Tractable Query Answering and Optimization for Extensions of Weakly-Sticky Datalog±

Mostafa Milani and **Leopoldo Bertossi**

Carleton University, School of Computer Science, Ottawa, Canada
{mmilani,bertossi}@scs.carleton.ca

Summary. We consider a semantic class, *weakly-chase-sticky* (WChS), and a syntactic subclass, *jointly-weakly-sticky* (JWS), of *Datalog*$^\pm$ programs. Both extend that of weakly-sticky (WS) programs, which appear in our applications to data quality. For WChS programs we propose a practical, polynomial-time query answering algorithm (QAA). We establish that the two classes are closed under magic-sets rewritings. As a consequence, QAA can be applied to the optimized programs. QAA takes as inputs the program (including the query) and semantic information about the "finiteness" of predicate positions. For the syntactic subclasses JWS and WS of WChS, this additional information is computable.

Datalog$^\pm$ **.** *Datalog*, a rule-based language for query and view-definition in relational databases [5], is not expressive enough to logically represent interesting and useful ontologies, at least of the kind needed to specify conceptual data models. *Datalog*$^\pm$ extends *Datalog* by allowing existentially quantified variables in rule heads ($\exists$-variables), equality atoms in rule heads, and program constraints [2]. Hence the "+" in *Datalog*$^\pm$, while the "−" reflects syntactic restrictions on programs, for better computational properties.

A typical *Datalog*$^\pm$ program, $\mathcal{P}$, is a finite set of rules, $\Sigma \cup E \cup N$, and an extensional database (finite set of *facts*), D. The rules in Σ are *tuple-generating-dependencies* (*tgds*) of the form $\exists \bar{x} P(\bar{x}, \bar{x}') \leftarrow P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n)$, where $\bar{x}' \subseteq \bigcup \bar{x}_i$, and $\bar{x}$ can be empty. E is a set of *equality-generating-dependencies* (*egds*) of the form $x = x' \leftarrow P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n)$, with $\{x, x'\} \subseteq \bigcup \bar{x}_i$. Finally, N contains *negative constraints* of the form $\bot \leftarrow P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n)$, where $\bot$ is false.

Example 1. The following *Datalog*$^\pm$ program shows a tgd, an egd, and a negative constraint, in this order: $\exists x \ \mathit{Assist}(y, x) \leftarrow \mathit{Doctor}(y)$; $\quad x = x' \leftarrow \mathit{Assist}(y, x),\ \mathit{Assist}(y, x')$; $\quad \bot \leftarrow \mathit{Specialist}(y, x, z),\ \mathit{Nurse}(y, z)$. □

Below, when we refer to a class of *Datalog*$^\pm$ programs, we consider only Σ, the tgds. Due to different syntactic restrictions, *Datalog*$^\pm$ can be seen as a class of sublanguages of *Datalog*$^\exists$, which is the extension of *Datalog* with tgds with $\exists$-variables [10].

The rules of a *Datalog*$^\pm$ program can be seen as an ontology $\mathcal{O}$ on top of D, which can be *incomplete*. $\mathcal{O}$ plays the role of: (a) a "query layer" for D, providing ontology-based data access (OBDA) [9], and (b) the specification of a completion of D, usually carried out through the *chase* mechanism that, starting from D, iteratively enforces the rules in Σ, generating new tuples. This leads to a possibly infinite instance extending D, denoted with $\mathit{chase}(\Sigma, D)$.

The answers to a conjunctive query $\mathcal{Q}(\bar{x})$ from D wrt. Σ is a sequence of constants $\bar{a}$, such that $\Sigma \cup D \models \mathcal{Q}(\bar{a})$ (or *yes* or *no* in case $\mathcal{Q}$ is boolean). The answers can be obtained by querying as usual the *universal* instance $chase(\Sigma, D)$. The chase may be infinite, which leads, in some cases, to undecidability of query answering [8]. However, in some cases where the chase is infinite, query answering (QA) is still computable (decidable), and even tractable in the size of D. Syntactic classes of *Datalog*$^{\pm}$ programs with tractable QA have been identified and investigated, among them: *sticky* [4, 7], and *weakly-sticky* [4] *Datalog*$^{\pm}$ programs.

Our Need for QA Optimization. In our work, we concentrate on the *stickiness* and *weak-stickiness* properties, because these programs appear in our applications to quality data specification and extraction [11], with the latter task accomplished through QA, which becomes crucial.

Sticky programs [4] satisfy a syntactic restriction on the multiple occurrences of variables (joins) in the body of a *tgd*. Weakly-sticky (WS) programs form a class that extends that of sticky programs [4]. WS-*Datalog*$^{\pm}$ is more expressive than sticky *Datalog*$^{\pm}$, and results from applying the notion of *weak-acyclicity* (WA) as found in data exchange [6], to relax acyclicity conditions on stickiness. More precisely, in comparison with sticky programs, WS programs require a milder condition on join variables, which is based on a program's *dependency graph* and the positions in it with finite rank [6].[1]

For QA, sticky programs enjoy *first-order rewritability* [7], i.e. a conjunctive query $\mathcal{Q}$ posed to $\Sigma \cup D$ can be rewritten into a new first-order (FO) query $\mathcal{Q}'$, and correctly answered by posing $\mathcal{Q}'$ to D, and answering as usual. For WS programs, QA is *PTIME*-complete in data, but the polynomial-time algorithm provided for the proof in [4] is not a practical one.

Stickiness of the Chase. In addition to (syntactic) stickiness, there is a "semantic" property of programs, which is relative to the chase (and the data, D), and is called "chase-stickiness" (ChS). Stickiness implies semantic stickiness (but not necessarily the other way around) [4]. For chase-sticky programs, QA is tractable [4].

Intuitively, a program has the chase-stickiness property if, due to the application of a tgd σ: When a value replaces a repeated variable in the body of a rule, then that value also appears in all the head atoms obtained through the iterative enforcement of applicable rules that starts with σ. So, that value is propagated all the way down through all the possible subsequent steps.

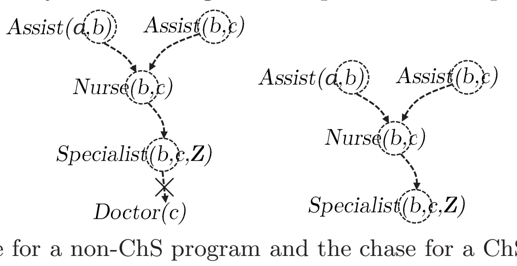

Fig. 1. The chase for a non-ChS program and the chase for a ChS program, resp.

[1] A position refers to a predicate attribute, e.g. *Nurse*[2].

Example 2. Consider $D = \{Assist(a,b), Assist(b,c)\}$, and the following set, Σ_1, of tgds: $Nurse(y,z) \leftarrow Assist(x,y), Assist(y,z)$; $\exists z\ Specialist(x,y,z) \leftarrow Nurse(x,y)$; $Doctor(y) \leftarrow Specialist(x,y,z)$. Σ_1 is not ChS, as the chase on the LHS of Figure1 shows: value b is not propagated all the way down to $Doctor(c)$. However, program Σ_2, which is Σ_1 without its third rule, is ChS, as shown on the RHS of Figure1. □

Weak-Stickiness of the Chase. Weak-stickiness also has a semantic version, called "weak-chase-stickiness" (WChS); which is implied by the former. So as for chase-stickiness, weak-chase-sticky programs have a tractable QA problem, even with a possibly infinite chase. This class is one of the two we introduce and investigate. They appear in double-edged boxes in Figure 2, with dashed edges indicating a semantic class.

By definition, weak-chase-stickiness is obtained by relaxing the condition for ChS: it applies only to values for repeated variables in the body of σ that appear in so-called *infinite positions*, which are semantically defined. A position is infinite if there is an instance D for which an unlimited number of different values appear in $Chase(\Sigma, D)$.

Given a program, deciding if a position is infinite is unsolvable, so as deciding in general if the chase terminates. Consequently, it is also undecidable if a program is WChS. However, there are syntactic conditions on programs [6, 12] that determine some (but not necessarily all) the finite positions. For example, the notion of position *rank*, based on the program's *dependency graph*, are used in [6, 4] to identify a (sound) set of finite positions, those with *finite rank*. Furthermore, finite-rank positions are used in [4] to define weakly-sticky (WS) programs as a syntactic subclass of WChS.

Finite Positions and Program Classes. In principle, any set-valued function S that, given a program, returns a subset of the program's finite positions can be used to define a subclass WChS(S) of WChS. This is done by applying the definition of WChS above with "infinite positions" replaced by "non-S-finite positions". Every class WChS(S) has a tractable QA problem.

S could be computable on the basis of the program syntax or not. In the former case, it would be a "syntactic class". Class $WChS(S)$ grows monotonically with S in the sense that if $S_1 \subseteq S_2$ (i.e. S_1 always returns a subset of the positions returned by S_2), then $WChS(S_1) \subseteq WChS(S_2)$. In general, the more finite positions are (correctly) identified (and the consequently, the less finite positions are treated as infinite), the more general the subclass of WChS that is identified or characterized.

For example, the function $S^{\perp}$ that always returns an empty set of finite positions, $WChS(S^{\perp})$ is the class of sticky programs, because stickiness must hold no matter what the (in)finite positions are. At the other extreme, for function $S^{\top}$ that returns all the (semantically) finite positions, $WChS(S^{\top})$ becomes the class WChS. (As mentioned above, $S^{\top}$ is in general uncomputable.) Now, if S^{rank} returns the set of finite-rank positions (for a program $\mathcal{P}$, usually denoted by $\Pi_F(\mathcal{P})$ [6]), $WChS(S^{rank})$ is the class of WS programs.

Joint-Weakly-Stickiness. The *joint-weakly-sticky* (JWS) programs we introduce form a syntactic class strictly between WS and WChS. Its definition appeals to the notions of *joint-acyclicity* and *existential dependency graphs* introduced in [12]. Figure 2 shows this syntactic class, and the inclusion relationships between classes of *Datalog*$^{\pm}$ programs.[2]

If S^{ext} denotes the function that specifies finite positions on the basis of the *existential dependency graphs* (EDG), implicitly defined in [12], the JWS class is, by definition, the class $\mathit{WChS}(S^{ext})$. EDGs provide a finer mechanism for capturing (in)finite positions in comparison with positions ranks (defined through dependency graphs): $S^{rank} \subseteq S^{ext}$. Consequently, the class of JWS programs, i.e. $\mathit{WChS}(S^{ext})$, is a strict superclass of WS programs, i.e. $\mathit{WChS}(S^{rank})$.[3]

QAA for WChS. Our query answering algorithm for WChS programs is parameterized by a (sound) finite-position function S as above. It is denoted with AL^S, and takes as input Σ, D, query $\mathcal{Q}$, and $S(\Sigma)$, which is a subset of the program's finite positions (the other are treated as infinite by default).

The customized algorithm AL^S is guaranteed to be sound and complete only when applied to programs in $\mathit{WChS}(S)$: $AL^S(\Sigma, D, \mathcal{Q})$ returns all and only the query answers. (Actually, AL^S is still sound for any program in WChS.) AL^S runs in polynomial-time in data; and can be applied to both the WS and the JWS syntactic classes. For them the finite-position functions are computable.

AL^S is based on the concepts of *parsimonious chase* (*pChase*) and *freezing nulls*, as used for QA with *shy Datalog*, a fragment of *Datalog*$^{\exists}$ [10]. At a *pChase* step, a new atom is added only if a homomorphic atom is not already in the chase. Freezing a null is promoting it to a constant (and keeping it as such in subsequent chase steps). So, it cannot take (other) values under homomorphisms, which may create new *pChase* steps. Resumption of the *pChase* means freezing *all* nulls, and continuing *pChase* until no more *pChase* steps are applicable.

Query answering with shy programs has a first phase where the *pChase* runs until termination (which it does). In a second phase, the *pChase* iteratively resumes for a number of times that depends on the number of distinct $\exists$-variables in the query. This second phase is required to properly deal with joins in the query. Our QAA for WChS programs (AL) is similar, it has the same two phases, but a *pChase* step is modified: after every application of a *pChase* step that generates nulls, the latter that appear in S-finite positions are immediately frozen.

Magic-Sets Rewriting. It turns out that JWS, as opposed to WS, is closed under the quite general magic-set rewriting method [5] introduced in [1]. As a

[2] Rectangles with dotted-edges show semantic classes, and double-edged rectangles show the classes introduced in this work. Notice that programs in semantic classes include the instance D, but syntactic classes are data-independent (for any instance as long as the syntactic conditions apply).

[3] The JWS class is different from (and incomparable with) the class of *weakly-sticky-join* programs (WSJ) introduced in [3], which extends the one of WS programs with consideration that are different from those used for JWS programs. WSJ generalizes WS on the basis of the weakly-sticky-join property of the chase [3, 4] and is related to repeated variables in single atoms.

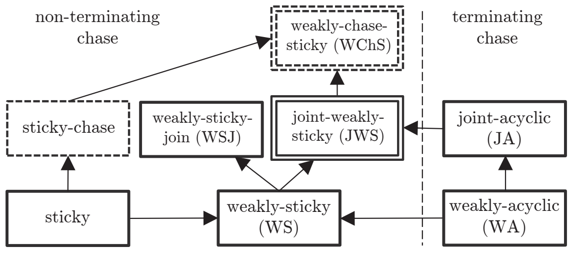

Fig. 2. Generalization relationships among program classes.

consequence, AL can be applied to both the original JWS program and its magic rewriting. (Actually, this also holds for the superclass WChS.)

It can be proved that (our modification of) the magic-sets rewriting method in [1] does not change the character of the original finite or infinite positions. The specification of (in)finiteness character of positions in magic predicates is not required by AL, because no new nulls appear in them during the AL execution. As consequence, the MS method rewriting can be perfectly integrated with our QAA, introducing additional efficiency.

Acknowledgments: We are very grateful to Mario Alviano and the DLV team for providing us with information and support in relation to existential Datalog. We also appreciate useful conversations with Andrea Cali and Andreas Pieris on Datalog±, and important comments from Andrea Cali on an earlier version of this paper.

References

1. M. Alviano, N. Leone, M. Manna, G. Terracina and P. Veltri. Magic-Sets for Datalog with Existential Quantifiers. *Proc. Datalog 2.0*, 2012, pp. 31-43.
2. A. Cali, G. Gottlob, and T. Lukasiewicz. Datalog±: A Unified Approach to Ontologies and Integrity Constraints. *Proc. ICDT*, 2009, pp. 14-30.
3. A. Cali, G. Gottlob and A. Pieris. Query Answering under Non-Guarded Rules in Datalog+/-. *Proc. RR*, 2010, pp. 1-17.
4. A. Cali, G. Gottlob, and A. Pieris. Towards more Expressive Ontology Languages: The Query Answering Problem. *Artificial Intelligence*, 2012, 193:87-128.
5. S. Ceri, G. Gottlob and L. Tanca. *Logic Programming and Databases.* Springer, 1990.
6. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 2005, 336:89-124.
7. G. Gottlob, G. Orsi, and A. Pieris. Query Rewriting and Optimization for Ontological Databases. *ACM TODS*, 2014, 39(3):25.
8. D. S. Johnson, and A. Klug. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *Proc. PODS*, 1984, pp. 164-169.
9. M. Lenzerini. Ontology-Based Data Management. Proc. AMW 2012, CEUR Proceedings, Vol. 866, pp. 12-15.
10. N. Leone, M. Manna, G. Terracina, and P. Veltri. Efficiently Computable Datalog$^\exists$ Programs. *Proc. KR*, 2012, pp. 13-23.
11. M. Milani, L. Bertossi and S. Ariyan. Extending Contexts with Ontologies for Multi-dimensional Data Quality Assessment. *Proc. DESWeb*, 2014, pp. 242-247.
12. S. Rudolph, and M. Krötzsch. Extending Decidable Existential Rules by Joining Acyclicity and Guardedness. *Proc. IJCAI*, 2011, pp. 963-968.