Query Processing with People

ABSTRACT

Some queries cannot be answered by machines only. Such queries involve human input for providing information that is not readily available on the web and for matching, ranking, or aggregating results. This paper describes the design and implementation of PeopleDB*, a database system that uses crowd sourcing to integrate human input in order to process queries that Google and Oracle cannot answer. To a large extent, PeopleDB is based on the same design principles as traditional database systems. SQL is used both as a language to ask complex queries and as a way to model data stored electronically and provided by human input. Furthermore, queries are automatically compiled and optimized. Special operators provide user interfaces in order to integrate and cleanse human input. While a great deal of concepts of traditional database systems can be leveraged, there are also important differences. Human input is virtually infinite so that the system is based on an open world assumption. Furthermore, performance and cost depend on a number of different factors: Humans can be trained, but they also can get bored. Real-life experiments conducted with PeopleDB and Mechanical Turk show that indeed PeopleDB is able to process queries that cannot be processed with traditional database systems. Furthermore, these experiments assess response time, cost (in $), and result quality depending on a number of different parameters such as the financial reward promised to humans and the way jobs are posted.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems; H.2.3 [**Database Management**]: Languages; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms

Human Factors, Languages, Design, Performance

*Name changed for double-blind reviewing.

SIGMOD'11, June 12–16, 2010, Athens, Greece.

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, web, and even scientific information. Still, as data creation and use becomes 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.

1.1 Impossible Queries

The most obvious case in which current database systems fail to produce a correct answer is when they do not contain the information that is required for answering the question. For example, the SQL 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 resulted in the I.B.M. information being skipped over or the record may have been inadvertently deleted.

There are other reasons why the above query may incorrectly return an empty answer. Perhaps a corresponding record was entered, but with a typo, so that the name was entered as "I.B.N." rather than "I.B.M.". Another possibility may be that the record was entered correctly, but the name used was "International Business Machines". While the previous examples are the result of errors, this latter "entity resolution" problem 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"; that is, 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. They do not natively tolerate inconsistencies and errors in either the data or the queries.

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

```
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 - the notion of relevance in this case is inherently subjective so one cannot reliably answer the question simply by applying relational operators on the database. The above queries, however, which are unanswerable by today's relational database systems, could easily be answered by people, especially people who have access to the Internet.

1.2 Answering Impossible Queries

Recently, there has been substantial interest (e.g. [3, 6, 31]) 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 (MT)[2] provide the infrastructure, connectivity and payment mechanisms that enable hundreds of thousands of people to perform paid work on the Internet. MT is used for many tasks that are easier for people than computers, from simple tasks such as labeling or segmenting images, tagging content, and answering polls, 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. In this paper, we start from the premise that the requirement is to answer database-style queries, expressed in SQL, and explore how to extend the range of situations in which such queries can provide high-quality answers. Thus, our goal is to examine how to judiciously and effectively add people into the process of query answering.

We argue for a principled integration of traditional database query processing and crowdsourcing so that the strengths of computer technology and human brainpower can be combined to create a system that is more capable than either computers or people alone.

1.3 Alternative Approaches

Query-granularity: A seemingly straightforward way to incorporate people into the query answering process would be to use a coarse-grained, per query approach. When a query is presented to the system, it could be analyzed to determine if it is answerable by the DBMS. If so, then it is processed as usual. Otherwise, it is sent to the crowd. Such an approach, however, has numerous drawbacks. First, on a system like MT, response times are measured, at best, in minutes, and the minimum pay for a task is $0.01. Thus, the costs and latencies of queries involving potentially large data sets can be prohibitive. Second, there are limits to the complexity of tasks that can reliably be done on a general microtask system such as MT, and it is easy to generate tasks that cannot be completed. Finally, since our input is SQL queries and schemas, and most people are not SQL-literate, there is a challenge in unambiguously translating SQL into requests that are understandable by workers.

Operator-granularity: An alternative approach is to construct "crowd query operators", that are effectively people-oriented analogs of the relational query operators. This approach is more flexible and more promising than the query-granularity approach but also has significant pitfalls. As query processing elements, people and computers differ in fundamental ways. Obviously, the types of tasks at which each excel are quite different. For example, asking a crowd to perform a Quicksort operation is feasible [20], but not particularly efficient. 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 care must be taken to properly maintain that relationship.

The above considerations lead us to adopt an operator-granularity approach, but one that takes into consideration the similarities and differences between human workers and machines in order to improve the utility and efficiency of the hybrid system.

1.4 PeopleDB

PeopleDB is an operator-granularity system that employs a limited number of crowd-oriented operators, chosen specifically to leverage the talents and capabilities of human computation. Recall the example "impossible" queries given above. These queries rely 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, in PeopleDB, we develop crowd-based implementations of query operators for finding and comparing data, and rely 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.

Our approach has three main benefits. First, it provides Physical Data Independence for the crowd. That is, application developers 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 PeopleDB, and in many cases will return more complete and correct answers than if run on a traditional DBMS. Second, because of its declarative programming environment and operator-based approach, PeopleDB can perform cost-based optimization, to improve query cost, time, and accuracy. Finally, as we discuss in subsequent sections, user interface design is a key factor in enabling questions to be answered by people. Our approach supports the automatic generation of effective user interfaces for tasks that are submitted to the crowd.

1.5 Paper Overview

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 from the nature of the crowdsourcing platform. For example, in our experiments (reported in Section 7), we quickly learned that the pool of workers available to us was orders of magnitude smaller than we expected, despite the large numbers of workers available in the entire system.

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

- We present the design of PeopleDB. We describe new query operators that encapsulate input from workers and alterna-

tive query plans for incorporating crowd input and traditional query operators.

- We develop simple SQL schema and query extensions that enable the integration of crowd-sourced data and processing.
- We show how the above extensions facilitate the automatic generation of effective User Interfaces for soliciting data and processing from the crowd.
- We present the results of experiments that show that PeopleDB is indeed able to answer queries that traditional relational systems cannot answer and give insights into the performance of workers and the system as a whole.

The remainder of this paper is organized as follows: Section 2 describes the crowdsourcing platform, and compares it with the resources cloud computing provides. Section 3 gives an overview of PeopleDB, followed by the PeopleSQL extension to SQL in Section 4, the user interface generation in Section 5, and query processing in Section 6. In Section 7 we present results from performance studies and demonstrate sample queries. Section 8 reviews related work, and finally Section 9 concludes and discusses the remaining research challenges.

2. CROWDSOURCING

Crowdsourcing platforms provide access to work forces as a utility. Similar to cloud computing services, they provide API-based, pay-as-you-go resources for accomplishing tasks. In this section we present an overview of a crowdsourcing platform and its interfaces. We then identify key differences compared to traditional computing platforms and outline resulting design issues. In subsequent sections we describe how PeopleDB is built based on these interfaces, services and design considerations.

2.1 Amazon Mechanical Turk

A crowdsourcing platform creates a marketplace on which requesters offer tasks and workers accept and work on the tasks. There are a number of emerging platforms, such as clickworker.com, microtask.com. We chose to build PeopleDB using one of the leading platforms, Amazon's Mechanical Turk (MT)[2]. In this discussion, we focus on this particular platform.

MT focusses on so-called microtasks. Microtasks usually do not require any special education and typically take no longer than one minute to complete; although in extreme cases, tasks can require up to one hour to finish [14]. In MT, 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 MT currently, workers from the whole world can participate but requesters must be holders of a US credit card. Amazon does not publish statistics about the marketplace, but in a marketing survey published in 2006 [1], they claimed that MT featured over 200,000 workers (referred to as turkers), and by all estimates, the marketplace has grown dramatically since then [24].

2.1.1 Mechanical Turk Basics

MT 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 and Job**: 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 MT, but we use it in this paper where necessary.
- **Assignment**: Every HIT can be replicated into several assignments. MT guarantees that a worker processes at most a single assignment for each HIT enabling the requester to get answers from different workers for quality assurance. Typical values are 3 or 5, with odd numbers being used to enable majority voting. Requesters pay workers for each assignment completed satisfactorily.
- **HIT Group**: MT 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 a picture. As we discuss below, workers typically locate HITs to perform based on HIT groups.

The basic MT workflow is as follows: A requester divides his or her information needs into jobs, the smallest unit of work. Then the requester packages these jobs into HITs and posts these HITs. Furthermore, the requester determines the number of assignments for each HIT in order to get reliable results. Requesters can specify requirements that workers must meet in order to be able to accept the HIT. MT groups HITs into HIT groups and posts them so that they are searchable by workers. A worker accepts and processes assignments. Requesters 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 if the requester found the answers useful; the completed assignment is rejected otherwise. 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 (plus an optional bonus), as well as a commission to Amazon.

Workers access MT through their web browsers and deal with two kinds of user interfaces. One is the main MT interface, which enables 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 assignment of the HIT. Obviously, the better this requester-defined user interface, the more productive the worker will be and the better the results.

In MT, requesters and workers have visible IDs, and so relationships and reputations can be established. 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.

2.1.2 Mechanical Turk APIs

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

- *createHIT(title, description, question, keywords, reward, duration, maxAssignments, lifetime)* $\rightarrow$ *HitID*: Calling this method creates a new HIT on the MT 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 MT to create HIT groups. The *question* parameter encapsulates the user interface that workers use to process this HIT, including HTML pages. *duration* indicates how long the worker has to complete an assignment after accepting it. *lifetime* 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. PeopleDB, however, does not currently use this capability so the details of this and several other parameters are omitted for brevity.

- *getAssignmentsForHIT(HitID) → list(assId, workerId , answer):* This method returns the answers 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 *assID* which is used by the requester to approve or reject that assignment (described next).
- *approveAssignment(assID) / rejectAssignment(assID)*: Approval triggers the payment of the reward to the worker and the commission to Amazon.
- *forceExpireHIT(HitID):* Expires a HIT immediately. Assignments which have already been accepted may be completed.

2.2 Comparing Clouds and Crowds

MT provides Human Computation as a service, much the way that a public cloud provider such as Amazon EC2 provides computational resources as a service. MT and EC2 both offer a pay-as-you-go pricing model: Simple HITs on MT are done for as little as $0.01 and Ipeirotis reported that the effective hourly wage on MT is $4.80 on an average [14]. For comparison, the largest EC2 instance costs $2.00 per hour which is in the same ballpark.

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 PeopleDB. However, while the metaphor of humans as computing resources is powerful, it is important to understand where this metaphor works, and where it breaks down. In this section we highlight some key issues that have influenced the design of PeopleDB:

Absolute 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 PeopleDB these differences have implications for optimization strategy, 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 cloud processors, 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 paying 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 implies that the PeopleDB design must 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 [19] 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 MT site, and others having to do with the affinity and learning issues discussed previously. For PeopleDB, 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 MT crowdsourcing system, we now turn to the design and implementation of PeopleDB.

3. OVERVIEW OF PEOPLEDB

Figure 1 shows the general architecture of PeopleDB. An application issues requests using PeopleSQL, a moderate extension of standard SQL. So, application programmers continue to build their applications in the traditional way and the complexities of dealing with the crowd are encapsulated by PeopleDB. PeopleDB sources information from the crowd that it does not have electronically. Furthermore, PeopleDB asks the crowd to process operations that it cannot process otherwise (e.g., tagging pictures). PeopleDB caches all answers from the crowd for future reference.

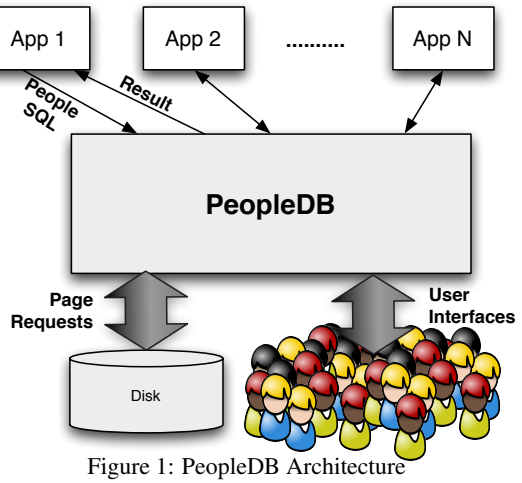


Figure 1: PeopleDB Architecture

In the previous section, we outlined design challenges for hybrid crowd/computer database systems. In the following, we describe how we address these challenges:

People Programming Language: The crowd requires its own programming language; i.e., user interfaces and instructions. PeopleDB addresses this issue by automatically generating user interfaces based on the available schema information. For this purpose, PeopleSQL extends SQL's data definition language. These extensions allow application developers to annotate tables and to generate powerful and expressive user interfaces from these annotations and the regular type definitions and constraints as specified in standard SQL. User interfaces are generated by PeopleDB at compile-time and the PeopleDB runtime involves posting HITs with these generated user interfaces. For complex and exceptional cases, PeopleDB provides tools that allow the application developer to customize the automatically generated user interface.

Crowd Input Management: Due to the Open World nature of our hybrid system and the issues of task ambiguity and answer quality, answers from the crowd can never be assumed to be complete or correct. PeopleDB addresses this via two mechanisms: user interface designs, and special query operators. The user interfaces generated for tasks 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). In query plans, *StopAfter* operators are used to restrict the amount of data obtained from the crowd and *GoodEnough* operators are used to control the quality of crowdsourced answers through voting-based and other mechanisms. These operators are discussed in more detail in Section 6.2.

Worker Community Management: As mentioned in Section 2.2, the requester/worker relationship evolves over time and care must be taken to cultivate and maintain an effective worker pool. PeopleDB assists the most important duties of a requester such as approving / rejecting assignments, paying on time, granting bonuses, and giving explanations if assignments are rejected. This way, PeopleDB helps to build communities for requesters which in the long run pays back in improved result quality, better response times, and lower cost.

Database Statistics and Optimization: The current PeopleDB prototype has only a simple heuristic optimizer. However, one advantage of our query operator-based approach is that it provides a mechanism for incorporating cost-based optimization into crowdsourced plans. Such optimization is part of our future work. However, initial measurements of response time, cost and quality (see Section 7) indicate that the development of accurate cost models for crowdsourced operators is not straight-forward.

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

4. PEOPLESQL

This section presents PeopleSQL, a SQL extension that supports crowdsourcing. As mentioned in the introduction, there are two cases that limit traditional systems: (a) incomplete data, and (b) subjective comparisons. In order to model incomplete data, PeopleSQL extends SQL's data definition language. In order to enable application programmers to ask queries with subjective comparisons, PeopleSQL extends SQL's query language with two new comparison operators. Both extensions are described in this section.

As mentioned in the previous sections, using SQL as a starting point has several advantages. First, SQL supports physical and logical data independence so that application programs need not be changed if, say, the API of the crowdsourcing platform changes or the application is ported to a different crowdsourcing platform. Second, existing relational optimization techniques can be leveraged (e.g., cost-based or adaptive query optimization techniques). Existing operators such as joins, aggregates and sorts need not be reimplemented. Finally, application programmers who know SQL need not be retrained.

4.1 PeopleSQL DDL

In PeopleDB, data can come from two sources: electronically (as in any other traditional DBMS) and from the crowd. To model crowd-sourced data, PeopleSQL provides a special keyword, *CROWD*. This keyword can be used to declare crowd-sourced columns, tables, and referential integrity constraints.

4.1.1 Crowd-sourced Columns

The following example illustrates that any column of a regular SQL table can be marked as crowd-sourced. Such columns are typically populated as a side-effect of processing queries that involve that column, but can also be modified using regular SQL DML operations. The semantics are as follows: If such a crowd-sourced attribute is accessed as part of a query and its value is a special defined *CNULL*, then PeopleDB will ask the crowd to provide a value. That value will then be memoized by PeopleDB for future use.

EXAMPLE 1 In the following "company" table, the *headquarter_address* is marked as crowd-sourced:

```
CREATE TABLE company (
  name STRING,
  headquarter_address CROWD STRING
);
```

As a side-effect of the following query, missing addresses are added for all companies registered in the database.

```
SELECT name, headquarter_address FROM company;
```

Note that populating data during query processing makes SQL `SELECT` operations a potentially side-effecting operation. Also note that crowd-sourced columns of regular tables do not create an open-world issue as described previously, as the data is bound by the number of records in the table. In contrast, as we describe in the next subsection, crowd-sourced tables do introduce this issue.

4.1.2 Crowd-sourced Tables

PeopleSQL allows application developers to source entire tables from the crowd. Such tables are virtually infinite because they are sourced as part of an open world. In order to limit the number of instances that are sourced from the crowd a query over a crowd-sourced table must either have a `LIMIT` clause that specifies the number of results or a `WHERE` clause that scopes the query to a bounded subset.

EXAMPLE 2 A crowd-sourced "department" table:

```
CREATE CROWD TABLE department (
  name STRING PRIMARY KEY,
  reception_phone_number STRING
);
```

EXAMPLE 3 As a side-effect of the following query, a new record could be added to the database if there isn't already a record for the Music department.

```
SELECT name, reception_phone_number
FROM department WHERE name = 'Music';
```

4.1.3 Crowd-sourced Referential Integrity

Referential integrity constraints can appear as part of crowd-sourced tables or columns. The next example, for instance, defines a crowd-sourced professor table with a foreign key to the (crowd-sourced) department table defined in the previous example.

EXAMPLE 4 A "professor" crowd table. Each "professor" is associated with one department (N:1 relationship).

```
CREATE CROWD TABLE professor (
  name STRING PRIMARY KEY,
  email STRING UNIQUE,
  department_name STRING
    REF department(name)
);
```

Columns that involve referential integrity constraints can also be crowd-sourced as a side-effect of query processing. For example, if a query asks for the *department_name* of all professors, then the *department_name* for each professor is sourced, if it has not been initialized or recorded previously. Crowd sourcing the *department_name* of a professor may involve creating a new department record in this example because *department* is also a crowd-sourced table. This case is discussed further in Section 5.2.

4.2 PeopleSQL Queries

The DDL extensions described enable PeopleSQL to obtain data from the crowd. As stated previously, crowd input is also needed for comparing data. To this end, PeopleSQL extends the SQL query language with two new functions: *CROWDEQUAL* and *CROWDORDER*. These two functions can be used in queries to ask the crowd to perform (subjective) comparisons.

4.2.1 CROWDEQUAL

The `CROWDEQUAL(lvalue, rvalue)` function takes two values as arguments and asks the crowd whether the two values are equal. Naturally, this function returns a Boolean value. The function is particularly useful when the data set is noisy and entity resolution is required, as demonstrated in the `I.B.M.` example of the introduction. As a syntactic sugar, `lvalue~rvalue` can be used and is equivalent to `CROWDEQUAL(lvalue, rvalue)`.

EXAMPLE 5 Continuing the example of the introduction, the following query asks for the address of the headquarters of "Big Blue":

```
SELECT headquarter_address FROM company
WHERE name ~ "Big Blue";
```

4.2.2 CROWDORDER

The `CROWDORDER(tuples, aspect)` function is used whenever the help of the crowd is needed to rank or order results.

EXAMPLE 6 The following PeopleSQL 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");
```

5. USER INTERFACE GENERATION

This section shows how we use PeopleSQL to automatically generate user interfaces (UIs) for the crowd at compile-time. In the following, we first discuss creating UIs for a single relation without any foreign keys to other tables. We then extend the techniques with foreign keys to so-called multi-relation interfaces. Finally, we explain how PeopleDB creates interfaces for crowd-sourced comparisons (i.e., the `CROWDEQUAL` and `CROWDORDER` functions).

5.1 Single Relation Interfaces

PeopleDB automatically creates user interfaces for all crowd-sourced tables or tables that contain crowd-sourced columns. To create such a user interface, PeopleDB compiles the PeopleSQL schema into HTML code. A similar approach is used by MS Access and Oracle Forms [22]. Basically, an HTML form is generated for every table and every attribute becomes a field in that form. Crowd-sourced attributes that are still unknown are translated into input fields; the values of all other attributes are given as reference data for the worker. The title of the HTML form is the name of the table and the instructions ask the workers to complete all missing information (i.e., enter values for the unknown crowd-sourced attributes).

Figure 2 (a) shows an example user interface for the schema presented in Example 2. In this example, the *Name* field is not editable because this attribute is not crowd-sourced.

In addition to HTML, JavaScript code is generated in order to check for the correct types of input provided by the workers. If the PeopleSQL involves a `CHECK` constraint that limits the domain of a crowd-sourced 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.

5.2 Multi-Relation Interfaces

Tables with foreign keys require additional attention to the user interface design. In the simplest case, the attribute containing the foreign key constraint refers to a non crowd-sourced table. in this case, the data inside the referenced table is subject to the closed-world assumption and assumed to be complete. The generated user interface then shows a drop-down box containing all the possible keys, or, if the domain is too large a JavaScript-based "suggest" (auto-completion) function is provided instead.

In the case where the foreign key constraint refers to a crowd-sourced table, one option is to treat it as a normal field and to resolve any integrity violations as a separate processing step. A more sophisticated option is to use a drop-down box or "suggest" function to enable previously obtained records to be shown and selected. Optionally, the form can also be denormalized to obtain the record and the foreign-key record at the same time using an additional *Add* button. Pressing *Add* shows the interface for the corresponding table, allowing the worker to fill in the data for a new record. The choice of whether or not to denormalize and thus, to enable the *Add* button can be made at run-time, based on the number of records previously obtained and cached for the crowdsourced table. Figure 2b shows the user interface generated to gather the E-Mail address and department information of a professor in this way (Example 4).

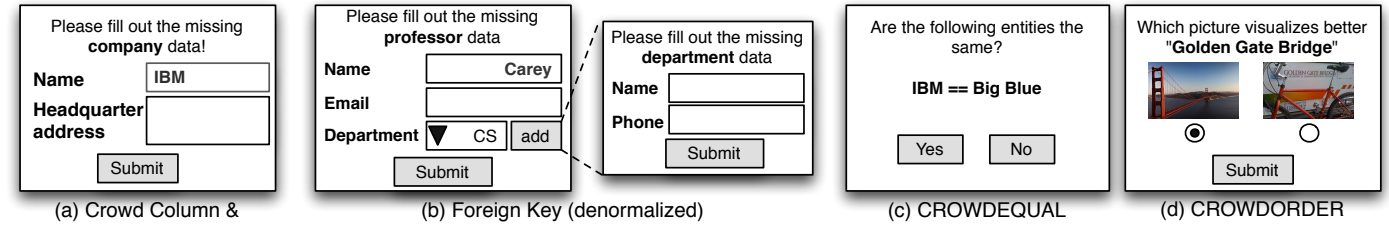
5.3 Crowd Comparison Functions

Figures 2c and 2d show example user interfaces generated for the `CROWDEQUAL` and `CROWDORDER` functions. These figures should be self-explanatory.

6. QUERY PROCESSING

Since PeopleDB is SQL-based and operator-based, query plan generation and execution follows a largely traditional approach. In this section, we focus on the specialized query processing techniques that enable crowdsourcing as specified in PeopleSQL.

As in a traditional DBMS, physical operators are iterators with an open(), next() and close() interface; i.e., each iterator processes its input an-item-at-a-time and only processes as much of its input as necessary. PeopleDB can use traditional query operators for nearly all data manipulation tasks, with two exceptions. First, special crowd-specific operators are needed for dealing with crowd-sourced data and comparisons. Second, as described in Section 4.1, the PeopleSQL DDL provides sufficient information to enable access to the crowd for missing data. Such on-demand access requires



(a) Crowd Column & Crowd Tables w/o Foreign Keys (b) Foreign Key (denormalized) (c) CROWDEQUAL (d) CROWDORDER

Figure 2: Example User Interfaces Generated by PeopleDB

a modification to query operators so that they call the crowdsourcing operators when they detect missing data.

6.1 Query Operators

In this section we present the basic crowdsourcing query operators and their use in query plans. An example compilation workflow for a query that uses these operators is shown in Figure 3. The query is a join of the two crowdsourced tables of Section 4.1.

We implement three crowd-specific iterators: MTProbe, MTJoin and MTFunction. All of these iterators take the standard HIT parameters (see Section 2.1.2) such as reward, maxAssignments, etc.

6.1.1 MTProbe

MTProbe is the basic operator for obtaining record and field values from the crowd. In addition to the standard HIT parameters, it takes a table name and a set of predicates and projections as input. Based on the table's name and the associated schema a HIT user interface (UI) is generated as described in Section 5 at compile-time. This compiled UI initially contains all of the fields of the specified table. If the table being probed contains foreign keys, the compiler must determine if the referenced table is a regular table or another crowdsourced table (as described in Section 5.2). In the former case, the possible foreign key values can be displayed as a drop-down menu or with an auto-completion function. In the latter case, there are three options. The first option is to simply remove (using the projection list) the foreign key from the generated UI. A second option is to treat the foreign key as a regular field, deferring referential integrity checking to a later point in the plan. The third approach is to use the "denormalized" scheme described in Section 5.2 to incorporate crowd look ups for data from any referenced tables as well, including, possibly following chains of foreign key/key relationships. The choice among these options is an optimization decision, which depends on issues such as the affinity of the worker pool to different types of tasks. An experiment regarding this choice is described in Section 7.2.3.

At runtime the projections are applied to the UI (i.e., unneeded display fields are removed) and predicate values are entered into their corresponding display fields in the form, which are then marked as non-changable. This delayed processing allows the MTProbe to be parameterized with values passed in at runtime, if required. Also, at runtime, multiple probe UIs can be combined into HITs and submitted to MT with the *createHIT* function described in Section 2.1.2. This batching means that it is possible to receive more than one record from a single HIT. Thus, the operator repeatedly issues *getAssignmentsForHIT* requests to MT and returns assignment results as soon as they are produced by a worker. Figure 3 (c) shows the use of MTProbe inside the example query. In this example, the foreign key is treated as a regular field in the MTProbe (i.e., the second option above).

6.1.2 MTJoin

MTJoin implements an indexed nested-loop join over two tables, at least one of which is crowdsourced. It takes a child iterator as an outer relation, a crowdsourced table-name as the inner relation, and one or more predicates as input. Based on the inner table-name it creates an interface similar to MTProbe. At run-time, for every outer tuple, the iterator creates one or more HITs for lookups on the inner. Based on the predicates and the field values of the outer tuple it presets and fixes additional fields in the form. Conceptually, MTJoin performs both an MTProbe and the join as the lookups on the inner table are collecting new records for that table as well as for the join. As with MTProbe, multiple jobs (i.e., look ups) are combined into a single HIT.

6.1.3 MTFunction

MTFunction implements the *CROWDEQUAL* and *CROWDORDER* comparison functions described in Section 4.2. It takes two records and a projection function as input. Based on the given projection, it determines the required attributes and creates a UI as described in Section 5.3. At run-time, MTFunction creates a HIT per given tuple pair and collects the data from MT. Note that MTFunction is not an iterator itself. Rather, it is a function, which can be used in iterators like sorting or selection iterators.

6.2 StopAfter and GoodEnough Operators

Dealing with the open-world assumption requires limiting the data acquired from the crowd. For example, a query asking for all pictures of the Golden Gate Bridge would require an infinite amount of time and budget. However, users are typically only interested in a limited number of pictures instead of all pictures, thus allowing the number of returned tuples to be restricted.

PeopleDB addresses the open-world assumption with the StopAfter operator [7]. This iterator stops pulling from the child iterator as soon as the stop *limit* has been reached. The limit can depend on the number of records, time, or budget used. Furthermore, when the limit is reached, the operator expires all outstanding HITs with the *forceExpireHIT* call. Analysis techniques such as those presented in [4] can be used to determine where in the plan StopAfter operators are needed.

PeopleDB's GoodEnough operator controls the answer quality. The operator only forwards those tuples which satisfy a quality threshold. Many different techniques can be applied to determine the quality of the result [21]. In the current implementation, we use majority voting. Thus, whenever the majority of the assignments for a job agree on an answer, the tuple with the majority vote is forwarded. If no majority can be reached, the tuple is discarded.

The optimizer can place both operators, StopAfter and GoodEnough, between any existing operators to control the flow.

6.3 Physical Plan Generation

We implemented a simple rule-based optimizer for PeopleDB. The optmizer implements several essential query rewrite rules such as predicate push-down, stopafter push-down [7], join-ordering and determining if the plan is bounded [4]. The last optimization is required to ensure that the amount of data from the crowd is bounded. A cost-based optimizer for PeopleDB, which must also consider the changing conditions on MT, remains future work.

Figure 3 shows the optimization process with the crowdsourced

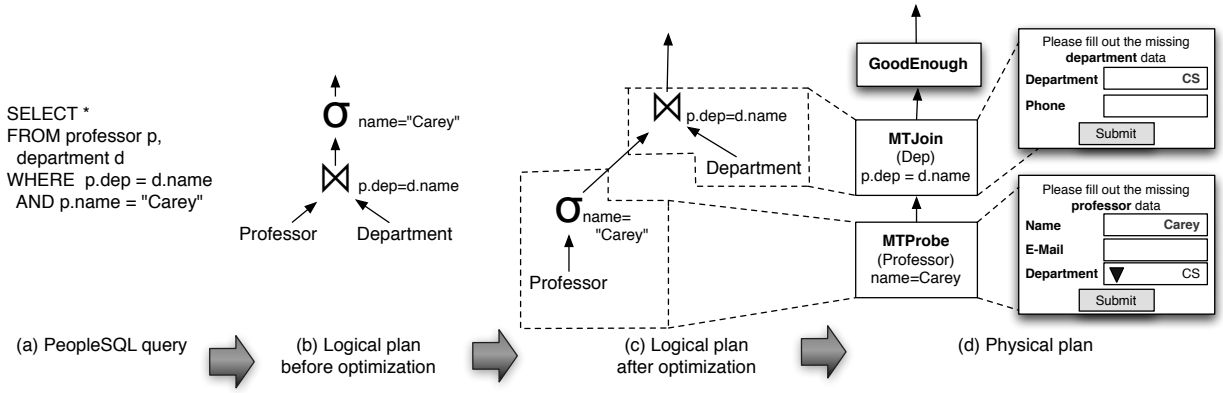


Figure 3: PeopleSQL Query Plan Generation

tables of Section 4.1. The query is first parsed and represented as a logical query plan. Figure 3 (a) to Figure 3 (b) visualize this process. The expression tree is then optimized using a set of simple heuristics. An example optimized logical plan is shown in Figure 3 (c).

The compiler then transforms the logical plan into a physical plan. In this process, first every join whose right-hand child is crowdsourced is mapped to an MTJoin, with the appropriate join predicates. All remaining accesses to crowd tables or crowd columns become MTProbe operators with appropriate predicates (i.e., selections) to allow a more precise questioning of the crowd.

Finally, in the last stage of the plan generation, the number of HITs, assignments, reward etc. are set for each crowd operator and the GoodEnough operator is introduced where necessary. Setting the parameters for the crowd operators as well as for the GoodEnough operators is also currently based on simple heuristics (e.g., the number of attributes the crowd is asked for). Exploring more sophisticated methods, for example how to split a budget or time-constraints, remains future work. An example of the final transformation from a logical to a physical query plan is given in Figure 3 (c) to (d).

7. EXPERIMENTS AND RESULTS

This section presents the results of experiments with PeopleDB and MT. Overall, we ran more than 25,000 HITs on MT in October 2010 thereby varying different parameters (e.g., price, jobs per HIT, etc.) and evaluated the response time and quality of the answers given by the workers. We conducted two kinds of experiments: Section 7.1 reports on results obtained with *micro benchmarks* used in order to study the behavior of workers. Section 7.2 reports on results of experiments with complex queries. Section 7.3 summarizes observations made during the experiments.

Obviously, the results presented in this section are highly dependent on the current state of MT and its community (October 2010) and the specific tasks and benchmark queries used. MT and its community are likely to evolve in the near future. Nevertheless, a number of important observations could be made.

7.1 Micro Benchmarks

The purpose of the first set of experiments was to study the behavior of workers on MT with regard to response time, quality of results, sensitivity to price and visibility of HITs on MT. All these experiments were carried out with a simple schema and a simple query. The schema involved a (regular) *businesses* table with two crowd-sourced columns:

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

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

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

Workers are not evenly distributed between all timezones so that the time of day can impact the availability of workers [24]. [13] concludes that the best timeframe to do experiments on MT is between 0600 and 1500 GMT. As a consequence, all experiments reported in this paper were carried out in that timeframe. We repeated all experiments four times and report on the average values in this section. If not stated otherwise, groups of 100 HITs were posted and each HIT involved five assignments. Furthermore, the reward was 1 cent by default and each HIT asked for the address and phone number of one business (i.e., 1 Job per HIT).

7.1.1 Experiment 1: Responsiveness, Vary HIT Groups

The first set of experiments studied the response time of assignments as a function of the number of HITs that are published at the same time. As mentioned in Section 2, MT automatically groups HITs of the same kind into a HIT group. Furthermore, HIT groups with many HITs receive more visibility because they are ranked higher when workers search for HITs. In this experiment, all HITs were of the same kind because they all asked for the *phone_number* and *address* of a particular business in the USA. While processing the query, we were able to control the number of HITs posted at each point in time and, therefore, we were able to control the size of the HIT groups published on MT. Each HIT contained only one assignment and the reward was 1 cent per HIT.

Figures 4 and 5 show the results of this experiment. Figure 4 shows that the workers responded faster and got the first HITs done quicker with a growing number of HITs per group. For instance, if a HIT group had 400 HITs, then the first HIT was completed almost instantaneously; the first 25 HITs were completed within a few minutes. Again, these results are expected because HIT groups with many HITs receive more visibility.

Figure 5 shows the fraction and absolute number of HITs that were completed within the first 30 minutes with a varying number of HITs per group. Again, bigger HIT groups got more visibility and, therefore, got more completed HITs in absolute terms; for instance, with groups of 400 HITs, on an average 223.8 HITs were completed within 30 minutes. In relative terms, however, the best results can be achieved with about 100 HITs per group. With such a setting, almost all HITs were processed within 30 minutes whereas many HITs remained stranded if the group is too big. Such

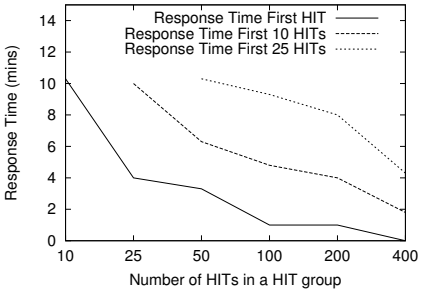


Figure 4: Resp. Time (min): Vary Hit Group

1 Asgn/HIT, 1 cent Reward

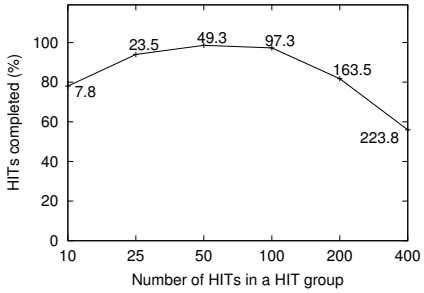


Figure 5: Completion (%): Vary Hit Group

1 Asgn/HIT, 1 cent Reward

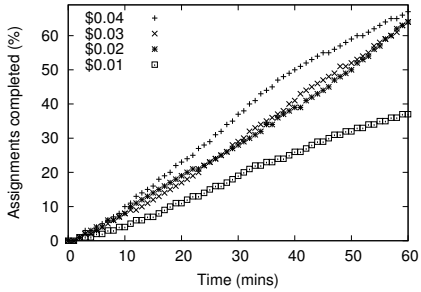


Figure 6: Completeness (%): Vary Reward

100 HITs/Group, 5 Asgn/HIT

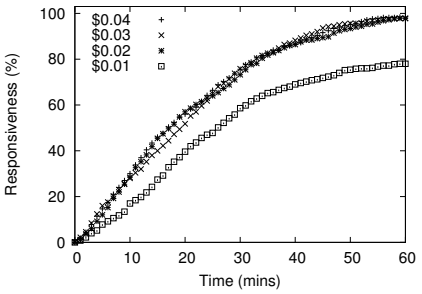


Figure 7: Responses (%): Vary Reward

100 HITs/Group, 5 Asgn/HIT

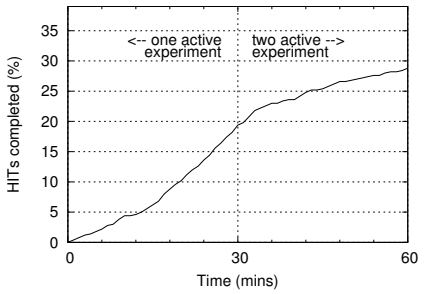


Figure 8: Completeness (%): W/O Interference

100 HITs/Group, 5 Asgn/HIT, 1 cent

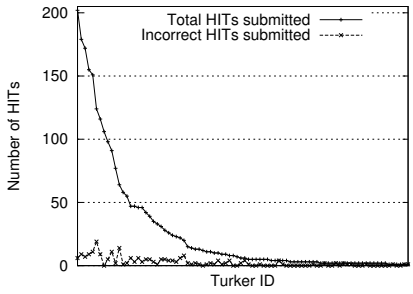


Figure 9: HITs/Result Quality by Worker

Any HITs/Group, 5 Asgn/HIT, Any Reward

results indicate that systems such as PeopleDB are needed to optimize the use of MT automatically without requiring the application programmer to carry that burden.

7.1.2 Experiment 2: Responsiveness, Vary Reward

The second set of experiments studied the response time as a function of the reward. In these experiments, 100 HITs were posted per 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 that were completed (i.e., all five assignments) as a function of time. The figure shows the graphs for HITs with a reward of 1 cent, 2 cents, 3 cents, and 4 cents. Obviously, all graphs are monotonic: the longer we wait, the more HITs are completed. The graph for a reward of 4 cents is on top indicating that with a reward of 4 cents more HITs are completed in the same amount of time than with a lower reward. The graph for 1 cent is at the bottom, and the graphs for 2 and 3 cents are in the middle. These results are as expected and confirm that larger rewards result in faster responses from the workers. Somewhat surprisingly, there is little difference between the graphs for 2 and 3 cents. So, there does not seem to be much incentive for requesters to give rewards of 3 cents (as opposed to 2 cents) for this particular kind of HIT.

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 were fully completed even for the highest reward of 4 cents (Figure 6). Second, if a requester is interested in getting at least one response for each HIT, there is little incentive to pay more than 2 cents. The graphs for 2, 3, and 4 cents are almost identical in Figure 7.

7.1.3 Experiment 3: Responsiveness, Interference

The third experiment shows how concurrent postings of similar HIT groups affect the completion rate of HITs. This experiment gives insight into the size of the MT community that is available to process a certain kind of HIT. If the community were large (i.e., abundant resources), the completion rate of HITs in one HIT group should not be affected by the presence of other concurrent HIT groups.

This experiment was carried out in the following way. First, we posted a HIT group with 100 HITs. After 30 minutes, we posted another HIT group with 100 HITs; this second HIT group contained the same kind of HITs as the first group (i.e., finding phone numbers and addresses of businesses) with the same reward i.e. 1 cent. Figure 8 plots the completion of HITs of the *first* HIT group in a period of 60 minutes. Figure 8 shows clearly that the completion rate was higher in the first 30 minutes (without a concurrent HIT group) than in the second 30 minutes (with a concurrent HIT group). The conclusion is that the worker community on MT is limited and that concurrent HIT groups do interfere and compete for resources in the MT community.

As a variant of this experiment (not shown in Figure 8), we gave a higher reward to the HITs of the first HIT group (i.e., 2 cents) and left the reward of the HITs of the second HIT group at 1 cent. In this case, the completion rate of the first HIT group was *not* affected by the presence of the *second* HIT group. So, as expected, the reward matters if HIT groups compete for resources.

7.1.4 Experiment 4: Result Quality

The fourth set of experiments studied the quality of the results

returned by the workers on MT and tried to answer the question whether workers who take more HITs are likely to make more or less errors. 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. A worker's answer that deviated from this ground truth was counted as an error. We normalized phone numbers in order to avoid errors due to layout differences (e.g., use of hyphens and brackets).

Figure 9 shows the results. For each worker, Figure 9 shows the number of HITs computed by that worker and the number of errors made by that worker. In Figure 9, the workers are plotted against the x-axis and the workers are sorted by the number of HITs they processed. As can be seen, the distribution of workers taking HITs is highly skewed; similar to a Zipf distribution. This result confirms previous studies that indicate that requesters build communities of workers that specialize on processing requests of this requester [14]. It seems that we were able to build a community of about a dozen fans.

Turning to the *Incorrect HITs* curve of Figure 9, no clear trend can be seen. In absolute numbers, heavy hitters, of course, make more errors. Looking at the error rate (Incorrect HITs / Total HITs, not shown), however, heavy hitters were as likely to make errors as workers that just took a few HITs. So, we could not see any training effects. The good news is that, we could not observe any spammers; i.e., workers who notoriously gave wrong answers. It seems that such spammers are attracted by high rewards and the rewards of a few cents per HIT that we gave were not enough to attract them.

We also investigated the dependency of errors on the reward given. We do not show these results because we could not draw any conclusions. It seems that at least in the price range that we chose (1 to 4 cents), the reward has no impact on the error rate of the workers. The results shown in Figure 9 includes HITs from the full reward range of 1 to 4 cents that we studied. Furthermore, error rates did not depend on the group size of HITs.

7.2 Impossible Queries

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

7.2.1 Entity Resolution on Company Names

For this experiment, we used the *Company* schema of Example 1 of Section 4.1 and populated it with the Fortune 100 companies, as derived from Wikipedia. Furthermore, we ran the following query in order to test the ability to carry out entity resolution:

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

We ran four different instances of this query with the four parameter settings of "the non-uniform" name shown in Table 1. 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. Table 1 shows the results in terms of result quality. It can be seen that in all four cases, the correct answer was returned. However, only in a single case (for BMW), the vote was unanimous. For reference, Table 1 also shows some of the wrong answers that workers produced.

In terms of response time, the 40 HITs (i.e., 120 assignments) for all four queries were completed in 39 minutes.

7.2.2 Ordering Pictures

In this experiment, we asked the query of Example 6 of Section 4; i.e., we asked for a ranking of pictures in different subject areas. Overall, we had 30 subject areas and asked to rank a set of eight pictures for each subject area. All pictures were obtained from Flickr under the Creative Common License. Each HIT involved the comparison of four pictures. Overall, all ranking could be done with 210 HITs, each with three assignments. It took 68 minutes to complete the whole 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, and the ranking of that picture according to the majority vote and sorting carried out by PeopleDB vs. the ranking according to a group of six experts who frequently visit the Golden Gate Bridge. Again, our conclusion is that the PeopleDB and the crowd solved the problem of ranking pictures.

7.2.3 Joining Professors and Departments

The last experiment presented in this paper compares the performance of two alternative plans for a join query. We populated the schema of Example 4 with 25 professors and 8 departments and asked the following SQL query (professors and their department information):

```
SELECT p.name, p.email, d.name,
       d.reception_phone_number
FROM   professor p, department d
WHERE  p.department_name = d.name AND
       p.name = "Professor A"
```

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 2b. For both plans, each HIT involved one job (e.g., an entry form for a single professor) and three assignments. The reward per HIT was 1 cent.

The execution of both plans differed in execution time, cost, and quality of results. In terms of execution time, the first plan could be executed in 206 minutes whereas the second plan was completed in 173 minutes. In terms of cost, the first plan cost $0.99, whereas the second plan 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 in the first plan. Again, these results confirm that a system like PeopleDB is needed in order to make the right optimization decision for varying response time, cost, and quality requirements; application developers should be relieved from the burden of finding the right plan for their purposes.

7.3 Observations

In total, our experiments involved 25817 assignments processed by 718 different workers. In addition to measuring the response time, cost, and quality of the results, we monitored several MT related websites. Overall, our experiments confirm the basic hypothesis of our work: It is possible to extend a relational database system and integrate human input in order to process queries that traditional systems cannot process. Furthermore, such an extended relational database system should make automatic decisions with regard to grouping HITs, giving rewards, generating user interfaces, and cleaning results by adjusting the number of assignments

Non Uniform Name	Query Result	Votes	Error Examples
Bayerische Motoren Werke	BMW	3	TATA Group, Gazprom, Boeing, Toyota
International Business Machines	IBM	2	Samsung, HP
Company of Gillette	P&G	2	Aviva, AIG, France Telecom
Big Blue	IBM	2	Microsoft

Table 1: Entity Resolution on Company Names

(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 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}.

per HIT. It is not productive to ask application developers to make these decisions and to hard-code such decisions into application code because the situation changes all the time. However, it is not as simple as the following high-level observations and anecdotes demonstrate.

First, we observe that the system has a memory that cannot be controlled like the memory of a typical server. Consequently, any crowd-enabled database systems like PeopleDB should keep state and monitor past events and actively manage relationships with workers. Requesters can track workers' past performance, but workers can also track the past behavior of requesters. To keep a community of workers happy, it might be better to be less strict in rejecting HITs and give out bonuses. For example, when we were strict in rejecting HITs, we received the following complaint from a worker on TurkOpticon [28]: "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 ..." Since then, we changed our reward strategy to be more lenient and be able to complete all experiments for this paper. Another example is that at one point a bug in PeopleDB triggered false alarms in security warnings for browsers on the workers' computers, and within hours our name appeared on Turker Nation [27].

Second, user interface design and precise instructions matter. For our first experiments with MT, we posted HITs with handcrafted user interfaces. In one HIT, we gave a list of companies and asked the workers to check the ones that equal "IBM". When no companies provided were IBM, we expected the workers to not check anything. To our surprise, the quality of the answers was very low. As soon as we added a checkbox that wrote "None of the above", the quality was high. Systems like PeopleDB help to avoid such obvious mistakes.

8. RELATED WORK

The design of PeopleDB is based on related work in two research communities: the database community and the emerging crowd-sourcing community.

Disregarding the crowd-specific extensions, PeopleDB is a regular relational database system. Therefore, PeopleDB leverages the standard, textbook architecture of building relational databases [23]. Furthermore, traditional query processing techniques can be applied in order to implement the crowd-specific extensions of PeopleDB. As shown in Section 6, all crowd-specific operators are implemented as iterators and integrated into the runtime system and compiler in the same way as traditional query operators.

There are a number of specialized database techniques that have impacted the design of PeopleDB. The "GoodEnough" operator was inspired by the "StopAfter" operator used to process Top N queries [7]. Furthermore, the "GoodEnough" operator incorporates data cleaning functionality as studied as part of probabilistic databases [26]. Since it is difficult to predict the behavior of the crowd at any given point in time, adaptive and dynamic query optimization technques such as those proposed in [5, 17, 29] appear to be attractive; we plan to study those as part of future work. Crowd-sourced comparisons such *CROWDEQUAL*, can be seen as special instances of expensive predicates whose optimization has been studied, among others, in [12]. Finally, there are some analogies between PeopleDB and federated databases such as those studied in the mid Nineties (e.g., [8, 11]): The crowd can be seen as a data source that is integrated by PeopleDB and the generated user interfaces can be regarded as wrappers for that data source.

Recently, there have been a number of studies that analyze the behavior of microtask platforms such as Mechanical Turk and systems that exploit such platforms. Ipeirotis, for instance, analyzed specifically the Mechanical Turk marketplace, thereby gathering statistics about HITs, requesters, rewards given, HIT completion rates, etc. [14]. Furthermore, the demographics of workers (e.g., age, gender, education, etc.) on Mechanical Turk have been studied [15, 24]. [13] reports on a model to predict the quality of image labeling tasks.

Systems that make use of crowd-sourcing platforms include CrowdSearch [31] and Soylent [6]. Hartmann has recently gained a great deal of attention by leveraging the crowd to write a paper about

crowd-sourcing [6]. The ESP game is the classic system that makes use of the crowd [30]. TurKit is a set of tools that allows to program iterative algorithms that are processed by the crowd [20]. This tool kit has been used to decipher unreadable hand-writing. Finally, Usher is a research project that studies the design of data entry forms for crowd-sourcing tasks [9, 10]. The goal of Usher is to provide workers with adaptive feedback in order to improve the accuracy of data entry.

Lately, there has also been work on building systems that combine the capabilities of humans and machines in order to solve problems that neither computers nor machines can solve alone. PeopleDB falls into this category of systems. To the best of our knowledge, PeopleDB is the only system in this category that is based on SQL and relational database technology. Dustdar et al. describe in a recent vision paper how BPEL and web services technology could possibly be used for this purpose [25].

An important building block of PeopleDB is the (semi-) automatic generation of user interfaces from meta-data (i.e., SQL schemas). As mentioned in Section 5, products like Oracle Forms and MS Access have taken a similar approach. In general, model-driven architecture frameworks (MDA) have been designed for exactly this purpose [18].

9. CONCLUSION

This paper presented the design of PeopleDB. PeopleDB is a classic relational database system that was extended to answer impossible queries with the help of microtask platforms. We identified two cases for which such extensions are needed: (a) unknown or incomplete data, and (b) subjective comparisons. PeopleDB extends SQL in order to meet both of these cases and it extends the query compiler and runtime system in order generate user interfaces and provide operators that crowdsource information using these interfaces. A number of experiments confirmed that PeopleDB can indeed answer impossible queries. Furthermore, experiments with micro-benchmarks provided insight into some of the performance, cost, and quality trade-offs an optimizer for a system like PeopleDB must consider in order to make good decisions when posting tasks. Specifically, such an optimizer must make decisions with regard to the grouping of tasks, the reward given to workers for completing a task, and the number of assignments allocated to each task for quality assurance. The right setting for all these parameters depends on the current status of the marketplace and involves careful monitoring and the maintenance of statistics. Another issue addressed by PeopleDB is the management of the relationship with workers. It is important to build a community of workers and to provide the workers of that community with timely and appropriate rewards and, if not, to give precise and understandable feedback.

The current version of PeopleDB is merely a proof-of-concept implementation. The first results of PeopleDB reported in this paper are highly encouraging. As future work, we plan to address a number of optimization issues which have not been addressed so far. For instance, we are planning a more sophisticated adaptive query optimizer that indeed adapts its plan as it goes along, depending on the state of the marketplace. Furthermore, we would like to explore *prefetching* strategies; i.e., asking for more information at marginal additional cost as part of processing one query in order to have significant cost and response time savings for future queries which are likely to involve the extra information. Of course, we planning to continuously improve user interface generation rules and PeopleDB's worker relationship management system. Another interesting avenue for future work is to explore use cases in which information may be available electronically, but it might nevertheless be beneficial to source it from the crowd using PeopleDB.

10. REFERENCES

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

[2] Amazon Mechanical Turk. http://www.mturk.com.

[3] S. Amer-Yahia et al. Crowds, clouds, and algorithms: exploring the human side of "big data" applications. In *SIGMOD*, 2010.

[4] M. Armbrust et al. PIQL: a performance insightful query language. In *SIGMOD*, 2010.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*, 2000.

[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. In *SIGMOD*, 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] K. Chen, J. M. Hellerstein, and T. S. Parikh. Designing adaptive feedback for improving data entry accuracy. In *UIST '10*, 2010.

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

[12] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD Conference*, 1993.

[13] E. Huang et al. Toward automatic task design: a progress report. In *HCOMP '10*, 2010.

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

[15] P. G. Ipeirotis. Demographics of Mechanical Turk. 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] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD*, 1998.

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

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

[20] G. Little et al. TurKit: tools for iterative tasks on mechanical Turk. In *HCOMP '09*, 2009.

[21] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[22] Oracle Forms. http://www.oracle.com/technetwork/developer-tools/forms/.

[23] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1998.

[24] J. Ross et al. Who are the crowdworkers?: shifting demographics in mechanical turk. In *CHI EA '10*, 2010.

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

[26] D. Suciu. Techniques for managing probabilistic data. 2007.

[27] Turker Nation. http://www.turkernation.com/.

[28] Turkopticon. http://turkopticon.differenceengines.com/.

[29] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *SIGMOD Conference*, 1998.

[30] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *CHI '04*, 2004.

[31] T. Yan, V. Kumar, and D. Ganesan. CrowdSearch: exploiting crowds for accurate real-time image search on mobile phones. In *MobiSys '10*, 2010.