Complete Approximations of Incomplete Queries

Ognjen Savković[1] Paramita Mirza[2] Alex Tomasi[1] Werner Nutt[1]

[1]Free University of Bozen-Bolzano
Piazza Domenicani 3
I-39100 Bozen-Bolzano, Italy
{fname.lname}@unibz.it

[2]Fondazione Bruno Kessler
Via Sommarive 18
I-38123 Trento, Italy
paramita@fbk.eu

ABSTRACT

We present a system that computes for a query that may be incomplete, complete approximations from above and from below.

We assume a setting where queries are posed over a partially complete database, that is, a database that is generally incomplete, but is known to contain complete information about specific aspects of its application domain. Which parts are complete, is described by a set of so-called table-completeness statements. Previous work led to a theoretical framework and an implementation that allowed one to determine whether in such a scenario a given conjunctive query is guaranteed to return a complete set of answers or not.

With the present demonstrator we show how to reformulate the original query in such a way that answers are guaranteed to be complete. If there exists a more general complete query, there is a unique most specific one, which we find. If there exists a more specific complete query, there may even be infinitely many. In this case, we find the least specific specializations whose size is bounded by a threshold provided by the user.

Generalizations are computed by a fixpoint iteration, employing an answer set programming engine. Specializations are found leveraging unification from logic programming.

1. INTRODUCTION

Completeness is one of the classical dimensions of data quality. Recently, it attracted increased attention in research (cf. [2, 3]). With this demonstration, we draw upon an approach by Razniewski and Nutt [7] and a subsequent implementation by the present authors [8]. Building upon previous work by Motro [6] and Levy [5], that work resulted in techniques to reason about the question whether a generally incomplete database D contains sufficient information to return a complete answer for a specific query Q.

This and the present work are motivated by a project to create a school information system in the Italian province of Bolzano, where decision-makers want to have guarantees about the completeness of query answers. In this setting, a database instance D may be *partially complete* in that it contains, e.g., all pupils at primary schools in Bolzano, but *generally incomplete* because the registration data from other schools have not yet been incorporated.

Proceedings of the VLDB Endowment, Vol. 6, No. 12

Following the approach in [7, 8], one can express which parts of which tables are complete, using so-called table-completeness (TC) statements. For example, one can write TC statements which say that the database instance contains "all primary schools" (prim), "all pupils attending a school in Bolzano" (schBol), and "all English learners that are pupils at a primary school" (primEng), respectively. Such TC statements could be generated by workflow engines or during ETL processes filling data warehouses. Intuitively, a TC statement like (schBol) says that for every pupil enrolled in a school in Bolzano, there is a recored in the pupil table of the database. A formal semantics has been provided in [5, 7].

If we know that a collection $\mathcal{C}$ of TC statements holds over a database, we can conclude that a certain query, say Q, will return the same set of answers as it would over a database that contains a complete record of the application domain. For instance, from (prim), (schBol), and (primEng) we can infer that we can retrieve all pupils at a primary school in Bolzano that learn English.

In [7], the authors reduced such completeness reasoning to query containment, while [8] reported on an implementation, the MAGIK system, that generalized these techniques to reasoning in the presence of keys, foreign keys (FKs), and finite domain constraints (FDCs). In addition, if completeness of Q does not follow from $\mathcal{C}$, MAGIK returns the weakest set of TC statements whose addition to $\mathcal{C}$ makes Q complete. Thus it would tell, in a way, which data need to be added to make Q complete.

In this demonstration, instead, we deduce how to *approximate* an incomplete query by complete queries from above, that is, by a more general query, and from below, by more specific queries. We can deal with single block SQL queries with equalities, which correspond to conjunctive queries and our algorithms take into account keys, FKs, and FDCs.

If Q, Q' are queries such that Q is contained in Q', then we say that Q is a *specialization* of Q' and Q' is a *generalization* of Q. One can prove that, given TC statements $\mathcal{C}$, if there is a complete generalization of a query Q, there is a most specific one. We call this the *most specific generalization* or MSG of Q. Our system computes such MSGs. One can also show that in general there is more than one *least specific specialization* (LSS) of Q. Specializations can be obtained by instantiating variables or by adding atoms or by both. In some cases, by continuous addition of atoms to Q, one can even generate less and less specific specializations. Therefore, our specialization algorithm is invoked with a bound on the number of atoms it can add to Q.

With our demonstrator we made the following contributions:

- We developed algorithms to compute for an incomplete query the most specific complete approximation from above (MSG) and, given a bound on the query size, the set of least specific complete approximations from below (LSSs);

- We implemented our algorithms using logic programming platforms. More specifically, (i) we extended earlier techniques for completeness reasoning, based on answer set programming, to compute most specific generalizations, and (ii) we leveraged Prolog unification to compute least specific specializations.

Section 2 walks through a possible demo session, Section 3 sketches our implementation techniques, and Section 4 gives an overview over the architecture.

2. SAMPLE DEMONSTRATION

The demonstrator analyzes queries that are posed over a fixed schema with keys, foreign keys (FKs), and finite domain constraints (FDCs) . It can be run either in *virtual* mode, where schemas can be defined and altered for demonstration purposes, or in *database* mode, where a schema can be imported from an existing PostgreSQL database, and where constraints are read from the catalog.

In this section, we walk through a possible session and explain the functionalities of the system. One starts by creating the schema of a virtual database, connecting to an existing database, or selecting an already existing schema. For each schema, the demo server keeps completeness constraints and queries from earlier sessions.

In our case, we choose the schema of the existing toy database "school-bolzano", modeling schools in the province of Bolzano (Figure 1). It consists of the four tables below, where primary keys are underlined:

$$school(\underline{sname}, type, district)$$
$$class(\underline{code}, level, scheme)$$
$$pupil(\underline{pname}, sname, code)$$
$$learns(\underline{pname}, \underline{lang}).$$

The table *school* records for each school the name, the type (e.g., primary or middle school), and its school district. The table *class* stores for each class, which is identified by a code, the level (e.g., 1st year) and its scheme (half day or full day). The table *pupil* contains for each pupil, identified by the name, the class the pupil attends and the school. Finally, the table *learns* records which pupil learns which language.

Once a schema is selected, the Completeness Reasoning window is opened (Figure 2 (a)). Here the upper panel contains the foreign key and finite domain constraints; the middle panel contains a list of TC statements; and the lower panel contains a list of queries. In virtual mode, all items from these three components can be created, edited or deleted. In database mode, this is impossible for the schema constraints.

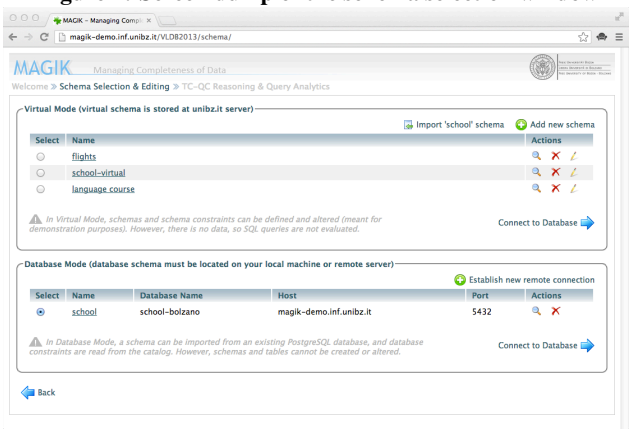
Each item in the window (FK, FDC, TC statement, or query) can be activated by selecting the corresponding check box. If constraints or completeness statements are activated, the reasoner takes them into account in its analysis. In this way, the different features and reasoning modes can be illustrated in the demo.

In our sample session, as documented in Figure 2(a), six TC statements are activated:

Complete Table	Completeness Condition
$tc_1 : school(S, 'primary', D)$	
$tc_2 : pupil(N, S, C)$	$school(S, T, 'Bolzano')$
$tc_3 : class(C, L, 'halfDay')$	
$tc_4 : class(C, L, 'fullDay')$	
$tc_5 : learns(N, 'English')$	$pupil(N, S, C), school(S, 'primary', D)$
$tc_6 : learns(N, 'English')$	$pupil(N, S, C), school(S, 'middle', D)$

Table completeness statements are written in a datalog-like syntax. Intuitively, a TC statement about a table asserts that the table

Figure 1: Screen dump of the schema selection window



is complete for all tuples satisfying some conditions expressed by selections and semi-joins. For instance, the first statement asserts that the *school* table contains all primary schools (in the application domain of the database). The second statement asserts that the table *pupil* contains records of all pupils that attend a school in the Bolzano district. Formally, the first statement says that the table *school* contains all tuples that over an ideal complete database satisfy the selection $\sigma_{\text{type}='primary'}(school)$. The second statment says that the table *pupil* contains all tuples that satisfy the semijoin $pupil \ltimes \sigma_{\text{district}='Bolzano'}(school)$ over such an ideal database. Syntactically, a TC statement for a table R consists of two parts: an R-atom, representing a selection on R, and, possibly, a condition, representing a semijoin with other tables. Strings starting with upper-case letters denote variables and others constants. In our example session, all TC statements are activated.

Queries are entered as SQL single block queries and, for the sake of this demo, are assumed to be evaluated under set semantics. The reason is that generalizations and specializations of queries often change the multiplicities of query answers, which would make the demo less interesting.

In the sample demo, we have entered four queries, Q_1 to Q_4. Once a query is selected it can be checked for completeness by pressing the button *Run Query*. Trying out one by one query, the completes reasoner shows that, given the activated TC statements (Figure 2 (a)), the first query can be answered completely and it displays green label that the query is complete. For the last three it found that completeness cannot be guaranteed and for each of them it displays red label that query is incomplete (Figure 2 (b)).

In addition, if a query is incomplete, one finds here the most specific generalization (MSG) of the query that is complete for the given TC statements. One also finds the least specific specializations (LSSs) that are complete and do not contain additional atoms. This helps a user understand which answers are potentially missing and which answers are be lost if the query is replaced with some complete approximation.

In the following, we discuss the completeness analysis for each query separately. While the demonstrator accepts queries in SQL syntax, we employ datalog syntax to save space.

Query Q_1: "Select the names of all pupils that attend a primary school in the Bolzano district":

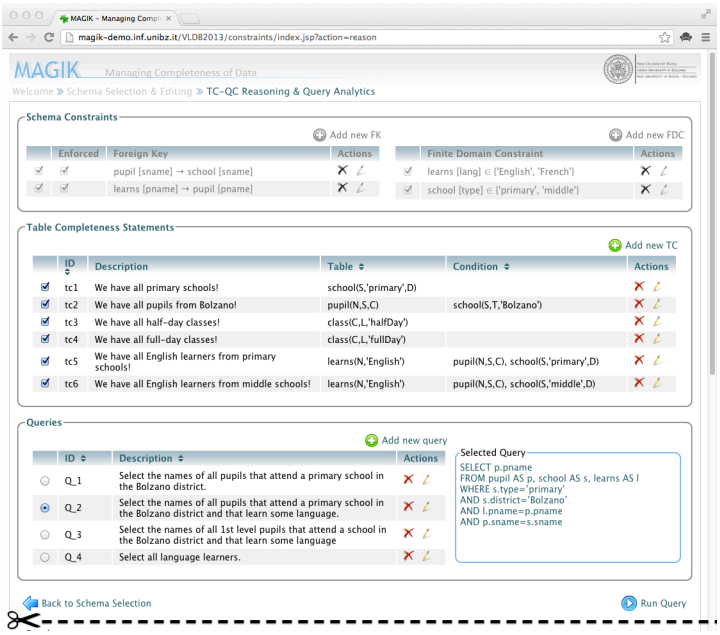
$$Q_1(N) :- pupil(N, S, C), school(S, 'primary', 'Bolzano').$$

This query is complete because our database contains all records of pupils from primary schools in the Bolzano district (tc_2) and it contains all such schools (tc_1).

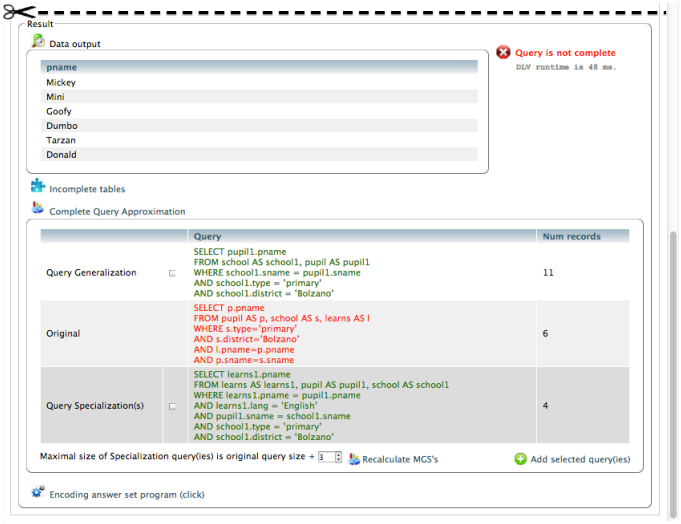
Query Q_2: "Select the names of all pupils that attend a primary school in the Bolzano district and that learn some language":

$$Q_2(N) :- pupil(N, S, C), school(S, 'primary', 'Bolzano'), learns(N, L).$$

Figure 2: Screen dumps of our demonstrator



(a) Input part of Reasoning window



(b) Output part of Reasoning window

This query is reported to be incomplete because there is no guarantee that we have all language learners. If we look at the output part of Reasoning window for this query (Figure 2 (b)), we see that the system has found one generalization and one specialization:

$$Q_2^{gen}(N) :- \ pupil(N,S,C), school(S,\text{'primary'},\text{'Bolzano'}).$$
$$Q_2^{spec1}(N) :- \ pupil(N,S,C), school(S,\text{'primary'},\text{'Bolzano'}), learns(N,\text{'English'}).$$

They query Q_2^{gen} was discovered by dropping the potentially "incomplete" *learns* atom. Then Q_2^{gen} is checked and found to be complete. Because a most specific generalization is always unique (if exists), the system stops.

Differently from the MSG approach, the LSS search algorithm tries to specialize "incomplete" atoms to make them "complete". In this case, the system unifies the query atom $learns(N,L)$ with $learns(N,\text{'English'})$ in tc_5.

The completion succeeds because the atoms in the condition of tc_5, namely $pupil(N,S,C), school(S,\text{'primary'},D)$, are more general than the corresponding atoms in Q_2. The condition of the TC statement had not been more general, then the algorithm would have attempted to unify the condition with the body of the query. In an alternate branch, the algorithm also tries to apply tc_6, but fails because the atom $school(S,\text{'middle'},D)$ does not unify. (It would fail as well if additional atoms were allowed in the specialization, due to the primary key constraint of *school*.)

The proposed approximations Q_2^{gen} and Q_2^{spec1} may introduce new answers or eliminate previous answers over the given database instance. To assess the quality of the approximations, the system returns the numbers of new and eliminated answers, respectively. If one of these numbers is close to the number of answers to the original query, this can be seen as a confirmation that the orginal answer was close to complete.

Query Q_3: "Select the names of all 1st level pupils that attend a school in the Bolzano district and that learn some language":

$$Q_3(N) :- \ pupil(N,S,C), school(S,\text{'primary'},\text{'Bolzano'}) \ class(C,\text{'1'},B), \ learn(N,L).$$

With this example we demonstrate the interaction between FKs and FDCs. On the basis of the given completeness assertions, the system proposes a generalization and a specialization.

For class the following FDC holds:

$$class[scheme] \in \{\text{'fullDay'}, \text{'halfDay'}\}.$$

The reasoner concludes from this together with tc_3 and tc_4 that the database is complete for all classes. It thus comes up with the generalization:

$$Q_3^{\text{gen}}(N) :- \ pupil(N,S,C), school(S,\text{'primary'},\text{'Bolzano'}), class(C,\text{'1'},B).$$

The result count over our example database reveals that Q_3 and Q_3^{gen} return the same number of answers—because every first year pupil in the bilingual province of Bolzano learns either Italian or German as an additional language. Consequently, the answer to Q_3 was already complete.

Analogously to Q_2, the specialization of Q_3 is the query that asks for all first level pupils that learn English. Again, finite domain reasoning is needed to conclude that the database is complete for all classes.

Query Q_4: "Select all language learners":

$$Q_4(N) :- \ learns(N,L).$$

This examples illustrates the effect of FKs and FDCs on generalization and specialization. Under the present completeness assertions, there is no complete generalization of Q_4. However, a generalization would be possible if there were a TC statement asserting completeness for the full *pupil* relation. In this case, the reasoner would return the query asking for all pupils.

If we switch off the foreign key constraints, it finds a specialization of size 1+2, which is the same as query Q_2^{spec1}. With all constraints activated, it returns as the least specific specialization

$$Q_4^{spec1}(N) :- \ learns(N,\text{'English'}).$$

Statements tc_5 and tc_6 guarantee completeness of the *learns* table for all pupils at primary and middle schools. Due to the FKs, for

every *learn* record, there is a corresponding *pupil*, and for every *pupil*, there is a corresponding *school*. Thus, the presence of *pupil* and *school* tuples is guaranteed. Moreover, due to the FDCs, each school is either primary or middle, so all possibilities are covered and $Q_4^{spec1}(N)$ is complete.

3. IMPLEMENTATION TECHNIQUE

The core of the demonstrator is the completeness reasoner. Its basic functionality is to determine whether a query is complete with respect to a set of TC statements. The complexity class of this problem Π_2^P, which is corresponds to the difficulty of problems expressible with Answer Set Programming (ASP). This functionality was already shown with the MAGIK demo [8], which translates a completeness problem instance into a set of disjunctive datalog rules, which are then processed by an ASP engine. Our demo uses MAGIK to determine whether a query is complete or not and extends it to compute generalizations.

To compute generalizations, we rely on an operator $f_{\mathcal{C}}$ underlying the algorithm in MAGIK. This operator exists for any set of TC statements $\mathcal{C}$. When reasoning without integrity constraints, $f_{\mathcal{C}}$ selects from a set of atoms B (e.g., the body of a conjunctive query Q), the subset $f_{\mathcal{C}}(B) \subseteq B$ of those atoms that are guaranteed to be complete by the statements in $\mathcal{C}$ under the hypothesis that *all* atoms in B are complete (which in general is not true). A query $Q(\bar{x}) :- B$ is complete with respect to $\mathcal{C}$ if and only if $f_{\mathcal{C}}(B)$ and B are equivalent. Clearly, the operator $f_{\mathcal{C}}$ is monotonic with respect to set inclusion as as it maps sets of atoms to subsets. It is also semantically monotonic in the sense that $f_{\mathcal{C}}(B)$ is at least as general as B. Repeated application therefore leads to a fixpoint. One can show that this fixpoint is the body of the most specific complete query containing Q if such a query exists. To take into account keys, FKs and FDCs, the operator has to be generalized, but it retains the property that it converges to the most specific generalization.

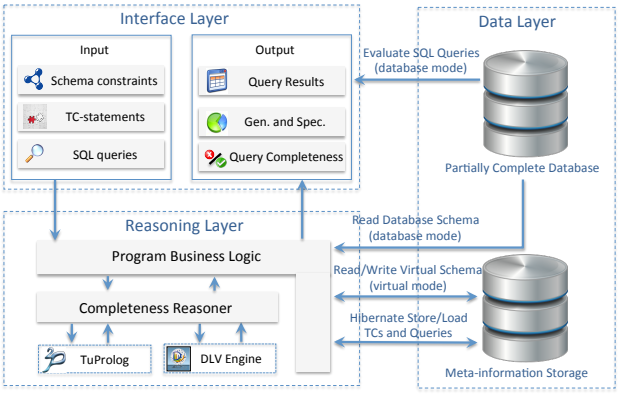
The operator $f_{\mathcal{C}}$ can be expressed by a collection of logic programming rules, where FKs are translated into rules with Skolem functions and FDCs into disjunctive rules. These rules together with a representation of the query are then passed to an ASP solver, which applies cautious reasoning to check if the query is complete.

Specializations are computed using Prolog unification. To specialize a query without adding atoms, each atom of the query is unified with a TC statement. If this succeeds, a more specific, but complete query is obtained. In the following steps, we nondeterministically add fresh atoms and apply the above specialization procedure again until the maximal allowed query size is reached. Prolog unification is used further to filter out the most general specializations.

4. SYSTEM ARCHITECTURE

The demo system is built as a web application written in Java. It is composed of three layers (Figure 3). The *interface layer* is implemented using Java Server Pages that are executed on an Apache-Tomcat Web server. The *reasoning layer* is the core of the system. The *completeness reasoner* encodes the problem of determining completeness and of finding query generalizations by calling the DLV [4] answer set engine. The problem of finding query specializations is encoded into Prolog and executed by TuProlog [1], a Java-friendly Prolog distribution. The *data layer* is stores the meta-information (virtual schemas, TC statements, and queries) using Hibernate as object-relational mapper. Database connections, which are also stored in this database, are used to establish connections with remote or localhost databases. When connecting to

Figure 3: The overall system architecture



a database, the system retrieves the schema and constraint information from the catalog. The connections also are used to execute queries and retrieve answers. At the moment, the system only communicates with PostgreSQL databases.

5. CONCLUSION

We aim to show how one can build a system that supports query answering over a partially complete database for which we have information about the complete parts, as it may arise when integrating data from different sources or generated by different business processes. Given a user query, our system computes queries that are guaranteed to be complete and approximate the user query from above and from below, taking into account keys, foreign keys, and finite domain constraints. The algorithms are implemented leveraging logical programming technology.

The current implementation is intended as a proof of concept. In future work, we will study how to make it scalable.

6. REFERENCES

[1] E. Denti, A. Omicini, and A. Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Science of Computer Programming*, 57(2):217 – 250, 2005.

[2] W. Fan and F. Geerts. Relative information completeness. In *PODS*, pages 97–106, 2009.

[3] W. Fan and F. Geerts. Capturing missing tuples and missing values. In *PODS*, pages 169–178, 2010.

[4] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM TOCL*, 7(3):499–562, 2006.

[5] A. Levy. Obtaining complete answers from incomplete databases. In *Proc. VLDB*, pages 402–412, 1996.

[6] A. Motro. Integrity = Validity + Completeness. *ACM TODS*, 14(4):480–502, 1989.

[7] S. Razniewski and W. Nutt. Completeness of queries over incomplete databases. In *VLDB*, 2011.

[8] O. Savković, P. Mirza, S. Paramonov, and W. Nutt. MAGIK: Managing Completeness of Data. In *CIKM*, pages 2725–2727, 2012.