A System Prototype for Approximate Query Answering over Incomplete Data

(DISCUSSION PAPER)

Nicola Fiorentino, Sergio Greco, Cristian Molinaro, and Irina Trubitsyna

DIMES, University of Calabria
{lastname}@dimes.unical.it

Abstract. Many database applications face the problem of querying incomplete data. In such scenarios, certain answers are a principled semantics of query answering. Unfortunately, the computation of certain query answers is a coNP-hard problem. To make query answering feasible in practice, recent research has focused on developing polynomial time algorithms computing a sound (but possibly incomplete) set of certain answers.
In this paper we present a system prototype implementing a suite of algorithms to compute sound sets of certain answers. The central tools used by our system are conditional tables and the conditional evaluation of relation algebra. Different evaluation strategies can be applied, with more accurate ones having higher complexity, but returning more certain answers, thereby enabling users to choose the technique that best meets their needs in terms of balance between efficiency and quality of the results.

1 Introduction

Incomplete information arises in many database applications, such as ontological reasoning [4, 5], inconsistency management [2, 3, 11, 15], data integration [7, 16], and many others.

A principled semantics of query answering over incomplete databases are *certain answers*, which are query answers that are obtained from all the complete databases represented by an incomplete database [17, 6, 18]. The following example illustrates the notion of a certain answer.

Example 1. Consider the database D consisting of the three unary relations `P` (Person), `S` (Student) and `E` (Employee) reported below, where $\perp$ is a null value.

P
john
mary

E
john
$\perp$

S
mary
bob

Copyright © 2019 for the individual papers by the papers authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors. SEBD 2019, June 16-19, 2019, Castiglione della Pescaia, Italy.

CEUR-WS.org/Vol-2400/paper-25.pdf

Under the missing value interpretation of nulls (i.e., a value for $\perp$ exists but is unknown), D represents all the databases obtained by replacing $\perp$ with an actual value.

A certain answer to a query is a tuple that is an answer to the query for every database represented by D. For instance, consider the query asking for the people who are not employees and students, which can be expressed in relational algebra as $\mathtt{P} - (\mathtt{E} \cap \mathtt{S})$. The certain answers to the query are $\{\langle \mathtt{john} \rangle\}$, because no matter how $\perp$ is replaced, $\langle \mathtt{john} \rangle$ is always a query answer.

For databases containing (labeled) nulls, certain answers to positive queries can be easily computed in polynomial time as follows: first a "standard" evaluation (that is, treating nulls as standard constants) is applied; then tuples with nulls in the result of the first step are discarded and the remaining tuples are the certain answers to the query. However, for more general queries with negation the problem of computing certain answers becomes coNP-hard.

To make query answering feasible in practice, one might resort to SQL's evaluation, but unfortunately, the way SQL behaves in the presence of nulls may result in wrong answers.

Specifically, as evidenced in [18], there are two ways in which certain answers and SQL's evaluation may differ: *(i)* SQL can miss some of the tuples that belong to certain answers, thus producing *false negatives*, or *(ii)* SQL can return some tuples that do not belong to certain answers, that is, *false positives*. While the first case can be seen as an under-approximation of certain answers (a sound but possibly incomplete set of certain answers is returned), the second scenario must be avoided, as the result might contain plain incorrect answers, that is, tuples that are not certain.

The experimental analysis in [13] showed that false positive are a real problem for queries involving negation—they were always present and sometimes they constitute almost 100% of the answers.

Example 2. Consider again the database D of Example 1. There are no certain answers to the query $\mathtt{P} - \mathtt{E}$, as the query answers are the empty set when $\perp$ is replaced with `mary`.

Assuming that `P` and `E`'s attribute is called `name`, the same query can be expressed in SQL as follows:

```
SELECT P.name
FROM P
WHERE NOT EXISTS (
    SELECT *
    FROM E
    WHERE P.name = E.name )
```

The evaluation of the SQL query above returns $\langle \mathtt{mary} \rangle$, which is not a certain answer. The problem with the SQL semantics is that every comparison involving at least one null evaluates to the truth value *unknown*, then 3-valued logic is used to evaluate the classical logical connectives (AND, OR, NOT), and eventually only those tuples whose condition evaluates to *true* are kept.

Going back to the query above, for the first tuple of P, namely john, the nested subquery finds the same tuple in E, and thus john is not returned.

For the second tuple of P, namely mary, the nested subquery gives the empty set and thus $\langle \texttt{mary} \rangle$ is returned by the overall query. The reason why the nested subquery returns the empty set when mary is considered is that mary is compared with john and the comparison evaluates to *false*, and mary is compared with $\bot$ and the comparison evaluates to *unknown* (because a null is involved). Thus, there is no tuple of E for which the comparison evaluates to *true* and the nested subquery returns the empty set.

Thus, on the one hand, SQL's evaluation is efficient but flawed, on the other hand, certain answers are a principled semantics but with high complexity. To deal with this issue, there has been recent work on evaluation algorithms with *correctness guarantees*, that is, techniques providing a sound but possibly incomplete set of certain answers [13, 17, 18, 12, 8]. The problem of computing sound (but possibly incomplete) sets of *consistent* query answers over inconsistent databases has been addressed in [9], but databases are assumed to be complete, while in this paper we consider incomplete databases with no integrity constraints.

We have developed novel evaluation algorithms with correctness guarantees leveraging conditional tables and the conditional evaluation of relational algebra [12, 8]. In conditional tables each tuple is associated with a condition and the conditional evaluation is a generalization of relational algebra that manipulate conditional tables. Conditions keep track of how tuples are derived and how nulls are used in comparison operators.

The basic idea is illustrated in the following example.

Example 3. Consider again the database and the query of Example 1. The conditional evaluation of the query is carried out by applying the "conditional" counterpart of each relational algebra operator. Rather than returning a set of tuples, the conditional evaluation of a relational algebra operator returns "conditional tuples", that is, pairs of the form $\langle t, \varphi \rangle$, where t is a regular tuple and φ is an expression stating under which conditions t can be derived.

Regarding the query of Example 1, first the conditional evaluation of $\texttt{E} \cap \texttt{S}$ is performed, which gives the conditional tuples $\langle \bot, \varphi_1 \rangle$ and $\langle \bot, \varphi_2 \rangle$, where φ_1 is the condition $(\bot = \texttt{mary})$ and φ_2 is the condition $(\bot = \texttt{bob})$. This intuitively means that the tuple $\langle \bot \rangle$ is derived when $\bot$ is mary or bob.

Then, the conditional evaluation of the difference operator is carried out, yielding the conditional tuples $\langle \texttt{john}, \varphi' \rangle$ and $\langle \texttt{mary}, \varphi'' \rangle$ where φ' and φ'' are the following conditions:

$$\varphi' = \neg((\texttt{john} = \bot) \wedge (\bot = \texttt{mary})) \wedge \neg((\texttt{john} = \bot) \wedge (\bot = \texttt{bob})),$$
$$\varphi'' = \neg((\texttt{mary} = \bot) \wedge (\bot = \texttt{mary})) \wedge \neg((\texttt{mary} = \bot) \wedge (\bot = \texttt{bob})).$$

This is the result of the conditional evaluation of the whole query.

Conditions are valuable information that can be exploited to determine which tuples are certain answers. As already mentioned, for a conditional tuple $\langle t, \varphi \rangle$,

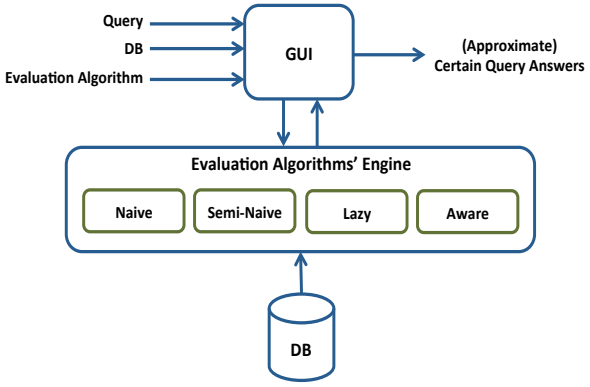

Fig. 1. System Architecture.

the expression φ says under which condition t can be derived. By condition evaluation we mean a way of associating φ with a truth value (*true*, *false*, or *unknown*). The aim is to ensure that if φ evaluates to *true*, then t is a certain answer. For instance, from an analysis of φ' in Example 3 above, one can realize that the condition is always true (i.e., it holds for every possible value $\perp$ stands for), and thus $\langle$`john`$\rangle$ is a certain answer.

Tuples' conditions can be evaluated in different ways: for instance, an eager strategy consists in evaluating conditions right after each relational algebra operator has been evaluated, while an opposite approach consists in evaluating conditions at the very end, that is, after the entire relational algebra query has been evaluated.

We have developed four different strategies leading to different evaluation algorithms, called *naive*, *semi-naive*, *lazy*, and *aware* evaluations. They have been implemented in the ACID system [8], which enables users to query incomplete databases and get under-approximations of the certain answers, choosing the evaluation strategy that is most suitable for the application at hand.

2 System Overview

The ACID system has been implemented in Java. The system architecture is depicted in Figure 1.

There are three main components: a graphical user interface (GUI), the evaluation algorithms' engine, and the database.

The GUI allows user to specify the query to be evaluated, the database, and the type of evaluation to be performed, that is, the approximation algorithm to be applied. The GUI displays the result of evaluating the specified query over the provided database according to the chosen evaluation algorithm. Different filters can be applied to the result (more details are discussed in the next section).

The system's engine supports the four evaluation algorithms mentioned in the previous section, with the naive algorithm being the most efficient but the

least accurate one, and the aware algorithm being the most accurate but the least efficient one. The basic ideas of the approximation algorithms are as follows:

- The *naive evaluation* evaluates tuples' conditions right after each relational algebra operator has been applied, using three-valued logic.
- The *semi-naive evaluation* behaves like the naive one, but it better exploits equalities in conditions (by propagating values into tuples and conditions) to provide more accurate results.
- The *lazy evaluation* improves upon the semi-naive one by postponing conditions' evaluation until the set difference operator is encountered in the query.
- The *aware evaluation* provides even more accurate results and behaves as follows: it performs the conditional evaluation of the entire query, then it uses a set of axioms to "simplify" conditions, and eventually it evaluates (simplified) tuples' conditions.

The ACID system manages relational databases possibly containing labeled nulls (in the literature, they have been called *naive tables*, *V-tables*, and *e-tables* [14, 1, 10]). Thus, the same (labeled) null can occur multiple times—e.g., this can be used to express that there are two employees with the *same* unknown salary.

The GUI provides information on the query, the database, and the evaluation strategy to the engine, which computes the approximate certain answers accessing the database. After the evaluation has been carried out, the engine returns the result to the GUI.

The ACID system provides also an API which allow third party applications to interact with the system.

We now go into the details of how to interact with the ACID system (cf. Figure 2).

A typical interaction with the system involves the following steps:

1. The user specifies the input databases. Specifically, for each table in the database, its location in the file system is provided. Tables are supposed to be in csv format.
2. The user specifies the query to be evaluated using standard SQL syntax. Queries can be loaded from and saved to files.
3. The user specifies the evaluation strategy that has to be applied to evaluate the query (indeed, the system supports also the "standard" evaluation mentioned in the introduction and the conditional evaluation of a query).
4. After the evaluation has been launched and has finished, the result and statistics are displayed. Specifically, the result is a set of a tuples, where each tuple is associated with a condition that is either `true` or `unknown`. Tuples associated with `true` are guaranteed to be certain answers to the input query. The result can be filtered with respect to the truth value of the tuples, thus displaying only true or only unknown tuples. The total number of true (resp. unknown) tuples is displayed as well as the execution time. Results can be saved to files.

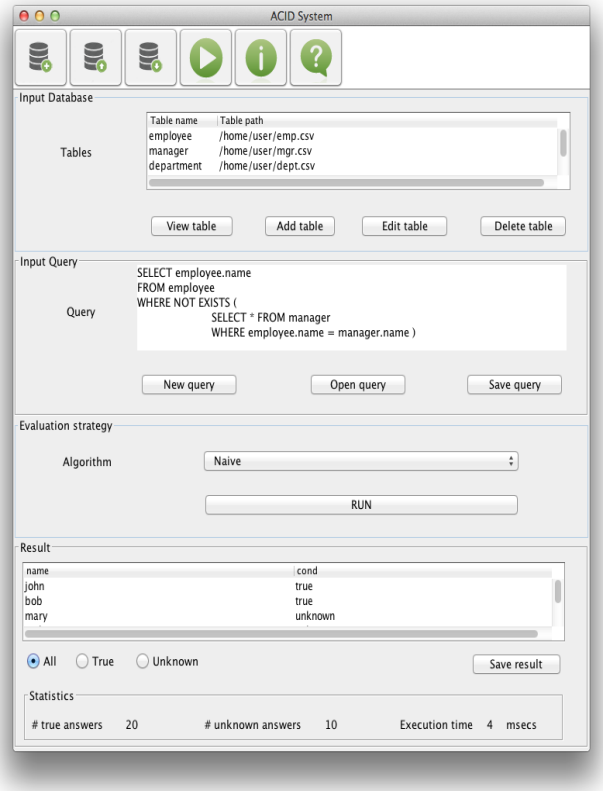

Fig. 2. ACID system's GUI.

3 Trading-off Quality with Runtime

With the same query and database, moving to more accurate strategie, that is, from the naive (resp. semi-naive, lazy) evaluation to the semi-naive (lazy, aware) one, users can see better results, that is, more tuples with condition *true* (i.e., more certain answers), but running times might get higher.

In general, using the system and analyzing the query syntax, users can figure out the strategy that is best suited for their purposes.

As an example, Table 1 reports the execution time, the number of true and unknown answers for three sample queries over a database with the same schema of the one in Example 1, with 1000 tuples per relation and 10% of nulls (randomly generated).

The queries are:

- $Q_{sn} = E - \sigma_{\$1=c}(S)$,
- $Q_{lazy} = P - (E \cap (\sigma_{\$1 \neq c}(S)))$, and

	Q_{sn}			Q_{lazy}			Q_{aware}		
	time	#true	#unk	time	#true	#unk	time	#true	#unk
Naive	518	741	167	788	710	189	2163	100	731
Semi-naive	615	763	145	1090	710	189	2347	100	731
Lazy	661	763	145	1579	737	162	5542	100	731
Aware	3580	763	145	4350	737	162	10376	231	600

Table 1. Runtime (msecs), number of true and unknown answers for three sample queries — 10% of nulls.

– $Q_{aware} = P - (E - S)$,

where c is a value randomly chosen from those in S.

The purpose of the first scenario is to exhibit a query (namely, Q_{sn}) that shows the benefits of going from the naive to the semi-naive evaluation—notice that, in this case, there is no benefit in applying the lazy or aware evaluation, as the structure of the query does not have features that can be exploited by them.

Likewise, the purpose of the second and third scenarios is to show the advantage of using the lazy (resp. aware) evaluation rather than the semi-naive (resp. lazy) one.

4 Conclusion

Certain answers are a principled manner to answer queries on incomplete databases. Since their computation is a coNP-hard problem, recent research has focused on developing polynomial time algorithms providing under-approximations. Leveraging conditional tables, we have developed a suite of novel approximation algorithms.

We have implemented them in the ACID system, which allows users to query incomplete information and get approximate answers with the flexibility of choosing the technique that best meets their needs in terms of balance between efficiency and quality of the result's approximation.

References

1. Abiteboul, S., Grahne, G.: Update semantics for incomplete databases. In: Proc. Very Large Data Bases (VLDB) Conference. pp. 1–12 (1985)
2. Arenas, M., Bertossi, L.E., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proc. Symposium on Principles of Database Systems (PODS). pp. 68–79 (1999)
3. Bertossi, L.E.: Database Repairing and Consistent Query Answering. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2011)
4. Bienvenu, M., Ortiz, M.: Ontology-mediated query answering with data-tractable description logics. In: Reasoning Web. pp. 218–307 (2015)
5. Calì, A., Gottlob, G., Lukasiewicz, T.: A general datalog-based framework for tractable query answering over ontologies. Journal of Web Semantics **14**, 57–83 (2012)

6. Console, M., Guagliardo, P., Libkin, L.: Approximations and refinements of certain answers via many-valued logics. In: Proc. International Conference on Principles of Knowledge Representation and Reasoning (KR). pp. 349–358 (2016)
7. De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: On reconciling data exchange, data integration, and peer data management. In: Proc. Symposium on Principles of Database Systems (PODS). pp. 133–142 (2007)
8. Fiorentino, N., Greco, S., Molinaro, C., Trubitsyna, I.: ACID: A system for computing approximate certain query answers over incomplete databases. In: Proc. International Conference on Management of Data (SIGMOD). pp. 1685–1688 (2018)
9. Furfaro, F., Greco, S., Molinaro, C.: A three-valued semantics for querying and repairing inconsistent databases. Annals of Mathematics and Artificial Intelligence **51**(2-4), 167–193 (2007)
10. Grahne, G.: The Problem of Incomplete Information in Relational Databases, Lecture Notes in Computer Science, vol. 554. Springer (1991)
11. Greco, S., Molinaro, C., Spezzano, F.: Incomplete Data and Data Dependencies in Relational Databases. Synthesis Lectures on Data Management, Morgan & Claypool Publishers (2012)
12. Greco, S., Molinaro, C., Trubitsyna, I.: Computing approximate certain answers over incomplete databases. In: Proc. Alberto Mendelzon International Workshop on Foundations of Data Management and the Web (AMW) (2017)
13. Guagliardo, P., Libkin, L.: Making SQL queries correct on incomplete databases: A feasibility study. In: Proc. Symposium on Principles of Database Systems (PODS). pp. 211–223 (2016)
14. Imielinski, T., Jr., W.L.: Incomplete information in relational databases. Journal of the ACM **31**(4), 761–791 (1984)
15. Koutris, P., Wijsen, J.: The data complexity of consistent query answering for self-join-free conjunctive queries under primary key constraints. In: Proc. Symposium on Principles of Database Systems (PODS). pp. 17–29 (2015)
16. Lenzerini, M.: Data integration: A theoretical perspective. In: Proc. Symposium on Principles of Database Systems (PODS). pp. 233–246 (2002)
17. Libkin, L.: How to define certain answers. In: Proc. International Joint Conference on Artificial Intelligence (IJCAI). pp. 4282–4288 (2015)
18. Libkin, L.: Certain answers as objects and knowledge. Artificial Intelligence **232**, 1–19 (2016)