Query-Oriented Data Cleaning with Oracles

Moria Bergman [1] Tova Milo [1] Slava Novgorodov [1] Wang-Chiew Tan [2]
[1]Tel-Aviv University [2]UC Santa Cruz
[1] {moriaben, milo, slavanov}@post.tau.ac.il [2] tan@cs.ucsc.edu

ABSTRACT

As key decisions are often made based on information contained in a database, it is important for the database to be as complete and correct as possible. For this reason, many data cleaning tools have been developed to automatically resolve inconsistencies in databases. However, data cleaning tools provide only best-effort results and usually cannot eradicate all errors that may exist in a database. Even more importantly, existing data cleaning tools do not typically address the problem of determining what information is missing from a database.

To overcome the limitations of existing data cleaning techniques, we present QOCO, a novel *query-oriented* system for cleaning data with oracles. Under this framework, incorrect (resp. missing) tuples are removed from (added to) the result of a query through edits that are applied to the underlying database, where the edits are derived by interacting with domain experts which we model as oracle crowds. We show that the problem of determining minimal interactions with oracle crowds to derive database edits for removing (adding) incorrect (missing) tuples to the result of a query is NP-hard in general and present heuristic algorithms that interact with oracle crowds. Finally, we implement our algorithms in our prototype system QOCO and show that it is effective and efficient through a comprehensive suite of experiments.

1. INTRODUCTION

Databases are accessed for the information they contain and key decisions are often made based on the results that are returned. Regardless of the query interface that is provided for accessing information (e.g., free-text search in an internet bookstore or form-based filters for choosing travel destinations in travel agency website), users naturally expect to obtain correct and complete results to a query that is posed against the database. In practice, however, the expectation of always obtaining correct and complete results is difficult to realize since many of the databases are constructed by (semi-)automatically aggregated data from different sources and are likely to contain some inaccuracies and inconsistencies even if individual sources are free of errors.

Even though data cleaning is a long standing problem that has attracted significant research efforts (e.g., see [20, 28, 52]) for a number of years, the state-of-the-art data cleaning techniques that have been developed cannot usually eradicate all errors in a database. YAGO [57] ontology is one such example of a database that was built by automatically extracted data from Wikipedia and other sources. Data cleaning techniques have been applied to the YAGO dataset and achieved an accuracy of about 95%, namely leaving 5% still erroneous [57]. Even highly curated databases [7] (i.e., databases that were constructed through extensive human effort of verifying and aggregating existing sources) such as Uniprot [60] and the CIA world fact book [13] are unlikely to be completely void of errors. At the same time, the sheer volume of such databases also makes it impossible to manually examine each piece of data for its correctness. Even more importantly, existing data cleaning tools do not usually address the problem of determining what information is missing from a database.

To complement the efforts and overcome the limitations of existing data cleaning techniques, we propose a novel *query-oriented* data cleaning approach with oracle crowds. In our framework, materialized views (i.e., views which are defined through user queries) are used as a trigger for identifying incorrect or missing information. Our premise is that users' queries (and their corresponding materialized views) provide relevant and focussed perspectives of the underlying database and hence, facilitates the discovery of errors. If an error (i.e., a wrong tuple or missing tuple) in the materialized view is detected, our system will interact minimally with a crowd of oracles by asking only pertinent questions. Given a view, we assume the crowd are relevant domain experts (hence, the name "oracle crowds[1]") who are likely to answer questions posed by our system correctly. The answers to the questions will help to identify how to clean the underlying database in order to correct the error in the materialized view. More precisely, the answers to a question will help to identify the next pertinent questions to ask and ultimately, a sequence of edits is derived and applied to the underlying database. The edits will bring the database closer to the state of the ground truth and, at the same time, correct the error in the materialized view. As we will describe, our algorithms effectively prune the search space and minimize the amount of interaction with the crowd while, at the same time, maximize the potential "cleaning benefit" derived from the oracles' answers.

Our algorithms are implemented in QOCO system. We emphasize that even though QOCO can be used as a standalone system for data cleaning, it can also be used to complement existing data cleaning techniques. Specifically, after the data is cleaned with traditional techniques, QOCO can be activated to monitor the views that are served to users/applications. Whenever an error is reported

[1]Even though there may not always be oracle crowds for every dataset, it is reasonable to assume that every important or curated dataset will have some domain experts who will be the oracles.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2737786.

Games

Date	Winner	Runner-up	Stage	Result
13.07.14	GER	ARG	Final	1:0
11.07.10	ESP	NED	Final	1:0
09.07.06	ITA	FRA	Final	5:3
30.06.02	BRA	GER	Final	2:0
12.07.98	ESP	NED	Final	4:2
17.07.94	ESP	NED	Final	3:1
08.07.90	GER	ARG	Final	1:0
11.07.82	ITA	GER	Final	4:1
25.06.78	ESP	NED	Final	1:0

Teams

Country	Continent
GER	EU
ESP	EU
BRA	EU
NED	SA
ITA	EU

Players

Name	Team	Birth year	Birth place
Mario Götze	GER	1992	GER
Andrea Pirlo	ITA	1979	ITA
Francesco Totti	ITA	1976	ITA

Goals

Name	Date
Mario Götze	13.07.14
Andrea Pirlo	09.06.06
Francesco Totti	09.06.06

Figure 1: World Cup Games database

in a view, QOCO can take over to clean the underlying database by interacting with the crowd.

An overview example Next, we illustrate the key ideas behind QOCO with an example database of World Cup Games. Assume that the data was extracted from some sport websites (e.g., [64, 63] and is therefore partially incorrect and incomplete due to errors in the automatic website scraping tools. Although one could argue that some of the errors in the data could be cleaned with previous automatic techniques (e.g., by comparing to FIFA official data [24]), the choice of this dataset for our running examples is deliberate. The true facts about World Cup Games are well-known, and it makes it easy for us to illustrate our key ideas without the need to delve heavily into the data. This choice also makes it easy for the reader to play the role of the crowd and/or user of QOCO.

We use D to denote the given dirty database and D_G to denote the correct ground truth database. A small sample of the World Cup Games dataset is depicted in Figure 1, which shows portions of four relations: Games lists the World Cup Games and stores the date, playing teams, stage of the tournament and the final score of each game. Teams records the teams' names and continents for teams that participated in various World Cup Games. Players records players who participated in the World Cup Games, including their team's name, birth year and birth place. Finally, Goals shows each player's name and a date of a game in which this player scored a goal. In the figure, the dark gray tuples are the wrong tuples in D (i.e., tuples that do not belong to the ground truth database D_G). The light gray tuples are the tuples that are missing from D (i.e., tuples that are in D_G but do not appear in D). All other tuples (marked in white) are correct (i.e., they belong to both D and D_G).

Consider a user query Q_1, defined below, which searches for European teams that won the World Cup at least twice.

$$(x) \text{ :- } \text{Games}(d_1, x, y, Final, u_1), \text{Games}(d_2, x, z, Final, u_2), \\ \text{Teams}(x, EU), d_1 \neq d_2.$$

When Q_1 is evaluated against the database D, the query result $Q_1(D)$ (i.e., materialized view of Q_1) consists of two tuples $\{(GER), (ESP)\}$. This output contains wrong answers such as Spain as well as missing ones such as Italy.

Note that in the absence of any knowledge about the ground truth (D_G), there are multiple ways to update D so that the wrong answers will no longer be part of the result. For example, to remove (ESP) from $Q_1(D)$, one can remove (ESP, EU) from Teams, or two of the four facts in Teams that represent a winning game of Spain. To add the missing tuple (ITA) to the result, one can add the tuple $(Italy, Europe)$ to Teams.

QOCO could ask the crowd whether Spain is in Europe, or if the tuples representing the World Cup finals in 1978, 1994, 1998, or 2010 are correct. Similarly, QOCO could ask the crowd whether there are tuples missing from the query result. Their replies would help QOCO correct the database.

In this simple example, it happens that there is a small number of tuples to verify for its correctness and hence, one or two questions would suffice. However, in general, the space of potential fixes may be large, and hence examining all of them may be prohibitively expensive. Our system employs an efficient novel algorithm to effectively prune the search space by carefully considering the order in which tuples should be examined. This way, the cleaning process is accelerated and the number of crowd questions is minimized.

Contributions. This paper makes the following contributions:

1. We formulate and present a novel query-oriented framework for data cleaning with oracle crowds. Under this framework, an incorrect (resp. missing) answer is removed from (added to) the result of a query through edits that are derived and applied to the underlying database. These edits are derived by minimally interacting with crowd oracles.
2. To assess the difficulty of the problem we first study two important special cases; determine the minimal interactions needed to derive a set of database edits to remove a tuple from a view, and resp. to add a tuple to the view. We show that these problems are NP-hard to solve even in these special settings.
3. We present heuristic algorithms for these special cases and show how these algorithms can be combined to arrive at a solution for the general query-oriented approach to data cleaning. Our algorithm for answer removal uses a greedy approach that is repeatedly applied until the right sequence of database edits can be completely determined. Our algorithm for answer addition exploits the fact that typically most relevant data does reside in the database. It greedily splits the query to identify, with the help of the crowd, where data is missing. To highlight the key ingredients of our solution we first consider a simplified setting using a single oracle who always answers correctly, then adapt the algorithm to support scenarios where there may be multiple crowd members who may not always provide correct answers.
4. We have implemented our solution in the QOCO prototype system and applied it on real use cases, demonstrating the efficiency of our query-oriented, oracle-based approach. We performed experimental evaluations on real datasets with both real and simulated crowd, that showed how our algorithms consistently outperformed alternative baseline algorithms, and effectively cleaned the data while asking fewer questions.

Outline of paper. Section 2 contains our preliminaries. We formalize and present our model and basic architecture in Section 3. Sections 4 and 5 examine two special cases of the query-oriented data cleaning problem, where hardness results along with heuristic algorithms are presented. The general algorithms is then presented in Section 6. The implementation of QOCO prototype system, as well as experimental results, are described in Section 7. Related work is in Section 8, and we conclude in Section 9.

2. PRELIMINARIES

Database and Queries We assume a relational schema $\mathbf{S}$ to be a finite set $\{R_1, \ldots, R_m\}$ of relational symbols, each with a fixed arity. A database instance D of $\mathbf{S}$ is a set $\{R_1^D, \ldots, R_m^D\}$, where R_i^D is a finite relation of the same arity as R_i. We use R_i to denote

both the relation symbol and the relation R_i^D that interprets it. We refer to a tuple t of a relation R or a fact $R(t)$ interchangeably.

Let $\mathcal{V}$ be a fixed set of variables and $\mathcal{C}$ be a fixed set of constants called the underlying vocabulary. A query Q over a relational schema $\mathbf{S}$ is an expression of the form

$Ans(\bar{u}_0)$:- $R_1(\bar{u}_1), ..., R_n(\bar{u}_n), E_1, ..., E_m$

where

- $R_1, ..., R_n$ are relation symbols in $\mathbf{S}$, Ans is a relation symbol not in $\mathbf{S}$.
- $\bar{u}_i$ is a vector $(l_1, ..., l_k)$, where $\forall j \in \{1, ..., k\}$, we have $l_j \in \mathcal{V} \cup \mathcal{C}$ and k is the arity of R_i.
- E_i is an expression of the form $l_j \neq l_k$, where $l_j \in \mathcal{V}$ and $l_k \in \mathcal{V} \cup \mathcal{C}$, and l_j (resp. l_k) occurs in some $R_i(\bar{u}_i)$.
- for every $l \in \bar{u}_0$, there exists some $0 < i$ such that $l \in \bar{u}_i$.

In other words, Q is a conjunctive query with inequalities. Ans $(\bar{u}_0)$ is the head of Q, denoted by $head(Q)$. The set $R_1(\bar{u}_1), \ldots,$ $R_n(\bar{u}_n)$, E_1, ..., E_m is the body of Q, denoted by $body(Q)$. We will omit Ans and simply write $(\bar{u}_0)$ for $head(Q)$. The variables and constants in $body(Q)$ are denoted by $Var(Q)$ and $Const(Q)$, respectively. Our results in this paper extend to unions of conjunctive queries with inequalities. However, for simplicity, we will only describe our results for conjunctive queries with inequalities.

EXAMPLE 2.1. *Consider again query Q_1 that finds the European teams that won the World Cup at least twice; i.e.,* (x) :- $\text{Games}(d_1, x, y, Final, u_1), \text{Games}(d_2, x, z, Final, u_2), \text{Teams}(x, EU)$; *$d_1 \neq d_2$. Notice that $Var(Q_1) = \{d_1, d_2, x, y, u_1, u_2\}$ and $Const(Q_1) = \{Final, EU\}$. The result, $Q_1(D)$, contains two answers $\{(GER), (ESP)\}$.*

Assignments and Results An *assignment* $\alpha : Var(Q) \to \mathcal{C}$ for a query Q is a mapping from the variables of Q to constants. An assignment for Q is *valid* w.r.t. database D if for every relational atom $R(\bar{u})$ in *body*(Q), it is the case that $R(\alpha(\bar{u}))$ is a fact in D and for every inequality atom E in *body*(Q), $\alpha(E)$ is true.

A *partial assignment* for Q is an assignment which may not be total. We say a partial assignment α for Q is *satisfiable* w.r.t. D if there is an extension of α to a total assignment α' for Q that is valid w.r.t. D.

Given a (partial) assignment α, the notation $\alpha(head(Q))$ refers to the tuple obtained from $head(Q)$ by replacing each occurrence of a variable l in $head(Q)$ by $\alpha(l)$. We denote by $\alpha(body(Q))$ the set of tuples and inequalities that are obtained by replacing every variable l in *body*(Q) with $\alpha(l)$.

Witness Let α be an assignment for Q that is valid w.r.t. database D. A *witness* for α consists of all facts in $\alpha(body(Q))$.

The set of all valid assignments for a query Q w.r.t. database D is denoted by $A(Q, D)$, and the result of evaluating a query Q on D, denoted by $Q(D)$, is the set $\cup_{\alpha \in A(Q,D)}\ \alpha(head(Q))$. Given a tuple $t \in Q(D)$, we write $A(t, Q, D)$ to denote the set of all valid assignments of Q w.r.t. D that yields t, $\{\alpha \in A(Q, D) \mid t = \alpha(head(Q))\}$. We refer to the witnesses for the assignments in $A(t, Q, D)$ as witnesses for t. Observe that a witness can in fact be extracted from a semiring of polynomials [30]. However, we use the term witness and witness set since we do not require the full generality of a provenance semiring.

Note that each tuple $t \in Q(D)$ induces a unique (partial) assignment that maps the variables in $head(Q)$ to the respective constants of t. With abuse of notation we refer to t also as a partial assignment, which is by definition satisfiable for D.

EXAMPLE 2.2. *Using the same query and notations from Example 2.1, recall that $Q_1(D)$ = {(GER), (ESP)}. Answer t = (GER) has two assignments, {α_1, α_2}, where α_1 = {$x \mapsto GER$, $y, z \mapsto ARG$, $d_1 \mapsto$ 13.7.14, $d_2 \mapsto$ 8.7.90, $u_1 \mapsto$ 1:0, $u_2 \mapsto$ 1:0}, and α_2 is similar except that d_1 and d_2 switches. Answer t uniquely defines the partial assignment $t = \{x \mapsto GER\}$ and t can be extended into all valid assignments in $A(t, Q_1, D)$. The assignment {$x \mapsto GER$, $y \mapsto ARG$, $d_1 = d_2 \mapsto$ 13.7.14, $u_1 \mapsto$ 1:0, $u_2 \mapsto$ 1:0} is invalid only because it does not satisfy the inequality $d_1 \neq d_2$. Partial assignment $\beta = \{x \mapsto ITA, y \mapsto FRA\}$ is non-satisfiable because it cannot be extended into a valid assignment w.r.t. D.*

3. PROBLEM DEFINITION

We now present the formal model and problem statements behind our framework for query-driven interactive data cleaning.

3.1 Model and Problem

Let D be our underlying database. Under the open world assumption, a fact that is in D is true and a fact that is not in D can be true or false. To model real-world data, we adopt the *truly open world assumption* where a fact that is in D can also be true or false, in addition to the assumption that a fact that is not in D can be true or false. In other words, we assume that a given database can contain mistakes, in addition to being incomplete. The truth of a tuple is given by the ground truth database D_G that contains all true tuples and only them. Hence, a database D is *dirty* w.r.t. D_G if $D \neq D_G$. The database D_G determines the *true answers* and *true result* of any query. The two databases D and D_G together determine the set of missing/wrong answers w.r.t. a given query.

DEFINITION 3.1. *Types of answers:*

- *(True Answer, True Result) A tuple t is a* true answer *to a query Q and database D if $t \in Q(D)$ and $t \in Q(D_G)$. We call $Q(D_G)$ the* true result *of Q.*
- *(Missing Answer) A tuple t is a* missing answer *to a query Q and database D if $t \in (Q(D_G) - Q(D))$.*
- *(Wrong Answer) A tuple t is a* wrong answer *to a query Q and database D if $t \in (Q(D) - Q(D_G))$.*

As in interactive systems such as data cleaning systems (e.g., [53, 21]) and mapping-design systems (e.g., [45]), we assume that a (domain expert) member has at least some knowledge about the ground truth. Hence, in our context, the crowd member could point out missing answers or wrong answers. Whenever a crowd member determines that a set of tuples is missing or wrong, the database D will be cleaned, with the help of the crowd, only as much as needed to obtain D' so that the missing answers occur in $Q(D')$ and the wrong answers no longer occur in $Q(D')$.

In our framework, the database D' is achieved through a sequence of updates that are generated through answers to questions posed to the crowd. Each question-and-answer is an *interaction* with the crowd and each update is called an *edit*. An *insertion edit* $R(\bar{a})^+$ inserts the tuple $\bar{a}$ into relation R in the database, while a *deletion edit* $R(\bar{a})^-$ removes the tuple $\bar{a}$ from R. An update to an existing tuple can be modeled by a deletion followed by an insertion. The result of updating D with an edit e is denoted by $D \oplus e$. We assume idempotent edits, that is, if $R(\bar{a})$ exists in D then $D \oplus R(\bar{a})^+ = D$ and the same for deletion, if $R(\bar{a})$ is missing from D then $D \oplus R(\bar{a})^- = D$. The minimization problem that we wish to solve is the following.

PROBLEM 3.2. (EDIT GENERATION PROBLEM) *Given a database instance D, a ground truth database D_G, and a query Q,*

interact with the crowd minimally to derive a sequence $e_1, \ldots, e_k$ of edits such that $Q(D') = Q(D_G)$, where $D' = D \oplus e_1 \oplus \ldots \oplus e_k$.

Observe that we may have $Q(D') = Q(D_G)$ even though $D' \neq D_G$. In other words, the database D' may produce the same query result as $Q(D_G)$ even when D' is still incomplete and/or dirty.

As we shall describe more precisely in subsequent sections under the special cases where a tuple in the output is to be removed or inserted, an interaction can be a boolean question with YES/NO answer. In general, however, the number of edits that is derived may not correspond to the number of interactions with the crowd as multiple edits may be derived with one, or even no questions. Our results show that even in these special cases, Problem 3.2 is intractable. Furthermore, the hardness holds under the assumption that a crowd member has full knowledge of the ground truth DB and every question yields exactly one correct edit on the dirty DB.

3.2 Basic Architecture

As mentioned above, we start by examining two special cases of Problem 3.2. The first case is when there is a single wrong answer. The goal here is to find a list of deletion edits to apply on D so that the resulting database D' will not yield the wrong answer under Q. The second case deals with a single missing answer. The goal here is to find a list of insertion edits so that the resulting database D' will yield the missing answer. We start by considering the following simple architecture (to be extended later).

Target actions There are two possible target actions that a user can specify: (*1*) remove a wrong answer from $Q(D)$ or, (*2*) add a missing answer to $Q(D)$.

Crowd In what follows, we assume there is a single crowd member who is a *perfect oracle*. A perfect oracle always speaks the truth and knows about D_G. Our techniques will be extended to multiple crowd members who may provide incorrect answers in Section 6.2.

Questions to crowd members We first consider a boolean question where we solicit YES or NO answers from the crowd to determine whether or not a tuple in D should be inserted or deleted. In the next sections, we will consider interactions with the crowd members through other types of questions. For now, every question posed to the crowd is of the form $\text{TRUE}(R(\bar{a}))$?, which means "Is $R(\bar{a})$ true?". A YES answer will generate an insertion edit $R(\bar{a})^+$ and a NO answer will generate a deletion edit $R(\bar{a})^-$. The result of updating D according to the answer for question q is denoted by $D \oplus ans(q)$, where $ans(q)$ is the edit that is generated respectively to the answer for q. Note that in general one may infer further edits based on the given answers and we will see some important examples for this later. But let us first assume for simplicity that no such inference is done and thus the length of the generated edit sequence equals to the number of questions asked. Hence, under these assumptions, minimizing the number of asked questions is the same as minimizing the number of edits.

Workflow The workflow that our system, called QOCO, follows can be intuitively described as follows. For now we assume that QOCO starts by receiving a target action on $Q(D_0)$, where D_0 is the initial database instance. QOCO will then generate a question and pose it to the crowd, and generate an edit based on the answer it receives from the crowd. The edit is then applied on D_0 to obtain a new database D_1. The question-answer-edit generation is repeated until QOCO determines that the desired target action can be achieved. The final database D_k, for some $k > 0$, is such that the target action is achieved with $Q(D_k)$.

At this point, a natural question is whether a strategy for generating questions to achieve a desired target action always exists. We first show that every edit on a database D tends to bring D "closer" to D_G. Formally, the *distance between two database instances D and D'*, denoted by $|D - D'|$, is defined to be the size of the symmetric difference between D and D' ($|D - D'| = |D' - D|$). Since the ground truth database is finite, and the perfect oracle provides only correct answers, there exists a naïve strategy that guarantees that the workflow described above always converges for a specific target action, as long as questions are never repeated and values of the domain can be systematically enumerated. The following two propositions can then be easily proved (see Appendix).

PROPOSITION 3.3. *Let e be an edit that is generated based on the oracle's answer to a question. We have $|(D \oplus e) - D_G| \leq |D - D_G|$.*

PROPOSITION 3.4. *Let D_G be a finite database instance that represents the ground truth, D be a database instance, Q be a query, and t be a target action. If the domain is ordered, then there is a finite number of questions $q_1, \ldots, q_k$ s.t. $Q(D \oplus \text{ans}(q_1) \oplus \ldots \oplus \text{ans}(q_k))$ achieves the desired target action.*

A naïve strategy that systematically enumerates all possible tuples in the domain, is of course too expensive to be practical. Hence, a more efficient solution is required. To illustrate the main principle of our algorithm, let us assume first, for simplicity, that there is just one wrong answer in the query result, and then that there is just one missing answer. We next describe our solutions to two subproblems that correspond to the target actions of removing a wrong answer or adding a missing answer. These will form the essence of our solution to the general problem.

4. REMOVING A WRONG ANSWER

The first sub-problem of Problem 3.2 corresponds to the target action of removing a wrong answer (i.e., delete a tuple $t \in Q(D)$ and $t \notin Q(D_G)$). The problem of identifying a set of k corrective updates that need to be performed for removing a wrong tuple from the query result can be formalized as follows.

PROBLEM 4.1. *(Deletion Question Search Problem) Given D, D_G, Q, $t \in (Q(D) - Q(D_G))$, generate at most k questions $q_1, ..., q_k$ of the type $\text{TRUE}(R(\bar{a}))$? s.t. $t \notin Q(D \oplus \text{ans}(q_1) \oplus ... \oplus \text{ans}(q_k))$.*

THEOREM 4.2. *Problem 4.1 is NP-hard.*

We prove the theorem by showing that the corresponding decision problem is NP-hard. For that we use reduction from a well known NP-hard problem, called the Hitting Set Problem [40] (See Appendix). The above result suggests that, in general, heuristics are needed in the design of an efficient algorithm for generating the right questions for removing a wrong answer from the output.

Before presenting our algorithm, we give some intuition for the solution. Recall that we have a wrong answer $t \in (Q(D) - Q(D_G))$, which we wish to delete. This answer is supported by a set of witnesses in D, $\{w_1, w_2, ...w_n\}$, each w_i is the set of facts in $\alpha_i(body(Q))$ where α_i is a valid assignment in $A(t, Q, D)$). Since t is a wrong answer, at least one tuple in each w_i must be false. We will use the crowd oracles to identify these false tuples. The challenge is thus to identify those false tuples with the fewest possible crowd questions. A minimal set of false tuples that covers all the witnesses is a *minimal hitting set*, defined below.

DEFINITION 4.3 (HITTING SET). *Consider the pair (U, S) where U is a universe of elements and S is a set of subsets of U. A set $H \subseteq U$ is a hitting set if H hits every set in S. In particular, $H \cap S' \neq \emptyset$ for every $S' \in S$. A minimal hitting set H, is s.t. $\forall e \in H$. $H \setminus \{e\}$ is not a hitting set.*

Algorithm 1: CrowdRemoveWrongAnswer

Input: A query Q, a database D and a wrong tuple t.
Output: A list of deletion edits.
Init: DeletionEdits = $\emptyset$, $S = wit(A(t, Q, D))$

```
1:  while S ≠ ∅ do
2:      foreach Singleton s = {R'(ā')} in S do
3:          DeletionEdits ← R'(ā')⁻
4:          Remove from S all sets that contain R'(ā')
5:      if S ≠ ∅ then
6:          R(ā) = MostFrequentTuple(S)
7:          if CrowdVerify(R(ā)) then
8:              S = {s \ {R(ā)} | s ∈ S}
9:          else
10:             Remove from S all sets that contain R(ā)
11:             DeletionEdits ← R(ā)⁻
12: return DeletionEdits
```

Consider a minimal hitting set where the universe U is exactly the facts in D and the sets in S are the witnesses for answer t. In general cases, there may exist more than one minimal hitting set for the witness set. Finding those hitting sets that contain only false tuples is not trivial, nor is identifying the smallest one among them (see the proof of Theorem 4.2). However, in certain cases, when there exists just one unique minimal hitting set, things become simpler: if there exist a unique minimal hitting set, all other hitting sets contain this set. Hence, it must be a set of false tuples which is safe to remove and no crowd questions are needed.

EXAMPLE 4.4. *Consider two witnesses $\{t_1\}$ and $\{t_1, t_2\}$ for some false answer. $\{t_1\}$ is a unique minimal hitting set, and thus it must be the case that t_1 is false. In contrast, the witnesses $\{t_1, t_2\}$ and $\{t_1, t_3\}$ have two minimal hitting sets; $\{t_1\}$ and $\{t_2, t_3\}$. Hence, no unique minimal hitting set exists, and to determine which are the false tuples the crowd must be consulted .*

The following theorem proves that it is not hard to identify when a unique minimal hitting set exists, and it shows how to find it. (The proof is given in the Appendix.) Our algorithm will use this technique to reduce the number of crowd questions.

THEOREM 4.5. *Given a pair (U, S), as described in Definition 4.3, a unique minimal hitting set exists if and only if the elements of the singleton sets of S forms a hitting set for S.*

Greedy deletion algorithm Our algorithm employs a greedy heuristic, asking the crowd first about tuples that occurs in the highest number of witnesses. This heuristic could be replaced by others, such as asking the crowd first about influential tuples [38] or, tuples with high causality/responsibility [44], or tuples which are least trustworthy (assuming that they have trust scores).

Out algorithm asks the crowd first about tuples that hit the largest number of witnesses. Intuitively, if a frequent tuple is indeed incorrect, deleting it from the database will eliminate all the witnesses in which it appears at once, whereas if found to be correct, it will provide a negative indication about the other tuples in those witnesses. This heuristic is repeatedly applied until either a unique minimal hitting set exists (and hence false tuples can be determined automatically), or all witnesses has been destroyed. This approach is described in Algorithm 1.

At the beginning of each iteration (lines 2-4), all tuples in the singleton sets are added to the deletion list. Then, all sets in S that contain these singleton tuples are eliminated from S. This routine (lines 2-4) guarantees, according to Theorem 4.5, that our algorithm will not pose questions to the crowd once there exists a unique minimal hitting set. Then, our algorithm greedily searches for tuples that, if found false, can eliminate the largest number of witnesses (lines 5-11). If the most frequent tuple is identified as correct, the algorithm removes it from all sets in S (line 8), otherwise it is added to the deletion list (line 11) and all sets in S containing this tuple are removed (line 10). Finally, the algorithm returns the list of deletion edits.

EXAMPLE 4.6. *To exemplify, consider the same query Q_1 that finds the European teams that won the World Cup at least twice. Assume that our expert crowd member examines the answers for the query and finds the answer (ESP) to be wrong. Note that this answer is supported by six witnesses, $w_1, w_2, \ldots, w_6$, in D.*

	Tuples of the witness
w_1	t_1 = Games(11.7.10, $ESP, NED, Final$, 1:0) t_2 = Games(12.7.98, $ESP, NED, Final$, 4:2) t_3 = Teams(ESP, EU)
w_2	t_2 = Games(12.7.98, $ESP, NED, Final$, 4:2) t_4 = Games(11.7.94, $ESP, NED, Final$, 3:1) t_3 = Teams(ESP, EU)
w_3	t_4 = Games(11.7.94, $ESP, NED, Final$, 3:1) t_1 = Games(11.7.10, $ESP, NED, Final$, 1:0) t_3 = Teams(ESP, EU)
w_4	t_1 = Games(11.7.10, $ESP, NED, Final$, 1:0) t_5 = Games(25.06.78, $ESP, NED, Final$, 1:0) t_3 = Teams(ESP, EU)
w_5	t_2 = Games(12.7.98, $ESP, NED, Final$, 4:2) t_5 = Games(25.06.78, $ESP, NED, Final$, 1:0) t_3 = Teams(ESP, EU)
w_6	t_4 = Games(11.7.94, $ESP, NED, Final$, 3:1) t_5 = Games(25.06.78, $ESP, NED, Final$, 1:0) t_3 = Teams(ESP, EU)

No singletons exist in the witness set and hence we jump to lines 6-11. Tuple t_3 is most frequent, appearing in all witnesses, and so the algorithm poses the question TRUE(t_3)? *to the crowd (line 7). Since t_3 is correct ($t_3 \in D_G$), the expert answers YES. True tuple t_3 = Teams(ESP, EU) is removed from each set (line 8). The remaining tuples in the witness set are now, respectively,*

{t_1, t_2}, {t_2, t_4}, {t_4, t_1}, {t_1, t_5}, {t_2, t_5}, {t_4, t_5}

As all tuples occur equally often in the witnesses, QOCO *will choose randomly between them. Suppose* QOCO *first poses the question* TRUE(t_5)? *(line 7). Since t_5 is a false tuple ($t_5 \notin D_G$), the expert answers NO. Consequently, all witnesses that contain tuple t_5 are removed from the witness set, and t_5 is added to the deletion list (lines 10-11). The remaining witnesses and candidate tuples are now, respectively,*

{t_1, t_2}, {t_2, t_4}, {t_4, t_1}

All tuples are again equally frequent, so suppose that QOCO *now chooses to pose the question* TRUE(t_1)? *(line 7), and hence our expert answers YES. At this point, the sets in S are reduced to:*

{t_2}, {t_2, t_4}, {t_4}

and there exists a unique minimal hitting set {t_2, t_4}. The remaining deletions can be automatically determined, without posing further questions to the crowd. Indeed, QOCO *adds the tuples in the singletons to the deletion list and removes all sets that contain these tuples (lines 2-4). This step eliminates all remaining sets, and the algorithm terminates and the deletion list of false tuples is returned.*

The list of deletion edits consists of false tuples that form a hitting set for the set of witnesses. In this example it is also a minimal hitting set, but generally, due to the greedy nature of the algorithm, this is not guaranteed. Note however that even when the set is not minimal the extra work is not wasted - the elimination of the redundant tuples does improve the correctness of the database, even

if not essential to the removal of the given wrong answer. More generally, observe that the set of questions being asked may not be minimal. For example, we asked the expert here about tuple t_3, that turned out to be correct and thus not added to the deletion list. Any algorithm that can be devised for this problem would confront the same issue (i.e., that a tuple picked by the algorithm as a candidate for deletion may be verified by the crowd to be correct), as the optimal strategy depends on D_G which is unknown.

5. ADDING A MISSING ANSWER

The second sub-problem of Problem 3.2 corresponds to the target action of inserting a missing answer to the output. Analogously, we formalize the problem of identifying a set of k corrective updates to be performed for adding a missing tuple to a query result as follows.

PROBLEM 5.1. *(Insertion Question Search Problem) Given D, D_G, Q, $t \in (Q(D_G) - Q(D))$, generate at most k questions $q_1, ..., q_k$ of the type* TRUE$(R(\bar{a}))$? *s.t.* $t \in Q(D \oplus \text{ans}(q_1) \oplus ... \oplus \text{ans}(q_k))$.

THEOREM 5.2. *Problem 5.1 NP-hard.*

To prove this we use reduction from another known NP-hard problem named One-3SAT [40], see Appendix. Similarly to the case of removing a wrong answer, the above result suggests that, even if we had access to the ground truth database D_G, heuristics are needed in the design of an efficient algorithm for generating the right questions to insert the relevant tuples to the database. To address this we will leverage the crowd's knowledge of D_G and ask them more general questions than the simple boolean ones we have used so far. We approach crowd oracles to directly identify some of the true tuples that should be inserted to the database to add a missing answer to the output.

A naïve approach could be asking an oracle to provide the facts in a witness of the missing answer. Recall that given a missing answer t we abuse the notation and t is also the induced partial assignment which maps the variables in $head(Q)$ to t. So, simply presenting to the crowd member $t(body(Q))$, and asking her to complete it into a witness for t, seems a possible solution. This way, the facts in this new witness can be inserted to D, and t becomes an answer in the output $Q(D)$.

For that, we define another crowd question[2], additional to the crowd question defined in Section 3.2. Given a partial assignment α for query Q, QOCO may ask the crowd COMPL(α, Q), which means: if α is satisfiable, complete $\alpha(body(Q))$ into a witness through a total valid assignment α' that extends α. Otherwise, do nothing. If an extended assignment α' is found, QOCO will interpret the facts in $\alpha'(body(Q))$ that are not in D as insertion candidates, and QOCO will create the compatible edit list of insertions.

In general, given a missing answer t, the number of tuples in $t(body(Q))$ which we ask the crowd member to complete (i.e., map the variables in those tuples), may be large. Note however that if the underlying database D is nearly clean and complete, most tuples in $t(body(Q))$ may exist in D, and hence only minor changes and small amount of mappings are in fact needed so that t will appear in the output. Hence, the crowd may be asked to do more work than needed in terms of the number of new tuples generated in the task. To avoid this we pursue the following approach.

5.1 Query split

It is well-known that crowdsourcing works best when tasks can be broken down into simpler pieces. An entire witness generation may be a too large task for the crowd. Hence, we suggest an approach that exploits both the crowd and the underlying database D which is likely to work well if the underlying database is mostly correct and complete w.r.t. the query Q. The goal is to help the crowd members by directing them with facts existing in the underlying database D. For that we present the notion of *splitting a query*. Before continuing we define a subquery for a given query.

DEFINITION 5.3 (SUBQUERY). *Let Q' be the following query $ans(u'_0)$:- $R'_1(u_1), ..., R'_k(u_k), E'_1, ..., E'_f$ and Q be the query $ans(u_0)$:- $R_1(u_1), ..., R_n(u_n), E_1, ..., E_m$. We say that Q' is a* subquery *of Q, denoted by $Q' \leq Q$, if the following hold.*

- $R'_1(u_1), ..., R'_k(u_k) \subseteq R_1(u_1), ..., R_n(u_n)$
- $E'_1, ..., E'_f \subseteq E_1, ..., E_m$

Splitting a query means decomposing the query into two or more subqueries where every relational atom in the body of the original query appears in the body of at least one subquery. We assume that the head of each subquery contains all the variables that appears in its body (i.e., no projection).

The intuition behind splitting a query into subqueries is to seek partial assignments for Q using assignments of its subqueries. These partial assignments are good candidates for being extended into a valid assignment (if satisfiable), and can be used to reduce the number of tuples we ask the crowd to complete. We next illustrate with an example, how splitting queries can be used to potentially reduce the required crowd work.

Before that, we define how we embed a given missing answer into the query Q, and thus obtain a new query. Given a query Q and a missing answer t, we denote by $Q|_t$ the query whose body is $t(body(Q))$ and its head consists of all the variables that appear in $t(body(Q))$. Our goal is to complete $t(body(Q))$, which amounts to finding a valid assignment for Q

EXAMPLE 5.4. *Consider the query Q_2, that finds all European players who scored a goal in a World Cup final game; formally it is* (x) :- Players(x, y, z, w), Goals(x, d), Games$(d, y, v, Final, u)$, Teams(y, EU). *Notice that tuple (ITA, EU) is missing from D (but is in D_G), and hence all Italian players are missing from the output. For our discussion we look at missing answer t = $(Pirlo)$ which defines the partial assignment $\{x \mapsto Pirlo\}$. Consider query $Q_2|_t$ that is* (z, w, d, v, u) :- Players$(Pirlo, y, z, w)$, Goals$(Pirlo, d)$, Games$(d, y, v, Final, u)$, Teams(y, EU). *We split $Q_2|_t$ into two subqueries*

Subquery	*Head*	*Body*
Q'	(y, z, w, d, u, v)	Players$(Pirlo, y, z, w)$ Goals$(Pirlo, d)$ Games$(d, y, v, Final, u)$
Q''	(y)	Teams(y, EU)

Notice there is one valid assignment for Q' w.r.t. D:
$\alpha_1 = \{y = w \mapsto \text{ITA}, z \mapsto 1979, d \mapsto 9.6.06, v \mapsto \text{FRA}, u \mapsto 5{:}3\}$, *and 3 valid assignment for Q'' w.r.t. D:*
$\alpha_2 = \{y \mapsto \text{GER}\}$, $\alpha_3 = \{y \mapsto \text{ESP}\}$, $\alpha_4 = \{y \mapsto \text{BRA}\}$.
Evaluating $body(Q_2|_t)$ under assignment α_1, induces four tuples:
Players$(Pirlo, ITA, 1979, ITA)$, Goals$(Pirlo, 9.6.06)$
Games$(9.6.06, ITA, FRA, Final, 5:3)$, Teams$(ITA, EU)$.
Notice that α_1 is a total assignment for $Q_2|_t$ (and not just for Q'). QOCO *system presents to the crowd members $\alpha_1(body(Q_2|_t))$, and the crowd members affirm that α_1 is a valid assignment w.r.t. D_G. Hence,* QOCO *concludes that the tuples in $\alpha_1(body(Q_2|_t))$ need to exist in D to make $(Pirlo)$ an answer. On the other hand, $\alpha_2, \alpha_3, \alpha_4$ are non satisfiable partial assignments for $Q_2|_t$ w.r.t. D. Hence, when* QOCO *system presents to the crowd $\alpha_i(body(Q_2|_t))$*

[2]Although strictly speaking, this is not a question but a task.

for $i = 1, 2, 3$, they answer that these assignments are non-satisfiable. This is due to tuples such as $\text{Players}(Pirlo, GER, z, w)$ *which occurs in* $\alpha_2(body(Q_2|_t))$ *or* $\text{Players}(Pirlo, ESP, z, w)$ *which occurs in* $\alpha_3(body(Q_2|_t))$*, that cannot be completed into a fact of* D_G*.*

If we consider the naïve approach a user would have to complete all 4 tuples in query Q_2 through binding 6 variables. On the other hand, Example 5.4 showed that the crowd task can be reduced to a question whether a given assignment is valid or satisfiable, and ask for its completion, if possible. Consequently, in this example, the tuples in $\alpha_1(body(Q|_t))$ are verified as true tuples, and QOCO can automatically conclude that $\text{Teams}(ITA, EU)$ should be inserted to D in order to add $(Pirlo)$ to the output $Q_2(D)$.

In this example we showed how a specific query split helped to reduce the amount of work done by the crowd and helped them to provide a witness for the missing answer. However, it can be tricky in general to decide how to split the query, and how to evaluate the subqueries to obtain a partial satisfiable assignment. Ideally, we should consider all subqueries of the query, but the number of subqueries is exponential, and so the amount of work required from the crowd is infeasible. We propose a greedy approach for splitting the query, which splits the query into two subqueries (see Subsection 5.2), evaluate each one of them and present to the crowd their partial assignments. The crowd is asked to verify if the partial assignment is also a valid total assignment, or complete the assignment if possible. We continue to split those subqueries recursively until a witness for the missing answer is found. If a valid assignment is not found through this process, we fall back to the naïve approach that asks the crowd to provide all tuples in the witness.

The assignments for subqueries of $Q|_t$ w.r.t. D can result in one of the cases: (*Case 1*) *A total valid assignment* (e.g., α_1 Example 5.4) or *a total invalid assignment*. In this case, the crowd needs to answer a simple true and false questions whether it is valid or not w.r.t. $Q|_t$ and D_G. (*Case 2*) *A partial unsatisfiable assignment* (e.g., α_2 Example 5.4). Such assignments can never be completed into one that "fits" D_G and hence, effectively useless. (*Case 3*) *A partial satisfiable assignment.* In this case, the crowd will be asked to complete the partial assignment into a valid assignment.

As we will show in Section 7, under our assumptions that D is mostly clean and complete, this heuristic is effective in helping the crowd to provide a valid assignment for the missing answer.

Greedy insertion algorithm We are now ready to describe our algorithm that uses the crowd as oracles to access D_G for the goal of adding a missing answer to the output.

Algorithm 2 uses two helper methods. CrowdVerify(X) is a crowdsourcing function that takes as input a set of tuples and inequalities. It verifies against crowd members whether facts (i.e., tuples with no variables) are true or false. It returns false if at least one of the tuples in the input is false, or if one of the inequalities is incorrect. CrowdComplete(α, Q) is another crowdsourcing function that takes as input a query Q and a partial assignment α for Q w.r.t. D. This method asks the crowd $\text{COMPL}(\alpha, Q)$. If α is satisfiable w.r.t. Q and D_G, the crowd member completes $\alpha(body(Q))$ to a valid assignment α', and then, CrowdComplete() method returns the corresponding list of insertion edits according to the witness $\alpha'(body(Q))$. Otherwise, it returns null.

The algorithm starts by splitting the input query and adding the subqueries to a queue (line 3). The main loop continues until a valid assignment for the missing answer is found, or if the subqueries queue is empty (line 4). At each iteration we pop a subquery from the queue (line 5). We evaluate the current subquery (line 6) and verify against the crowd whether it is a valid total assignment for the input query Q (lines 8-10) or a partial assignment that should

Algorithm 2: CrowdAddMissingAnswer

```
   Input: A query Q, a database D and a missing tuple t.
   Output: void
   Init: Queries = ∅, InsertionActions = ∅
 1: TrueTuples = {R(ā)^+ | R(ā) ∈ body(Q|_t) ∧ ā consists of
   only constants}
 2: D = D ⊕ TrueTuples
 3: Queries ← Split(Q|_t)
 4: while Q|_t(D) = ∅ && Queries ≠ ∅ do
 5:     CurrQ = pop(Queries)
 6:     foreach Assignment α in A(CurrQ, D) do
 7:         if CrowdVerify(α(body(Q|_t))) then
 8:             if α is a total assignment of Q|_t then
 9:                 D ⊕ {R(ā)^+ | R(ā) ∈ α(body(Q|_t))}
10:                 return
11:             else
12:                 InsertionActions = CrowdComplete(α, Q|_t)
13:                 if InsertionActions ≠ ∅ then
14:                     D ⊕ InsertionActions
15:                     return
16:     if body(CurrQ) has more than 1 tuple then
17:         Queries ← Split(CurrQ)
18: InsertionActions = CrowdComplete(Q|_t, id)
19: D ⊕ InsertionActions
20: return
```

be completed with the crowd (lines 12-15). If not, we split the current query (lines 16-17) and continue to the next iteration. If the algorithm fails to find a partial assignment that can be extended into a valid assignment for Q it posts to the crowd a question to provide a witness for the missing answer (line 18). In line 19 the algorithm executes the insertions of the true missing tuples.

5.2 Implementations of Split()

The Split() method, which appears in lines 3 and 17, is the heart of our algorithm. This is the heuristic discussed before, which breaks the input query into two subqueries.

Split() method can be implemented in different ways and we next describe two approaches that we examined and experimented on.

Data-directed approach This approach exploits provenance metadata [12], when available, to split the query. If we have database provenance for the query result, we can instrument methods similar to the WhyNot? system in [58]. In that work, they try to provide explanations to missing answers by identifying the manipulation operation(s) in the query plan that are responsible for excluding the missing answers. As opposed to WhyNot? we are not interested in explaining why an answer is missing, or describing the possible ways in which it potentially could be added but we wish to identify the *correct* edits to the underlying database that will add the answer to the result. Nevertheless, we can still exploit the output of WhyNot? system to wisely split a query. Our input to WhyNot? system is a query with no projection and no answers (i.e., $Q|_t$ or one of its subqueries), and we ask "Why no answers?". When we get the manipulation operation(s) that are responsible for excluding the missing answer, we split accordingly. We omit here the details for lack of space, but illustrate in Figure 2 (right) such a split. In this example the WhyNot? mechanism outputs a join operation, and QOCO splits the query atoms accordingly, further adding to each subquery all the inequality involving its variables.

Query-directed approach In the absence of provenance information that points to the missing data items, one can try to use the

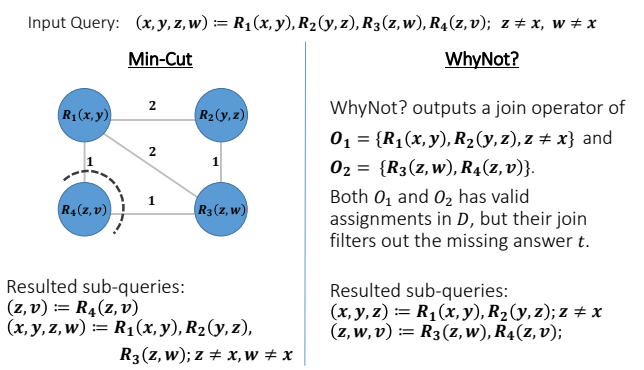


Figure 2: Example of split using different methods

structure of the query to guide the split. For example consider the query as a weighted graph, described shortly. We may use this graph to split the query in a manner that will produce two subqueries that their graphs are connected (up to certain limitations). This way we are more likely to avoid situations where the same variable appears in both subqueries, and avoid the loss of inequalities (as happens in the WhyNot?-based split in Figure 2 (right), where the inequality $w \neq x$ does not appear in the subqueries because variables w and x are divided into different subqueries).

In our graph, vertices represent tuples in the query's body and edges occur between vertices that represent tuples with joint variables, or tuples with variables that share an inequality. Moreover, the edges of the graph are weighted with the number of variables that appear in both relations represented by its adjacent nodes, plus the number of inequalities that are relevant to the variables of those same two nodes. On this graph we can look for a Min-Cut [18] that will define how to split the query. Figure 2 (left) illustrates a split of the query after finding a Min-Cut of the query graph.

As one can see in Figure 2, different split methods may result with different subqueries. In the provenance approach we are more likely to obtain subqueries that their evaluated results on the database are not empty. In this example (right), we assumed a join operation was returned from WhyNot? (recall that we ask "Why no answers?" for $Q|_t$), which means that both subqueries has results when evaluated on the database. In the query-directed split we are more likely to have a small number of variables that appear in both subqueries and a larger number of inequalities that are captured within the subqueries. In the same example (left), the inequality $w \neq x$ appears in the second subquery, while it is meaningless for both subqueries in the other split. This way, we can satisfy more constraints in each subquery. One could also apply heuristic to keep relational atoms with key-foreign key relationship together.

Interestingly, as shown in the experiments, since the ideal split depends in practice on what data is missing, rather than on the query structure, sophisticated structure-based analysis as the one above often does not perform better than a simple random split.

6. THE GENERAL ALGORITHM

The principles described above extend naturally to handle multiple wrong/missing answers and multiple imperfect experts. We detail these two extensions next.

6.1 Iterative Cleaning

In the general case our systems first needs to identify the set of wrong answers (resp., multiple missing answers). Then it can continue to process the actions of deleting (resp., inserting) an answer

Algorithm 3: Main Algorithm

Input: A query Q, and an underlying database D
Output: A clean and complete database D w.r.t. Q and D_G
Init: $VerifiedResults = \emptyset$, FirstIter = true

```
1:  while FirstIter || Q(D) \ VerifiedResults ≠ ∅ do
Del 2:  foreach Tuple t in Q(D) \ VerifiedResults do
3:      if CrowdVerify(Q(D), t) then
4:          VerifiedResults ← t
5:      else
6:          D ⊕ CrowdRemoveWrongAnswer(Q, D, t)
Ins 7:  foreach Tuple t in CrowdComplete(Q(D)) do
8:      CrowdAddMissingAnswer(Q, D, t)
9:      VerifiedResults ← t
10:     FirstIter = false
11: return
```

using the suggested solutions of Problems 4.1 and 5.1. For that we define two more crowd questions:

- TRUE(Q,t)?: Is the result tuple $t \in Q(D_G)$?
- COMPL$(Q(D))$: Complete $Q(D)$ into $Q(D_G)$.

A perfect oracle will answer YES to a question TRUE(Q,t)? if and only if $t \in Q(D_G)$. An answer to a question COMPL$(Q(t))$ is a missing answer $t \in Q(D_G)$, or null if $Q(D) \subseteq Q(D_G)$.

Note however that when both types of errors exist, fixing one type (e.g., a wrong answer) may lead to the occurrence of new errors of the second type (e.g., missing answers), and vice versa. For example, the deletion of false tuples that were the cause of an incorrect answer in the output, may cause the deletion of correct answers (that were previously there due some false tuples). Adding correct witnesses for these answers may in turn generate other incorrect answers (due to newly formed incorrect witness sets), and so on.

EXAMPLE 6.1. *Consider again query Q_2 in Example 5.4, and the missing answer $(Pirlo) \notin Q_2(D)$. QOCO needs to execute the insertion edit of $\{\text{Teams}(ITA, EU)^+\}$ on database D to add $(Pirlo)$ to $Q_2(D)$. Note that D contains the false tuple $\text{Goals}(Totti, 9.6.06)$. Thus, if we add true tuple $\text{Teams}(ITA, EU)$, the wrong answer $(Totti)$ will be added to the output of Q_2, as a side effect.*

A key observation which goes back to Proposition 3.3, is that each step in this tuple addition/deleting sequence brings the database closer to the ground truth database, and thus our algorithm, that iteratively handles these newly generated wrong/missing answers, is guaranteed to converge to the correct query result.

The algorithm enters the outer loop (line 1) in one of two cases; (*1*) during the first iteration, to cover the case when $Q(D)$ is empty but $Q(D_G)$ is not. (*2*) when $Q(D) \setminus$ VerifiedResults $\neq \emptyset$, which means that there are unverified answers in $Q(D)$ that must be verified against the crowd. This algorithm is iterative. It first handles wrong answers in the deletion part (lines 2-6). It identifies incorrect tuples in $Q(D)$ by asking TRUE(Q,t)? (line 3) and then calls CrowdRemoveWrongAnswer from Section 4 (line 6) to execute the deletion algorithm on wrong answers. Afterwards, it continues to the insertion part (lines 7-9). It uses the crowd to find tuples that should be added to $Q(D)$ using the method CrowdComplete$(Q(D))$ (line 7) that poses questions of the type COMPL$(Q(D))$ to the crowd. Then it calls CrowdAddMissingAnswer from Section 5 (line 8) to execute the needed insertion edits.

The helper method CrowdComplete$(Q(D))$ (line 7), that poses questions of the type COMPL$(Q(D))$, needs to know when to stop

posting these questions (i.e., when $Q(D)$ is complete). In [59] the authors developed statistical tools to enable developers to reason about query completeness. We use their technique as a black-box, called enumeration black-box, to decide when the query result is complete. This black-box notifies QOCO once posing additional crowd questions asking to add missing answers is no longer necessary, because the query result is complete with high probability.

6.2 Multiple Imperfect Experts

For simplicity, up to this point, we assumed a single perfect oracle. We next extend our framework to support multiple crowd (imperfect) experts working in parallel. This extension has two different aspects; parallelism, and dealing with the fact that humans, even if experts, are imperfect and may make mistakes.

Imperfect experts Recall that QOCO system has four types of questions. Two boolean questions that verify tuples and answers, and two open questions (tasks) that ask to complete partial assignments of queries to complete the result sets of queries. To deal with potential errors in boolean questions, we use below a simple estimation method where each question is posed to a fixed-size sample of the crowd members and the answers are averaged. More generally one could use any black-box (e.g., of [2, 47]) to determine the number of users to be asked and how to aggregate their answers (e.g., using majority vote). Regarding open questions, once a single expert provides an answer, the system poses an additional set of boolean questions to verify that the obtained answer is correct (using the black-box aggregator). More precisely, if tuple t is an answer to $\text{COMPL}(Q(D))$, the system will ask several experts the closed question $\text{TRUE}(Q, t)$?. If a set of tuples S is the answer to some question $\text{COMPL}(\alpha, Q)$, the system poses the question $\text{TRUE}(R(\bar{a}))$? for each tuple $R(\bar{a}) \in S$. Note that the iterative nature of our algorithm provides further protection against wrong insterions/deletions: If some wrong (resp. correct) tuple was mistakenly inserted (deleted), it may be removed (added) in the next iterations, if it caused for a new wrong/missing answer.

Parallelism Recall that Algorithm 3 consists of two components: deletion (lines 2-6), and insertion (lines 7-9). It also has the outer loop (lines 1-10) that iteratively runs deletion and insertion components over and over again until termination. We would like to be able to maximize the use of all available crowd members at any point, to speed up the computation. Thus, we run the deletion and insertion parts in parallel. To allow that, we need a dedicated variable $Res_{complete}$ to hold the set of tuples in $Q(D)$ at the time when the insertion loop is done according to enumeration black-box (line 7). We use this variable to add another condition $Q(D) \neq Res_{complete}$ to the outer loop (line 7). It is necessary because the deletion part, which is now executed simultaneously with the insertion part, might delete correct answers. We further use parallel foreach loops, in both deletion and insertion components. We verify the correctness of all tuples in $Q(D)$ at the same time (line 3), or post together multiple completion questions (line 7). For a summarized list of the modifications to Algorithms 1, 2 and 3, previously discussed, see Appendix.

7. IMPLEMENTATION

We have implemented all the techniques described in the previous sections in QOCO. QOCO is implemented in PHP (back-end), JavaScript (front-end) and uses MySQL as the database engine. The system architecture is detailed in the Appendix.

Crowd members in popular crowdsourcing platforms such as Amazon Mechanical Turk or CrowdFlower are not always experts for the domains we use in our experiments. These platforms also do not allow to dynamically compute the questions to the crowd based on previously collected answers. We have thus implemented our own system and recruited our crowd through the relevant social networks to ensure that they have the necessary expertise for judging the truthfulness and completeness of the query results.

Our experiments are based on two different real-world datasets. A smaller database used as a showcase for the usefulness of our approach, and a larger dataset for comparing the performance to alternative baseline algorithms w.r.t. varying parameters. In our experiments we measured the efficiency of the algorithms in terms of number of questions posed the crowd. We also measured the running time required to select to next question. For our datasets this was always not more than one or two seconds and negligible for the user interaction, and we thus omit the exact measures here.

7.1 DBGroup database

The first database is our real-life DB group database (DBGroup), recording information on our group members, their research activities, publications, academic events, achievements etc. This dataset was created about 10 years ago and continuously maintained by various group members since, and currently contains around 2000 tuples. The DBGroup database is used to generate reports about the group's research projects and achievements for various purposes, e.g., to be included in periodic grant reports. Thus, each item in the database has attributes that relate it to relevant grants and topics.

To showcase how QOCO can be used to effectively clean the database we have launched it with queries used to generate past reports, and were (positively) surprised by the results. Even though we expected the database, being maintained and curated by past and present members and extensively used for reports, to be correct, QOCO helped to discover several unnoticed mistakes.

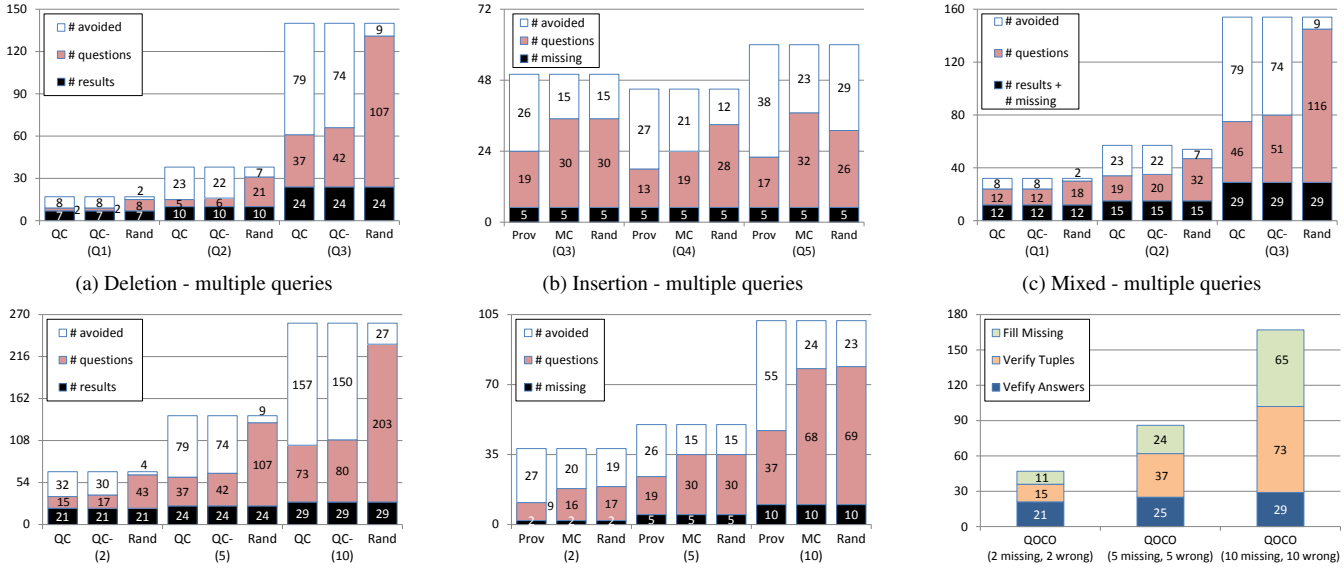
To illustrate we present four queries from the last grant report.

- *Q1:* Find all keynotes and tutorials on topics related to ERC.
- *Q2:* Find all current group members financed by ERC.
- *Q3:* Find all students who participated in conferences in the past 30 months, where the travel was sponsored by ERC.
- *Q4:* Find all publications with the topic "crowdsourcing" published in the last 30 months.

With QOCO, we ran the above queries and our group members played the role of crowd experts. The black-box used for a decision mechanism was simple: three users were required for determining the correctness of an answer and the majority vote was taken. The whole experiment for this report took less then one hour from the posting of the call for participation in the group's social network. During the process, we discovered 5 wrong answers (1 wrong keynote and 4 wrong group members) and 7 missing answers (1 missing keynote, 1 missing member, and 5 missing conferences). Consequently, QOCO cleaned the DBGroup database and removed 6 wrong tuples and added 8 missing tuples, which we have later manually verified to be all indeed correct edits.

7.2 Soccer database

The second database we experimented on is about Soccer games, including in particular the World Cup games. The Soccer database contains data about the games, goals, players, teams (national), clubs, etc. and consists of around 5000 tuples. This is a real-life database derived using automatic website scraping tools from sites [63, 64]. We first cleaned the Soccer database by comparing the data with reference data from FIFA official data [24] and used this as our ground truth, then added to the dataset some controlled noise (to be explained in Section 7.2). Our experiments were carried out with 3 different notions of crowd; (*1*) a simulated perfect



(a) Deletion - multiple queries (b) Insertion - multiple queries (c) Mixed - multiple queries

(d) Deletion - varying # of wrong answers ($Q3$) (e) Insertion - varying # of missing answers ($Q3$) (f) Mixed - Types of questions ($Q3$)

Figure 3: Results with a simulated oracle. Notation: QOCO (QC), QOCO$^-$ (QC-), Random (Rand), Provenance (Prov), Min-Cut (MC).

oracle, namely an implemented oracle that consults with the ground truth Soccer database, (*2*) a real person who is a perfect expert; in fact, to make sure the results that we obtain are consistent, we repeated the experiments with three distinct people whom we know to be true Soccer experts, and (*3*) real crowd of imperfect experts consisting of soccer fans. Surprisingly, in all our experiments the perfect experts provided the exact same results as the simulated perfect oracle. First, the Soccer games database is at a reasonable scale and it was not difficult to find ardent fans who are extremely knowledgeable about the World Cup and soccer games in general. Second, these fans are usually competitive and will do all it takes (e.g., searching the answers on Google) for being the "master" of FIFA. As we shall explain, our results with imperfect experts are somewhat different.

The parameters we considered in our experiments are below.

Degree of data cleanliness refers to the ratio of number of true tuples in the dataset (i.e., $|D \cap D_G|$) to the total number of tuples in the dataset plus the tuples missing from the dataset (namely $|D| + |D_G - D|$). For example, if the *data cleanliness* is 50%, then the number of true tuples in the dataset is exactly the same as the total number of false and missing tuples. To simulate a dirty database which contains wrong and missing tuples, we add false tuples and remove true tuples to the cleaned Soccer data. We vary the cleanliness of our datasets from 60% to 95%. The default value is 80%.

Noise skewness refers to the ratio of the number of false tuples in the dataset (i.e., $|D - D_G|$) to the number of these false tuples plus the number of the missing true tuples (namely $|D - D_G| + |D_G - D|$)) . We vary it from 100% where we have only false tuples and no true missing tuples, through 50% where the number of false and missing true tuples are equal, to 0% where we have only missing tuples and no false tuples. For our experiments with the deletion algorithm (Algorithm 1), the default value is 100% and for experiments with our insertion algorithm (Algorithm 2), the default value is 0%. For experiments in the general case (Algorithm 3), the default value is 50%.

Degree of result cleanliness is similar to the *Data Cleanliness* parameter, but considers the cleanliness of the *query result*, and thus refers to the ratio between $|Q(D) \cap Q(D_G)|$ and $|Q(D)| + |Q(D_G) - Q(D)|$.

We illustrate our findings with the following five representative queries that are inspired by World Cup trivia quizzes from various websites e.g., [26, 25]. These queries have varying result sizes, from the smallest to largest:

- *Q1:* Find all European teams who lost at least two finals.
- *Q2:* Find all teams from the same continent that played at least twice against each other.
- *Q3:* Find all non-Asian teams that reached the World Cup knockout phase and won at least once.
- *Q4:* Find all teams that lost two games with the same score.
- *Q5:* Find all teams that won at least two games, while one of the opponents was a South American team.

Next, we describe the alternative baseline algorithms that we compared to our solution. Specifically, we compared the number of different crowd actions (i.e., boolean questions and open questions) posed by our solution (Algorithms 1, 2, and 3) to that posed by the competing algorithms running on the same input.

Deletion baseline algorithms For the case of deletions, we have two baseline algorithms (*Random* and QOCO$^-$) that decide which tuples among the witnesses to verify against the crowd. In our experimental setup, QOCO executes in a loop and iteratively asks the crowd whether a given answer in $Q(D)$ is correct. If the crowd deems that an answer in $Q(D)$ is incorrect, then one of these baseline algorithms is used to determine which tuples in the witnesses of the wrong tuple should be verified against the crowd.

- *Random* - a naïve algorithm that randomly picks a tuple, among the tuples in the witnesses of the wrong answer, to verify next.
- QOCO$^-$ - a simplified version of our deletion algorithm that greedily picks the most frequent tuple among the tuples in the witnesses of the wrong answer, but does not identify when a unique minimal hitting set exists. Consequently, it continues posing further questions to verify the remaining tuples.

Insertion baseline algorithms Recall that the core of Algorithm 2 is the Split() method. Hence, we would like to study the effect of using different methods for splitting a query on the performance of Algorithm 2. We consider the following alternatives.

- *Naïve* - the naïve approach does not split the query.
- *Random* - randomly splits the given query into two subqueries.
- *Min-Cut* - splits the given query according to structure-based process presented in Subsection 5.2.
- *Provenance* - uses data provenance and the WhyNot? algorithm from [58], as discussed in Subsection 5.2.

Results for Perfect Oracle. Figure 3 shows the results of simulated perfect oracle (and thus also for a real perfect expert), for insertion, deletion, and the mixed general case algorithms with varying parameters. We refer to the version of Algorithm 3, that uses a combination of our deletion algorithm (i.e., Algorithms 1) with the Provenance-based insertion algorithm (i.e., Algorithms 2), as the "Mixed" algorithm. Each graph is the result of an experiment for an algorithm with a change in one of the parameters (mentioned in parenthesis), while the remaining parameters are assigned default values. For graphs 3a,...,3e the bottom part of each bar (in black) represents the lower bound count, i.e., the number of query answers that must be verified - for deletion algorithms , the number of missing answers - for insertion algorithms, and respectively, the number of query answers and missing answers for the mixed case. The middle part of each bar (in red) represents the actual number of verification questions (i.e., deletion questions) or the number of filled variables (i.e., insertion questions) or their sum when relevant. The top portion of each bar (in white) represents the number of questions saved or avoided relative to the upper bound. For example, for deletion, the total number of questions that one would ask with the naïve algorithm corresponds to the number of distinct tuples in the witness set of the answer that is to be deleted. Hence the total is always constant for a given query. Figure 3a shows that the total number of possible questions is 17 for Q1, 38 for Q2, and 140 for Q3. For insertion, the total number of questions is what would be asked by the naïve algorithm that does not split the query (that is, the highest number of unique variables that the expert needs to provide, in the worst case). We next detail the results.

Deletion algorithms. In all our experiments, QOCO had better performance results than its competitors. We present only a few representative graphs in the figure. We show only the graphs of queries Q1, Q2, and Q3 since the trends in queries Q4 and Q5 are similar. Figure 3a shows how the performance varies across queries Q1, Q2, and Q3. The difference between QOCO and QOCO$^-$ due QOCO's ability to identify unique hitting sets becomes more apparent as the size of the query grows (i.e., both number of results and number of tuples). In all cases, the two perform better than the Random algorithm that verifies all tuples of all witnesses. Figure 3d demonstrates how different levels of noise (i.e., varying degree of data cleanliness and result cleanliness) affects the performance. The numbers mentioned at the bottom of the graph, i.e., (2), (5), (10) represent the number of wrong answers among the answers in the result $Q(D)$. The gap between the performance of QOCO and the Random algorithm increases with the noise level.

Insertion algorithms. In all our experiments the split-based algorithms performed better than the Naive (the upper bound in the graphs), and the provenance-based split performed best. Interestingly however, there was no clear winner between the Min-Cut approach and the Random split. To illustrate Figure 3b shows the results for queries Q3, Q4 and Q5 (Queries Q1 and Q2 show similar trends to Q3 and are thus omitted). As we can see, the provenance based algorithm always performs best, whereas for Min-Cut and Random, in Q3 the two perform the same, in Q4 Min-Cut is better than Random, and in Q5 the opposite holds. This confirms our intuition that the ideal split depends in practice on what data is missing, rather than on the structure of the query.

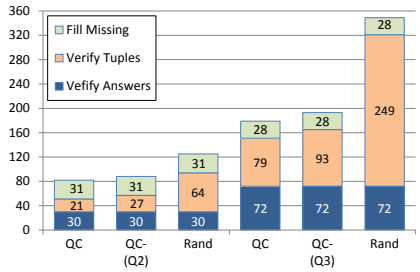


Figure 4: Experimental results - real experts ($Q2$ and $Q3$)

Figure 3e illustrates the performance of the algorithms for increasing degrees of noise for a specific query (Q3). Here again, for all setups of parameters, the provenance based algorithm performed better than both Min-Cut and Random algorithms, and among the these two, Min-cut was marginally better than Random for this query, but overall there was no clear winner.

Mixed algorithms. As mentioned before, we consider here an implementation of Algorithm 3 that uses QOCO deletion algorithm (as opposed to the weaker QOCO$^-$ or Random) and the Provenance-based insertion algorithm. Results with other insertion algorithms are not presented for space constraints because they yield weaker results than the provenance based algorithm. As shown in Figure 3c, QOCO performed better than its competitors. In figure 3f we use the Mixed algorithm, and the same query (Q3), but vary the number of missing and wrong answers. We demonstrate the distribution of different types of question presented to the crowd: verification of a result answer, verification of a tuple (a tuple contained in a false answer's witness), and filling blanks that refer to both adding a missing answer and completing a missing tuple. As expected, the graph shows that the number of tuples and answers that are verified increases as the number of errors increases.

Results for Real Experts Crowd. Figure 4 illustrates that QOCO is also effective with the real crowd (imperfect experts for a certain domain). The trends we observed are similar to the set of experiments with a simulated oracle (or a perfect expert). We aggregated the results from 3 experts using majority vote rule for all the answers from the crowd (our chosen implementation for the black-box aggregator). Recall that for each answer to an open question, QOCO poses to the crowd 2 additional closed verification questions (see Section 6.2). The way we count crowd answers is slightly different for boolean (closed) questions and open ones. Answers to closed questions increase the counter by one, while answers to open questions increase the counter by the number of (unique) variables that the expert provided their values. In addition, since the decisions are made by majority vote, once two experts give the same answer, a decision can be made and a third answer is no longer needed. Hence the total number of crowd answers in Figure 4 may be smaller than 3 times the number presented in the matching graph for the single perfect expert experiment. The results of the two illustrated queries contained 5 missing and 5 wrong answers (but the computation of Q3 involves more tuples hence the corresponding larger number of crowd questions). 60% of the errors in each query (the more popular and well-known answers) were identified and corrected within an hour from the time the queries were posted on the social network. 90% was fixed within another hour, and the whole experiment completed within 3.5 hours, identifying all errors. Observe that the "fill missing" numbers are identical across different algorithms of Q2 (resp. Q3). This is because we use the same provenance-based insertion algorithm and hence, the same (number of) questions are posed to the experts.

8. RELATED WORK

Data cleaning As mentioned in the Introduction, numerous data cleaning techniques have been proposed in the past. Cleaning problems can be classified between single-source and multi-source problems and between schema and instance related problems [52]. Existing tools address problems such as deduplication [17], entity resolution [1, 4], and schema matching [46]. Common technical approaches are, inter alia, clustering and similarity measures [4]. Data mining tools for outlier detection [22] or association rules [55] are also used to improve data, to complete missing values, correct illegal values and identify duplicate records. [66] used query aware approach in a different context of determining uncertain objects in probalistic data. The goal there is to generate a deterministic representation that optimize the quality of answers to queries/triggers that execute over the determinized data. In the context of data cleaning [62] introduced the idea of cleaning only a sample of data to obtain unbiased query results with confidence intervals. Experimental results have indicated that only a small sample needs to be cleaned to obtain accurate results. QOCO is similar in spirit to [62] in that it uses the crowd to correct query results, but unlike QOCO, [62] does not propagate the updates back to the underlying database. Another critical difference from [62], as well as from prior work on data cleaning, is that the open world assumption that we support allows to add missing true tuples to the database.

Crowdsourcing Crowdsourcing, or human computation, is a model where humans perform small tasks to help solve challenging problems. Incentives can range from small payments to public recognition and social reputation to the desire to help scientific progress [51]. It is a powerful tool that has been employed for database cleaning tasks such as entity/conflict resolution [61, 65], duplicate detection [6, 11], schema matching [67, 35, 43], and filling up missing data [50, 49, 27]. These complimentary techniques can be used for the initial data cleaning and then refined by our approach. As previously mentioned, extensive research has also been devoted to develop algorithms to ensure the quality of answers, both for individual answers (e.g., outlier detection [42, 59, 16]) and aggregated answers (e.g., using error probability, or an average weighted by trust [48, 47, 54]). In addition, previous works propose different methods for evaluating crowd workers' quality, e.g., to filter spammers and identify domain experts [42, 39, 36, 29, 37]. These methods too are complementary to our work and can be used here as a preliminary step to select our experts. Depending on the task, crowdsourced solutions may require a massive work force and may be expensive and time consuming. For instance, verifying that an ontology corresponds to an experts model of the scientific domain requires checking every relationship in the ontology [19]. As mentioned, we propose our query-oriented approach as means to focus resources to the most relevant portions of the underlying data.

View updates The problem of translating updates on the view into source updates so that the updates on the view are effectively captured is called the *view update problem*. Some of the works (e.g., [14, 9, 41]) compute the necessary updates on the source that will remove or insert the desired output tuple and, at the same time, minimizes the changes to the output. Our work follows more closely to the idea of automatically identifying "minimal updates" on the *source* that will correct one or more tuples in the query result [9]. However, a minimal source update does not always reflect the actual ground truth D_G. In fact, even though a minimal update to the source database will fix the query result, it may not correct the *database*. Even worse, it may actually further corrupt the database. Our algorithm in QOCO interactively leverages oracle crowds to identify a correct sequence of updates which may not always be "minimal" in the sense [9], to be performed. Such updates correct tuples in the underlying database in addition to fixing the errors in the query result.

Provenance The topic of data provenance (i.e., the origins or source of data) has been extensively studied in the past. Various notions of data provenance, such as lineage [15], why and where provenance [8], provenance semirings [30], have been proposed. Related to data provenance is the topic on explaining the reason(s) a tuple is in the result. Naturally, the data provenance of an output tuple can be used to explain the existence of the tuple in the output. In addition, other works, such as [3, 56], have considered how to explain the result or differences in aggregates in the result. More recently, there has been a number of research on deriving the changes that are needed to the underlying databases [34, 33, 32] or query [5, 10, 58, 31] so that a missing tuple appears in the output. The focus of our work, however, is neither to compute the data provenance nor to provide (all) explanations to the why or why-not questions but rather, to identify correct updates to apply to the underlying database to rectify the error in the output. Even if the explanations from prior work can be translated into corrective updates, their results do not suggest which corrective updates to apply.

9. CONCLUSIONS

We present QOCO, a novel query-oriented system for cleaning data with crowd oracles. An incorrect (resp. missing) tuple is removed from (resp. added to) the result of a query through updates on the underlying database, where the updates are derived by interacting with crowd oracles. The system uses a set of novel algorithms for minimizing the required interaction. Our experimental results on real-world datasets demonstrate the promise that QOCO is an effective and efficient tool for data cleaning.

There are several challenging directions for future work. Specifically, we plan to extend QOCO by supporting richer view languages, such as queries with aggregates and negation. Aggregates introduce significant complications as there are potentially numerous ways to achieve the same aggregate (e.g., to SUM to 100) and pruning the search space to identify the correct updates is challenging. On a related direction, we plan to investigate how constraints such as key and foreign key constraints can be incorporated into our framework. The presence of such constraints will require a more nuanced calculation of the (potential) interactions with the crowd, that take into account the dependencies among tuples and possible constraints violation. In addition, we plan to consider richer crowd interactions by allowing composite crowd questions where, for example, the correctness of several tuples is posed in a single question. Composite questions can potentially reduce the number of questions posed in general. We plan to investigate how to decide what kind of composite questions to ask and when to ask such questions. The crowd incentive and rewards model can also be improved. Currently we compute a user's effort based on the number of tuples and variables she adds. The model can be enhanced to account for the frequency and novelty of a particular answer (e.g., rare facts are harder to find), and motivate users to provide less-trivial answers in general. Another extension is to consider a richer crowd interaction paradigm that allows for attributes modification or tuples merge, in addition to deletions and insertions. Finally, an intriguing question is how to incrementally maintain the cleaned data when the underlying ground truth changes.

Acknowledgements We are grateful to the anonymous reviewers for their insightful comments. This work has been partially funded by the European Research Council under the FP7, ERC grant MoDaS, agreement 291071 and by the Israel Ministry of Science. Tan is partially supported by NSF grant IIS-1450560.

10. REFERENCES

[1] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *Proceedings of the VLDB Endowment*, 7(11), 2014.

[2] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD*, pages 241–252, 2013.

[3] T. Arora, R. Ramakrishnan, W. G. Roth, P. Seshadri, and D. Srivastava. Explaining program execution in deductive systems. In *DOOD*, pages 101–119, 1993.

[4] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(1):5, 2007.

[5] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based why-not provenance with nedexplain. In *EDBT*, pages 145–156, 2014.

[6] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. *KDD*, pages 39–48, 2003.

[7] P. Buneman, J. Cheney, W. C. Tan, and S. Vansummeren. Curated databases. In *ACM PODS*, pages 1–12, 2008.

[8] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[9] P. Buneman, S. Khanna, and W. C. Tan. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

[10] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[11] M. Charika, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. *PODS*, pages 268–279, 2000.

[12] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[13] Cia world fact book. https://www.cia.gov/library/publications/the-world-factbook/.

[14] Y. Cui and J. Widom. Run-time translation of view tuple deletions using data lineage. Technical Report 2001-24, Stanford InfoLab, 2001.

[15] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2):179–227, 2000.

[16] T. Dasu and T. Johnson. Exploratory data mining and data cleaning. In *Wiley*, 2003.

[17] P. Domingos. Multi-relational record linkage. In *In Proceedings of the KDD-2004 Workshop on Multi-Relational Data Mining*. Citeseer, 2004.

[18] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *JACM*, 19(2):284–264, 1972.

[19] J. Evermanna and J. Fangb. Evaluating ontologies: Towards a cognitive measure of quality. *Information Systems*, 35(4):391–403, 2010.

[20] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Synthesis Lectures on Data Management. 2012.

[21] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Cerfix: A system for cleaning data with certain fixes. In *PVLDB*, pages 1375–1378, 2011.

[22] U. M. Fayyad. Mining databases: Towards algorithms for knowledge discovery. *IEEE Data Eng. Bull.*, 21(1):39–48, 1998.

[23] U. Feige, M. Langberg, and K. Nissim. On the hardness of approximating NP witnesses. In K. Jansen and S. Khuller, editors, *Approximation Algorithms for Combinatorial Optimization*, volume 1913 of *LNCS*, pages 120–131. Springer, 2000.

[24] Fifa official site. http://www.fifa.com/.

[25] Fifa trivia quizzes and games. http://www.sporcle.com/games/g/fifaworldcup.

[26] Fifa world cup trivia. http://en.trivia.fifa.com/.

[27] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: Answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[28] V. Ganti and A. D. Sarma. *Data Cleaning: A Practical Perspective*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[29] S. Ghosh, N. Sharma, F. Benevenuto, N. Ganguly, and K. Gummadi. Cognos: crowdsourcing search for topic experts in microblogs. *SIGIR*, pages 575–590, 2012.

[30] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *ACM PODS*, pages 31–40, 2007.

[31] Z. He and E. Lo. Answering why-not questions on top-k queries. In *ICDE*, pages 750–761, 2012.

[32] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *PVLDB*, 3(1):185–196, 2010.

[33] M. Herschel, M. A. Hernández, and W. C. Tan. Artemis: A system for analyzing missing answers. *PVLDB*, 2(2):1550–1553, 2009.

[34] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[35] N. Q. V. Hung, N. T. Tam, Z. Mikló, K. Aberer, A. Gal, and M. Weidlich. Pay-as-you-go reconciliation in schema matching networks. In *ICDE*, pages 220–231, 2014.

[36] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. *HCOMP*, pages 64–67, 2010.

[37] J. Jiao, J. Yan, H. Zhao, and W. Fan. Expertrank: An expert user ranking algorithm in online communities. *NISS*, pages 674–679, 2009.

[38] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *ACM SIGMOD*, pages 841–852. ACM, 2011.

[39] D. R. Karger, S. Oh, and D. Shah. Iterative learning for reliable crowdsourcing systems. *NIPS*, pages 1953–1961, 2011.

[40] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, Plenum Press, New York, 1972.

[41] B. Kimelfeld, J. Vondrák, and R. Williams. Maximizing conjunctive views in deletion propagation. In *ACM PODS*, pages 187–198, 2011.

[42] A. Marcus, D. Karger, S. Madden, R. Miller, and S. Oh. Counting with the crowd. *PVLDB*, 6(2):109–120, 2012.

[43] R. Mccann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, pages 110–119, 2008.

[44] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1):34–45, 2010.
[45] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.
[46] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, volume 98, pages 24–27. Citeseer, 1998.
[47] A. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: Algorithms for filtering data with humans. In *SIGMOD*, pages 361–372, 2012.
[48] A. G. Parameswaran, S. Boyd, H. Garcia-Molina, A. Gupta, N. Polyzotis, and J. Widom. Optimal crowd-powered rating and filtering algorithms. *VLDB*, 7(9):685–696, 2014.
[49] H. Park, R. Pang, A. Parameswaran, H. Garcia-Molina, N. Polyzotis, and J. Widom. An overview of the deco system: data model and query language; query processing and optimization. In *SIGMOD*, pages 22–27, 2012.
[50] H. Park and J. Widom. Crowdfill: collecting structured data from the crowd. In *SIGMOD*, pages 577–588, 2014.
[51] M. J. Raddick, G. Bracey, P. L. Gay, C. J. Lintott, P. Murray, K. Schawinski, A. S. Szalay, and J. Vandenberg. Galaxy zoo: exploring the motivations of citizen science volunteers. *Astronomy Education Review*, 9(1), 2010.
[52] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.
[53] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.
[54] V. C. Raykar, S. Yu, L. H. Zhao, A. Jerebko, C. Florin, G. H. Valadez, L. Bogoni, and L. Moy. Supervised learning from multiple experts: whom to trust when everyone lies a bit. *ICML*, 2009.
[55] C. Sapia, G. Höfling, M. Müller, C. Hausdorf, H. Stoyan, and U. Grimmer. On supporting the data warehouse design by data mining techniques. In *Proc. GI-Workshop Data Mining and Data Warehousing*, page 63. Citeseer, 1999.
[56] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 42–53, 1999.
[57] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge unifying wordnet and wikipedia. In *WWW*, pages 697–706, 2007.
[58] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD Conference*, pages 15–26, 2010.
[59] B. Trushkowsky, T. Kraska, M. J. Franklin, and P. Sarkar. Crowdsourced enumeration queries. In *ICDE*, pages 673–684, 2013.
[60] Uniprot. http://www.uniprot.org/.
[61] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(10):1483–1494, 2012.
[62] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD*, pages 469–480, 2014.
[63] Open football. https://github.com/openfootball/.
[64] World cup history. http://www.worldcup-history.com/.
[65] S. E. Whang, H. Garcia-Molina, and P. Lofgren. Question selection for crowd entity resolution. *PVLDB*, 6(6):349–360, 2013.
[66] J. Xu, D. V. Kalashnikov, and S. Mehrotra. Query aware determinization of uncertain objects. *IEEE Trans. Knowl. Data Eng.*, 27(1):207–221, 2015.
[67] C. J. Zhang, Z. Zhao, L. Chen, H. V. Jagadish, and C. C. Cao. Crowdmatcher: crowd-assisted schema matching. In *SIGMOD*, pages 721–724, 2014.

APPENDIX

A. PROOFS

We provide below the proofs for the theorems and propositions presented in the paper.

PROPOSITION 3.3. *Let e be an edit that is generated based on the oracle's answer to a question. We have $|(D \oplus e) - D_G| \leq |D - D_G|$.*

PROOF. The edit e is either $R(\bar{a})^+$ or $R(\bar{a})^-$. Since $D \oplus e$ either removes a false tuple from D, or adds a true tuple to D, or leaves D unchanged, the above inequality holds. □

PROPOSITION 3.4. *Let D_G be a finite database instance that represents the ground truth, D be a database instance, Q be a query, and t be a target action. If the domain is ordered, then there is a finite number of questions $q_1, \ldots, q_k$ s.t. $Q(D \oplus \text{ans}(q_1) \oplus \ldots \oplus \text{ans}(q_k))$ achieves the desired target action.*

PROOF. Since the domain consists of values with an order, one can systematically enumerate all possible facts. For every fact f, we ask the question TRUE(f)? to the crowd and apply the corresponding edits to the database until the target action t is achieved. It is easy to see that all facts in $(D - D_G) \cup (D_G - D)$ will be asked after a finite number of steps and hence, the desired target action must be achieved after a finite number of steps. □

THEOREM 4.2. *Problem 4.1 is NP-hard.*

PROOF. We prove the theorem by showing that the corresponding decision problem is NP-hard. The deletion question decision problem asks: Given D, D_G, Q, $t \in (Q(D) - Q(D_G))$, does there exist at most k questions $q_1, ..., q_k$ of the type TRUE$(R(\bar{a}))$? s.t. $t \notin Q(D \oplus ans(q_1) \oplus ... \oplus ans(q_k))$? Clearly, the deletion question search problem is at least as hard as the deletion question decision problem.

Our reduction makes use of the Hitting Set Problem which asks: Given an instance (U, S) where U is a universe of elements and S is a set of subsets of U, and a positive number k, does there exists a hitting set H for (U, S) such that $|H| \leq k$?

We will reduce an instance of the decision version of the Hitting Set Problem with (U, S) and k to the deletion question decision problem as follows: The underlying vocabulary is $\{u_1, \ldots, u_{|U|}, S_1, \ldots, S_{|S|}, d\}$, for each $u_i \in U$, $S_i \in S$ and d stands for a constant that is different from the rest of the values. The instance D consists of $|U| + 1$ relations. For each element $u_i \in U$, $1 \leq i \leq |U|$, there is a unary relation schema $R_i(X_i)$ where the relation R_i has two facts $R_i(u_i)$ and $R_i(d)$. In addition, we have the relation schema $R(Z, A, X_1, \ldots, X_{|U|})$. For every set $S_i \in S$, we record in R the characteristic vector of S_i that describes the elements which occur in S_i. The instance D_G consists of the facts $\{R_1(d), ..., R_{|U|}(d)\}$. We define the query Q to be (z) :- $R(z, y, w_1, \ldots, w_{|U|}), R_1(w_1), ..., R_{|U|}(w_{|U|})$.

Notice that $Q(D)$ consists of a single tuple (d), while $Q(D_G) = \emptyset$. The input target action is to delete tuple $(d) \in Q(D)$. Recall that given an assignment α for a result tuple t, we call the set of tuples in $\alpha(body(Q))$ the witness for α, (or simply, a witness for t). Notice

that every witness for (d) w.r.t. D contains a characteristic vector of a different $S_i \in S$.

For example, if (U, S) is such that $U = \{u_1, u_2, ..., u_4\}$ $S = \{S_1 = \{u_2, u_3, u_4\}, S_2 = \{u_1, u_2\}\}$, then D consists of the following facts $R_1(u_1)$, $R_1(d)$, $R_2(u_2)$, $R_2(d)$, $R_3(u_3)$, $R_3(d)$, $R_4(u_4)$, $R_4(d)$, and 2 facts of the relation R, one for each $S_i \in S$. The set S_1 corresponds to the fact $R(d, S_1, d, u_2, u_3, u_4)$, and the set S_2 corresponds to the fact $R(d, S_2, u_1, u_2, d, d)$. The instance D_G consists of the facts $\{R_1(d), R_2(d), R_3(d), R_4(d)\}$, and the query Q is (z) :- $R(z, y, w_1, w_2, w_3, w_4), R_1(w_1), R_2(w_2), R_3(w_3), R_4(w_4)$. In this example tuple (d) has 2 different assignments and 2 witnesses (one for each $S_i \in S$):

$w_1 = R(d, S_1, d, u_2, u_3, u_4), R_1(d), R_2(u_2), R_3(u_3), R_4(u_4)$

$w_2 = R(d, S_2, u_1, u_2, d, d), R_1(u_1), R_2(u_2), R_3(d), R_4(d)$

It is easy to verify that the input to the deletion question decision problem can be constructed in polynomial time in the size of (U, S) and k.

We now show that there is a hitting set of size at most k if and only if there exists at most k questions $q_1, ..., q_k$ such that $t \notin Q(D \oplus ans(q_1) \oplus ... \oplus ans(q_k))$.

Suppose H is a hitting set of size at most k. It is straightforward to verify that one can pose the questions TRUE$(R_i(u_i))$? for every $u_i \in H$ in any order. Recall that D_G doesn't contain facts of type $R_i(u_i)$, for all i. Thus, the answer to every question TRUE$(R_i(u_i))$? for every $u_i \in H$ is NO, and a deletion edit $R_i(u_i)^-$ is generated. Since H is a hitting set, its elements hit every set in S, and hence their respective relations of the form $R_i(u_i)$ hit every witness of the tuple (d). The result of applying all such deletion edits to D is a database D' such that $(d) \notin Q(D')$.

For the converse, suppose there exists at most k questions q_1, ..., q_k such that $t \notin Q(D \oplus ans(q_1) \oplus ... \oplus ans(q_k))$. We can assume wlog that each q_i, $1 \leq i \leq k$, must be a question on facts of D since questions regarding facts not in D are unnecessary towards the removal of (d) from the output. Similarly, we can assume wlog that these questions will not be about facts of D_G, because they are unnecessary towards the removal of the wrong answer (d) as well.

Furthermore, we can assume wlog that each question is of the form TRUE$(R_i(u_i))$? where $1 \leq i \leq |U|$. If the question is of the form TRUE$(R(\bar{t}_i))$?, where $\bar{t}_i$ is a tuple of relation R, we can always replace this question with the question TRUE$(R_j(u_j))$? for some element u_j contained in the set represented by $R(t_i)$ in order to destroy the same witness. Since the deletion edits consists of facts of relations of type R_j and $t \notin Q(D \oplus ans(q_1) \oplus ... \oplus ans(q_k))$, it must be that the set of all subsets that correspond to the deletion edits forms a hitting set for U and this hitting set has size at most k. □

THEOREM 4.5. *Given a pair (U, S), as described in Definition 4.3, a unique minimal hitting set exists if and only if the elements of the singleton sets of S forms a hitting set for S.*

PROOF. Let M denote the elements that occur in all singleton sets of S. Clearly, any hitting set must contain M. If M itself is a hitting set, then M is also a unique minimal hitting set because none of the elements of M can be removed. We show next that if a unique minimal hitting set exists, then M must be a hitting for S. Assume by contradiction that the hitting set must include, in addition to the elements in M, some other element that hits a set $s' \in S$. Then, s' must contain at least 2 elements and the elements of s' do not occur among M. If this is the case, different elements of s' will constitute to different minimal hitting sets. □

THEOREM 5.2. *Problem 5.1 NP-hard.*

PROOF. We prove the theorem by describing a reduction from the following NP-hard problem, which we call One-3SAT. An instance of the One-3SAT problem is a satisfiable 3CNF formula Φ where each clause in Φ has three literals. The problem of constructing a satisfying assignment for a satisfiable formula Φ is NP-hard [23]. We will show that if there is a polynomial time algorithm for generating at most k questions such that $t \in Q(D \oplus ans(q_1) \oplus ... \oplus ans(q_k))$, then one can construct a satisfying assignment for Φ in polynomial time.

Suppose there is a polynomial time algorithm P for the output insertion question search problem. Given a 3SAT formula Φ, we construct an input instance for the output insertion question search problem as follows: The database instance D is the empty database. Construct a relation for every clause in Φ as follows. For every clause $\varphi_i \in \Phi$, where $1 \leq i \leq |\Phi|$, construct a relation $R_i(A, X_{i1}, X_{i2}, X_{i3})$ where X_{ij} denotes the jth literal in φ_i and A is a fresh attribute that does not occur among the literals of Φ. In database D_G, every relation R_i, $1 \leq i \leq |\Phi|$, consists of facts that represent the satisfying assignments of the corresponding clause φ_i. For example, given $\varphi_1 = (X_1 + X_2 + X_4)$, there are 7 facts (all of $R_1(d, 1, 1, 1)$ to $R_1(d, 0, 0, 1)$ except $R_1(d, 0, 0, 0)$) in D_G. For the clause $\varphi_2 = (X_1 + X_2 + \neg X_3)$, there are 7 facts in D_G, all except for the tuple $R_2(d, 0, 0, 1)$. The value d is a fresh constant that is different from 0 or 1. Recall that the crowd member must answer YES to questions of the form TRUE$(R_i(\bar{a}))$?, if $R_i(\bar{a}) \in D_G$. Now, define the query Q to be (x) :- $R_1(x, x_{1,1}, x_{1,2}, x_{1,3}), \ldots, R_{|\Phi|}(x, x_{|\varphi|,1}, x_{|\varphi|,2}, x_{|\varphi|,3}))$, where the variables $x_{i,1}, x_{i,2}, x_{i,3}$ correspond to the literals in clause $\varphi_i \in \Phi$ for $1 \leq i \leq |\Phi|$. For example, the corresponding query for φ_1 and φ_2 is (x) :- $R_1(x, X_1, X_2, X_4)$, $R_2(x, X_1, X_2, X_3)$ and $Var(Q) = \{x, X_1, X_2, X_3, X_4\}$. The output $Q(D)$ is $\emptyset$ since D is empty. The wanted target action is to insert tuple (d), because $(d) \in Q(D_G)$ and $t \notin Q(D)$. It is straightforward to verify that the input to the output insertion question search problem can be constructed in polynomial time in the size of Φ.

We now execute our polynomial time algorithm P on D, D_G, Q, and (d) with $k = |\Phi|$. We show that a solution to $P(D, D_G, Q, (d), k)$ leads to a satisfying assignment for Φ. For (d) to appear in the output, there must be at least one tuple per relation R_i, where $1 \leq i \leq |\Phi|$, such that the tuples from different relations join according to Q. Since we are only allowed $|\Phi|$ questions, a solution to $P(D, D_G, Q, (d), k)$ must contain exactly one question TRUE$(R_i(t_j))$? for a tuple in each relation and the answer to the question must be YES. Furthermore, since these tuples must join together to produce (d), we can conclude that the satisfying assignment for each clause together form a satisfying assignment for Φ. □

B. THE GENERAL ALGORITHM

The list of modifications to Algorithms 1, 2 and 3, designed to support multiple imperfect experts in parallel, described in Section 6.2, are:

1. Post all closed questions multiple times according to the aggregator black-box. These questions are in Algorithm 1 in line 7, Algorithm 2 line 7, and Algorithm 3 line 3.
2. Verify an answer to each opened question with closed verification questions posted to the crowd. These questions appear in Algorithm 2 lines 12, 18, and Algorithm 3 line 7.
3. Run both deletion and insertion parts in parallel. To allow that, we add another condition $Q(D) \neq Res_{complete}$ to the outer "while' loop in Algorithm 3 line 1.

4. Use parallel foreach loops, in both deletion part (Algorithm 3 lines 2-6) and insertion part (Algorithm 3 lines 7-9).

C. SYSTEM ARCHITECTURE

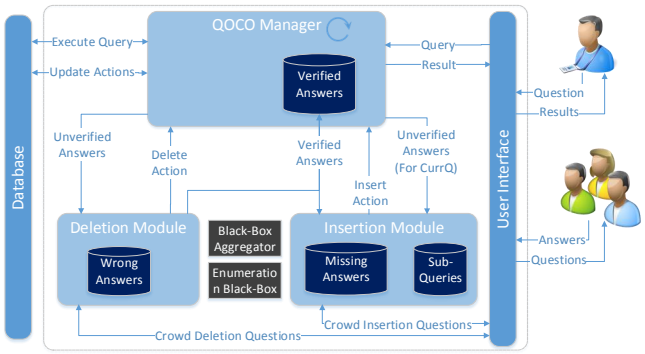


Figure 5: QOCO architecture

Figure 5 illustrates QOCO's system architecture. We describe the major components next. The crowd, who are answering questions, and the user (requester) who is running the query over the DB, interact with QOCO through the *User Interface*. In our experiments with simulated oracle, the *User Interface* was replaced with a simulator (called the *ground truth*) that returned answers to queries from the ground truth database). There are three core modules. *QOCO Manager* is responsible for interacting with the *Database*, and managing iterations and deletions as described in Algorithm 3. It receives the query from the requester and executes it on the *Database*. It performs the needed insert and delete actions identified by the *Deletion module* (Algorithm 1) and *Insertion* module (Algorithm 2).