPEL and web services technology could be used for this purpose [25].

9. CONCLUSION

This paper presented the design of CrowdDB, a relational query processing system that uses microtask-based crowdsourcing to answer queries that cannot otherwise be answered. We highlighted two cases where human input is needed: (a) unknown or incomplete data, and (b) subjective comparisons. CrowdDB extends SQL in order to address both of these cases and it extends the query compiler and runtime system with auto-generated user interfaces and new query operators that can obtain human input via these interfaces. Experiments with CrowdDB on Amazon Mechanical Turk demonstrated that human input can indeed be leveraged to dramatically extend the range of SQL-based query processing.

We described simple extensions to the SQL DDL and DML for crowdsourcing. These simple extensions, however, raise deep and fundamental issues for the design of crowd-enabled query languages and execution engines. Perhaps most importantly, they invalidate the closed-world assumption on which SQL semantics and processing strategies are based. We identified implementation challenges that arise as a result and outlined initial solutions.

Other important issues concern the assessment and preservation of answer quality. Quality is greatly impacted by the motivation and skills of the human workers as well as the structure and interfaces of the jobs they are asked to do. Performance, in terms of latency and cost, is also effected by the way in which CrowdDB interacts with workers. Through a series of micro-benchmarks we identified key design parameters such as the grouping of tasks, the rewards paid to workers, and the number of assignments per task. We also discussed the need to manage the long-term relationship with workers. It is important to build a community of workers and to provide them with timely and appropriate rewards and, if not, to give precise and understandable feedback.

Of course, there is future work to do. Answer quality is a pervasive issue that must be addressed. Clear semantics in the presence of the open-world assumption are needed for for both crowdsourced values and tables. While our existing approach preserves SQL semantics in principle, there are practical considerations due to cost and latency that can impact the answers that will be returned by any crowdsourced query processor.

There are also a host of exciting implementation challenges. Cost-based optimization for the crowd involves many new parameters and considerations. Time-of-day, pricing, worker fatigue, task user-interface design, etc. all impact performance and quality. Adaptive optimization techniques will clearly be needed. Caching and managing crowdsourced answers, if done properly, can greatly improve performance as well. Answer quality assessment and improvement will require the development of new techniques and operators.

The combination of human input with high-powered database processing not only extends the range of existing database systems, but also enables completely new applications and capabilities. For this reason, we expect this to be a fruitful area of research and development for many years.

Acknowledgements

This work was supported in part by AMPLab founding sponsors Google and SAP, and AMPLab sponsors Amazon Web Services, Cloudera, Huawei, IBM, Intel, Microsoft, NEC, NetApp, and VMWare.

10. REFERENCES

[1] Pictures of the Golden Gate Bridge retrieved from Flickr by akaporn, Dawn Endico, devinleedrew, di_the_huntress, Geoff Livingston, kevincole, Marc_Smith, and superstrikertwo under the Creative Commons Attribution 2.0 Generic license.

[2] Amazon. AWS Case Study: Smartsheet, 2006.

[3] Amazon Mechanical Turk. http://www.mturk.com, 2010.

[4] S. Amer-Yahia et al. Crowds, Clouds, and Algorithms: Exploring the Human Side of "Big Data" Applications. In *SIGMOD*, 2010.

[5] M. Armbrust et al. PIQL: A Performance Insightful Query Language. In *SIGMOD*, 2010.

[6] M. S. Bernstein et al. Soylent: A Word Processor with a Crowd Inside. In *ACM SUIST*, 2010.

[7] M. J. Carey and D. Kossmann. On saying "Enough already!" in SQL. *SIGMOD Rec.*, 26(2):219–230, 1997.

[8] S. S. Chawathe et al. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*, 1994.

[9] K. Chen et al. USHER: Improving Data Quality with Dynamic Forms. In *ICDE*, pages 321–332, 2010.

[10] A. Doan, R. Ramakrishnan, and A. Halevy. Crowdsourcing Systems on the World-Wide Web. *CACM*, 54:86–96, Apr. 2011.

[11] L. M. Haas et al. Optimizing Queries Across Diverse Data Sources. In *VLDB*, 1997.

[12] J. M. Hellerstein et al. Adaptive Query Processing: Technology in Evolution. *IEEE Data Eng. Bull.*, 2000.

[13] J. M. Hellerstein and J. F. Naughton. Query Execution Techniques for Caching Expensive Methods. In *SIGMOD*, pages 423–434, 1996.

[14] E. Huang et al. Toward Automatic Task Design: A Progress Report. In *HCOMP*, 2010.

[15] P. G. Ipeirotis. Analyzing the Amazon Mechanical Turk Marketplace. http://hdl.handle.net/2451/29801, 2010.

[16] P. G. Ipeirotis. Mechanical Turk, Low Wages, and the Market for Lemons. http://behind-the-enemy-lines.blogspot.com/2010/07/mechanical-turk-low-wages-and-market.html, 2010.

[17] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[18] G. Little. How many turkers are there? http://groups.csail.mit.edu/uid/deneme/?p=502, 2009.

[19] G. Little et al. TurKit: Tools for Iterative Tasks on Mechanical Turk. In *HCOMP*, 2009.

[20] A. Marcus et al. Crowdsourced Databases: Query Processing with People. In *CIDR*, 2011.

[21] Microsoft. Table Column Properties (SQL Server), 2008.

[22] A. Parameswaran et al. Human-Assisted Graph Search: It's Okay to Ask Questions. In *VLDB*, 2011.

[23] A. Parameswaran and N. Polyzotis. Answering Queries using Humans, Algorithms and Databases. In *CIDR*, 2011.

[24] J. Ross et al. Who are the Crowdworkers? Shifting Demographics in Mechanical Turk. In *CHI EA*, 2010.

[25] D. Schall, S. Dustdar, and M. B. Blake. Programming Human and Software-Based Web Services. *Computer*, 43(7):82–85, 2010.

[26] Turker Nation. http://www.turkernation.com/, 2010.

[27] Turkopticon. http://turkopticon.differenceengines.com/, 2010.

[28] T. Yan, V. Kumar, and D. Ganesan. CrowdSearch: Exploiting Crowds for Accurate Real-time. Image Search on Mobile Phones. In *MobiSys*, 2010.