An ASP Approach to Query Completeness Reasoning

Werner Nutt
Free University of Bozen-Bolzano
(*e-mail:* nutt@inf.unibz.it)

Sergey Paramonov
KU Leuven
(*e-mail:* sergey.paramonov@cs.kuleuven.be)

Ognjen Savković
Free University of Bozen-Bolzano
(*e-mail:* savkovic@inf.unibz.it)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We address the problem to determine whether a query over a partially complete database can be answered completely, which arises in data integration and decision support. Using so-called table completeness statements, one asserts which parts of a database are complete. The question then is whether these are sufficient to retrieve the same answers as if the database had complete information about the domain of application. Previous work in the area of databases has characterized the complexity of the problem, but did not come up with a practical implementation.

In this paper we explore ASP engines as a possible platform to execute completeness reasoning problems. We first generalize the problem by taking into account finite domain constraints and then translate it into rules that may have disjunctions in the heads. The translation allows us to encode completeness problems into cautious reasoning in ASP. We implemented our encoding in two state of the art solvers and tested it on examples that involve many disjunctions, but allow for significant optimizations. It turned out that both engines did not take advantage of the possibilities for optimization.

1 Introduction

In database applications such as information integration and decision support, one is in interested in data sets that are complete, in the sense that the data represent all relevant facts that hold in the real world. In many situations, though, it is only possible to guarantee partial completeness of the data, which means that for certain aspects of the application domain the data are complete, but not for others. In such a situation, one would like to know at least whether the available data are sufficient to answer a given query completely, that is, whether the answers to the query over the avalable data are the same as if the data set were complete.

An example is the management of school data in the province of Bolzano, which motivated the work reported here. Data in the school information system of the province are often incomplete because each school individually is responsible for inserting its data into the system and because for certain kinds of data the contribution is optional. Decision makers, however, need to know

whether or not the statistics on which they base the allocation of resources to schools are derived from complete data.

Motro (1989) was the first to formalize incomplete databases and completeness of queries. Levy (1996), in addition, introduced a format for assertions that say which parts of a relational database are complete. We call these assertions *table completeness* or *TC* statements. He raised the problem to determine whether a set of such TC statements imply that some given query can be answered completely. Razniewski and Nutt (2011b) showed how to reduce this *completeness reasoning problem* to containment of conjunctive queries (Chandra and Merlin 1977) and gave a comprehensive analysis of its complexity, considering several variants of queries and assertions. In particular, they showed that for the most expressive queries and assertions they considered, completeness reasoning is Π_2^P-complete. Such a degree of difficulty is reached, for instance, if queries and assertions are expressed by conjunctive queries and if finite-domain constraints hold over the database (Razniewski and Nutt 2011a). This work, however, did not lend itself immediately to a practical implementation.

When implementing a new reasoning procedure, one may either start from scratch, or map the reasoning problem to an existing formalism, for which implementations exist. Answer Set Programming (ASP) allows one to declaratively express problems in Π_2^P (Leone et al. 2006) and researchers have produced some powerful solvers. This suggests ASP as a promising platform for completeness reasoners.

In this paper we develop an approach to implement completeness reasoning for conjunctive queries and for TC statements that are expressed by means of conjunctive queries. Moreover, we take into account the finite domain constraints that hold over a database. Our approach consists of two steps. First, we develop *syntactic characterizations* of when a collection of TC statments entails the completeness of a query. Then we build upon the characterization to encode every completeness reasoning problem into a logic program such that the resulting program entails a test fact under cautious reasoning if and only if the the original problem has the answer "yes." The two-step approach opens up the possibility to prove the correctness of the encoding. Complete proofs can be found in (Paramonov 2013), but are not included in the paper due to space limitations.

We have experimented with our encoding using two ASP engines, `clasp` (Gebser et al. 2012) and `dlv` (Leone et al. 2006). The tests involve finite domain constraints, which are encoded using disjunction in the head of rules. On the tests, both systems perform similarly: (i) their running times are of the same order of magnitude, and (ii) the running times are exponential in the input size. We argue that both systems forgo opportunities of optimisation, which in one of the tests could already be achieved with straightforward and established optimization techniques.

The remainder of the paper is organized as follows. In Section 2, we recall basic definitions from database theory and fix our notation. Section 3 formally introduces completeness reasoning. In Section 4, we characterize completeness reasoning in the absence of finite domain constraints and present our encoding for this case. In Section 5, characterization and encoding are generalized to take account finite domain constraints. Section 6 reports on our experiments and Section 7 concludes.

2 Preliminaries

Relational Databases and Conjunctive Queries. A database *schema* is a set of relation symbols Σ, each with an arity $ary(R)$. In the following, we assume the schema to be fixed. We assume an infinite set of constants *dom*, the *domain*. For a relation R with arity n, an *atom* is an expression

$R(t_1,\ldots,t_n)$, where $t_1 \ldots t_n$ are either elements of *dom* or variables. We denote constants with lower-case and variables with upper-case letters. A database *instance* D is a finite set of ground atoms. We sometimes refer to the atoms in an instance as facts. For a relation R, we denote as $R(D)$ the set $\{\bar{t} \mid R(\bar{t}) \in D\}$ of all tuples occurring in an R-atom. A *condition* is a set of atoms.

A *conjunctive query* is written as $Q(\bar{X}) \leftarrow B$, where B is a condition and $\bar{X}$ is a tuple of variables, each occuring also in B. We call B the *body* of Q, the variables in $\bar{X}$ the *distinguished variables* and the other variables in B the *nondistinguished variables*. Given a conjunctive query $Q(\bar{X}) \leftarrow B$ and an instance D, an answer to Q is a tuple $\alpha\bar{X}$, where α is an assignment of domain values to all variables such that $\alpha B \subseteq D$. The *set* of all answers to Q over D is written as $Q(D)$.

Finite Domain Constraints. Database systems allow one to formulate conditions, so-called database integrity constraints, that all instances of a database have to satisfy. Many of them can be captured in logic and allow for more inferences when reasoning about instances and queries. In this paper, we consider finite domain constraints.

A *finite domain* (FD) constraint has the form $Dom(R,p,V)$, where R is a relation, $1 \leq p \leq ary(R)$ is an argument position of R, and V is a finite set of constants. An instance D satisfies such a constraint if for every $R(\bar{t}) \in D$ we have that $\bar{t}[p] \in V$, where $\bar{t}[p]$ is the value in the p-th position of $\bar{t}$. We will use $\mathscr{F}$ to denote sets of FD constraints.

Answer Set Programming. We consider extended answer set programs (Gelfond and Lifschitz 1991) allowing for (proper) disjunction in a head of rule:

$$A_1 \mid \ldots \mid A_k \leftarrow B_1,\ldots,B_m$$

where the A's and B's are atoms. A rule with empty head (the empty disjunction) is a *denial*. (We avoid the common term "integrity constraint" in this paper to avoid confusion with integrity constraints over databases.) A fact is a rule with empty body (we omit "$\leftarrow$"). An *answer-set program* is a finite set of such rules. For the theory of answer set programs we refer to (Simons et al. 2002; Baral 2003).

3 Data Completeness

Running Example. Our examples build upon a toy schema from the school world with the three relations

$$pupil(\underline{name},level,code) \qquad class(\underline{level},\underline{code},scheme) \qquad learns(\underline{name},\underline{lang}).$$

Here, $pupil(fred,1,a)$ means that Fred is a pupil in class 1a; $class(1,a,halfDay)$ means that class 1a follows a half-day scheme; and $learns(fred,english)$ means that Fred learns English.

The schema shows attribute names to make the relations more intuitive, although with our notation we refer to the argument positions of relations only by way of numbers. We consider the FD constraints that classes have a level between 1 and 5, expressed as $Dom(class,1,\{1,2,3,4,5\})$, classes have a code among a, b, and c, expressed as $Dom(class,2,\{a,b,c\})$, and that the possible schemes are half-day or full-day, expressed as $Dom(class,3,\{halfDay,fullDay\})$.

The conjunctive query $Q_{1\text{hd}}$, defined by the rule

$$Q_{1\text{hD}}(N) \leftarrow pupil(N,1,C),class(1,C,halfDay) \tag{1}$$

asks for "the names of all pupils that attend a class at level 1 following a half-day scheme".

Query and Table Completeness. When stating that data is incomplete, one must have a conceptual complete reference. We model an *incomplete database* in the style of (Levy 1996) as a pair of database instances $\mathscr{D} = (D^i, D^a)$, where $D^a \subseteq D^i$. Here, D^i is called the *ideal* state and D^a the *available* state. In an application, the state stored in a DBMS is the available state, which represents only a part of the facts that hold in reality. The facts holding in reality constitute the ideal state, which however is unknown. (Later on we will introduce table completeness statements as a way to express meta-information about the extent to which the available state captures the ideal state.)

Data are accessed by posing queries. We would like to know whether a database has sufficient information to answer a query completely, that is, whether the query is *complete.* If we can infer from meta-information that a query is complete, we know that the answer we receive over the available database is the same as the one we would get over the (hypothetical) ideal database.

We write $\mathit{Compl}(Q)$ to denote that Q is complete. This statement is satisfied by an incomplete database $\mathscr{D} = (D^i, D^a)$, written $\mathscr{D} \models \mathit{Compl}(Q)$, if $Q(D^i) = Q(D^a)$.

With *table completeness* (TC) statements we specify that parts of a table are complete. A TC statement, written $\mathit{Compl}(R(\bar{s}); G)$, has two components, a relational atom $R(\bar{s})$ and a condition G. In the sequel, we will denote a TC statement generically as C. As an example, consider

$$C_{\text{hD}} = \mathit{Compl}(\mathit{pupil}(N, L, C); \mathit{class}(L, C, \mathit{halfDay})) \tag{2}$$

$$C_{\text{lev1}} = \mathit{Compl}(\mathit{class}(1, C, S); \mathit{true}). \tag{3}$$

The first statement asserts that the table *pupil* contains all records of pupils attending a half-day class, while the second asserts that the table *class* contains all records of classes at level 1.

In practice, TC statements can be generated in several ways. One source can be business rules. For instance, in the area of Bolzano, vocational schools use the the province school information system to manage grades of pupils. Therefore, grades are complete for pupils from vocational schools. Completeness statements can also be derived from information about the business processes that generate data. For instance, if the administration of a school finishes the registration procedure for the new school year, this can be recorded and tranlated into a completeness statement.

A TC statement C has an *associated query*, which is defined as $Q_C(\bar{s}) \leftarrow R(\bar{s}), G$. The statement C is satisfied by $\mathscr{D} = (D^i, D^a)$, written $\mathscr{D} \models \mathit{Compl}(R(\bar{s}); G)$, if $Q_C(D^i) \subseteq R(D^a)$. Note that the ideal instance D^i is used to determine those tuples in the ideal version $R(D^i)$ that satisfy G and that the statement C is satisfied if these tuples are present in the available version $R(D^a)$. We refer to the query associated to C as Q_C. For instance, the query associated to C_{hD} is $Q_{C_{\text{hd}}}(N, L, C) \leftarrow \mathit{pupil}(N, L, C), \mathit{class}(L, C, \mathit{halfDay})$.

For any set $\mathscr{C}$ of TC statements we define an operator $T_{\mathscr{C}}$ that maps database instances to instances. If C is a TC statement about R, then we define $T_C(D) := \{R(\bar{t}) \mid \bar{t} \in Q_C(D)\}$. For $\mathscr{C}$ we define

$$T_{\mathscr{C}}(D) := \bigcup_{C \in \mathscr{C}} T_C(D). \tag{4}$$

For any instance D, the pair $(D, T_{\mathscr{C}}(D))$ is an incomplete database satisfying $\mathscr{C}$ and $T_{\mathscr{C}}(D)$ is the smallest set (wrt set inclusion) for which this holds.

We say that $\mathscr{C}$ entails the completeness of Q if every incomplete database $\mathscr{D}$ that satisfies the statements in $\mathscr{C}$, also satisfies $\mathit{Compl}(Q)$. In this case, we write $\mathscr{C} \models \mathit{Compl}(Q)$. The *completeness reasoning problem* is to check, given $\mathscr{C}$ and Q, whether the above entailments hold. An instance

of this problem is to check whether $\{C_{\text{hD}}, C_{\text{lev1}}\} \models \mathit{Compl}(Q_{\text{1hD}})$ (which intuitively holds and which we prove to hold in Section 4).

We say that $\mathscr{D} = (D^i, D^a)$ *satisfies* $\mathscr{F}$ if D^i satisfies $\mathscr{F}$. Note that, due to $D^a \subseteq D^i$, this implies that also D^a satisfies $\mathscr{F}$. We say that $\mathscr{C}$ entails the completeness of Q wrt $\mathscr{F}$, and write $\mathscr{C} \models_{\mathscr{F}} \mathit{Compl}(Q)$, if every $\mathscr{D}$ that satisfies both $\mathscr{C}$ and $\mathscr{F}$, also satisfies $\mathit{Compl}(Q)$.

From (Razniewski and Nutt 2011b) we know that the completeness reasoning problem can be reduced to query containment (cf. (Chandra and Merlin 1977; Klug 1988)). Reasoning in the presence of FD constraints can be reduced to containment with respect to FD constraints (Razniewski and Nutt 2011a).

4 Encoding Query Completeness into ASP

In the rest of the paper, we always consider a set of TC statements $\mathscr{C}$ and a conjunctive query Q defined by the rule $Q(\bar{X}) \leftarrow R_1(\bar{t}_1), \ldots, R_n(\bar{t}_n)$.

We want to encode the completeness check "$\mathscr{C} \models \mathit{Compl}(Q)$?" into the question whether a program entails a fact for a boolean test predicate. While developing our approach, we illustrate it with an example. Consider the query Q_{1hD} in (1) and the set $\mathscr{C}_{\text{hD,lev1}} = \{C_{\text{hD}}, C_{\text{lev1}}\}$ comprising the TC statements in (2) and (3). Suppose $\mathscr{D} = (D^i, D^a)$ satisfies $\mathscr{C}_{\text{hD,lev1}}$ and Q_{1hD} returns an answer, say n', over the ideal instance D^i. Then D^i contains two atoms of the form $\mathit{pupil}(n', 1, c')$ and $\mathit{class}(1, c', \mathit{halfDay})$. Now, due to C_{hD}, also D^a contains $\mathit{pupil}(n', 1, c')$, and due to C_{lev1}, the atom $\mathit{class}(1, c', \mathit{halfDay})$ is in D^a, too. Consequently, Q_{1hD} returns n' also over D^a. Since D^i and D^a were arbitrary, this shows that $\mathscr{C}_{\text{hD,lev1}} \models \mathit{Compl}(Q_{\text{1hD}})$.

Using the $T_{\mathscr{C}}$ transformation in (4), we can generalize this approach to a completeness test. We define the set of facts D_Q, which we call the *canonical database* of Q, obtained by freezing the atoms in the body of Q. ("Freezing" variables is a well-known concept in logic programming and database theory, which allows one to treat a variable like a constant.) Thus,

$$D_Q = \{R_1(\theta\bar{t}_1), \ldots, R_n(\theta\bar{t}_n)\}, \tag{5}$$

where θ is the substitution that maps each variable X to the "frozen version" of θX of X. To D_Q we apply $T_{\mathscr{C}}$ and check whether Q can retrieve the frozen tuple of distinguished variables.

Theorem 1 (Characterization)**.** *Let $\mathscr{C}$ be a set of TC statements, and $Q(\bar{X}) \leftarrow B$ be a conjunctive query. Then*

$$\mathscr{C} \models \mathit{Compl}(Q) \quad \Longleftrightarrow \quad \theta\bar{X} \in Q(T_{\mathscr{C}}(D_Q)). \tag{6}$$

This reasoning can be performed by an answer set program for arbitrary Q and $\mathscr{C}$. We start from the set of facts D_Q. Then we extend our signature by two additional relation symbols R^i and R^a for every R in Σ, to be able to reason about the ideal and available instances. We also introduce a *copy rule* $r_R\colon R^i(\bar{X}) \leftarrow R(\bar{X})$ for every relation and denote the set of all such copy rules as P_Σ. Thus, every answer set of D_Q and P_Σ contains an "ideal" copy of D_Q.

Next, we capture the reasoning with TC statements by introducing for each $C \in \mathscr{C}$, $C = \mathit{Compl}(R(\bar{s}); G)$, the datalog rule r_C, defined as:

$$R^a(\bar{s}) \leftarrow R^i(\bar{s}), G^i, \tag{7}$$

where the i in G^i indicates that all relation symbols are replaced by their ideal version. For example, $r_{C_{\text{hD}}}$ is the rule $\mathit{pupil}^a(N, L, C) \leftarrow \mathit{pupil}^i(N, L, C), \mathit{class}^i(L, C, \mathit{halfDay})$ and $r_{C_{\text{lev1}}}$ is the

rule $class^a(1,C,S) \leftarrow class^i(1,C,S)$. We collect all rules for statements in $\mathscr{C}$ in the set

$$P_{\mathscr{C}} = \{r_C \mid C \in \mathscr{C}\}. \tag{8}$$

Intuitively, D_Q is a prototypical instance D where Q returns an answer, namely, the prototypical answer $\theta\bar{X}$. Applying the rules in P_Σ turns D into an ideal database D^i_Q. The application of the rules in $P_{\mathscr{C}}$ then amounts to computing $T_{\mathscr{C}}(D^i_Q)$ (see Theorem 1). It remains to check whether Q returns the answer $\theta\bar{X}$ also over the available instance $T_{\mathscr{C}}(D^i_Q)$.

To this end, we introduce the boolean *test query* Q_s, which is obtained from Q by replacing each relation symbol by its available version and freezing the *distinguished* variables. Formally, Q_s is defined by the rule $r_Q = Q_s \leftarrow R^a_1(\delta\bar{t}_1),\ldots,R^a_n(\delta\bar{t}_n)$, where δ is the substitution that maps every distinguished variable to its frozen version. In our example, the test query is $Q_{1\text{hD}} \leftarrow pupil^a(n',1,C), class^a(1,C,halfDay)$.

We end up with programs $P_{\mathscr{C}}$, encoding $\mathscr{C}$, and $P_Q := D_Q \cup P_\Sigma \cup \{r_Q\}$, encoding Q.

Theorem 2. *Let Q be a conjunctive query and $\mathscr{C}$ be a set of TC statements. Then*

$$\mathscr{C} \models Compl(Q) \quad \Longleftrightarrow \quad Q_s \text{ is in the answer set of } P_{\mathscr{C}} \cup P_Q.$$

The theorem can be proved by formalizing the intuition presented above. Recall that in the definition of the test query, we *freeze* the distinguished variables because we want to test whether Q returns $\theta\bar{X}$, and we do *not freeze* the non-distinguished variables because we do not want to impose constraints on *how* $\theta\bar{X}$ is retrieved. To see why this definition works, consider the query

$$Q_{\text{eng}}(N) \leftarrow learns(N,english), learns(N,L), \tag{9}$$

which asks for "all learners of English, which in addition learn some language" (which may be English). Suppose, our data are complete for all learners of English, expressed by the TC statement C_{eng}, whose rule form is $learns^a(N,english) \leftarrow learns^i(N,english)$. The ideal copy of the canonical database $D_{Q_{\text{eng}}}$ is $D^i_{Q_{\text{eng}}} = \{learns^i(n',english),\ learns^i(n',l')\}$. With our TC rule, we can derive the fact $learns^a(n',english)$, but not $learns^a(n',l')$. Still, this is enough to succeed for our test query $Q^s_{\text{eng}} \leftarrow learns^a(n',english), learns(n',L)$, since the variable L can be bound to *english*. Thus, the check in Theorem 2 returns the intuitively expected answer "yes".

5 Completeness Reasoning with Finite Domain Constraints

Finite domain constraints make it necessary to reason by cases. Consider the query

$$Q_{\text{levEng}}(N,L) \leftarrow pupil(N,L,C), learns(N,english), \tag{10}$$

which asks for the name and the level of pupils learning English. Suppose that our data is complete for all English learners, expressed by the statement C_{eng} from above, and for all pupils at each level from 1 to 5, expressed by the TC statements $C_{\text{pLev}l}$, where $l = 1,\ldots,5$, with rule representation $pupil^a(N,l,C) \leftarrow pupil^i(N,l,C)$. These TC statements alone do not entail completeness of Q_{levEng}, because in principle there could be pupils at other levels. If in addition we know that the only levels are 1 to 5, expressed by the constraint $F_{\text{pLev}} = Dom(pupil,2,\{1,2,3,4,5\})$, we can conclude the completeness of the query.

The test in Theorem 2, though, will not work here, because the level rules $r_{C_{\text{pLev}l}}$ only fire in the presence of $pupil^i$ atoms where the level is one of the constants 1 to 5. Therefore, a reasoning procedure has to instantiate the levels in all ways permitted by the FD constraint before applying the TC rules. If the test query succeeds for all instantiations, then we can conclude completeness.

In the following, we develop concepts that allows us to characterize completeness wrt FD constraints and then provide an encoding of the characteristic condition.

Let $\mathscr{F}$ be a fixed set of FD constraints. We say that a variable Y occurring at position p in an R-atom in Q is *constrained* by $Dom(R,p,V)$ to a value in V. For instance, the variable L in Q_{levEng} is constrained by F_{pLev}. We say that a substitution γ is an $\mathscr{F}$*-case of* Q if γ maps the variables of Q to constants such that (i) $\gamma Y \in F$ whenever Y is constrained to F by some FD constraint in $\mathscr{F}$, and (ii) γY is the frozen version of Y otherwise. For instance, $\{N/n', L/1, C/c'\}$ is an $\mathscr{F}_{\text{pLev}}$-case of Q_{levEng}, where $\mathscr{F}_{\text{pLev}} = \{F_{\text{pLev}}\}$. Note that our definition requires that a variable that is constrained by more than FD constraint has to respect *all* such constraints. By $\Gamma_{\mathscr{F},Q}$ we denote the set of all $\mathscr{F}$-cases of Q. We now generalize Theorem 1 to FD constraints.

Theorem 3 (Characterization FD Constraints)**.** *Let $\mathscr{C}$ be a set of TC statements, $\mathscr{F}$ be a set of FD constraints, and $Q(\bar{X}) \leftarrow B$ be a conjunctive query. Then*

$$\mathscr{C} \models_{\mathscr{F}} Compl(Q) \quad \Longleftrightarrow \quad \gamma\bar{X} \in Q(T_{\mathscr{C}}(\gamma B)) \text{ for every case } \gamma \text{ in } \Gamma_{\mathscr{F},Q}. \tag{11}$$

Condition (11) stipulates a test comprising the following steps: (1) Instantiate the query body B, which is a set of atoms, in all ways possible permitted by the FD constraints. This amounts to considering all possible ways in which an answer to Q can be retrieved. (2) For each such instantiation γB, compute $T_{\mathscr{C}}(\gamma B)$, the set of atoms that must be present in any available database if the ideal database contains γB. (3) Evaluate Q over $T_{\mathscr{C}}(\gamma B)$ and check whether the result contains, $\gamma\bar{X}$, the prototypical answer to Q over γB. We now show how to encode (1) $\mathscr{F}$-cases, (2) the operator $T_{\mathscr{C}}$, and (3) the final test.

The starting point is again the canonical database D_Q. Since D_Q consists of ground atoms, which cannot be instantiated, we have to mimic the instantiation by cases. We introduce a new binary predicate *val*, where intuitively $val(t,v)$ indicates that the term t is instantiated by the value v. We want to keep the encodings of Q, $\mathscr{F}$, and $\mathscr{C}$ independent from each other. Therefore, no information as to which variable can be instantiated by which value should influence the encoding of Q. To achieve this, we introduce for each term in D_Q, both for original constants and for frozen variables, the fact $val(t,t)$. The set of all such facts is denoted as Val_Q. Then we require that every term can have at most one *val*-value other than itself, which can be seen as a requirement for *val* to be functional. This can be expressed by the denial r_{fun} below:

$$\leftarrow val(X,Y), val(X,Z), X \neq Y, X \neq Z, Y \neq Z. \tag{12}$$

To mimic the instantiation of the query body by cases, we introduce for each constraint $F = Dom(R,p,\{a_1,\ldots,a_m\})$ the disjunctive rule r_F as

$$val(X_p,a_1) \mid \cdots \mid val(X_p,a_m) \leftarrow R^i(X_1,\ldots,X_p,\ldots X_n), \tag{13}$$

which nondeterministically binds the term in position p to one of the values in $\{a_1,\ldots,a_m\}$. We collect the rules encoding $\mathscr{F}$ into the program $P_{\mathscr{F}} := \{r_F \mid F \in \mathscr{F}\} \cup \{r_{\text{fun}}\}$.

We keep the rules P_Σ that copy D_Q into the ideal database D^i_Q. To make the TC rules applicable to facts with *val*-bindings, we need to unfold them. For an atom $A = R(t_1,\ldots,t_n)$, the *unfolding* consists of the atom $A^u = R(Y_1,\ldots,Y_n)$, where each argument is replaced by a fresh variable, and the set of *val*-bindings $U^A = \{val(Y_1,t_1),\ldots,val(Y_n,t_n)\}$. We have already translated a TC statement C into the rule $r_C\colon R^a(t_1,\ldots,t_n) \leftarrow R^i(t_1,\ldots,t_n), G^i$. The *unfolded rule* for C is then

$$r^u_C\colon \quad R^a(Y_1,\ldots,Y_n) \leftarrow R^i(Y_1,\ldots,Y_n), (G^i)^u, U^{R^i(t_1,\ldots,t_n)}, U^{G^i}, \tag{14}$$

where the unfolding $(G^i)^u$ of the condition G^i and the set of value atoms U^{G^i} are defined analogously to the case of single atoms. For example the unfolded rule for C_{hD} is

$$r^u_{C_{\text{hD}}}: \quad pupil^a(N_1,L_1,C_1) \leftarrow pupil^i(N_1,L_1,C_1), val(N_1,N), val(L_1,L), val(C_1,C), \\ class^i(L_2,C_2,S_2), val(L_2,L), val(C_2,C), val(S_2,halfDay).$$

The program consisting of the unfolded rules is denoted as $P^u_{\mathscr{C}} = \{r^u_C \mid C \in \mathscr{C}\}$.

Let $Q(X_1,\ldots,X_n) \leftarrow B$ be the query which we want to check for completeness. We have to modify the test query in two ways, first by unfolding, and second by adding *val*-atoms to encode the check whether Q returns $\gamma\bar{X}$ over $T_{\mathscr{C}}(\gamma B)$. Thus, the rule $r_{Q^u_s}$ for the test query Q^u_s is

$$Q^u_s \leftarrow (B^a)^u, U^{B^a}, val(x_1,X_1),\ldots,val(x_n,X_n), \tag{15}$$

where the x_j are the frozen versions of the X_j. For example, the test query for $Q_{1\text{hD}}$ in (1) is

$$Q^u_{1\text{hD}} \leftarrow pupil^a(N_1,L_1,C_1), val(N_1,N), val(n',N), val(L_1,1), val(C_1,C), \\ class^a(L_2,C_2,S_2), val(L_2,1), val(C_2,C), val(S_2,halfDay),$$

where n' is the frozen version of N. Let $P^u_Q := D^i_Q \cup Val_Q \cup \{r_{Q^u_s}\}$ be the set of rules about Q.

Theorem 4. *Let Q be a conjunctive query, $\mathscr{C}$ a set of TC statements, and $\mathscr{F}$ a set of FD constraints. Then*

$$\mathscr{C} \models_{\mathscr{F}} Compl(Q) \quad \Longleftrightarrow \quad Q^u_s \text{ is in every answer set of } P_{\mathscr{C}} \cup P_{\mathscr{F}} \cup P^u_Q.$$

6 Experiments

In the absence of finite domain constraints, completeness is NP-complete. In this case, only one answer set exists and checking query completeness boils down to applying TC rules in all possible ways to the ideal canonical database D_Q in order to satisfy the body of the Q_s rule.

Finite domain (FD) constraints make the problem more difficult, raising the complexity to Π^P_2-completeness. This is reflected by the disjunctive rules in our encoding, which introduce multiple answer sets. Our FD constraints are independent of each other. Therefore, the number of answer sets to be checked for an instance is equal to the product of the sizes of the finite domains involved. Consequently, when considering a sequence of problems where the number of FD constraints grows linearly, the number of answers sets grows exponentially. In principle one cannot avoid generating exponentially many answer sets, due to the Π^P_2-completeness. However, in many cases a problem is structured in such a way that the FD constraints can be elaborated independently of each other, or that they can be ignored completely.

To check whether state-of-the-art ASP solvers are taking advantage of such possibilities for optimization, we set up two test cases for completeness reasoning in the presence of FD constraints: the first case allows one to work on each constraint in isolation and the second can be solved without considering FD constraints at all. We ran them using two state-of-the-art solvers: `dlv` and `clasp` on a laptop machine with a 2.4 GHz processor and 8 GB of RAM.

Test 1. We wanted to see whether the reasoners can identify that some disjunctive rules are independent from each other and treat them consecutively. For the test, we checked the query

$$Q_1(N) \leftarrow pupil(N,L,C), learns(N,Lang), class(L,C,halfDay) \tag{16}$$

in the presence of finite domain constraints for the attributes *level* and *code* of *pupil* and *lang* of *learns*. We varied the FD constraints so that the cardinality of *level* and *code* ranged between 1

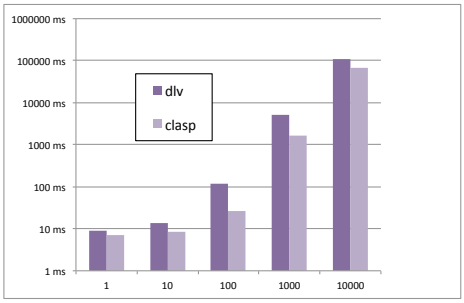

Fig. 1. Test 1

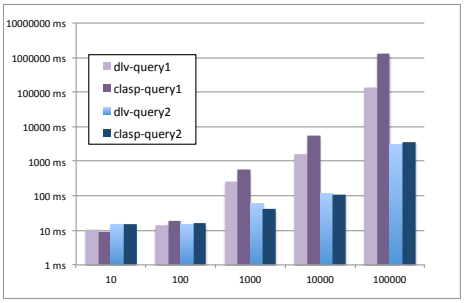

Fig. 2. Test 2

and 10, the cardinality of *lang* ranged between 10 and 100, and their product ranged from 10^1 to 10^5. For each possible *level* $\$l$ and *code* $\$c$ we had a TC rule $pupil^a(N,\$l,\$c) \leftarrow pupil^i(N,\$l,\$c)$, while for each possible *lang* value $\$lg$, we had the rule $learns^a(N,\$lg) \leftarrow learns^i(N,\$lg)$. In addition, we had the rule $class^a(L,C,S) \leftarrow class^i(L,C,S)$. The first type of rule says, for instance, that we are complete for all pupils of level $\$l$ and code $\$c$.

We ran the completeness test for cardinalities with products 10^i, $1 \leq i \leq 5$. Conceptually, according to Theorem 3, for such a test 10^i many cases have to be checked, that is, 10^i many answer sets. In principle, one could derive the completeness of Q_1 by independently verifying completeness for each atom in the query. The cardinalities above were chosen such that no more than 100 cases per atom had to be considered. Figure 1 shows that the execution time grows proportionally to the number of answer sets. This suggests that all possible instantiations were performed and checked and the possible optimization did not take place.

Test 2. Next, we wanted to see whether the reasoners were able to identify disjunctive rules that are irreleveant for a completeness check.

We used again query Q_1 from above, but simplified our TC rules. There were only three rules, which state completeness of each of the tables *pupil*, *learns*, and *class*, that is, $pupil^a(N,L,C) \leftarrow pupil^i(N,L,C)$, $learns^a(N,Lang) \leftarrow learns^i(N,Lang)$, and $class^a(L,C,S) \leftarrow class^i(L,C,S)$. Again, we added FD constraints for *level*, *code* and *lang*. Similar to the first test, we varied the FD constraints so that their cardinality ranged between 1 and 100 and their product ranged from 10^1 to 10^5. As before, 10^i many answer sets had to be checked in principle. However, an optimization should be easier here, since each of the TC rules is applicable without firing any of the disjunctive finite domain rules. A system that applies such rules lazily should show a running time polynomial in the size of the input, which consisted of the three TC rules and three disjunctive rules with a maximum of 100 atoms in the head (see rule pattern (13)).

Surprisingly, Figure 2 shows that now the running time grows even faster than the number of answer sets, both for `clasp` and `dlv`. To understand whether the joins in Q_1 prevented an optimization, we ran the test with the join-free query $Q_2(N) \leftarrow pupil(N,L,C)$, where we modified the FD constraints on *level* and *code* so that again between 10^1 to 10^5 answer sets had to be considered. As shown in Figure 2, the times grew more slowly, but still exponentially.

With the straightforward optimization of first attempting to solve a problem by deterministic rules and to resort to non-deterministic rules only if necessary, this instance could have been solved in polynomial time. We conclude that none of the two systems implements such an optimization.

7 Conclusion

Previous work on query completeness led to a theoretical framework in which the central concepts and the reasoning tasks could be defined, a practical way to realise completeness reasoners was missing, though. With our encoding of completeness reasoning tasks into disjunctive logic programs, we have opened the possibility to implement such reasoners by harnessing existing answer set engines. We built a demonstrator system, MAGIK, that realizes our approach, which is publically accessible on the Web[1] (Savković et al. 2012).

Our current work indicates that ASP is also helpful for additional functionalities beyond mere yes/no answers of a completeness checker. By analysing the answer sets generated by a query check, we can find out which parts of a database would need to be completed to guarantee completeness of the query.

Our performance tests, however, showed that our encoding into ASP is not necessarily scalable. ASP engines have difficulties with combinations of disjunctions, since apparently they do not exploit situations where a conclusion can be reached without applying a disjunctive rule at all. We needed disjunctions to reason about finite domains. A challenge for the ASP community that may arise from this work is to provide better support for finite domain reasoning.

Acknowledgements: This work was partially supported by the ESF project 2-299-2010 "Schul-Informationssystem – Wir verbinden Menschen" and by the project "Managing Completeness of Data (MAGIC)", funded by the province of Bolzano.

References

BARAL, C. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA.

CHANDRA, A. AND MERLIN, P. 1977. Optimal implementation of conjunctive queries in relational databases. In *Proc. 9th STOC*.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Multi-threaded ASP solving with clasp. *TPLP 12,* 4-5, 525–545.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

KLUG, A. 1988. On conjunctive queries containing inequalities. *J. ACM 35,* 1, 146–160.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM TOCL 7,* 3, 499–562.

LEVY, A. 1996. Obtaining complete answers from incomplete databases. In *Proc. VLDB*. 402–412.

MOTRO, A. 1989. Integrity = Validity + Completeness. *ACM TODS 14,* 4, 480–502.

PARAMONOV, S. 2013. Query completeness—A logic programming approach. Tech. Rep. KRDB13-2, KRDB Research Center, Free Univ. Bozen-Bolzano. http://www.inf.unibz.it/krdb/pub/tech-rep.php.

RAZNIEWSKI, S. AND NUTT, W. 2011a. Checking query completeness over incomplete data. Tech. Rep. KRDB11-2, KRDB Research Center, Free University of Bozen-Bolzano.

RAZNIEWSKI, S. AND NUTT, W. 2011b. Completeness of queries over incomplete databases. In *VLDB*.

SAVKOVIĆ, O., MIRZA, P., PARAMONOV, S., AND NUTT, W. 2012. MAGIK: Managing Completeness of Data. In *CIKM*. 2725–2727.

SIMONS, P., NIEMELÁ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artif. Intell. 138,* 1-2 (June), 181–234.

[1] `http://magik-demo.inf.unibz.it`