Datalog Queries Distributing over Components

Tom J. Ameloot[*1], Bas Ketsman[†1], Frank Neven[1], and Daniel Zinn[2]

1 **Hasselt University & transnational University of Limburg**
Martelarenlaan 42, Hasselt, Belgium
`firstname.lastname@uhasselt.be`

2 **LogicBlox, Inc**
1416 NW 46th St., Suite 301, Seattle, WA 98103, USA
`daniel.zinn@logicblox.com`

Abstract

We investigate the class $\mathcal{D}$ of queries that distribute over components. These are the queries that can be evaluated by taking the union of the query results over the connected components of the database instance. We show that it is undecidable whether a (positive) Datalog program distributes over components. Additionally, we show that connected Datalog$^\neg$ (the fragment of Datalog$^\neg$ where all rules are connected) provides an effective syntax for Datalog$^\neg$ programs that distribute over components under the stratified as well as under the well-founded semantics. As a corollary, we obtain a simple proof for one of the main results in previous work [19], namely, that the classic win-move query is in $\mathcal{F}_2$ (a particular class of coordination-free queries).

1998 ACM Subject Classification H.2.3 Query Languages, H.2.4 Distributed databases

Keywords and phrases Datalog, stratified semantics, well-founded semantics, coordination-free evaluation, distributed databases

Digital Object Identifier 10.4230/LIPIcs.ICDT.2015.308

1 Introduction

A Datalog program is called connected when the graph of every rule is connected; here, the graph of a rule views the variables of the rule as vertices and each positive body atom as a hyperedge. For instance, the canonical program computing the transitive closure of a binary relation

$$TC(x,y) \leftarrow E(x,y)$$
$$TC(x,y) \leftarrow E(x,z), TC(z,y)$$

is connected, while the program

$$A(x,y) \leftarrow E(x,z), E(y,z')$$

is not as $E(x,z)$ and $E(y,z')$ do not share a common variable. The definition of connectedness can also be extended to Datalog$^\neg$ (with negation), where the negative body atoms of a rule do not contribute to the graph of this rule. While connected Datalog programs are very natural, as a logic they are not well-understood. The earliest reference to connected Datalog is by Guessarian [11] who obtained a decidability result for boundedness of a subclass of

[*] Postdoctoral Fellow of the Research Foundation – Flanders (FWO)
[†] PhD Fellow of the Research Foundation – Flanders (FWO)

18th International Conference on Database Theory (ICDT'15).
Editors: Marcelo Arenas and Martín Ugarte; pp. 308–323

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

connected Datalog programs. Ameloot et al. [4] obtained that every connected stratified Datalog$^\neg$ program distributes over components, that is, the program can be evaluated by taking the union of the query results over the connected components of the database instance (cf. Section 3 for a formal definition). We denote by $\boldsymbol{\mathcal{D}}$ the class of queries that distribute over components.

In this paper, we investigate the relationship between connected Datalog$^\neg$ and the class $\boldsymbol{\mathcal{D}}$. This investigation is motivated by a general theme in model theory that considers the relationship between syntactic and semantic properties of logic. In the context of Datalog, results of this type have for example been obtained for the class of queries preserved under homomorphisms, denoted by $\mathcal{H}$. For instance, Feder and Vardi [10] showed that all queries in semi-positive Datalog$^\neg$ that are preserved under homomorphisms can already be expressed in Datalog itself. That is, semi-positive Datalog$^\neg \cap \mathcal{H}$ = Datalog. Dawar and Kreutzer [9] showed that the latter result can not be extended to least fixed-point logic (LFP): LFP $\cap \mathcal{H} \not\subseteq$ Datalog. The main result of this paper is that both under the stratified as well as under the well-founded semantics, we have connected Datalog$^\neg$ = Datalog$^\neg \cap \boldsymbol{\mathcal{D}}$. Additionally, we show that, even when we forbid negation in rules, it is undecidable whether a given (positive) Datalog program is in $\boldsymbol{\mathcal{D}}$. Our main result therefore shows that connected Datalog$^\neg$ is an effective syntax for queries in Datalog$^\neg \cap \boldsymbol{\mathcal{D}}$ (both under the stratified and under the well-founded semantics).

Apart from the model theoretic motivation mentioned above, the results in this paper also provide more insight in some of the recent results concerning coordination-free evaluation. Datalog has attracted quite a bit of attention as a declarative programming language for distributed systems, see e.g. [14, 1, 16]. In fact, Hellerstein [12] argues that the theory of declarative database query languages can provide a foundation for the next generation of parallel and distributed programming languages. In this respect, programs (queries) are specified on a logical level over a global schema and are computed by multiple computing nodes over which the input database is distributed. These nodes can perform local computations and communicate asynchronously with each other via messages. The model operates under the assumption that messages can never be lost but can be arbitrarily delayed. As the global barriers raised by the need for synchronization are an inherent source of inefficiency in such systems, a number of researchers started investigating classes of queries that can be evaluated in a coordination-free manner [8, 19, 5, 3, 4]. In a coordination-free evaluation, communication between nodes can only transfer data and can not be used to coordinate.[1] Zinn, Green, and Ludäscher [19] introduced various classes of coordination-free queries: $\boldsymbol{\mathcal{F}}_0$, $\boldsymbol{\mathcal{F}}_1$, and $\boldsymbol{\mathcal{F}}_2$. Membership of the classical non-monotonic win-move query in $\boldsymbol{\mathcal{F}}_2$ is one of the main results in [19], where the authors describe a distributed query evaluation strategy that is specific for the win-move query. The results in this paper provide a more in-depth explanation of that result. Indeed, letting $\boldsymbol{\mathcal{V}}$ denote the class of so-called *value-driven* queries that have nonempty output only on inputs containing values, we explain that every query in $\boldsymbol{\mathcal{D}} \cap \boldsymbol{\mathcal{V}}$ is also in the class $\boldsymbol{\mathcal{F}}_2$. This implies that every connected Datalog$^\neg$ program in $\boldsymbol{\mathcal{V}}$ is in $\boldsymbol{\mathcal{F}}_2$ as well. Since win-move is a connected Datalog$^\neg$ program, and is value-driven, it then follows immediately that win-move is in $\boldsymbol{\mathcal{F}}_2$.

In this paper, we also briefly discuss *semi-connected* Datalog$^\neg$ programs, a relaxation of connected Datalog$^\neg$ introduced in previous work [4]: under the well-founded semantics, the queries expressible by semi-connected Datalog$^\neg$ programs remain in the class $\boldsymbol{\mathcal{F}}_2$.

[1] We refer to [5, 4] for a formal definition of coordination-freeness.

Outline. This paper is organized as follows. Section 2 presents preliminaries on databases, and on Datalog$^\neg$ together with its stratified and well-founded semantics. Section 3 recalls distribution over components; additionally, we present an undecidability result and we discuss the relationship with weaker forms of monotonicity. Next, Section 4 discusses a syntactic restriction of Datalog$^\neg$, called *connected Datalog*$^\neg$; and, we show that under the stratified semantics, this restriction captures the queries that both distribute over components and that are expressible in Datalog$^\neg$. This capturing result is subsequently generalized in Section 5 to the well-founded semantics. Section 6 briefly discusses how a relaxation of the connectedness restriction behaves under the well-founded semantics. We conclude in Section 7. Many of the technical proofs are moved to the appendix due to space limitations. Proof sketches are given in the main body.

2 Preliminaries

2.1 Database Basics

A *(database) schema* σ is a finite set of pairs (R, k), also denoted as $R^{(k)}$, where R is a relation name and $k \in \mathbb{N}$ its associated arity. We assume an infinite universe **dom** of atomic data values. A *fact* is a pair $(R, \bar{a})$, also denoted as $R(\bar{a})$, where R is a relation name and $\bar{a}$ is a (possibly empty) tuple of values over **dom**. We say that fact $R(a_1, \ldots, a_k)$ is *over* database schema σ if $R^{(k)} \in \sigma$. If $k = 0$ then the fact is called *nullary*. A *(database) instance* I over σ is a finite set of facts over σ. The *active domain* of a fact $\boldsymbol{f}$, denoted $adom(\boldsymbol{f})$, is the set of values occurring in $\boldsymbol{f}$. For an instance I, we define $adom(I) = \bigcup_{\boldsymbol{f} \in I} adom(\boldsymbol{f})$. For a subset $\sigma' \subseteq \sigma$, we write $I|_{\sigma'}$ to denote the maximal subset of I that is over σ'.

A *query* $\mathcal{Q}$ *over input schema* σ_1 *and output schema* σ_2 is a function that maps instances over σ_1 to instances over σ_2. We consider only queries that are *generic*: these queries are independent of the concrete data values. More formally, a query $\mathcal{Q}$ is generic if for all inputs I, and all permutations ρ of **dom**, we have $\rho(\mathcal{Q}(I)) = \mathcal{Q}(\rho(I))$.

2.2 Datalog with Negation

We recall here the language Datalog with negation [2], denoted Datalog$^\neg$.

Atoms and rules

We assume a separate universe **var** of variables. An *atom* is a pair $(R, \bar{u})$, also denoted as $R(\bar{u})$, where R is a relation name and $\bar{u}$ is a (possibly empty) tuple of variables over **var**. We say that an atom $R(u_1, \ldots, u_k)$ is *over* a database schema σ if $R^{(k)} \in \sigma$. A *rule* φ is a tuple $(head_\varphi, pos_\varphi, neg_\varphi)$ where $head_\varphi$ is an atom, and pos_φ and neg_φ are both sets of atoms. We call $head_\varphi$ the *head*; and we call pos_φ and neg_φ respectively the *positive body atoms* and the *negative body atoms*. We only consider rules φ where each variable in $head_\varphi$ and neg_φ also occurs in pos_φ. We say that φ is *over* a database schema σ when all its atoms are over σ. A rule φ may also be written in the conventional syntax, e.g., when $head_\varphi = T(\bar{u})$, $pos_\varphi = \{R_1(\overline{u_1}), \ldots, R_m(\overline{u_m})\}$ and $neg_\varphi = \{S_1(\overline{v_1}), \ldots, S_n(\overline{v_n})\}$ then we may write φ as:

$$T(\bar{u}) \leftarrow R_1(\overline{u_1}), \ldots, R_m(\overline{u_m}), \neg S_1(\overline{v_1}), \ldots, \neg S_n(\overline{v_n}).$$

The ordering of atoms to the right of the arrow is arbitrary.

A *valuation* for rule φ is a function V that maps each variable in φ to a value in **dom**. Applying V to atoms of φ results in facts: we substitute each variable u by $V(u)$. We say

that V is *satisfying for φ on an instance I*, when $V(pos_\varphi) \subseteq I$ and $V(neg_\varphi) \cap I = \emptyset$. In that case, the pair (φ, V) is said to *derive* the fact $V(head_\varphi)$ on instance I.

Programs

A *Datalog program with negation over a schema σ* is a set P of rules over σ. The class of such programs is denoted by Datalog$^\neg$. For a Datalog$^\neg$ program P, we also write $sch(P)$ to denote the (minimal) schema that P is over. We define $idb(P) \subseteq sch(P)$ as the relations of $sch(P)$ that appear in rule heads. We also define $edb(P) = sch(P) \setminus idb(P)$.[2] Intuitively, $edb(P)$ can be seen as the input relations for P. Various semantics can be given to Datalog$^\neg$ programs. In this paper we use the stratified semantics and the well-founded semantics.

2.3 Stratified Semantics

We recall the stratified semantics of Datalog$^\neg$ [2].

Semi-positive programs

We call a Datalog$^\neg$ program P *semi-positive* when its rules only apply negation to relations in $edb(P)$. More formally, for all rules φ in P, the set neg_φ is over $edb(P)$. The semantics of such a program P can be defined as follows. Consider the following function T_P, called the *(inflationary) immediate consequence operator* of P: T_P maps any instance J over $sch(P)$ to $J \cup A$ where $A = \{V(head_\varphi) \mid \varphi \in P, V \text{ is a satisfying valuation for } \varphi \text{ on } J\}$. Now, for an input I over $edb(P)$, consider the following infinite sequence of instances: $I_0, I_1, I_2, \ldots$ where $I_0 = I$ and $I_i = T_P(I_{i-1})$ for all $i \geq 1$. Because T_P only adds facts, and is limited to $adom(I)$, there is an index k such that $I_k = I_{k+1}$, i.e., there is a fixpoint. The output of P on I is defined as this fixpoint.

Syntactic stratification

Let P be a Datalog$^\neg$ program. We call P *syntactically stratifiable* (or simply *stratified*) if we can partition the rules of P into a sequence of Datalog$^\neg$ subprograms $P_1, \ldots, P_n$ such that:

- Rules with the same head relation occur in the same subprogram;
- In each subprogram P_i, relations R of positive body atoms either belong to $edb(P)$ or all rules computing R must be in some subprogram P_j with $j \leq i$; and,
- In each subprogram P_i, relations R of negative body atoms either belong to $edb(P)$ or all rules computing R must be in some subprogram P_j with $j < i$.

Each subprogram is also called a *stratum*. Note that negation is only applied to relations computed in strictly lower strata. So, each stratum by itself is a semi-positive program. Given a syntactic stratification $P_1, \ldots, P_n$, the output of P on an input I over $edb(P)$, denoted $P(I)$, is defined as $P_n(P_{n-1}(\ldots(P_1(I))\ldots))$, i.e., we first apply stratum P_1, then stratum P_2, etc. All syntactic stratifications give the same result [2].

We say that a query $\mathcal{Q}$ with input schema σ_1 and output schema σ_2 is *computed by a stratified Datalog$^\neg$ program* P if for all inputs I for $\mathcal{Q}$, we have $\mathcal{Q}(I) = P(I)|_{\sigma_2}$ using the stratified semantics of P.

[2] The abbreviations "idb" and "edb" respectively stand for *intensional schema* and *extensional schema*.

2.4 Well-founded Semantics

Let P be a Datalog$^\neg$ program. We define the well-founded semantics of P using the *alternating fixpoint computation* [18].

Negation on assumptions

Let φ be a rule in P, and let J be an instance over $sch(P)$. A valuation V for φ is said to be *J-neg-satisfying for φ on an instance I* if $V(pos_\varphi) \subseteq I$ and $V(neg_\varphi) \cap J = \emptyset$. In contrast to the semantics of semi-positive programs from above, J-neg-satisfaction tests negative body atoms against the fixed database instance J. Next, consider the following function T_P^J, called the *(inflationary) immediate consequence operator of P with assumptions J*: function T_P^J maps each instance K over $sch(P)$ to $K \cup A$ where $A = \{V(head_\varphi) \mid \varphi \in P,\ V$ is a J-neg-satisfying valuation for φ on $K\}$. Now, for an instance I over $sch(P)$, consider the following infinite sequence of instances: $I_0, I_1, I_2, \ldots$ where $I_0 = I$ and $I_i = T_P^J(I_{i-1})$ for all $i \geq 1$. Because T_P^J only adds facts, and is limited to $adom(I)$, there is an index k such that $I_k = I_{k+1}$, i.e., there is a fixpoint, denoted as $\hat{T}_{P,I}^J$.

Antimonotone operator

Let I be an input over $edb(P)$. Let $\Gamma_{P,I}$ denote the function that maps each instance J over $sch(P)$ to $\hat{T}_{P,I}^J$. Note that for any two instances J_1 and J_2 with $J_1 \subseteq J_2$, the J_2-neg-satisfying valuations are also J_1-neg-satisfying, so $\Gamma_{P,I}(J_2) \subseteq \Gamma_{P,I}(J_1)$. For this reason, we call $\Gamma_{P,I}$ *antimonotone*. We similarly see that $\Gamma_{P,I}(\Gamma_{P,I}(J_1)) \subseteq \Gamma_{P,I}(\Gamma_{P,I}(J_2))$, so function $\Gamma_{P,I} \circ \Gamma_{P,I}$ is monotone. Next, consider the following infinite sequence of instances: $I_0, I_1, I_2, \ldots$ where $I_0 = \emptyset$ and $I_i = \Gamma_{P,I}(I_{i-1})$ for all $i \geq 1$.[3] Since $\Gamma_{P,I} \circ \Gamma_{P,I}$ is monotone, the subsequence $I_0, I_2, I_4, \ldots$ converges to a fixpoint, i.e., there is a number $k \geq 0$ such that $I_{2k} = I_{2k+2}$.[4] This implies that index $2k+1$ is a fixpoint for the subsequence with uneven indices, i.e., $I_{2k+1} = I_{2k+3}$. Now, the *well-founded semantics of P on I* produces both a set of *true* facts and a set of *true or undefined* facts, denoted as $P_{\mathrm{t}}(I)$ and $P_{\mathrm{t}\vee\mathrm{u}}(I)$ respectively. Formally, they are defined as $P_{\mathrm{t}}(I) = I_{2k}$ and $P_{\mathrm{t}\vee\mathrm{u}}(I) = I_{2k+1}$. Intuitively, for the facts in $P_{\mathrm{t}\vee\mathrm{u}}(I) \setminus P_{\mathrm{t}}(I)$, we can not compute whether they are definitely present in the output or definitely absent from the output. So, the well-founded semantics essentially assigns one of three truth values to facts over $sch(P)$: true, false, and undefined.

We say that a query $\mathcal{Q}$ with input schema σ_1 and output schema σ_2 is *computed by P under the well-founded semantics* if for all inputs I for $\mathcal{Q}$, we have $\mathcal{Q}(I) = P_{\mathrm{t}}(I)|_{\sigma_2}$.

▶ **Example 1.** We recall the well-known *win-move* Datalog$^\neg$ program P [2]:

$$win(x) \leftarrow move(x, y), \neg win(y).$$

The win-move program represents a game as follows. The input relation *move* is viewed as a graph. A game on this graph starts with one node x of the graph marked with a flag. Next, two players, called 1 and 2, take turns to move the flag from the currently flagged node to one of its successor nodes, and player 1 always gets the first turn. A player looses when

[3] There is an alternating fixpoint: the inner fixpoint is given by $\Gamma_{P,I}(J) = \hat{T}_{P,I}^J$, the outer fixpoint is obtained by iterating $\Gamma_{P,I}$. Applying $\Gamma_{P,I}$ to an underestimate yields an overestimate and vice versa.

[4] To see this, we start with $I_0 = \emptyset \subseteq I_2$. Next, since $\Gamma_{P,I} \circ \Gamma_{P,I}$ is monotone, we have $I_2 = \Gamma_{P,I} \circ \Gamma_{P,I}(I_0) \subseteq \Gamma_{P,I} \circ \Gamma_{P,I}(I_2) = I_4$. This reasoning can be repeated to see $I_4 \subseteq I_6$, etc. Since derived facts are restricted to $adom(I)$, we eventually arrive at a fixpoint.

there are no successor nodes during his or her turn. Now, we say that player 1 *has a winning strategy at node* x, if player 1 can always force a win when starting at node x. That is, no matter how player 2 moves, eventually, player 1 will move the flag to a node where player 2 cannot move anymore.

The relation *win* computes the nodes for which player 1 has a winning strategy. For example, letting $\sigma = \{win^{(1)}\}$, on the input $I = \{move(a,b), move(b,a), move(a,c)\}$, we have $P_{\mathrm{t}}(I)|_\sigma = P_{\mathrm{t}\vee\mathrm{u}}(I)|_\sigma = \{win(a)\}$; the winning strategy for player 1 is to move the flag from a to c. As another example, consider the instance $J = I \cup \{move(c,d)\}$. We have $P_{\mathrm{t}}(J)|_\sigma = \{win(c)\}$ and $P_{\mathrm{t}\vee\mathrm{u}}(J)|_\sigma = \{win(a), win(b), win(c)\}$. Facts in $P_{\mathrm{t}\vee\mathrm{u}}(J) \setminus P_{\mathrm{t}}(J)$ represent drawn positions, that is, neither player can force a win and the game goes on indefinitely. The absence of $win(d)$ indicates that player 1 has no winning strategy at node d (because player 1 can not make a move there).

The win-move program is non-monotone: $win(a) \in P_{\mathrm{t}}(I)$ but $win(a) \notin P_{\mathrm{t}}(J)$. ◀

3 Distribution over Components

We recall distribution over components [4]. We call an instance J *connected* if for all values $a, b \in adom(J)$, there exists a sequence $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_n$ of facts in J such that $a \in adom(\boldsymbol{f}_1)$, $b \in adom(\boldsymbol{f}_n)$, and $adom(\boldsymbol{f}_i) \cap adom(\boldsymbol{f}_{i-1}) \neq \emptyset$ for all $i \in \{2, \ldots, n\}$. Possibly $n = 1$. Intuitively, any two values are connected by at least one chain of facts, where subsequent facts share at least one value.[5]

Now, for an instance I, we call a subinstance $J \subseteq I$ a *component* of I if *(i)* J includes all nullary facts of I; *(ii)* J is connected and is maximal with this property in I. This implies that $adom(J) \cap adom(I \setminus J) = \emptyset$. We write $co(I)$ to denote the set of components of I.[6] For example, the components of $I = \{R(a,b), R(b,c), S(c), T(d), U(\,)\}$ are $\{R(a,b), R(b,c), S(c), U(\,)\}$ and $\{T(d), U(\,)\}$.

We say that a query $\mathcal{Q}$ *distributes over components* if for all inputs I for $\mathcal{Q}$ we have $\mathcal{Q}(I) = \bigcup_{J \in co(I)} \mathcal{Q}(J)$, i.e., the centralized output of $\mathcal{Q}$ on I is precisely obtained when we parallelize $\mathcal{Q}$ over the components of I. Let $\boldsymbol{\mathcal{D}}$ denote the class of queries that distribute over components.

3.1 Undecidability

To gain additional insight into distribution over components, we consider decidability of this semantical property for the concrete setting of Datalog$^\neg$. First, we call a Datalog$^\neg$ program *positive* if its rules contain no negative body atoms. To denote the class of such programs, we simply write 'Datalog'. For evaluating Datalog programs, we will assume the intuitive semantics of semi-positive Datalog$^\neg$ programs (cf. Section 2.3). Interestingly, despite the restriction,

▶ **Theorem 2.** *Membership in $\boldsymbol{\mathcal{D}}$ is undecidable for queries computable by Datalog programs.*

Proof. First, from previous work by Shmueli [17], we know that it is impossible to decide whether two Datalog programs P_1 and P_2, each with a single non-nullary output relation, are equivalent. This problem was shown to be undecidable by a reduction from equivalence

[5] Equivalently, one could demand that the Gaifman graph of J is connected, where we view $adom(J)$ as the set of vertices and each fact of J as a hyperedge.

[6] The nullary facts are given to each component because there is no natural preference for how to distribute these facts to any particular component.

of context-free grammars. We point out that this reduction actually constructs *connected* programs (see Section 4.1 for a formal definition). So, equivalence, and thus containment, of two connected Datalog programs, each with a single non-nullary output relation, is undecidable. Our proof below reduces this latter containment problem to deciding whether a Datalog program distributes over components.

Let P_1 and P_2 be two connected Datalog programs with the same *edb* schema σ_1 and each having one k-ary intended output relation, denoted A_1 and A_2 respectively, where $k \geq 1$. Both programs may use auxiliary *idb* relations, but for convenience we assume that $idb(P_1)$ and $idb(P_2)$ have no relation names in common. We define the following auxiliary program P', where T and S are relation names not yet used in P_1 and P_2, and all variables are assumed to be pairwise different:

$$T(\bar{u}) \leftarrow A_1(\bar{u}), S(z).$$
$$T(\bar{u}) \leftarrow A_2(\bar{u}).$$

Now consider the program $P^* = P_1 \cup P_2 \cup P'$. Note that $edb(P^*) = \sigma_1 \cup \{S^{(1)}\}$. Although program P^* is positive, it is *not* connected due to the first rule of P'. The S-atom plays the role of a guard: relation A_1 flows into relation T if S is nonempty. Let $\mathcal{Q}$ be the query computed by P^* over output schema $\sigma_2 = \{T^{(k)}\}$. To finish the proof, we show that $\mathcal{Q} \in \boldsymbol{\mathcal{D}}$ if and only if P_1 is contained in P_2.

Suppose that P_1 is contained in P_2. First, we define program P^c as P^* but without the first rule of P'. Note that P^c is connected. For any input I over $edb(P^*)$, since $A_1(\bar{a}) \in P_1(I)$ implies $A_2(\bar{a}) \in P_2(I)$ by containment, we have $P^c(I)|_{\sigma_2} = P^*(I)|_{\sigma_2} = \mathcal{Q}(I)$. And since P^c is a connected program, we can apply Proposition 6 (in Section 4.2), to know $\mathcal{Q} \in \boldsymbol{\mathcal{D}}$.

Suppose that P_1 is not contained in P_2. In particular, there is some input I over σ_1 for which there is a tuple $\bar{a}$ with $A_1(\bar{a}) \in P_1(I)$ and $A_2(\bar{a}) \notin P_2(I)$. Letting d be a new value outside $adom(I)$, we define the instance $I' = I \cup \{S(d)\}$. During the computation of $P^*(I')$, the first rule of subprogram P' has access to $A_1(\bar{a})$ and $S(d)$, giving $T(\bar{a}) \in P^*(I')$. But, the first rule of P' can never be satisfied on any $J \in co(I')$, because by choice of value d, component J does not simultaneously contain non-nullary facts over σ_1 and $\{S^{(1)}\}$. Moreover, for any $J \in co(I')$, we have $A_2(\bar{a}) \notin P_2(J)$: since program P_2 is unaware of S-facts, we have $A_2(\bar{a}) \notin P_2(I')$ and monotonicity of P_2 implies that $A_2(\bar{a})$ can not be produced on any subset of I'. Overall, we have $T(\bar{a}) \notin P^*(J)$ for all $J \in co(I')$. Hence, $\mathcal{Q} \notin \boldsymbol{\mathcal{D}}$. ◀

We now readily observe that:

▶ **Corollary 3.** *Membership in $\boldsymbol{\mathcal{D}}$ is undecidable for queries computable by stratified Datalog$^\neg$ programs.*

3.2 Weaker Forms of Monotonicity

We briefly relate $\boldsymbol{\mathcal{D}}$ to the classes $\boldsymbol{\mathcal{M}}_{\text{distinct}}$ and $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$ [4]. The class $\boldsymbol{\mathcal{M}}_{\text{distinct}}$ consists of the *domain-distinct-monotone* queries: for such queries $\mathcal{Q}$, we have $\mathcal{Q}(I) \subseteq \mathcal{Q}(I \cup J)$ for all instances I and J where each $\boldsymbol{f} \in J$ satisfies $adom(\boldsymbol{f}) \nsubseteq adom(I)$.[7] Intuitively, $\mathcal{Q}$ behaves monotonically when adding facts that contain at least one new value.

We observe that $\boldsymbol{\mathcal{D}} \nsubseteq \boldsymbol{\mathcal{M}}_{\text{distinct}}$: over a schema $\{R^{(1)}, S^{(2)}\}$, consider the query $\mathcal{Q}_1 = R - \pi_1(S)$, where π_1 projects onto the first component. To see $\mathcal{Q}_1 \in \boldsymbol{\mathcal{D}}$, note that when

[7] This implies that J contains no nullary facts.

forming components, the S-facts are always grouped together with those R-facts they subtract from. To see $\mathcal{Q}_1 \notin \boldsymbol{\mathcal{M}}_{\text{distinct}}$, consider the instances $I = \{R(a)\}$ and $J = \{S(a,b)\}$; note that $\mathcal{Q}_1(I) \not\subseteq \mathcal{Q}_1(I \cup J)$.

We also observe that $\boldsymbol{\mathcal{M}}_{\text{distinct}} \not\subseteq \boldsymbol{\mathcal{D}}$: over a schema $\{R^{(1)}, S^{(1)}\}$, consider the query $\mathcal{Q}_2$ that computes the cross product $T = R \times S$. Query $\mathcal{Q}_2$ is monotone, hence $\mathcal{Q}_2 \in \boldsymbol{\mathcal{M}}_{\text{distinct}}$. To see $\mathcal{Q}_2 \notin \boldsymbol{\mathcal{D}}$, consider the instance $I = \{R(a), S(b)\}$. On the full input I, query $\mathcal{Q}_2$ produces $T(a,b)$, but this fact is not produced on component $\{R(a)\}$ nor on component $\{S(b)\}$.

Next, the class $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$ consists of the *domain-disjoint-monotone* queries: for such queries $\mathcal{Q}$, we have $\mathcal{Q}(I) \subseteq \mathcal{Q}(I \cup J)$ for all instances I and J where J contains no nullary facts and $adom(I) \cap adom(J) = \emptyset$.[8]

We observe that $\boldsymbol{\mathcal{D}} \not\subseteq \boldsymbol{\mathcal{M}}_{\text{disjoint}}$: over a schema $\{R^{(1)}\}$, take the query $\mathcal{Q}_3$ that outputs true (in a nullary relation T) if $R = \emptyset$. We see that $\mathcal{Q}_3 \in \boldsymbol{\mathcal{D}}$: if there are multiple components in an input I then each component contains one R-fact, implying that relation T remains empty on each component, giving the same result as the centralized execution $\mathcal{Q}_3(I)$. Also, $\mathcal{Q}_3 \notin \boldsymbol{\mathcal{M}}_{\text{disjoint}}$, because on the instances $I = \emptyset$ and $J = \{R(a)\}$, we have $\mathcal{Q}(I) = \{T(\,)\} \not\subseteq \mathcal{Q}(I \cup J) = \emptyset$.

We also observe that $\boldsymbol{\mathcal{M}}_{\text{disjoint}} \not\subseteq \boldsymbol{\mathcal{D}}$: take the same query $\mathcal{Q}_2$ from above. Since $\boldsymbol{\mathcal{M}}_{\text{distinct}} \subseteq \boldsymbol{\mathcal{M}}_{\text{disjoint}}$, we have $\mathcal{Q}_2 \in \boldsymbol{\mathcal{M}}_{\text{disjoint}}$. But $\mathcal{Q}_2 \notin \boldsymbol{\mathcal{D}}$ as shown above.

When we exclude queries like $\mathcal{Q}_3$, the remaining queries of $\boldsymbol{\mathcal{D}}$ are included in $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$. Formally, we call a query $\mathcal{Q}$ *value-driven* if for all inputs I for $\mathcal{Q}$ with $adom(I) = \emptyset$, we have $\mathcal{Q}(I) = \emptyset$. Intuitively, the query produces nothing in absence of values. Let $\boldsymbol{\mathcal{V}}$ denote the class of such queries. We observe that $\boldsymbol{\mathcal{D}} \cap \boldsymbol{\mathcal{V}} \subseteq \boldsymbol{\mathcal{M}}_{\text{disjoint}}$: for a query $\mathcal{Q} \in \boldsymbol{\mathcal{D}} \cap \boldsymbol{\mathcal{V}}$, *(i)* for an input I with $adom(I) = \emptyset$, we have $\mathcal{Q}(I) = \emptyset \subseteq \mathcal{Q}(I \cup J)$ for all instances J; and *(ii)* for an input I with $adom(I) \neq \emptyset$, and an instance J without nullary facts and with $adom(I) \cap adom(J) = \emptyset$, we have $co(I) \subseteq co(I \cup J)$, so using $\mathcal{Q} \in \boldsymbol{\mathcal{D}}$, we see $\mathcal{Q}(I) = \bigcup_{K \in co(I)} \mathcal{Q}(K) \subseteq \bigcup_{L \in co(I \cup J)} \mathcal{Q}(L) = \mathcal{Q}(I \cup J)$.

4 Connected Datalog

We can not decide for queries computed by stratified Datalog$^\neg$ programs whether they distribute over components (Corollary 3). However, in this section we show there is a fragment of stratified Datalog$^\neg$ that captures precisely the queries of $\boldsymbol{\mathcal{D}}$ expressible in stratified Datalog$^\neg$.

4.1 Connected Syntax

We recall the language of *connected Datalog*$^\neg$, denoted con-Datalog$^\neg$ [4]. We extend the definition in this previous work, however, to explicitly deal with nullary relations. Nullary relations allow more programming flexibility, and they allow boolean computation in absence of input values, e.g., when only a set of nullary facts is given.

As notational convenience, for an atom $\boldsymbol{a}$ we write $var(\boldsymbol{a})$ to denote the set of variables occurring in $\boldsymbol{a}$. Also, for a rule φ we write $var(\varphi)$ to denote the set of variables in φ. Now, very similarly to connected database instances, we say that a rule φ is *connected* when for any two variables $u, v \in var(\varphi)$ there is a sequence of atoms $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ in pos_φ such that

[8] The queries in $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$ are conceptually similar to the first order sentences preserved under closed extensions, studied by Compton [7].

$u \in var(\boldsymbol{a}_1)$, $v \in var(\boldsymbol{a}_n)$, and $var(\boldsymbol{a}_i) \cap var(\boldsymbol{a}_{i-1}) \neq \emptyset$ for all $i \in \{2, \ldots, n\}$. Possibly $n = 1$. Negative body atoms do not contribute to the connectedness of a rule. Note that rules without variables are always connected.

Next, for a Datalog$^\neg$ program P, we say that nullary relations of $edb(P)$ are *global (for all components)* because the nullary input facts are given to all components by definition. Similarly, we say a nullary relation of $idb(P)$ is *global* if all its rules, and the rules of the idb-relations it depends on, do not use variables.[9] So, the term "global" means that these nullary relations will have the same contents on every component. Also, we say that a nullary relation $S^{(0)} \in idb(P)$ is *value-detecting* when *(i)* for each non-nullary relation $R^{(k)} \in edb(P)$, program P contains a rule isomorphic to '$S(\,) \leftarrow R(u_1, \ldots, u_k)$', using pairwise distinct variables; and, *(ii)* there are no other rules for S in P. Such value-detecting relations can only be used to see if the input contains values.

Now, we say that a Datalog$^\neg$ program P is *connected* when

1. every rule of P is connected by itself; and,
2. the only nullary relations used in rule bodies are global or value-detecting.

Nullary relations that are neither global nor value-detecting may still be used for directly representing output.

Note that the win-move program from Example 1 is connected. Below we consider other examples of connected programs.

▶ **Example 4.** Consider the following semi-positive con-Datalog$^\neg$ program P with $edb(P) = \{A^{(1)}, B^{(1)}, R^{(2)}\}$:

$$reach(x) \leftarrow A(x).$$
$$reach(y) \leftarrow reach(x), R(x, y).$$
$$T(x) \leftarrow reach(x), \neg B(x).$$

Thinking of relation R as edges of a graph, this program computes all nodes reachable from set A but outside set B. ◀

▶ **Example 5.** As an example using nullary relations, here is a stratified con-Datalog$^\neg$ program P, with $edb(P) = \{R^{(2)}, S^{(0)}, T^{(0)}, U^{(1)}\}$, whose meaning is discussed below:

$$xor(\,) \leftarrow S(\,), \neg T(\,).$$
$$xor(\,) \leftarrow \neg S(\,), T(\,).$$
$$V(x) \leftarrow xor(\,), U(x).$$
$$V(y) \leftarrow V(x), R(x, y).$$
$$values(\,) \leftarrow R(x, y).$$
$$values(\,) \leftarrow U(x).$$
$$W(\,) \leftarrow \neg values(\,), xor(\,).$$

Suppose that V and W are the output relations. For the nullary relations of $idb(P)$, note that *xor* is global, *values* is value-detecting, and W is neither global nor value-detecting (and hence may not be used in rule bodies). In the presence of values, again thinking of relation R as edges of a graph, program P finds in relation V the nodes reachable from U on condition that the exclusive or $S \oplus T$ is true. In absence of values, P outputs $S \oplus T$ in relation W.

[9] Focusing on $idb(P)$, a relation R *depends on* another relation S if there is a path from R to S in the so-called *dependency graph* of P, where the relations of $idb(P)$ are the vertices and there is an edge from a relation A to a relation B if a rule with head relation A uses B in its body (either positively or negatively).

Note that V and W are never simultaneously nonempty (although they can be simultaneously empty). The output behavior strongly depends on the presence or absence of values; value-detecting relations are needed to achieve this effect. ◀

4.2 Results

We recall the following result [4]:

▶ **Proposition 6.** Every query computable by a stratified con-Datalog$^\neg$ program distributes over components.

The sketch below provides an intuitive understanding.

Proof (sketch). The positive body atoms in connected rules are all strung together. This way, connected rules can only combine facts from the same component, causing derived non-nullary facts to be connected to their originating component. Essentially, a con-Datalog$^\neg$ program derives facts "inside" components. So, the program does not notice when we separate the components.

For completeness, we also discuss the details of nullary relations. First, note that a value-detecting relation S is nonempty on the entire (nondistributed) input if and only if relation S is nonempty on *all* individual components: this property is trivially true when there is only one component; and, when there is more than one component, they each have non-nullary facts.

Next, since each component contains by definition all nullary input facts, nullary facts derived purely from nullary input facts can be seen as "global flags". These global flags may be injected into per-component computations (as represented by rules with variables).[10]

Lastly, nullary relations that are neither global nor value-detecting, can be seen as per-component flags. The syntactic restriction prevents using such flags in further computation. Without this restriction, per-component flags could be combined in a cross-component fashion, preventing distribution over components. ◀

Within stratified Datalog$^\neg$, a new result is that the converse direction also holds:

▶ **Proposition 7.** Every query computable by a stratified Datalog$^\neg$ program, and distributing over components, can be computed by a stratified con-Datalog$^\neg$ program.

Proof (sketch). Let $\mathcal{Q}$ be a query computable by a stratified Datalog$^\neg$ program P, with the additional assumption that $\mathcal{Q}$ distributes over components. Let σ_1 and σ_2 denote the input and output schema of $\mathcal{Q}$. The proof is constructive: we transform P into a stratified con-Datalog$^\neg$ program $\alpha(P)$ such that for all inputs I over σ_1, we have $\alpha(P)(I)|_{\sigma_2} = \mathcal{Q}(I)$, i.e., $\alpha(P)$ also computes $\mathcal{Q}$. The main idea behind $\alpha(P)$ is that it uses connected rules to separate the original computation over the components (as sketched for Proposition 6). Concretely, $\alpha(P)$ is defined as a union of four subprograms: $\alpha(P) = P^{\uparrow} \cup P^{\#} \cup P^{\downarrow} \cup P^{\text{null}}$. In particular, there are two parts: subprogram $(P^{\uparrow} \cup P^{\#} \cup P^{\downarrow})$ is executed in case there are values in the input, and otherwise the subprogram P^{null} is executed (on just the nullary input facts). Roughly speaking, the order $P^{\uparrow}$, $P^{\#}$, P^{null}, $P^{\downarrow}$ aligns with a syntactic stratification for $\alpha(P)$. Below we explain each subprogram in turn. We also provide an illustration in Example 8. As notation, for any schema σ, we define the extended schema $\#(\sigma) = \{R^{(k+1)}_{\#} \mid R^{(k)} \in \sigma\}$.

[10] This usage is illustrated by Example 5.

First, $P^{\uparrow}$ transforms the input instance I over σ_1 to its *component-extended version* over $\#(\sigma_1)$, denoted $\#(I)$: each original input fact is tagged at the front with the "identifier" of the component it belongs to.[11] However, we have no choice-mechanism that allows us to pick just one value for this identifier; hence, each fact is tagged with *all* values occurring in its component. To illustrate, if $I = \{R(a), R(c), S(a,b), T(\,)\}$, having two components $\{R(a), S(a,b), T(\,)\}$ and $\{R(c), T(\,)\}$, then $P^{\uparrow}$ produces $\#(I) = \{R_{\#}(a,a), R_{\#}(b,a), R_{\#}(c,c), S_{\#}(a,a,b), S_{\#}(b,a,b), T_{\#}(a), T_{\#}(b), T_{\#}(c)\}$.

Next, letting $\boldsymbol{A}$ be a variable not yet used in P, the program $P^{\#}$ is obtained from P by changing each atom $R(\bar{u})$ (including head atoms) to $R_{\#}(\boldsymbol{A}, \bar{u})$. Note that $P^{\#}$ is over $\#(sch(P))$. The presence of variable $\boldsymbol{A}$ guarantees that all rules in $P^{\#}$ are connected. Moreover, satisfying valuations now only use sets of facts whose first value is the same, i.e., the facts share the same component-identifier. Hence, when we execute $P^{\#}$ over $\#(I)$, the computation proceeds in a per-component fashion. Because the original program P distributes over components, program $P^{\#}$ correctly simulates P when the input contains values (see below for more discussion). To obtain output over σ_2, the third program $P^{\downarrow}$ projects the relations of $\#(\sigma_2)$ back to σ_2.

The subprogram $(P^{\uparrow} \cup P^{\#} \cup P^{\downarrow})$ will only do something if there are values in the input: at the very least, variable $\boldsymbol{A}$ needs to be assigned a value when evaluating $P^{\#}$. But even if $adom(I) = \emptyset$, in which case there is only one component, the original program P could still do useful boolean operations (e.g., as in Example 5). This computation is preserved by program P^{null}, that contains only the rules of P without variables, after extending the output rules with an additional negative body atom $\neg values(\,)$, where *values* is a value-detecting nullary relation outside $sch(P)$. The atom $\neg values(\,)$ acts as a guard, so the output rules will not fire when there are values. ◀

▶ **Example 8.** We illustrate the construction used in the proof of Proposition 7. Consider the following Datalog$^{\neg}$ program P with $edb(P) = \{R^{(1)}, S^{(1)}, T^{(2)}\}$ and $idb(P) = \{U^{(2)}, V^{(2)}\}$:

$$U(x,y) \leftarrow R(x), S(y).$$
$$V(x,y) \leftarrow U(x,y), T(x,y).$$

Assuming that V is the output relation, although the first rule of P is not connected, P distributes over components because of the join with input relation T. The transformed version of P is $\alpha(P) = P^{\uparrow} \cup P^{\#} \cup P^{\downarrow} \cup P^{\text{null}}$. Note that $P^{\text{null}} = \emptyset$ as P contains no rules without variables. Next, the program $P^{\uparrow}$ that tags input facts with their component values, contains the following rules, where '*con*' is an auxiliary relation to detect which values are connected:

$$con(x,x) \leftarrow R(x). \qquad R_{\#}(\boldsymbol{A},x) \leftarrow R(x), con(x,\boldsymbol{A}).$$
$$con(x,x) \leftarrow S(x). \qquad S_{\#}(\boldsymbol{A},x) \leftarrow S(x), con(x,\boldsymbol{A}).$$
$$con(x,y) \leftarrow T(x,y). \qquad T_{\#}(\boldsymbol{A},x,y) \leftarrow T(x,y), con(x,\boldsymbol{A}).$$
$$con(x,y) \leftarrow con(y,x).$$
$$con(x,y) \leftarrow con(x,z), con(z,y).$$

Next, program $P^{\#}$ is as follows:

$$U_{\#}(\boldsymbol{A},x,y) \leftarrow R_{\#}(\boldsymbol{A},x), S_{\#}(\boldsymbol{A},y).$$
$$V_{\#}(\boldsymbol{A},x,y) \leftarrow U_{\#}(\boldsymbol{A},x,y), T_{\#}(\boldsymbol{A},x,y).$$

[11] Nullary facts are tagged with all identifiers, since by definition they belong to all components.

Lastly, to project output back to V, program $P^{\downarrow}$ contains the rule: $V(x, y) \leftarrow V_{\#}(\boldsymbol{A}, x, y)$.

Intuitively, whenever the rule for relation $U_{\#}$ combines a fact $R_{\#}(c, a)$ and a fact $S_{\#}(c, b)$ where $a \neq b$, the shared tag c implies (through program $P^{\uparrow}$) that there is some T-fact connecting values a and b in the input, i.e., $R(a)$ and $S(b)$ belong to the same component. So, the rule for relation $U_{\#}$ works "inside" components, considering fewer pairs of R-facts and S-facts compared to the original rule for relation U. But, since P distributes over components (assuming output relation V), the output of $\alpha(P)$ is the same as P for all inputs. ◀

Let Datalog$^{\neg s}$ and con-Datalog$^{\neg s}$ denote the classes of queries computable by respectively stratified Datalog$^{\neg}$ programs and stratified con-Datalog$^{\neg}$ programs. By combining Proposition 6 and Proposition 7, we may write:

▶ **Theorem 9.** *Datalog*$^{\neg s} \cap \boldsymbol{\mathcal{D}} =$ *con-Datalog*$^{\neg s}$.

5 Connected Well-founded Datalog

In the following, we extend our results on class $\boldsymbol{\mathcal{D}}$ and stratified Datalog$^{\neg}$ to the well-founded semantics. The proofs for the well-founded semantics build upon the results for the stratified semantics by constructing, for each Datalog$^{\neg}$ program P and an input instance I, a stratified Datalog$^{\neg}$ program that simulates the well-founded semantics of P for the specific instance I. We start with the following result:

▶ **Proposition 10.** Every query computable by a con-Datalog$^{\neg}$ program under the well-founded semantics distributes over components.

Proof (sketch). Let $\mathcal{Q}$ be a query computable by a con-Datalog$^{\neg}$ program P under the well-founded semantics. Let I be an input for $\mathcal{Q}$. We have to show $\mathcal{Q}(I) = \bigcup_{J \in co(I)} \mathcal{Q}(J)$. We first transform P to a *stratified* con-Datalog$^{\neg}$ program $u_k(P)$, where each successive stratum simulates an outer step in the alternating fixpoint computation of P, where k indicates that $2k$ steps are simulated in total.[12] This technique is inspired by the *doubled program* construction [15]. Although only a constant number of steps can be simulated this way, for the specific instance I, we can choose k sufficiently large so that $u_k(P)$ simulates $P_{\mathrm{t}}(I)$ and $P_{\mathrm{t}}(J)$ for each $J \in co(I)$. Letting σ denote the output schema of $\mathcal{Q}$, we have $u_k(P)(I)|_{\sigma} = \mathcal{Q}(I)$ and $u_k(P)(J)|_{\sigma} = \mathcal{Q}(J)$ for each $J \in co(I)$. Next, because $u_k(P)$ is a stratified con-Datalog$^{\neg}$ program, we can apply Proposition 6 to know $u_k(P)(I)|_{\sigma} = \bigcup_{J \in co(I)} u_k(P)(J)|_{\sigma}$, resulting in $\mathcal{Q}(I) = \bigcup_{J \in co(I)} \mathcal{Q}(J)$, as desired. ◀

Now, Proposition 10 combined with the inclusion $\boldsymbol{\mathcal{D}} \cap \boldsymbol{\mathcal{V}} \subseteq \boldsymbol{\mathcal{M}}_{\text{disjoint}}$ from Section 3.2, gives rise to the following corollary:

▶ **Corollary 11.** *Every query computable by a con-Datalog*$^{\neg}$ *program under the well-founded semantics, and being value-driven, is domain-disjoint-monotone.*

Corollary 11 can be used to obtain an alternative proof for one of the main results in previous work by Zinn et al. [19], namely, that the win-move query (Example 1) is in the class $\boldsymbol{\mathcal{F}}_2$, as we now explain. First, $\boldsymbol{\mathcal{F}}_2$ is the class of queries that can be computed in a coordination-free manner under so-called *domain-guided distribution policies*: here, nodes of a network are made responsible for values of **dom**, and each input fact $\boldsymbol{f}$ is distributed to all those nodes responsible for at least one value of $adom(\boldsymbol{f})$. Coordination-freeness means that

[12] The 'u' in $u_k(P)$ stands for *unrolling*.

for any input, there exists a domain-guided distribution policy under which the nodes do not have to share input facts in order to compute the query. Now, since the win-move Datalog$^\neg$ program is connected, and this program is value-driven, Corollary 11 gives us that win-move is in $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$. Further applying the result $\boldsymbol{\mathcal{M}}_{\text{disjoint}} = \boldsymbol{\mathcal{F}}_2$ [4], we obtain that win-move is in $\boldsymbol{\mathcal{F}}_2$.

We also have the converse result of Proposition 10:

▶ **Proposition 12.** Every query computable by a Datalog$^\neg$ program under the well-founded semantics, and distributing over components, can be computed by a con-Datalog$^\neg$ program under the well-founded semantics.

Proof (sketch). Let $\mathcal{Q}$ be a query computable by a Datalog$^\neg$ program P under the well-founded semantics. Let $\alpha(P) = P^\uparrow \cup P^\# \cup P^\downarrow \cup P^{\text{null}}$ be the con-Datalog$^\neg$ program as defined in the proof for Proposition 7. If P is not stratified then $P^\#$, and by extension $\alpha(P)$, is also not stratified. We now outline the main arguments to demonstrate that $\alpha(P)$ computes the query $\mathcal{Q}$ under the well-founded semantics. Let I be an input for $\mathcal{Q}$. If I contains no values then the output of $\alpha(P)$ on I is just the output of P^{null} on I, and P^{null} correctly simulates P on such inputs. We also sketch the main steps for the case that I contains values. Let σ denote the output schema of $\mathcal{Q}$. First, because P computes $\mathcal{Q}$ under the well-founded semantics, we have $\mathcal{Q}(I) = P_{\text{t}}(I)|_\sigma$. Next, as in the proof of Proposition 10, we convert P to a stratified Datalog$^\neg$ program $u_k(P)$, with sufficiently large $k \in \mathbb{N}$ such that: $\mathcal{Q}(I) = u_k(P)\,(I)|_\sigma$. Now, considering the transformation $\bar{\alpha}(D) = D^\uparrow \cup D^\# \cup D^\downarrow$ for any Datalog$^\neg$ program D, it can be shown that $\bar{\alpha}$ may be applied to the right hand side, to obtain: $\mathcal{Q}(I) = \bar{\alpha}(u_k(P))\,(I)|_\sigma$. Subsequently, we can use a technical lemma to know that operations u_k and $\bar{\alpha}$ commute, i.e., $\bar{\alpha}(u_k(P))\,(I)|_\sigma = u_k(\bar{\alpha}(P))\,(I)|_\sigma$. We can up front also choose k large enough to correctly simulate the well-founded semantics of $\bar{\alpha}(P)$ on I, so that $u_k(\bar{\alpha}(P))\,(I)|_\sigma = \bar{\alpha}(P)_{\text{t}}\,(I)|_\sigma$. Finally, since P^{null} is constructed to output nothing when $adom(I) \neq \emptyset$, we have $\mathcal{Q}(I) = \bar{\alpha}(P)_{\text{t}}\,(I)|_\sigma = \alpha(P)_{\text{t}}\,(I)|_\sigma$, as desired. ◀

Let Datalog$^{\neg wf}$ and con-Datalog$^{\neg wf}$ denote the classes of queries computable under the well-founded semantics by respectively Datalog$^\neg$ programs and con-Datalog$^\neg$ programs. By combining Proposition 10 and Proposition 12, we may write:

▶ **Theorem 13.** *Datalog*$^{\neg wf} \cap \boldsymbol{\mathcal{D}} =$ *con-Datalog*$^{\neg wf}$.

6 Semi-connected Well-founded Datalog

Previous work has considered a relaxation of connected Datalog$^\neg$, called *semi-connected* [4]. We refer to Section 4.1 for the definition of connected Datalog$^\neg$, including the notions of *global* and *value-detecting* nullary relations. Now, we say that a Datalog$^\neg$ program P is *semi-connected* if we can partition the rules of P into two subprograms P_1 and P_2 such that:

1. P_1 is a con-Datalog$^\neg$ program;[13]
2. P_2 is a semi-positive program satisfying the following conditions: *(i)* $idb(P_2) \cap sch(P_1) = \emptyset$, and *(ii)* nullary relations occurring in rule bodies of P_2 are either global or value-detecting within the entire program P.

Note that the entire schema of P_1 can be used as input for P_2. So, P_2 can negate relations of $idb(P_1)$, as demonstrated by Example 15. Subprogram P_1 or P_2 could be empty. Subprogram

[13] Note that this includes restrictions on nullary atoms in rule bodies.

P_1 is not necessarily stratified, but we may view P_2 as a last computation step of P that possibly uses non-connected rules. If P is stratified then we may view P_2 as the last stratum. We denote the language of semi-connected Datalog$^\neg$ programs as semicon-Datalog$^\neg$.

Recall the query classes $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$ and $\boldsymbol{\mathcal{V}}$ from Section 3.2. Queries of $\boldsymbol{\mathcal{V}}$ computable by stratified semicon-Datalog$^\neg$ programs are in $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$ [4]. We can now confirm that this result is maintained under the well-founded semantics:

▶ **Theorem 14.** *Every query computable by a semicon-Datalog$^\neg$ program under the well-founded semantics, and being value-driven, is in $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$.*

Proof (sketch). Let $\mathcal{Q}$ be a query that is computed by a semicon-Datalog$^\neg$ program P under the well-founded semantics, and being value-driven. Let I and J be two inputs for $\mathcal{Q}$ such that J contains no nullary facts and $adom(I) \cap adom(J) = \emptyset$. We show that $\mathcal{Q}(I) \subseteq \mathcal{Q}(I \cup J)$. Following the proof idea for Proposition 10, we convert P to a stratified program $u_k(P)$ with k sufficiently large to correctly simulate the alternating fixpoint computation of P on the instances I and $I \cup J$. Importantly, if P is semi-connected then $u_k(P)$ is also semi-connected. Letting σ be the output schema of $\mathcal{Q}$, it can be shown that $u_k(P)(I)|_\sigma \subseteq u_k(P)(I \cup J)|_\sigma$, using similar techniques as in previous work [4]. Next, because $u_k(P)$ correctly simulates the well-founded semantics of P on instances I and $I \cup J$, and $\mathcal{Q}$ is computed by P, we obtain $\mathcal{Q}(I) \subseteq \mathcal{Q}(I \cup J)$, as desired. ◀

We illustrate the use of Theorem 14 with the following example.

▶ **Example 15.** Building upon the win-move example (Example 1), the following program outputs *true* (in nullary relation T) if there are at least two nodes at which player 1 has a winning strategy:

$$win(x) \leftarrow move(x, y), \neg win(y).$$
$$same(x, x) \leftarrow move(x, y).$$
$$same(y, y) \leftarrow move(x, y).$$
$$T() \leftarrow win(x), win(y), \neg same(x, y).$$

Note that negation on relation *same* simulates nonequality. This program is not stratified due to the embedding of the win-move program; so, we apply the well-founded semantics. Moreover, this program is not connected due to the last rule. But, the program is still semi-connected. It is also value-driven. Hence we can apply Theorem 14 to know that the query expressed by this program is in $\boldsymbol{\mathcal{M}}_{\text{disjoint}}$. Next, using $\boldsymbol{\mathcal{M}}_{\text{disjoint}} = \boldsymbol{\mathcal{F}}_2$ [4], we know this query can be computed in a coordination-free manner under domain-guided distribution policies. ◀

Relating to the class $\boldsymbol{\mathcal{D}}$, the following simple example shows that not all queries computable by semicon-Datalog$^\neg$ programs under the well-founded semantics distribute over components:[14]

▶ **Example 16.** Consider the following semicon-Datalog$^\neg$ program P, with $edb(P) = \{R^{(1)}\}$ and $idb(P) = \{S^{(1)}, T^{(2)}\}$, where T is the intended output relation:

$$S(x) \leftarrow R(x).$$
$$T(x, y) \leftarrow S(x), S(y).$$

[14] The query of Example 15 also does not distribute over components, e.g., on an input consisting of two disjoint *move*-subgraphs, in each of which there is precisely one winning node.

The first rule is connected, whereas the second rule is not. For the input $I = \{R(a), R(b)\}$, the output of P on I under the well-founded semantics (or stratified semantics) is $\{T(a,a), T(b,b), T(a,b), T(b,a)\}$. The facts $T(a,b)$ and $T(b,a)$, however, can not be computed when we distribute P over the components $\{R(a)\}$ and $\{R(b)\}$. ◀

7 Discussion

In this paper, we have shown that although membership of positive Datalog programs in $\boldsymbol{\mathcal{D}}$ is undecidable, connected Datalog$^\neg$ provides an effective syntax for Datalog$^\neg \cap \boldsymbol{\mathcal{D}}$ both under the stratified as well as under the well-founded semantics. In addition, the latter result provides an alternative explanation for why the non-monotonic win-move query is in $\boldsymbol{\mathcal{F}}_2$.

In theory, any query in $\boldsymbol{\mathcal{D}}$ (and therefore any query in connected Datalog$^\neg$) can be evaluated without any communication over a network using a distribution where every computing node is assigned, as a local instance, a union of connected components of the global database instance (and every connected component is assigned to at least one computing node). However, as finding connected components is expensive, it is unlikely that there are many datasets for which such a distribution of data is practical. Still, it would be interesting to investigate properties of Datalog$^\neg$ programs that imply distributions of data that give rise to communication-free evaluation.

Hull and Yoshikawa [13] introduced a declarative formalism in the style of stratified Datalog$^\neg$ in the context of object databases. Using their formalism, Cabibbo [6] showed, among other things, that semi-positive Datalog$^\neg$ extended with value invention captures the class of all queries preserved under extensions. The latter type of result can be seen as evidence that semi-positive Datalog$^\neg$ is a core fragment of Datalog$^\neg$ for the class of queries preserved under extensions. In analogy, we expect that con-Datalog$^\neg$ is somehow the right Datalog$^\neg$ fragment for $\boldsymbol{\mathcal{D}}$ and conjecture that con-Datalog$^\neg$ extended with value invention captures the class $\boldsymbol{\mathcal{D}}$.

References

1 S. Abiteboul, Z. Abrams, S. Haar, and T. Milo. Diagnosis of asynchronous discrete event systems: Datalog to the rescue! In *PODS*, pages 358–367. ACM Press, 2005.

2 S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

3 P. Alvaro, N. Conway, J.M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *IEEE 30th International Conference on Data Engineering*, pages 52–63. IEEE, 2014.

4 T.J. Ameloot, B. Ketsman, F. Neven, and D. Zinn. Weaker forms of monotonicity for declarative networking: A more fine-grained answer to the CALM-conjecture. In *PODS*, pages 64–75. ACM Press, 2014.

5 T.J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15:1–15:38, 2013.

6 L. Cabibbo. The expressive power of stratified logic programs with value invention. *Information and Computation*, 147(1):22–56, 1998.

7 K.J. Compton. Some useful preservation theorems. *Journal of Symbolic Logic*, 48:427–440, 1983.

8 N. Conway, W.R. Marczak, P. Alvaro, J.M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1:1–1:14. ACM Press, 2012.

9 A. Dawar and S. Kreutzer. On Datalog vs. LFP. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming*, pages 160–171. Springer, 2008.

10 T. Feder and M.Y. Vardi. Homomorphism closed vs. existential positive. In *LICS*, pages 311–320. IEEE Computer Society, 2003.

11 I. Guessarian. Deciding boundedness for uniformly connected datalog programs. In S. Abiteboul and P.C. Kanellakis, editors, *ICDT*, volume 470 of *Lecture Notes in Computer Science*, pages 395–405. Springer, 1990.

12 J.M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.

13 R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *VLDB*, pages 455–468. Morgan Kaufmann Publishers Inc., 1990.

14 T. Jim and D. Suciu. Dynamically distributed query evaluation. In *PODS*, pages 28–39. ACM Press, 2001.

15 D.B. Kemp, D. Srivastava, and P.J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theor. Comput. Sci.*, 146(1&2):145–184, 1995.

16 B.T. Loo, T. Condie, M. Garofalakis, D.E. Gay, J.M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: Language, execution and optimization. In *SIGMOD*, pages 97–108. ACM Press, 2006.

17 O. Shmueli. Equivalence of Datalog queries is undecidable. *The Journal of Logic Programming*, 15(3):231–241, 1993.

18 A. Van Gelder. The alternating fixpoint of logic programs with negation. *J. Comput. Syst. Sci.*, 47(1):185–221, 1993.

19 D. Zinn, T.J. Green, and B. Ludäscher. Win-move is coordination-free (sometimes). In *ICDT*, pages 99–113. ACM Press, 2012.