Declarative Platform for Data Sourcing Games

Daniel Deutch
Ben-Gurion University*
deutchd@cs.bgu.ac.il

Ohad Greenshpan
Tel-Aviv University*
ohad.greenspan@gmail.com

Boris Kostenko*
Tel-Aviv Univeristy
boris.kostenko@gmail.com

Tova Milo*
Tel-Aviv University
milo@cs.tau.ac.il

ABSTRACT

Harnessing a *crowd* of users for the collection of mass data (data sourcing) has recently become a wide-spread practice. One effective technique is based on *games* as a tool that attracts the crowd to contribute useful facts. We focus here on the data management layer of such games, and observe that the development of this layer involves challenges such as dealing with *probabilistic* data, combined with *recursive* manipulation of this data. These challenges are difficult to address using current declarative data management frameworks, and we thus propose here a novel such framework, and demonstrate its usefulness in expressing different aspects in the data management of Trivia-like games. We have implemented a system prototype with our novel data management framework at its core, and we highlight key issues in the system design, as well as our experimentations that indicate the usefulness and scalability of the approach.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

Keywords

Databases, Crowdsourcing, Games, Probabilistic

1. INTRODUCTION

Harnessing a *crowd* of users for the collection of mass data (referred to as data sourcing) has recently become a wide-spread technique [7, 25]. Specifically, the work of [32, 21] suggested the use of *games* as a tool that attracts the crowd to contribute facts. In the internet era, such techniques have the potential of generating large databases that are otherwise very difficult to construct.

However, the design of games that fulfill this potential is not trivial and involves significant challenges. Consider a Trivia-like game that is run e.g. to obtain a simple database of capital cities, contributed independently by various users over the Web. The players are presented with questions on capital cities, and their answers are added to the database. Players gain scores by contributing correct facts to the database. This database is in turn used to answer queries posed by (a possibly different set of) users. The problem is that some of the facts may be wrong, and some may be contradicting, for example two different users claiming different cities to be the capital of England.

Even in this simple settings, several dilemmas arise in designing the data layer of the game: for instance, how to choose the questions that the game poses to players to maximize the expected knowledge gain? Which players to pose these questions to? How to decide whether a player is correct, to update her score? How to settle contradictions in the collected data when answering queries on it?

For each of these questions, hard-coded solutions can be employed. But no single solution is guaranteed to always achieve superior results, and the quality of results also depends on the type of data set in hand, and as always, hard-coded solutions are inflexible, difficult to adapt and deploy. Therefore it is desirable to use a *declarative framework for the data layer of crowdsourcing games which allows for rapid adjustments, modification and optimization.* The development of such framework is the goal of the present paper. We stress here that we focus on the *data layer* of games. The full design of games involves many additional important issues such as Human-computer Interaction, communication layer and others, that are outside of the scope of this paper.

There are some conceptual difficulties in the design of such framework. One difficulty lies in the *uncertainty* on which data items are correct; this uncertainty is due to the lack of an authoritative opinion, and it is common to capture it with *probabilities*. Declarative frameworks that deal with probabilistic data have been presented in [17, 1, 12, 18]. However the design of these frameworks does not address an additional difficulty that arises in our context, due to *recursive* dependencies between uncertain information: to identify the questions that should be posed to users, we must first know which data pieces are correct, which require validation, and which are completely missing. On the other hand, in order to know how the data should be cleaned, we need to know which users can be trusted, this depending on their contribution (correct and incorrect) to the aggregated dataset.

For instance, one possible solution is based on a set of probabilistic, PageRank-style rules. A first such rule may randomly decide in which fact to believe, using a distribution that is based on the current credibility of users that contributed the facts. The credibility of users can then be re-computed, via a second rule, according to which users that supported facts that were decided to be correct, may

*This work has been partially funded by the Israel Ministry of Science and by the European Research Council under the European CommunityâĂŹs Seventh Framework Programme/ERC grant MoDaS.

WWW 2012, April 16–20, 2012, Lyon, France.
ACM 978-1-4503-1229-5/12/04.

now be considered more credible. We may again choose in which facts to believe, based on these new calculated credibility scores, etc. The results of these recursive process can be used for the different tasks listed above. The believed answers can be used to answer queries; the computed credibility of users can be used to identify preferred users to get information from; and the questions that they will be asked can correspond to facts with high level of uncertainty (close to 50%).

However, this is only one possible solution, and there are many plausible others. In particular, there is a rich literature on *data cleaning* [29, 3, 15, 5]. For example, a simple approach decides between two contradicting facts according to their support [29]; another approach suggests the application of "transformation" rules [3] to fix parts of the data. [5] presents a technique to solve key violations using probabilistic choice over possible Database repairs. A recent paper [15] suggests to gradually clean data based on "corroboration", i.e. the trust in the users providing the data. This is in fact a non-probabilistic (yet recursive) variant of the PageRank-style policy described above. Similarly, the questions chosen may focus on facts with low entropy [28], and the users to which these question are posed may be those that gained high credibility in related facts (rather than overall high credibility).

So, we observed that *recursion and probabilistic data lie at the core of the developed techniques*. However, current declarative frameworks either support only probabilistic rules (e.g. [18, 5]), or only recursion (e.g. datalog-based frameworks such as [20]), but *not both*. Consequently, the development of a novel framework is required. We next briefly explain the principles underlying our framework.

Our framework suggests an interface that is based on SQL, but is augmented by a particular operator that allows to introduce probabilities, and supports recursive rules invocation. This syntax allows for a very easy implementation of the various techniques described above. The underlying model is that of *Markov Chain Monte Carlo* (MCMC) [27]. The idea is that we are given probabilistic rules and a query on the data. The former defines probabilistic transitions between possible database instances, serving as states of the Markov Chain. The query possible results are *sampled* (hence the Monte Carlo algorithm) in each database instance that is defined by the rules to be "clean" (i.e. non-contradictory). The output is a set of tuples that appeared in the query results, each accompanied with a probability that reflects the fraction of its appearance in the observed samples. We explain the details of this framework, and exemplify its usefulness in capturing different aspects of the game's data layer, in Sections 2 and 3.

We have implemented the framework and used it as the data layer for a data sourcing game called Trivia Masster. (A first prototype of the system was demonstrated in [10]). Several practical issues rise in the implementation, pertaining to optimizing the execution time, deciding convergence etc. We explain these practical challenges and our solutions in Section 4.

Finally, we provide in Section 5 an experimental study of techniques implemented using the framework. The goal of the study is not to study the performance of a particular technique, but rather to (1) show that common techniques are feasible to execute using our framework and (2) show how the generic declarative framework allows to easily compare different techniques. To that end we compare both the quality and runtime of various techniques.

The rest of this paper is organized as follows. In Section 2 we describe the foundations of our framework, including our query language. In Section 3 we show how common techniques can be expressed by the language. In Section 4 we provide more details on our implementation and solutions to practical challenges. Section 5 details our experimental study. In Section 6 we provide an overview of related work, and we conclude in Section 7.

2. FRAMEWORK FOUNDATIONS

In this section we provide details on our declarative framework for data management for data sourcing games. The framework is based on the theoretical advancements in [11], towards the evaluation of recursive queries with probabilistic choices. We are introducing here a significant extension to the language, in three respects: first, the language presented in [11] allowed rules to be defined using relational algebra (enriched with a special *repair-key* [19] operator used to introduce probabilistic choices), and needed to be extended to SQL (with repair-key) to allow for an easier design of rules by the system administrators. Second, the queries accounted for in [11] were limited to boolean ones. In contrast, the queries here may be arbitrary SQL queries, including select-project, aggregation and joins. Last, as we will observe, some techniques require explicit constructs that allow to use the probability obtained for the result of a sub-query.

To formally define and exemplify the (extended) language, we start by recalling the repair-key operator; then we show how to use repair-key to enrich SQL operators and obtain a language for describing probabilistic rules, and finally we explain the semantics of query evaluation.

2.1 Repair-key

We start by exemplifying the use of repair-key [19] in our context, then turn to the formal definition.

EXAMPLE 2.1. *Consider the relation Capitals in Table 1. It includes facts that describe capital cities of various countries, and a numeric value reflecting the authority of the user that submitted this fact (we will explain in the sequel how authority values are computed). The primary key here is the attribute Country; but note that the table contains primary key violations. The* repair-key *construct [19] allows to solve such contradictions by choosing one repair to each key value (i.e. one capital for each country), in a probabilistic way, with the probabilities dictated by the user authorities. The following expression has this effect:*

$$\textit{repair-key}_{Country@Authority}(Capitals)$$

The output of evaluating this expression is a probabilistic choice out of 4 possible databases, each consisting of a single tuple for China and a single tuple for the Netherlands; the probability of a tuple to be chosen is defined to be the sum of authorities of the users that supported it, relative to all users that contributed tuples with the same key value (e.g. $\frac{2}{2+4}$ *for (China,Beijing)). The probability of a given database is the multiplication of probabilities of all tuples appearing in it.*

More formally, let $\vec{A}, P$ be column names from the schema of a relation R, where $\vec{A}$ is a vector of columns and P is a single column, containing only numeric values which are

Country	Capital	Authority
China	Beijing	2
Netherlands	Amsterdam	2
Netherlands	Amsterdam	5
Netherlands	Hague	3
China	Shanghai	4

Table 1: Capitals

all greater than zero. Intuitively, the P value of a tuple is the level of trust in the tuple correctness. For each distinct value $\vec{a}$ of $\vec{A}$ appearing in tuples of R, denote the set of tuples in which $\vec{a}$ is the key value by $T_{\vec{a}}$. For each such $\vec{a}$, we sample exactly one tuple $\vec{t}$ from $T_{\vec{a}}$ with the probability distribution given by (normalized) column P That is, the probability of $\vec{t}$ to be chosen is $\frac{\vec{t}.P}{\sum_{\vec{t'} \in T_{\vec{a}}} \vec{t'}.P}$. If there are tuples $(\vec{b}, p_1), \ldots, (\vec{b}, p_n)$ in which all attributes apart for the P column share the same value, we first replace them by a single tuple $(\vec{b}, \sum_i p_i)$, then apply the probabilistic choice. For instance, in the above example, the two tuples having Amsterdam as the capital city of the Netherland (with authorities values of 2 and 5) are replaced by a single tuple with authority value 7. The operation repair-key$_{\vec{A}@P}(R)$ [19] then samples one maximal *repair* of the key $\vec{A}$. That is, the application of a repair-key construct generates a set of possible worlds (samples) with a single tuple for each key value; the probability of a possible world is the product of probabilities of chosen tuples within their groups $T_{\vec{a}}$, that is, the groups are assumed independent.

2.2 SQL enriched with Repair-key

We next consider the incorporation of the repair-key operator in a full-fledged query language. We start by recalling the semantics of relational algebra enriched by the repair-key construct [19], then explain the syntax and semantics of SQL enriched by repair-key.

Relational Algebra With Repair-key. In [19] the author introduced the repair-key construct into conventional relational algebra. The reader is referred to [19] for exact definitions. Intuitively, the semantics is as follows. We evaluate the relational algebra expression in the standard manner except that whenever the evaluation reaches an application of the repair-key construct, its application generates a set of possible worlds, each with an accompanying probability. The further relational algebra operations in the expression are applied in each possible world independently (possibly generating more worlds, in the presence of further repair-key operators, etc.). The result of evaluating such an enriched query Q over a relation R is thus a *probabilistic database* (see e.g. [26, 2, 1]), and we denote it by $Q(R)$. $Q(R)$ is a set of relations, each corresponding to a possible world and associated with a probability value.

SQL with repair-key. We then enrich the SQL syntax to account for the repair-key operator in a straightforward way. The new operation `REPAIR-KEY[a @ P] ON R` is the counterpart of repair-key$_{a@P}(R)$. Similarly to the case of relational algebra, the enriched SQL Query is evaluated in the standard manner, except that the evaluation of a `REPAIR-KEY` operation results in the generation of new possible worlds, in which further query operations are applied, and so on.

User	Authority
Alice	2
Bob	5
Carol	3
Dan	4

Table 2: Users

Country	Capital	User
China	Beijing	Alice
Netherlands	Amsterdam	Alice
Netherlands	Amsterdam	Bob
Netherlands	Hague	Carol
China	Shanghai	Dan

Table 3: CapitalsByUsers

EXAMPLE 2.2. *Consider the relation Users in Table 2, containing authorities values of different users, and the relation CapitalsByUsers where each tuple corresponds to a fact (a country and its capital) submitted by a user. Note that in contrast to the relation in Table 1, the authority value assigned to each fact does not appear in this table, and one has to first join this table with the User table, and only then the repair-key operation can be performed to probabilistically choose capitals. This is achieved by the following rule.*

```
DROP BelievedCapitals;
INSERT INTO BelievedCapitals
REPAIR-KEY[Country @ Authority] ON
(SELECT Country, Capital, Authority
 FROM CapitalsByUsers AS CBU, Users AS U
 WHERE CBU.User = U.User);
```

Note that here, the output of the selection sub-query is the relation used as input to the repair-key construct. The output of the query (probabilistic choice of capitals) is written into an additional relation, named BelievedCapitals. The need for using the Drop command to omit old copies of BelievedCapitals will become apparent below, when we introduce the while language and repeatedly apply this rule.

2.3 While language

We next define our probabilistic while language, used for recursive application of SQL with repair-key queries. The query now consists of three parts: (1) a set of update queries U, which implement the data cleaning rules and are written in the enriched SQL as explained above, (2) a boolean condition C that is used to decide whether a given instance is "clean", i.e. non-contradictory and (3) a query Q written in standard SQL, that is in fact the question of interest, and needs to be evaluated on clean instances of the Database (e.g. "what is the capital of China?").

We explain the language semantics in 3 steps. First, we assume that U consists of a single update query and Q is a boolean query (predicate). Then we consider the case where Q is not necessarily boolean, and finally we consider the case where U contains multiple update queries.

Given the input database, and an update SQL query U (possibly enriched with repair-key), the evaluation under the While language semantics follows the following program:

```
State := the input Database;
forever do
   State := U(State);
```

Note that if U contains an application of the repair-key construct, then $U(state)$ is a database that is probabilistically chosen out of several options. Thus the while-loop is a *random walk* over a *Markov Chain* (MC), whose states are the possible database instances, and the transition probabilities are determined by the repair-key operation (and the corresponding attribute values used by the repair-key construct as the probability column). In principal, we would like to define a query semantics that is based on the probability that a query is satisfied in a random state in the course of a random walk over this MC. We note however that, since we allow arbitrary SQL queries here (in contrast to the relational algebra rules in [11]) this MC may not be ergodic [14], in which case the above probability is not well defined. We thus refine the MC definition, as follows: we introduce some (low and configurable) probabilities ϵ_1, ϵ_2 to stay in the current state or to return to the initial database state; with probability $1 - \epsilon_1 - \epsilon_2$ we make another probabilistic choice, this time based on applying the repair-key construct as above. The refined MC is called the MC induced by the query and update rules, and we can show that it is ergodic.

The following definition is then adapted from [11]:

DEFINITION 2.3. *Given an input database D and a query (U, C, Q) where U is an update query written in SQL enriched with repair key, and C, Q are boolean queries, Let $M(U, D)$ denote the Markov Chain induced by the query U and the initial (input) Database instance D, as defined above. Let $seq = [s_1, ..., s_k]$ be a sequence of states in $M(U, D)$, such that s_1 is the initial database state and s_i is a possible transition from s_{i-1}, associated in $M(U, D)$ with its probability p_i. We denote the length of seq by $len(seq)$. The probability of seq, denoted $Pr(seq)$, is $\prod_{i=1,...,k} p_i$. The probability of being in a given database state s, at an arbitrary point in time, in an infinite random walk over database instances, is then defined as*

$$Pr(s) = \lim_{k\to\infty} \sum_{\{seq|len(seq)=k\}} Pr(seq) \cdot \frac{|\{i|s_i = s, 1 \le i \le k\}|}{k}.$$

Finally, the result of evaluating (U, C, Q) over D (denoted $P((U, Q); D)$ is defined as $P((U, C, Q); D) = \frac{\sum_{\{s|Q(s)\wedge C(s)\}} Pr(s)}{\sum_{\{s|C(s)\}} Pr(s)}$, where $Q(s)$ ($C(s)$) means that the boolean query Q (C) holds for the database instance s.

We can show (following our construction, see [14]) that the limit used in the above definition exists and is finite.

In most of our examples, the condition C is simply a boolean query that verifies that there is no primary key violation in any of the tables. For brevity, whenever this is the case in the sequel, we omit the explicit description of C.

Multiple Rules. Definition 2.3 assumes that the query Q consists of a single update query (rule). In practice, we may need to use multiple rules that interact in a recursive manner. In this case, the semantics is such that in each iteration of the while-loop, we evaluate all queries "in parallel", on the *old* Database state, and only then update the Database state with the queries results.

EXAMPLE 2.4. *Re-consider the rule in Example 2.2, and assume now that U additionally contains the following rule.*

```
UPDATE Users
SET Authority = (SELECT CorrectFacts
                 FROM Q1
                 WHERE Q1.user = Users.User)

Q1 = SELECT user, COUNT(DISTINCT country)
     AS CorrectFacts FROM Q2
     GROUP BY user;

Q2 = SELECT user,country,capital
     FROM UserCapitals UC
     WHERE EXISTS
             (SELECT *
              FROM BelievedCapitals BC
              WHERE BC.country = UF.country AND
                    BC.capital = UF.capital);
```

The auxiliary query Q1 computes for each user the number of facts that were submitted by her and "survived" the cleaning phase, i.e. appear in BelievedCapitals (we assume for simplicity that every user contributed at least one such fact); it does that by using another auxiliary query, Q2, that associates each surviving fact with the user that submitted it. Then the number of correct facts submitted by the user is assigned as her new authority value.

The while-loop now entails a repeated execution of the two rules in U: in each iteration, the effect of the first rule is a probabilistic choice of facts that populate BelievedCapitals table, based on the current users authorities. The second rule then re-computes the of users' authorities, based upon the number of facts that they contributed and were copied to BelievedCapitals (meaning that they are believed to be true). The new authority values will affect the probability of each fact submitted by these users to be chosen by the repair-key construct and consequently to appear in BelievedCapitals, which again affect the users' authorities and so forth.

Further consider a boolean query Q that asks whether Shanghai is the capital of China. In this case, the query answer will reflect the fraction of iterations throughout the while-loop (as the iterations number goes to infinity) in which (Shanghai,China) appeared as a tuple in BelievedCapitals.

Non-boolean queries. Definition 2.3 assumes that the query Q is boolean, while in general Q may be an arbitrary SQL query. We assume here that the possible query answers contain only values of the active domain. In this case, the query answer consists of all tuples that appear in the evaluation of Q on any clean database *state* obtained during the while-loop, and its probability is the fraction of these states in which it appeared, namely:

DEFINITION 2.5. *Let (U, C, Q) be a query and let D be the input Database. The output database (denoted $Res((U, C, Q), D)$) contains a set of pairs of the form (tuple, probability). The set of tuples in the result is the set of tuples $\{t \mid \exists state.C(state) \wedge state \in U^*(D) \wedge t \in Q(state)\}$, where $U^*(D)$ is the set of all database states that can be obtained by one or more subsequent applications of U, starting from a database instance D; and $C(state)$ means that the condition C holds for the Database instance state. The probability value assigned to t is $P((U, C, T); D)$, where T is the boolean query asking for the existence of the tuple t in the Database.*

EXAMPLE 2.6. *Let Q be the following query:*

```
SELECT * FROM BelievedCapitals
WHERE Country = 'China';
```

The output consists of all possible capitals of China according to users, along with their probabilities to appear in clean Database instances during the random walk induced by U. Consider query evaluation for Q with respect to the input Database and cleaning rules given above. At the first step, exactly one possible capital of China is chosen to appear in BelieveCapitals (Shanghai is chosen with probability of $\frac{4}{6}$, Beijing with probability of $\frac{2}{6}$), and the same for Netherlands (Amsterdam is chosen with probability of $\frac{7}{10}$, Hague is chosen with probability of $\frac{3}{10}$). Then, the user scores are updated according to the chosen tuples; specifically, note that Alice will win a point whenever Amsterdam is chosen (and this is a likely event, due to the support of Bob). These points reflect our increase of trust in Alice answers, and can lead to an increased probability of the (Beijing,China) tuple (also submitted by Alice) appearing in further clean instances.

Nesting queries. As we shall observe in section 4, exact computation of query probabilities can be computationally expensive. Thus we will resort to approximations. To allow (approximated) probabilities to be used in more complex queries. To this end, we introduce a new construct called $ApproxProb(U, C, Q)$, with the following semantics: when given a query (U,C,Q) as described above, the result of $ApproxProb(U, C, Q)$ is a relation with a new, unique column called *Prob*, that contains for each tuple its approximated probability to hold, computed as the fraction of the worlds in which it appeared in the while-loop as described in the semantics above. This construct can then be used as a sub-query in any query. We note that the ApproxProb construct requires fixing some parameters related to the approximation algorithm (see section 4); these are preconfigured by the designer.

Remark. Note that according to our semantics, a query answer consists of all tuples that may appear in possible answers to the query, and their individual probabilities of appearing in the query result. In particular, the answer does not capture dependencies in-between the appearances of these tuples. For instance, in the above example, the answer consists of a set of possible capitals of China, each with its associated probability; but it is clear that in every possible world there exists only a single capital of China.

3. EXPRESSING GAME-RELATED TECHNIQUES

To illustrate the expressiveness and flexibility of the language, and consequently the usefulness of the framework, we show in this section how different useful policies for the various aspects of the game can be expressed using our formalism. We explain the policies using the same database described in Section 2; for brevity, we also use an additional relation called UserCapitals, which is obtained as the inner join of CapitalsByUsers and Users (i.e. each tuple in it consists of a fact, a contributing user, and the authority value of this user). Also, for coherence of presentation, we focus first implementing different techniques for the task of question answering. Then, we show that other game-related tasks are solvable by very slight modifications of these implementations.

3.1 User-based PageRank

In section 2 we described an implementation of a PageRank-style algorithm; we next describe a variation of it, showing the ease of modifying/refining a given technique. Note that in the original policy, at each iteration we probabilistically chose between facts based on their relative support. In contrast, a different possible approach is to first (probabilistically) decide on a set of users who are believed to be accurate, and consider all of their submitted facts as true (for the current iteration).

Implementing this approach requires a slight change in the DB schema (w.r.t the schema given in Section 2: we add to the Users table a new *Correctness* attribute; the idea is that each user is now associated with two tuples, with correctness values of 0 and 1, and possibly different authority values for both. For each user we will probabilistically choose one of these tuples; a choice of the tuple with correctness value 1 indicates that the user is considered trusted for this iteration, and all her submitted facts are considered true.

Then, to implement the refined policy, we make changes to the update rules shown in section 2), as follows. We replace the rule for choosing facts in Example 2.2, by the following rule (and we combine this rule with a slight refinement of the rule in Example 2.4, see below):

```
DROP BelievedCapitals;
INSERT INTO BelievedCapitals
SELECT Capital, Country, Authority
FROM UserCapitals AS UC, Q1 AS U
WHERE UC.User = U.User;

Q1 = SELECT * FROM
(REPAIR-KEY[User @ Authority] ON Users)
WHERE Correctness=1;
```

Query Q1 selects the subset of users considered to be reliable for this iteration. The obtained table is then joined with UserCapitals, to obtain all facts submitted by these users, and use them to update the BelievedCapitals table. The rule described in Example 2.4 is then also refined, to update also the authority values of incorrectness (correctness=0) tuple of each user, by counting her submitted facts that do not appear in the clean instance (details omitted).

The correctness condition C requires that the BelievedCapitals table is consistent, i.e. each country is associated with exactly one capital city (this is easy to implement in SQL), and the query to be sampled is the question to be answered, e.g. to output the probability of possible answers for capital cities of England we can simply use the query:

```
Q=SELECT * FROM BelievedCapitals
  WHERE Country = 'England'
```

3.2 Non-probabilistic Techniques

We now turn to the implementations of some non-probabilistic techniques presented in [15]. This also demonstrates the flexibility of the approach.

Fixpoint Computation. Note that in the above examples, the policies stored the authority values of users, and used

them to probabilistically decide on correct facts. In contrast, the authorities score of *facts* (i.e. to what extent we believe they are correct) are not stored but rather calculated on the fly and used in the application of repair-Key. However, these scores can be materialized and stored as well. In the fix-point policies described in [15], the scores of facts (users) are "frozen" and used for the calculation of users (facts) scores. In general, the score of a fact (user) is expressed as an aggregation of the scores of users submitting it (facts submitted by the user). To implement this, an additional table storing the reliability of facts (referred to as FactsScores) should be added to the database schema. Then, we replace the update rules described in Section 2, with two rules as follows. The first rule is:

```
DROP BelievedCapitals;
INSERT INTO BelievedCapitals
SELECT Country, Capital, SUM(Authority)
FROM UserCapitals AS UC, Users AS U
WHERE UC.User = U.User
GROUP BY Country, Capital;
```

According to this rule, the score of a fact is calculated as the sum of all scores of users introducing it to the system. The second rule (omitted for space constraints) similarly computes the users scores, based on the scores of the facts they contributed. The two rules define a recursive process of updating the user scores based on facts scores, and facts scores based on user scores, and so on, run in a forever loop.

The condition C dictating when to sample is the constant True (we always sample) and the query Q corresponds to the question asked (e.g. selection of all capitals as above).

Majority and Voting. In [15] the authors also describe simple non-recursive, non-probabilistic methods, namely *Majority and Voting*. The Majority policy decides between facts according to the majority choice (weighted by authorities). The voting policy allows users not only to introduce facts but also state that some given facts are inaccurate. We note that these policies do not require neither probabilistic decisions nor recursive computation, and can be implemented in standard SQL; the implementation is straightforward and is omitted for lack of space. Our framework allows standard SQL queries and can be used to implement these policies.

3.3 Solving Additional Contradiction Types

We have so far seen examples where the contradictions were yielded by a primary-key violation; but in practice there are be many other types of errors/constraint violations that one may wish to account for. We next consider several such examples.

Foreign-key violation. We start by exemplifying rules that account for *foreign-key* violations. To continue with our running example, assume now the existence of a Parliaments table, storing the names of world parliaments and the cities they are located in [1]; further assume that there is a foreign key constraint indicating that cities in the Parliaments table must appear as capital cities. In such case, we may use the following update rule, together with the rules in Examples 2.2 and 2.4:

[1]Other examples for foreign key constraints rise when we incorporate the use of external data, e.g. from Wikipedia

```
INSERT INTO BelievedCapitals
SELECT UC.Country, UC.Capital, UC.User
FROM UserCapitals AS UC, Q1 AS PAR
WHERE UC.Capital = PAR.Capital
AND UC.Country = PAR.Country;

Q1 = SELECT Capital FROM Parliaments
WHERE Capital NOT IN
(SELECT Capital FROM BelievedCapitals);
```

Q1 returns cities that appear in the Parliaments table but not in BelievedCapitals; the main query then looks for corresponding facts in UserCapitals, as it may be the case that these facts were submitted by users, but then omitted by the rule of Example 2.2 [2]. Since the omission of these facts cause a foreign key constraint violation, they are returned to BelievedCapitals, possibly causing now a primary-key violation, which will be handled in the next iteration by the rule of Example 2.2, and so on.

While the above rule solves all foreign constraints encountered at each iteration (consequently possibly leading to primary key violations), one may settle the tension between the primary key and foreign key constraints by choosing to add only a subset of the facts required according to the Parliaments table. To this end, recall that we have exemplified above (Section 3.1) the use of repair-key for choosing a subset of tuples; a similar policy may be applied here.

Multiple Answers. The methods described above are appropriate for cases with a primary key constraint, i.e. cases in which for every question there exists one correct answer in the dataset. However, in some cases there are no such key constraints, e.g. "Names of Tennis Players". In this case, a simple technique is to choose probabilistically a subset of (all) user answers, based on their popularity support. For that, we use similar schemas to the ones described above: answers are stored in a table UserTennisPlayers with Attributes User, PlayerName; BelievedTennisPlayers has a single attribute PlayerName, and it will store answers that are believed to be correct. We also use an auxiliary table T1 (T2) which contains a single attribute named Correctness, and a single tuple with value 1 (0) that enables the system to select a subset of the cartesian product, as described below. We then define the following rule:

```
INSERT INTO BelievedTennisPlayers
SELECT PlayerName FROM
(REPAIR-KEY[(PlayerName) @ ProbVal] ON
 ((Q1 CROSSJOIN T1) UNION (Q2 CROSSJOIN T2)))
WHERE Correctness = 1;

Q1 = SELECT PlayerName,
SUM(Authority) AS ProbVal
FROM UserTennisPlayers, Users
WHERE UserTennisPlayers.User = Users.User
GROUP BY PlayerName;
```

[2]We assume that the rule described here is fired at the end of each iteration, after the rules of Examples 2.2 and 2.4. It is easy to achieve such synchronization e.g. via the use of a dedicated table, with a tuple whose value dictates which rules can now fire. We omit the formal description of this synchronization for simplicity of presentation

```
Q2 = SELECT PlayerName,
Q3.TotalAuth - SUM(Authority) AS ProbVal
FROM UserTennisPlayers, Users, Q3
WHERE UserTennisPlayers.User = Users.User
GROUP BY PlayerName;

Q3 = SELECT SUM(Authority) AS TotalAuth
FROM Users;
```

Q1 sums up the authorities values of users that submitted each fact; Q2 computes for every fact, the total authority values of users minus the sum of authorities computed in Q1. Then, the main query creates the cartesian product of Q1 (Q2) with T1 (T2) and unions the results, then applies repair-key. Thus, for every fact we will choose tuple with Correctness value 1 with probability that is the sum of authorities of the users that supported it, divided by the sum of all authorities of users.

3.4 Additional Game-Related Tasks

So far we have focused on techniques for question answering. As noted in the introduction, there are other tasks that the game designer must face, including the assignment of user scores, the choice of questions to pose to gain maximal information, the choice of users that are most likely to answer them correctly, etc. Interestingly, these can be achieved by modifying (slightly) any one of the policies suggested above for question answering. We show this next for simple techniques for these tasks. First, consider the identification of questions to pose to users. Here we can use a simple method that selects the questions for which all possible answers have similar (defined by some threshold) probabilities, and also the number of users that have answered this is below some threshold. These are questions for which there exists a high level of uncertainty. To implement this algorithm, we can choose any of the rules suggested above, and feed them (along with a corresponding sampling condition and a query Q of interest, e.g. selecting all capitals of all countries) to the *ApproxProb* construct, whose output will be in this case the set of possible capitals with probability value assigned to each of them; then the result can easily be queried via standard SQL queries to identify questions (countries in our example) for which all answers have similar probabilities. Identifying the questions with low number of users and joining the results is then straightforward.

To identify which users are more credible (and thus questions should be directed to them), we can use the same update rules, but change the query Q to query the Users table for users with authority greater than some threshold. The probability of this to be the case, for each user, can then be used for choosing the "most knowledgable" users, to pose questions to. More sophisticated algorithms to select users suitable for particular questions can be similarly implemented and we omit the details for space constraints.

4. SYSTEM IMPLEMENTATION

The data management platform presented in this paper was used as the data layer of a data sourcing game called Trivia Masster, demonstrated in [10]. Trivia Masster players are presented with Trivia questions in a variety of topics. The answers are used to construct a database of facts and the players are assigned scores based of the quality and value of their contribution. The constructed database is in turn used to answer queries posed by (a possibly different set of) users. The demonstration [10] only provided a high-level description of the system; here we describe some implementation issues that were addressed to put the platform into practical use, in the context of Trivia Masster.

Query Evaluation Algorithm. It was shown in [11] that computing exact probabilities (for the settings presented there) is not possible in PTIME (w.r.t. the database size) unless P=NP. Recall that the language used here is an extension of [11]; consequently, the same hardness results hold here as well. We thus employ an approximation algorithm that is an adaptation of the one presented in [11] (extended to account to the additional expressive power of our language), and stands as a particular case of a *Markov Chain Monte Carlo* [27] algorithm. Recall that our rules may include a repair-key construct, in which case their application may result in many possible results (other Database Instances), each with some probability. Now, consider a Markov Chain (MC) induced by the rules as defined in section 2. Our algorithm performs a random walk over this MC, starting from its initial state and at each point randomly choosing the next state, according to the distribution on the MC transitions. Whenever the walk reaches a state that corresponds to an instance D that satisfies the given condition, our algorithm evaluates the query over D and keeps record of the tuples that appear in the result of this evaluation. The algorithm halts when convergence is reached; we discuss in the sequel how to define and determine convergence. The returned result is then the set of tuples that occurred in evaluation results, where each tuple is associated with a quantity that is the number of its occurrences divided by the number of clean Database Instances observed. This quantity is an approximation for the tuple probability [11].

Online vs. Offline processing. Our query evaluation algorithm may be naïvely employed whenever a query is run against the Database. Such a naïve rule-based execution until convergence takes a few hours. We next explain how to preempt some of the processing offline.

For simplicity, let us first assume that all evaluated queries are simple select-project queries (i.e. there are no joins and no aggregates). Then, at the offline stage, for each relation R in the database, we evaluate a query that selects all tuples from R (`SELECT * FROM R`), and using the given update rules. The result of this query evaluation is the set of all tuples of R, each associated with a value reflecting its (approximated) probability of appearing in a (correct) Database Instance. This set of tuples can be considered as an extended relation $ext(R)$ that has the same schema as R, but with an additional attribute of probability. Now, in online time, given a select query Q over R posed by the user, we simply evaluate it (in the usual sense of SQL queries evaluation) over $ext(R)$; the result consists of all tuples that are in the result of evaluating Q over R, but each is now also associated with its probability of appearing in a clean instance. If Q also projects over some attributes, then we need to slightly rewrite it into a query $ext(Q)$ that also sums the values in the (newly added) probability attribute, grouped by the projected attributes; then we evaluate $ext(Q)$ over $ext(R)$ to get the desired results. Since this corresponds to "conventional" SQL query evaluation, the online performance is satisfactory (see Section 5).

When the posed queries may include joins, the offline processing step becomes somewhat more intricate. It is well-known [9] that for probabilistic databases, computing query results for each individual relation and then joining the results can cause errors in the computation of probabilities. This is due to probabilistic dependencies between tuples that are ignored by the above naïve computation, but may occur in presence of (e.g. foreign key) constraints over the data, or the use of projection in the query [9]. This is also a challenge in our settings. Consequently, we refine the MCMC algorithm described above, as follows: we repeatedly apply the update rules, as in the above algorithm, but whenever we reach an instance where sampling should be performed, we also perform all joins (using all join conditions) that occur in the queries supported by the given application, and accumulate the count of tuples that appeared in the joined relation. The output of this process now contains, in addition to the ext(R) extended relations for each original relation R, also relations of the form ext(R_1 JOIN R_2 JOIN...R_N). Queries with joins may then be translated to conventional SQL queries over the extended database, and then evaluated in the same exact manner as the one explained above for select-project queries.

Data Inserts. Our database rapidly grows as users answer Trivia questions and consequently add facts to the database. This means that the offline processing must be run sufficiently often, to account for these inserts. Our experiments show that on large data sets, the offline processing step runs for a few hours on a simple laptop, which allows one to run it daily. Naturally, a more desirable solution involves the development of an optimized incremental algorithm; this is a challenging future research.

Convergence. Recall that our query evaluation algorithm requires a procedure that decides convergence. There are many possible ways to define convergence [8]. One definition is based on stability, i.e. that the (approximated) probability values computed for the different tuples does not change by more than a given ϵ over the course of i samples (where ϵ and i are configurable). A difficulty lies in the fact that for the offline processing, there is no single concrete query in hand; and testing convergence for all possible queries is inefficient. Thus, we instead tested convergence of the *user scores* (relative to each other), and found that such convergence is a good predictor for the stability of the queries results. This solution is tailor-made to the context of the game, where the rule always involve computation of user scores. The convergence condition we used here requires that the user scores for 99% of the users (or more) do not vary more in more than 10% for a sequence of 10 iterations (or more).

5. EXPERIMENTS

As described in the previous section, we have used the platform as the data layer for Trivia Masster , allowing the game designers to define rules for cleaning and supporting queries over the data collected in crowdsourcing games. The real-life data collected using Trivia Masster is of relatively small scale. Thus, for our experiments we have used the schema on which the game is built (see [10]) and generated a larger scale synthetic data (see Section 5.1). The experiments tested both the performance and quality of different techniques, when implemented and run using our platform. The experiments were designed to address two main goals. First, we wished to examine the scalability and quality of results of algorithms implemented using our framework. We show that they scale well and are thus practical to use. Second, we note that the declarative nature of the platform allows the game designers to easily compare and modify the implemented policies, and to thus get insights that eventually lead to a better game design. Our experiments show concrete examples of such insights we have derived, providing further indications for the usefulness of the platform.

The experiments were performed on an Intel Core2 Quad CPU 2.66GHz and 4GB RAM. We next describe the parameters varied in the generation of synthetic data for the experiments, and then describe the experimental results, first for experiments on performance and then on quality.

5.1 Data Generation

We have aimed to generate synthetic data that reflects different scenarios that may be encountered in real-life. For that, we have started with a domain of 1000 possible questions, to each of which we have generated one correct answer and a domain of possible incorrect answers. Then, we have simulated players by distributing (a) the number of questions answered by every individual (simulated) player answers, and (b) the percentage of these questions that she has answered *correctly* (reffered to in the sequel as the user correctness rate). We have experimentred with various such distributions, as well as with the number of users. The combination of number of usres and distribution of their answers determines the database size.

We next detail the values that we have considered for each of these parameters. The default values for all except the users number (for which we have considred a large number) are based on interpolations of the values we have observed in the (small) real-life data we have collected using the game.

- Number of users. We varied the number of users from 0 to 50k. The default value is 50k users.
- The number of questions answered by a single user varied from 10 to 100. The default value is 100. Note that combined with the default number of users, this yields a default database size of 5M facts.
- The distribution of correct facts among users. We have considered uniform datasets where all users have the same correctness rate (varied from 0% to 100%) , and profiled datasets where the users are split to 10% "experts" (90% correctness rate) and 90% non-experts (10% correctness rate). An incorrect answer is chosen uniformly among the possible (incorrect) answers. The default is a uniform dataset, where all users have 10% correctness rate.
- The policies (update rules) that are used. We have implemented all policies presented in Section 3.

5.2 Performance

In the previous section we have discussed online vs. offline processing, and noted that most of the computation can be preempted offline. Indeed, the online processing time that we have observed for all settings is marginal (several dozens of milliseconds). Thus, we only present the results for the offline processing.

We show here only a representative sample of the results, namely those obtained for probabilistic PageRank (see Section 3.1), fixpoint and majority policies (section 3.2) and

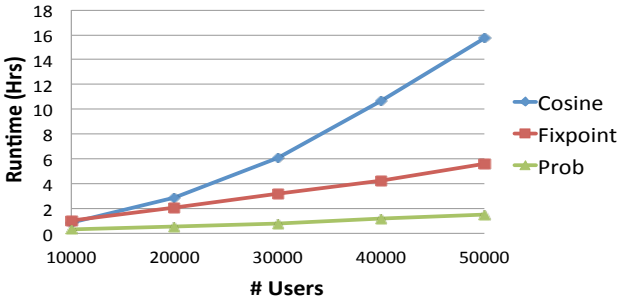

Figure 1: Scalability of Query Evaluation

Cosine algorithm [15] whose implementation was omitted for lack of space. Each performance experiment was run 10 times and the graphs show the average results. Figure 1 shows the performance of the different algorithms as a function of the database size. In this experiment we varied the number of users from 0 to 50k, yielding a maximum of 5M facts. The running time of the majority algorithm was marginal (split seconds) and is thus omitted from the figure. We observe that the running time of the PageRank algorithms (both the probabilistic and fixed-point versions) algorithms increases moderately with respect to the Database size. The running time of the Cosine algorithm increases more drastically. This can be explained by complex SQL queries in the Cosine algorithm implementation. In this implementation, a score of a user is defined based on the sum of scores the facts he provided have. Obviously, this query involves aggregate operators that are run over the joined data sets of users and facts. In contrast, the PageRank implementation consists of simple queries which converge much faster, and the majority algorithm performs a trivial computation.

Figure 2 shows the effect of number of users on the number of *iterations* until convergence. Observe that more iterations are required for the Probabilistic PageRank to converge. However, these iterations are short and the overall time is shorter than the time of the other algorithms. The figure shows a noticeable increase for the Probabilistic Pagerank as the database size increases while it is insignificant for the non-probabilistic algorithms.

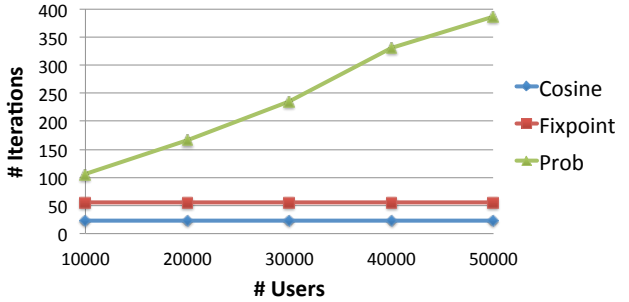

Figure 2: Number of Iterations

Figure 3 shows the algorithms running time when varying the number of questions each user answers (this of course also affects the database size) while playing the trivia game. Again, the performance results for the majority algorithm are omitted as it operates in split-seconds. For all algorithms we can observe a rather moderate increase of runtime as the number of answers increases.

5.3 Quality

So far we have considered the running time of the various techniques. It is also interesting to compare the *quality* of the results that they achieve. To measure quality, we considered the 1000 possible questions (presented earlier to the

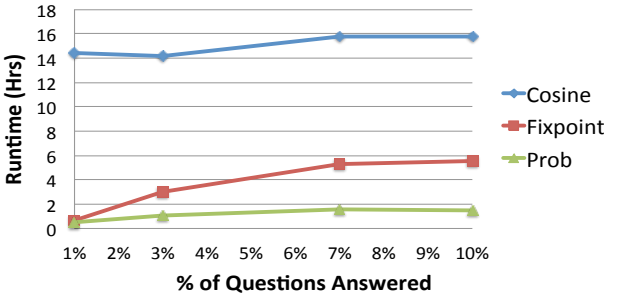

Figure 3: Varying % of Questions Answered

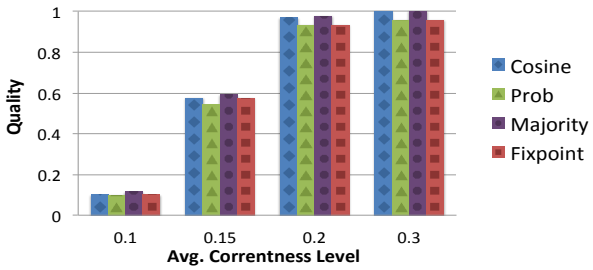

Figure 4: Quality for varying correctness level

players), as queries on the database that are evaluated using our algorithm with some (varying) implemented technique as update rules. The quality is then the fraction of queries for which the system computes the correct answer.

Figure 4 shows the quality of the results produced by the different techniques. In this experiment we have also varied the correctness level for the users (but always used a uniform correctness level among all users) in order to see how it affects the quality of algorithms results. We observe that the difference in quality between the different algorithms (for the particular dataset that we have used) is marginal (less than 10 % relative difference). Furthermore, there is an expected correlation between the quality of results and the correctness level. It is interesting to see that for the when the correctness level reaches 20%, the algorithms return the correct answer almost always. Consider, for example, the probabilistic PageRank algorithm in a case with 15 possible answers for a given question, and 20 % correctness. Here the correct answer is chosen with a probability of 0.2 while the rest 14 answers are chosen with probability of $\frac{0.8}{14} = 0.057$ each (recall that we use a uniform distribution in the generation of incorrect answers), and the algorithm will eventually conclude on the correct answers.

We have also experimented with profiled datasets, where the correctness level varies among different users (The figure is omitted due to lack of space). Here the quality results differ much more significantly, as the PageRank techniques (probabilistic and fixpoint) outperform majority and cosine by over 100% (relative difference). The reason is that the PageRank algorithm gains more confidence in users that provide correct answers, and is thus more suited for the "experts" vs."non-experts" case of distribution.

6. RELATED WORK

Crowdsourcing is an emerging paradigm that harnesses a mass of users to perform various types of tasks [7, 32, 21, 30, 25]. *Games* are particular tools that attract the crowd to contribute [32, 21] to such tasks. In particular games were used in [31] for collecting large amount of information on images for which metadata is unavailable. This work was followed by a set of games known as Games With a Purpose [32] that aims at harnessing the crowd for various difficult tasks such as object recognition in images and collection of

user preferences [16]. Other methods showed how people can also help to improve the quality of search engines [21] and complete missing information in social networks, such as tags associated with its members [4]. We have focused in this paper on the data layer of a particular kind of crowdsourcing games, namely *datasourcing* games that aim to use the collective wisdom to construct a large database of facts.

Much research has been recently directed in the databases community to the development of DB platforms that allow for declarative specification of the crowdsourced data components [13, 23]. These platforms are providing declarative lagunage support and tools to define what data will be retrieved from the crowd (the choice of questions to ask players in our case), for example by adding a *CROWD* operator to a certain column in a *Create* SQL statement of a table [13] and how the flow of such tasks should be managed [23]. While the choice of data to ask the crowd for can be done declaratively in these frameworks, the other tasks described in this paper such as aggregating the data (and deciding which data is correct), choosing which users to ask, assigning scores to users etc. are not addressed or done in a hard-coded manner (see, for example, the *Combiner* operator implemented as *MajorityVote* in [22]). Our platform allows declrative formulation of policies for these tasks and is therefore complementary to the platforms presented above.

Policies for determining correctness of data in presence of contradictions often appear in the context of *data cleaning*. We have already mentioned some data cleaning policies and showed how to implement them using our framework. Many other policies have been proposed in the literature: In [3] the authors present a different approach for cleaning by using string-transformation rules for correcting errors in the data. [5] presents a technique to solve key violations using probabilistic choice over possible Database repairs. [24] discusses techniques for evaluating the trustworthiness of information sources. These solutions are all hard-coded, in contrast to the generic declarative framework that we propose here.

Information integration often entails fusion of data from various sources (e.g, [6]). This requires the identification of common objects and the resolution of possible conflicts. Such (possibly probabilistic) data fusion algorithms may also benefit from the declarative framework described in the present paper and we intend to study its application to this domain in future work.

Last, we note that there are some declarative frameworks that support queries on probabilistic data (e.g. [1, 17, 12, 18]). In particular, in [18] the authors present a declarative framework for probabilistic rules, based on Markov Chains. However, all of these works do not allow the definition of recursive rules, hence for example cannot express the PageRank-style cleaning rules described here.

7. CONCLUSION

We presented a novel declarative framework for the data management layer of data-sourcing games. At the core of our framework is a declarative language that allows to express probabilistic and recursive policies, which we demonstrated to be useful for different data management aspects of such games. We further described implementation issues addressed when putting the platform into practical use, in the context of the Trivia Masster game, and reported the results of our experimental study with respect to the system.

The design of dedicated optimization algorithms for the framework is a challenging future research. In particular, we intend to study incremental maintenance for rapidly adjusting to changes and additions to the facts database, thereby avoiding a full off-line (re)evaluation.

8. REFERENCES

[1] P. Agrawal, O. Benjelloun, A. Das Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB '06*.

[2] L. Antova, C. Koch, and D. Olteanu. "Query Language Support for Incomplete Information in the MayBMS System". In *Proc. VLDB*, 2007.

[3] A. Arasu, S. Chaudhuri, and R. Kaushik. Learning string transformations from examples. *PVLDB*, 2(1), 2009.

[4] M. Bernstein, D. S. Tan, G. Smith, M. Czerwinski, and E. Horvitz. Collabio: a game for annotating people within social networks. In *UIST '09*. ACM.

[5] G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. In *VLDB '10*.

[6] J. Bleiholder and F. Naumann. Declarative data fusion - syntax, semantics, and implementation. In *ADBIS*, '05.

[7] D. C. Brabham. Crowdsourcing as a Model for Problem Solving: An Introduction and Cases. *Convergence*, 14(1), 2008.

[8] M. K. Cowles and B. P. Carlin. Markov chain monte carlo convergence diagnostics: A comparative review. *Journal of the American Statistical Association*, 91, 1996.

[9] N. Dalvi and D. Suciu. "Efficient query evaluation on probabilistic databases". In *Proc. VLDB*, 2004.

[10] D. Deutch, O. Greenshpan, B. Kostenko, and T. Milo. Using markov chain monte carlo to play trivia (demo). In *ICDE*, 2011.

[11] D. Deutch, C. Koch, and T. Milo. On probabilistic fixpoint and markov chain query languages. In *PODS '10*.

[12] R. Fink, D. Olteanu, and S. Rath. Providing support for full relational algebra in probabilistic databases. In *ICDE*, 2011.

[13] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD Conference*, 2011.

[14] D. Freedman. *Markov Chains*. Springer-Verlag, 1983.

[15] A. Galland, S. Abiteboul, A. Marian, and P. Senellart. Corroborating information from disagreeing views. In *WSDM '10*.

[16] S. Hacker and L. von Ahn. Matchin: eliciting user preferences with an online game. In *CHI '09*.

[17] J. Huang, L. Antova, C. Koch, and D. Olteanu. Maybms: a probabilistic database management system. In *SIGMOD Conference*, 2009.

[18] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *SIGMOD '08*.

[19] C. Koch. Approximating predicates and expressive queries on probabilistic databases. In *PODS '08*.

[20] N. Leone and et al. "The INFOMIX system for advanced integration of incomplete and inconsistent data". In *Proc. SIGMOD*, 2005.

[21] H. Ma, R. Chandrasekar, C. Quirk, and A. Gupta. Improving search engines using human computation games. In *CIKM '09*.

[22] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller. Human-powered sorts and joins. *PVLDB*, 5(1), 2011.

[23] A. Marcus, E. Wu, S. Madden, and R. C. Miller. Crowdsourced databases: Query processing with people. In *CIDR*, 2011.

[24] A. Marian and M. Wu. Corroborating information from web sources. *IEEE Data Eng. Bull.*, 34(3), 2011.

[25] Amazon's mechanical turk. https://www.mturk.com/.

[26] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE '07*.

[27] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Springer-Verlag, Inc., 2005.

[28] Shannon and Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, 1949.

[29] Q. Su, D. Pavlov, J.-H. Chow, and W. C. Baker. Internet-scale collection of human-reviewed data. In *WWW '07*.

[30] Top coder. http://www.topcoder.com/.

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

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