Win-Move is Coordination-Free (Sometimes)

Daniel Zinn[1]
daniel.zinn@logicblox.com

Todd J. Green[1,2]
green@cs.ucdavis.edu

Bertram Ludäscher[2]
ludaesch@cs.ucdavis.edu

[1] LogicBlox, Inc.
1349 W Peachtree St NW
Atlanta, GA 30309 USA

[2] Dept. of Computer Science
University of California, Davis
Davis, CA 95616 USA

ABSTRACT

In a recent paper by Hellerstein [15], a tight relationship was conjectured between the number of strata of a Datalog$^\neg$ program and the number of "coordination stages" required for its distributed computation. Indeed, Ameloot et al. [9] showed that a query can be computed by a coordination-free relational transducer network iff it is *monotone*, thus answering in the affirmative a variant of Hellerstein's CALM conjecture, based on a particular definition of coordination-free computation. In this paper, we present three additional models for declarative networking. In these variants, relational transducers have limited access to the way data is distributed. This variation allows transducer networks to compute more queries in a coordination-free manner: e.g., a transducer can check whether a ground atom A over the input schema is in the "scope" of the local node, and then send either A or $\neg A$ to other nodes.

We show the surprising result that the query given by the well-founded semantics of the unstratifiable *win-move* program is coordination-free in some of the models we consider. We also show that the original transducer network model [9] and our variants form a strict hierarchy of classes of coordination-free queries. Finally, we identify different syntactic fragments of Datalog$^\neg_\forall$, called *semi-monotone* programs, which can be used as declarative network programming languages, whose distributed computation is guaranteed to be eventually consistent and coordination-free.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems, Distributed databases*

General Terms

Languages, Theory

Keywords

Datalog, distribution, relational transducer, monotonicity

ICDT 2012, March 26–30, 2012, Berlin, Germany.

1. INTRODUCTION

The popularization of cloud computing as a scalable, distributed computing paradigm has reinvigorated work on programming distributed systems and databases. One of the thornier sources of inefficiency in such systems, identified first in [15] and reiterated in [9, 18], is the presence of *global coordination barriers* in distributed computation.

For example, consider a stratified Datalog$^\neg$ program with a rule of the form $A \leftarrow B, \neg C$. If a new fact C becomes known, previously derived atoms A may no longer be derivable, i.e., the program is *non-monotonic*. Therefore, the standard bottom-up evaluation procedure for such programs involves a staged, stratum-at-a-time evaluation [2], with computation of a given stratum allowed to proceed only when computation of all lower strata is complete. In the distributed context, each stratum apparently constitutes a synchronization barrier where all nodes must wait for all other nodes from lower strata to finish. Moreover, it seems that no results can be returned to the user or client application until computation reaches the last stratum.

In contrast, in Datalog programs without negation, the rules can be evaluated *in any order* until a fixpoint is reached, since such positive programs are *monotone*. Even though this "disorderly" evaluation strategy can proceed in a nondeterministic fashion, the resulting fixpoint is uniquely determined, independent of the chosen evaluation order [5]. Thus, positive Datalog programs lend themselves to a coordination-free evaluation. This has been exploited, e.g., by Loo et al. [19] for declarative networking, using a pipelined semi-naive strategy, which, as Hellerstein [15] puts it, "*makes monotonic logic embarrassingly parallel*". In contrast, nonmonotonic stratified programs seem to require distributed coordination to proceed from one stratum to the next: "*Nonmonotonic stratum boundaries are global barriers: in general, no node can proceed until all tasks in the lower stratum are guaranteed to be finished*" [15]. Guided by this intution, Hellerstein conjectured:

Conjecture 1.1 (CALM [15]) A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

The CALM[1] conjecture is informal—it requires precise notions of eventually consistent, coordination-free, etc.—but seems plausible nevertheless. Ameloot et al. [9] have shown that a version of the conjecture holds for transducer networks with a particular notion of coordination-freeness. For

[1]Consistency And Logical Monotonicity

stratified Datalog$^\neg$, Hellerstein [15] suggests that the parallel runtime "*might be best measured by the number of strata it must proceed through sequentially*", which thus would yield a simple, elegant notion of coordination complexity, and further support the intuition that "*there is deep connection between non-monotonic reasoning and parallel coordination*".

A close correspondence between coordination complexity and number of strata in Datalog$^\neg$ programs would also nicely mimic the strict class hierarchy of queries expressible by Datalog$^\neg$ programs with n strata: positive Datalog programs are coordination-free and can be executed in any order within a single stratum, while programs with n strata could have a disorderly rule evaluation within a stratum but would require n global coordination barriers between strata. Extrapolating these ideas even further, *unstratifiable* Datalog$^\neg$ programs, evaluated under the well-founded semantics [22], would then require an *unbounded* number of coordination stages, corresponding to the number of rounds of the alternating fixpoint computation [14], i.e., the number of coordination barriers would no longer be fixed for a given program (as for stratified Datalog$^\neg$) but instead depend on the database instance.[2]

In this paper, we revisit and shed new light on the CALM conjecture. We show that, surprisingly, the canonical example of an unstratifiable program, the *win-move* program, has a coordination-free distributed evaluation strategy in some of the declarative networking variants we consider.

A Hierarchy of Relational Transducer Networks

The original model, denoted $\mathcal{N}_0$, are the transducer networks defined by Ameloot et al. [9]. Only monotone queries are coordination-free in this model. In our first variant, $\mathcal{N}_1$, transducers have some limited knowledge about the horizontal data distribution: a transducer can check whether an input atom A (i.e., a ground fact $R(\bar{x})$ from the Herbrand base) is in its "scope". If that is the case, then the local node can determine whether A or $\neg A$ is true using only local computations, and pass on this information to other nodes. In $\mathcal{N}_1$, semi-positive Datalog$^\neg$ programs are coordination-free. In our second variant, $\mathcal{N}_2$, we require partitioning policies to be *element-determined*, i.e., there is a function F, mapping each domain element to a set of nodes. A ground input atom $R(x_1, \ldots, x_k)$ is then in the scope of all nodes $F(x_i)$, for all $i = 1, \ldots, k$. We show that in this model the unstratifiable win-move program is coordination-free.

Finally, in $\mathcal{N}_3$, transducers have access to a system relation `adom` containing the active domain of the global input. As it turns out, this knowledge has a dramatic effect on the class of coordination-free programs, making all queries coordination-free.

Our main result, Theorem 6.2, states that the classes of queries that are coordination-free under the network models $\mathcal{N}_0$, $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$, form a strict hierarchy. Furthermore, the class of coordination-free queries under type $\mathcal{N}_1$ coincides with the queries computable by a semi-positive Datalog$^\neg$ program.

Games and Coordination

After presenting preliminaries in Section 2, we first motivate our approach for minimizing coordination in the presence of non-monotonic constructs (negation), on the example of the well-known win-move program (Section 3). We start from the "doubled program" [16] version of win-move and translate it into Datalog$^{\neg}_{\forall}$, which is a Datalog$^\neg$ variant proposed by Abiteboul and Vianu [3] with negation in the head (to indicate deletions) and $\forall$-quantification in the body. The key idea of our approach to deal with the inherent difficulties in programming distributed systems is to define a *disorderly* semantics for Datalog$^{\neg}_{\forall}$ with built-in non-determinism that corresponds to non-determinism caused by network *message re-ordering* or by *asynchronously* incorporationg incoming updates to the local database state as in [19]. We show that our transformed program is confluent and terminating under this semantics, and that the result agrees with both the deterministic and non-deterministic semantics of Datalog$^{\neg}_{\forall}$ given by Abiteboul and Vianu. Since the deterministic and non-deterministic semantics in general disagree for arbitrary programs, another contribution is the identification of a "well-behaved" fragment of Datalog$^{\neg}_{\forall}$, which we call semi-monotone Datalog (Section 4).

Section 5 introduces the basic transducer network model of Ameloot et al. [9], called $\mathcal{N}_0$ here, and then describes our variants $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$; Section 6 presents our main result, the hierarchy of classes of coordination-free queries implied by these network models. The remaining sections contain discussions of related work and concluding remarks.

2. BACKGROUND AND PRELIMINARIES

2.1 Datalog with Negation

A *relational schema* σ consists of a finite set of relation symbols $r_1, \ldots, r_k$ with associated arities $\alpha(r_i) \geq 0$. Let *dom* be a fixed and countable underlying domain. A *database instance (database)* over σ is a finite structure

$$D = (U, r_1^D, \ldots, r_k^D)$$

with finite universe $U \subseteq dom$ and relations $r_i^D \subseteq U^{\alpha(r_i)}$. We will often identify database instances as sets of ground facts in the standard way, assuming U to be the active domain.

A *Datalog$^\neg$ program* P is a finite set of rules of the form

$$A \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m$$

where the head A, positive body literals B_i and negative body literals C_i are relational atoms, i.e., of the form

$$r(x_0, \ldots, x_l)$$

with $r \in \sigma_P$ being a relation symbol with $\alpha(r) = l$ and each x_i is either a variable or a constant from *dom*. The signature σ_P of P is partitioned into a set $idb(P)$ of *intensional* relation symbols of P occurring in some head of P and $edb(P)$ of *extensional* relation symbols occurring only in the bodies of rules. We require negation to be safe, i.e., a variable occurring in a negative body literal should also occur in a positive literal in the body. Among the predicates of $idb(P)$, one is distinguished as the *output predicate*.

A *(positive) Datalog* program is a Datalog$^\neg$ program that does not have negative literals, i.e., $m = 0$ for all rules. A Datalog$^\neg$ program P is *semi-positive* if all negatively used relations are among $edb(P)$.

Fix a Datalog$^\neg$ program P. The *predicate dependency graph* for P is the directed graph $G(P) = (\sigma_P, E^+ \cup E^-)$ whose vertices are the relation symbols of P, and such that

[2]Well-founded Datalog$^\neg$ expresses the fixpoint queries [2], and is strictly more expressive than stratified Datalog$^\neg$ [17].

$(r, r') \in E^+$ (resp. E^-) if r occurs positively (resp. negatively) in the body of a rule in P that has r' in the head. P is *recursive* if $G(P)$ contains a cycle. P is *stratifiable* if $G(P)$ does not contain a cycle with a negative edge $(r, r') \in E^-$. Any semi-positive Datalog$^\neg$ program is trivially stratifiable.

2.2 Well-Founded Semantics

Let P be a Datalog$^\neg$ program, including a given database D as a set of facts. Fix some Herbrand interpretation $J \subseteq \mathcal{B}_P$, where $\mathcal{B}_P$ denotes the set of all ground instances of atomic formulas of P. The *immediate consequences* under the assumptions J are given by:

$$T_{P,J}(I) := \{H \mid (H \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m) \in \mathit{ground}(P), \; I \models B_1 \wedge \cdots \wedge B_n, J \models \neg C_1 \wedge \cdots \wedge \neg C_m\}.$$

Since J is fixed, $T_{P,J}$ is a monotone operator. Let $\Gamma_P(J) := \mathrm{lfp}(T_{P,J})$ be its least fixpoint. The operator Γ_P is antimonotone (observe how J is used in $T_{P,J}$, i.e., $J_1 \subseteq J_2$ implies $\Gamma_P(J_2) \subseteq \Gamma_P(J_1)$). It follows that $\Gamma_P^2 (:= \Gamma_P \circ \Gamma_P)$ is a monotone operator, so it has a least and a greatest fixpoint. These are used to define the 3-valued *well-founded model* W_P, a mapping of ground atoms to $\{\mathsf{true}, \mathsf{false}, \mathsf{undefined}\}$ as follows:

$$W_P(A) := \begin{cases} \mathsf{true} & \text{if } A \in \mathrm{lfp}(\Gamma_P^2) \\ \mathsf{false} & \text{if } A \notin \mathrm{gfp}(\Gamma_P^2) \\ \mathsf{undefined} & \text{if } A \in \mathrm{gfp}(\Gamma_P^2) \setminus \mathrm{lfp}(\Gamma_P^2) \end{cases}$$

We recall also that the well-founded semantics is a conservative extension of the stratified semantics, i.e., the well-founded semantics is two-valued for stratifiable Datalog$^\neg$ programs P; and in these cases, well-founded semantics agrees with stratified semantics.

2.3 Alternating Fixpoint

The construction of W_P given above is called the *alternating fixpoint* computation of the well-founded model [14] and involves a nested fixpoint: The inner fixpoint is given by $\Gamma_P(J) = \mathrm{lfp}(T_{P,J})$, the outer fixpoints are obtained by iterating the antimonotone operator Γ_P. The sequence $\Gamma_P^0, \Gamma_P^1, \ldots$ given by

$$\Gamma_P^0 := \emptyset \quad \text{and} \quad \Gamma_P^{i+1} := \Gamma_P(\Gamma_P^i), \tag{1}$$

alternates between underestimates of true, and overestimates of true or $\mathsf{undefined}$ atoms, respectively. More precisely, the subsequence Γ_P^{2k} converges to the least fixpoint (the set of true atoms) from below, while Γ_P^{2k+1} converges to the greatest fixpoint (the set of true or undefined atoms) from above. The key idea is that applying the antimonotone operator Γ_P to an underestimate yields an overestimate and vice versa.

2.4 Doubled Program

A standard refinement of the alternating fixpoint technique [16] initializes the first underestimate with the set of definitely true facts, i.e., those that do not depend on any negative subgoals. Furthermore, previously computed under- and over-estimates are used to "seed" the fixpoint computation exploiting monotonicity of $T_{P,J}(I)$ in I. Here, computing a new underestimate starts from the old underestimate (since the sequence of underestimates is monotonically increasing), and computing a new overestimate starts from the last underestimate since underestimates are always subsets of the overestimates. Let P^+ be the subset of rules of a program P that do not contain negation, and $T^\omega_{P,J}(I)$ denote the least fixpoint obtained by iterating $T_{P,J}$ on I. The *doubled program* approach can be described as:

$$\begin{aligned} U_0 &:= T^\omega_{P^+,\emptyset}(\emptyset) \\ V_0 &:= T^\omega_{P,U_0}(U_0) \\ U_i &:= T^\omega_{P,V_{i-1}}(U_{i-1}), \quad i > 0 \\ V_i &:= T^\omega_{P,U_i}(U_i), \quad i > 0 \end{aligned}$$

This computation is performed until the sequence of underestimates U_k and overestimates V_k become stationary. It is equivalent with the definition given in (1) in the sense that $\mathrm{lfp}(\Gamma_P^2)$ equals the fixpoint ${U_i}^\omega$ and $\mathrm{gfp}(\Gamma_P^2)$ equals the fixpoint ${V_i}^\omega$.

We use the doubled program approach as a starting point for our parallel win-move evaluation (Section 3). However, note that while in this scheme an underestimate U_{i+1} is computed "incrementally" (using the previous underestimate U_i), an overestimate V_{i+1} is always computed "almost from scratch" (without using the previous overestimate V_i but only U_i), which is somewhat inefficient. A number of papers (e.g., [11, 12, 24, 10]) have explored techniques for incrementally computing overestimates V^{i+1} by directly inferring the "to-be-deleted" facts from V^i; we discuss the potential relevance of such techniques to our problem in Section 7.

2.5 Production Rules Semantics

A number of *production rule* semantics have been defined for Datalog$^\neg$ and extensions thereof [23, 3, 4]. These procedural semantics are easy to compute: The program is understood as a set of rules that are fired until a fixpoint is reached. A rule can fire if its body is true for the current database instance. The rule heads are interpreted as updates, i.e., positive heads are insertions, while negative literals are deletions. Here, we consider Datalog$_\forall^{\neg\neg}$ [3], a language whose syntax extends Datalog$^\neg$ by allowing negative literals in the head (deletions), and $\forall$-quantification of variables in the rule body. Thus, Datalog$_\forall^{\neg\neg}$ rules are of the form

$$A \leftarrow \forall \bar{X} \; B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m$$

where A is a *positive* or *negated* ($\neg$) relational atom[3], each variable not in $\bar{X}$ occurs in at least one positive atom in the body, and the variables in $\bar{X}$ occur only in the body and only in negated atoms.

It is convenient to allow Datalog$_\forall^{\neg\neg}$ programs to perform updates to extensional relations; or, equivalently, to allow for some intensional relations to come "pre-initialized." Thus we assume that the schema σ_P of a Datalog$_\forall^{\neg\neg}$ program P is partitioned into a set $\mathit{edb}(P)$ of relation symbols occurring only in the bodies of rules, and two sets $\mathit{idb}(P)$ and $\mathit{eidb}(P)$ of relation symbols occurring in the heads of rules. The input to P will be an instance consisting of $\mathit{edb}(P)$ and $\mathit{eidb}(P)$ tuples. The distinguished output predicate for P must come from $\mathit{idb}(P)$.

A Datalog$_\forall^{\neg\neg}$ program may be interpreted under either of two distinct semantics, the deterministic and the nondeterministic semantics. Each is procedural in the sense

[3] In the presentation of Abiteboul and Vianu [3], heads may have multiple atoms, giving additional expressive power under their non-deterministic semantics, but the more restricted version here is sufficient for our purposes.

that it is defined in terms of a fixpoint operator. We will return to these semantics, and present a third "even more non-deterministic" alternative, in Section 4.

Intuitively, the *deterministic* semantics involves firing all applicable rules in parallel at each step of the computation. Rules with positive heads are treated as insertions, while those with negative heads are considered deletions. (Conflicting insertions and deletions are ignored.) The *non-deterministic* semantics for Datalog$^{\neg}_{\forall}$, on the other hand, involves firing just a single ground rule at each step of the computation, with the rule chosen non-deterministically.

Formally, given a Datalog$^{\neg}_{\forall}$ program P and database instance I, we define the *deterministic immediate consequence operator* $A^{\text{det}}_P()$ as follows:

$$A^{\text{det}}_P(I) := (I \cup (\Delta I^+ \setminus \Delta I^-)) \setminus (\Delta I^- \setminus \Delta I^+) \text{ with}$$
$$\Delta I = \{\, \nu(head) \mid I \models \nu(body) \text{ for } (head \leftarrow body) \in P \,\}$$

The deterministic semantics $\mathcal{S}_P(I)$ of a Datalog$^{\neg}_{\forall}$ program P applied to an input instance I is defined as the least fixpoint of iterating $A^{\text{det}}_P(\cdot)$ on I. In case the fixpoint does not exist $P(I)$ is undefined. Note, that for these procedural semantics we do *not* encode the EDB facts as body-less rules in P. Instead, they are introduced by seeding the fixpoint computation. Since the procedural semantics allow deletions, this is significant.

The non-deterministic semantics is based on the notion of *immediate successor* [23] of a set of facts using a rule, defined as follows. Let $r = head \leftarrow body$ be a Datalog$^{\neg}_{\forall}$ rule. Let I be a set of facts and ν be a consistent variable assignment for the variables in r such that I implies $\nu(body)$. Then an instance I' is an *immediate successor* of I if it can be obtained from I by (a) deleting the fact A if $\nu(head) = \neg A$, or (b) by inserting A if $\nu(head) = A$. An instance J is an *eventual successor* of I using the rules of P if there exists a sequence $I_0 = I, \ldots, I_n = J$ such that for each i, I_{i+1} is an immediate successor of I_i using some rule in P.

Definition 2.1 Let P be a Datalog$^{\neg}_{\forall}$ program, and let I be a source instance. The *result* of applying P to I under the non-deterministic semantics is the set containing an instance J iff J is an eventual successor of I using the rules of P, and there is no immediate successor $J' \neq J$ of J using some rule in P.

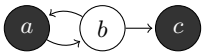
3. BASIC APPROACH

In this section, we present the key intuitions for a parallel and "disorderly" evaluation strategy for the *win-move game*:

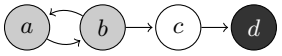
```
win(X) ← move(X,Y), ¬win(Y).        (2)
```

Here, we have a database instance over an active domain of *positions* and having a single binary `move` predicate. A tuple `move(a,b)` can be read as indicating that "from position `a` a player can move to position `b`." In the game, two players `White` and `Black` take turns making moves, starting from a given position, with `White` playing first. A player *loses* at position X if she cannot move; and she *wins* at X if she can move to a position which the opponent loses.

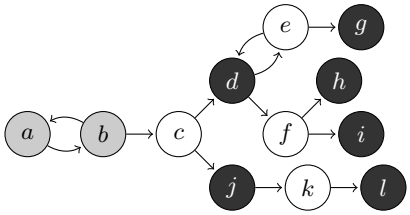
Evaluating program (2) under the well-founded semantics, the `true` facts in `win` are the positions X such that `White` has a winning strategy for the game starting at X, while the `false` facts in `win` are the positions for which `Black` has a winning strategy. The `undefined` facts in `win` are the *drawn* positions for which neither player has a winning strategy,



(a) c is initially lost, therefore b is won and thus a is lost.



(b) d is initially lost, therefore c is won. From b, the optimal strategy is to move back to a; thus a and b are draws.



(c) g, h, i, l are initially lost. Thus, e, f, k are won (for each of the won positions, there exists an adjacent lost position). d and f are lost, since all adjacent positions win. c is won either by moving to d or j. Positions a and b are draws.

Figure 1: Examples of solved win-move games

that is, the two players can move in cycles without either one being able to force the other into a lost position. See Figure 1 for several examples of `move` graphs together with their solutions.

Our strategy uses a multi-step program transformation whose first step is just the standard doubled program approach (cf. Section 2). Thus, we introduce two relations that we shall call `won` and `may_win`, in which we compute the set of *definitely won* and *not definitely lost* positions, respectively. Here, `won` is the *underestimate* of the `win` relation, while `may_win` is the *overestimate* of the `win` relation.

```
won(X) ← move(X,Y), ¬may_win(Y).        (P_u)
may_win(X) ← move(X,Y), ¬won(Y).        (P_v)
```

Example 3.1 Consider Figure 1(c). The doubled program computation proceeds as follows to determine positions c, e, f and k as win; a and b as draws and the remaining positions d, g, h, i, j and l as lost positions.

	estimates
U_0	`won` = $\{\}$
V_0	`may_win` = $\{a, b, c, d, e, f, j, k\}$
U_1	`won` = $\{e, f, k\}$
V_1	`may_win` = $\{a, b, c, e, f, k\}$
U_2	`won` = $\{c, e, f, k\}$
V_2	`may_win` = $\{a, b, c, e, f, k\}$

Note that in the doubled program evaluation, P_u and P_v are evaluated sequentially in a strictly alternating order. Directly mapping this approach to a distributed setting would seem to require reaching a global consensus that a given step is complete before the next step is allowed to proceed. To avoid the need for such coordination, we will derive a new set of rules in which all rules can be applied to ground facts *in any order*. The new set of rules will be a Datalog$^{\neg}_{\forall}$ program, whose result turns out to agree with the doubled program whether we evaluate it under the deterministic semantics of Abiteboul and Vianu, under their non-deterministic semantics, or under the disorderly semantics that we introduce in this paper.

As a first step in this derivation, we introduce an auxiliary binary relation `good_move` in which we record moves that, so far as we know, may be able to be played to avoid a certain defeat. Initially, *any* move *out* of a node is such a move, and all interior nodes may still win. That is, we initialize our relations `may_win` and `good_move` as follows[4]:

```
good_move(X,Y) ← move(X,Y).
may_win(X) ← move(X,Y).
```
(P_{init})

Now, intuitively, we can successively recognize won positions by detecting an adjacent lost position, i.e., one that may not win anymore (or is not in the over-estimate). Further, whenever a position has been discovered to be won, moving into that position is definitely not a good move since now the opponent would be left in a winning position; thus the edge should be deleted from our `good_move` relation. Finally, if no good moves remain from a position, then it is certain that the position is lost.

```
1  won(X) ← move(X,Y), ¬may_win(Y).
2  ¬good_move(X,Y) ← won(Y), move(X,Y).
3  ¬may_win(X) ← ∀Y ¬good_move(X,Y),move(X,_).
```
(P_{wm})

Example 3.2 Continuing with the example of Figure 1(c), we now consider two possible computations according to the non-deterministic semantics for Datalog$^{\neg}_{\forall}$. In either case, we repeatedly choose a valuation of variables that makes the body true for a rule in P_{wm} and the current database state, and apply the corresponding insertion or deletion. The two computations A and B are represented in Figure 2 as follows: `move` is an EDB relation, so it stays constant. The relation `good_move` is initialized with the contents of `move`, and during the computation we are subsequently deleting facts from `good_move`. For `may_win` it is similar: `may_win` is initialized with non-leaf nodes, some of which are going to be deleted during the computation. `won` begins empty. The numbers in the columns A (resp. B) indicate the order in which a fact is derived into `won` or deleted from `good_move` and `may_win` in the computation trace A (resp. B). For example consider computation A. The first three derivations performed are to successively declare positions `e`, `f`, and `k` won according to rule (1) of P_{wm}. We can then apply rule (2) to delete `(d,e)` from `good_move` since `e` is lost (step 4), similar for `(d,f)`, and `(j,k)`. Now, `d` does not have a good-move out anymore, and thus can be deleted from `may_win` according to rule (3), same for `j`. Finally, we can apply rule (1) again to declare `c` won, since, for example, `d` is not in `may_win` anymore. Now, no applicable rule would change any of the IDB relations anymore. We end up with the winning positions in `won`; the drawn positions are those in `may_win` that are not also in `won`, in our case `a` and `b`.

In the computation A, we derive `won(c)` last[5]. However, the different order B of evaluating P_{wm} derives `won(c)` already after only four rule applications. Intuitively, this is due to the fact that there is a short "winning path" from c through j, k, and l, which does *not* have to wait for the deletion of d from `may_win`.

[4]The rule to compute `may_win` is equivalent to computing the first overestimate of the alternating fixpoint procedure.
[5]Observe how this order roughly corresponds to the alternating fixpoint order.

move

X	Y
a	b
b	a
b	c
c	d
c	j
d	e
d	f
e	d
e	g
f	h
f	i
j	k
k	l

good_move

X	Y	A	B
a	b		
b	a		
b	c		
c	d		
c	j		
~~d~~	~~e~~	4	9
~~d~~	~~f~~	5	7
e	d		
e	g		
f	h		
f	i		
~~j~~	~~k~~	6	2
k	l		

may_win

X	A	B
a		
b		
c		
~~d~~	7	8
e		
f		
~~j~~	8	3
k		

won

X	A	B
e	1	5
f	2	6
k	3	1
c	9	4

Figure 2: Two non-deterministic evaluations `A` and `B` of P_{wm}, solving the win-move game in Figure 1(c).

Furthermore, both computations A and B reach the same result, although P_{wm} contains negation and is non-monotonic.

The fact that computations A and B from the example evaluate P_{wm} differently, yet reach the same conclusion, is not a coincidence: as we will show in Section 4, it is a consequence of the structure of P_{wm} that holds for *any* example of a win-move graph and order of evaluation. Moreover, we will also show there that the result is in accordance with the well-founded semantics.

Since the order of picking rules for evaluation does not change the final result, we have much more flexibility when we distribute the computation, as discussed next.

3.1 Distributed Execution

We assume the contents of the relations `move`, `won`, `may_win`, and `good_move` to be horizontally distributed across multiple nodes. A *global* database state thus only exists virtually as a union over all *local* databases. We would like each node to be able to take part in the computation, i.e., execute rules and update the global state. To do so, it is necessary that a node can judge by only investigating its *local* state whether the *global* database implies a certain valuation for a body of a rule. Consider a rule that has only positive body literals, e.g., `a(X) ← b(X),c(X)`. Clearly, if a node N has `b(1)` and `c(1)` amongst its local data, than `a(1)` can be derived. If `a(1)` should be stored at a different node N', then an insertion request for `a(1)` is sent to N', who in turn inserts it into its local database. However, consider a rule such as `won(X) ← move(X,Y),¬may_win(Y)`. To be applicable with a binding of `X`=k,`Y`=l on the *global* database state, it is in general not enough to check whether the fact `may_win(`l`)` is missing in the local state, since it could be in one of the other partitions located at a different node. Here, we exploit the fact that data is often not partitioned arbitrarily, but for example, according to a hash-function, or a range-partitioning scheme. In our example here, we assume a globally known, fixed partitioning function h that maps each fact of the active domain to a certain host. In our example, this simply means that each node is responsible for a certain set of positions and its adjacent edges. The tuple `may_win(`l`)`, when existing, is thus stored at node $N := h($`may_win(`l`)`$)$. Now, since node N knows that `may_win(`l`)` *should* be stored *locally*, it can deduce that the valuation `X`=k,`Y`=l is applicable if `may_win(`l`)` is not present in its local database.

Each node applies valuations according to the program

P_{wm}, without coordination. Not only is the final result deterministic, but also can the computation potentially proceed even if communication to other nodes has been lost (or is slow). Returning to example Figure 1(c). Assume a partitioning function that assigns positions a, b, and c to node N_1; j, k, l to node N_2; and the remaining positions to node N_3. Now, consider the case in which the network is partitioned such that N_3 cannot communicate with the other two nodes. Surprisingly, in this case, running program P_{wm} on each node individually with communication only between nodes N_1 and N_2 will compute the complete result! The intuitive reason is that the only dependence between the subgraphs a, b, c, j, k, l and d, e, f, g, h, i is the edge between c and d. But then, the fact that position c is won, can be established from the fact that j is lost and does not require the cooperation of node N_3. From the view of nodes N_1 and N_2, the final result of d is irrelevant in deciding that c is won. Similarly, d can be established to be lost by node N_3 independently.

4. DISORDERLY EVALUATION MODEL

In this section, we formalize the ideas illustrated in the previous section. We proceed in three steps. First, we define an "even more non-deterministic" semantics for Datalog$^{\neg}_{\forall}$ which we call the *disorderly* semantics. The basic idea is to relax the non-deterministic semantics by allowing deferred application of updates, to model network delay effects. Second, we identify a syntactic fragment of Datalog$^{\neg}_{\forall}$, called semi-monotone Datalog, where the deterministic, non-deterministic, and disorderly semantics of Datalog$^{\neg}_{\forall}$ coincide; and which is *eventually consistent* in a certain precise sense. Third, we demonstrate that our construction for evaluating win-move is eventually consistent and correct.

4.1 The Disorderly Semantics

Intuitively, in a distributed system, each node only has local knowledge about the global state. As in declarative networking [19], we assume that the "global" database is horizontally distributed accross various nodes in the system. Each node runs a copy of the *same* Datalog$^{\neg}_{\forall}$ program, but has access only to its own *local* data. During the distributed computation, a node fires rules of the program based on its locally available data, requests updates that are either applied locally or shipped to other nodes. Crucially, even rules using negation in the body are fired using just locally available information; correctness of the scheme will rely on source data being partitioned in such a way that the local node always sees a conservative underestimate of the relevant negative information, making it safe to fire such rules. To capture the network message delays and reordering which arise in distributed systems, we include a global bag (multiset) of *pending updates* in the description of the state of the distributed system, manipulated by local nodes in a non-deterministic fashion.

Fix a Datalog$^{\neg}_{\forall}$ program P with schema σ. An *update* is a positive or negative ground literal over $eidb(P) \cup idb(P)$. A *state* of the computation is a pair (I, U) comprising a database instance I over σ and a bag (multiset) U of requested updates over σ. U can be thought of as a collection of pending or deferred updates.

Like the non-deterministic semantics, the disorderly semantics is based on the notion of an *immediate successor*, but this time of states rather than sets of facts. Let (I, U) be a state, let r be a Datalog$^{\neg}_{\forall}$ rule,

$$H \leftarrow \forall \bar{X} B_1, \ldots, B_n,$$

in P, and let ν be a consistent valuation of the free variables in r such that $I \models \forall \bar{X} \nu(B_1) \wedge \cdots \wedge \nu(B_n)$. Then a state (I', U') is an *immediate successor* of (I, U) using r if one of the following holds:

request $I' = I$ and $U' = U \uplus \{\nu(H)\}$, where $\uplus$ denotes bag union

insert A' is an update from U, $I' = I \cup \{A'\}$, and $U' = U - \{A'\}$, where $-$ denotes bag difference

delete $\neg A'$ is an update from U, $I' = I \setminus \{A'\}$, and $U' = U - \{\neg A'\}$, where $-$ denotes bag difference

A state (J, V) is an *eventual successor* of state (I, U) using the rules of P if there exists a sequence $(I_0, U_0) = (I, U), \ldots, (I_n, U_n) = (J, V)$ such that for each i, (I_{i+1}, U_{i+1}) is an immediate successor of (I_i, U_i) using some rule in P.

Under the disorderly semantics, a program has a set of possible outcomes (due to non-deterministic choices). Intuitively, the instance J is part of this result set if a state (J, U) can be reached for which the only further updates that can be requested or applied are "no-ops."

Definition 4.1 Let P be a Datalog$^{\neg}_{\forall}$ program, and let I be a source instance. The *result* of applying P to I under the disorderly semantics is the set of all instances J such that there exists V satisfying (i) (J, V) is an eventual successor of $(I, \emptyset)$ using the rules of P, and (ii) (J, V) is a *terminal state*, i.e., it has no eventual successor (J', V') of (J, V) with $J' \neq J$.

The disorderly semantics is "even more non-deterministic" than the non-deterministic semantics in the following sense:

Proposition 4.2

1. *For any Datalog$^{\neg}_{\forall}$ program P and source instance I, if J is in the result of P under the non-deterministic semantics, then J is in the result of P under the disorderly semantics.*
2. *There exists a Datalog$^{\neg}_{\forall}$ program P, a source instance I, and an instance J such that J is in the result of P under the disorderly semantics, but not under the non-deterministic semantics.*

PROOF. (1) follows from the observation that any computation under the non-deterministic semantics can be emulated by strictly alternating update derivation and update application. To prove (2), consider the Datalog$^{\neg}_{\forall}$ program

$$P = \boxed{\begin{array}{l} \texttt{r} \leftarrow \neg\texttt{r}, \ \neg\texttt{s}. \\ \texttt{s} \leftarrow \neg\texttt{r}, \ \neg\texttt{s}. \end{array}}$$

applied to the empty source instance. Under the non-deterministic semantics, the result is $\{\{\texttt{r}\}, \{\texttt{s}\}\}$, while under the disorderly semantics, the result is $\{\{\texttt{r}\}, \{\texttt{s}\}, \{\texttt{r}, \texttt{s}\}\}$. □

Practical Considerations

For practical reasons, we are interested in Datalog$^{\neg}_{\forall}$ programs P that compute a single, deterministic result, even

while allowing the computation of that result to be carried out in a non-deterministic or disorderly fashion. This is captured by the notion of a *functional fragment* [4] of Datalog$^{\neg}_{\forall}$:

Definition 4.3 A Datalog$^{\neg}_{\forall}$ program P is *functional*, under a given semantics, if the result under that semantics has cardinality ≤ 1 for any source instance I.

However, even functional programs that compute exactly one result can be undesirable from a practical point of view, when they admit divergent evaluation sequences that never reach a terminal state. Consider, for example, the following functional Datalog$^{\neg}_{\forall}$ program:

$$P = \boxed{\begin{array}{ll} \texttt{t} \leftarrow \neg\texttt{a}. & \texttt{p} \leftarrow \texttt{t}. \\ \texttt{a} \leftarrow \texttt{b}. & \neg\texttt{p} \leftarrow \texttt{t}. \end{array}}$$

Evaluated under the disorderly semantics, the result of P on $\{b\}$ is the single output $\{a, b\}$. Yet, if the first rule is used to derive `t` in a state (I, U) (by for example, executing it first), then no eventual successor of (I, U) will satisfy requirement (ii) of Definition 4.1. This is problematic because we intuitively, would want to reach the final result *eventually* even though some "bad choices" have been made while picking the order of evaluation.

Before defining a notion of termination, we first dispatch with one technical issue. Nothing in the formal semantics we have presented so far rules out the possibility that, in a given evaluation trace, the same update will be derived *ad infinitum*, a given rule will never be given a chance to fire, or a certain pending update will never be applied. To rule out such pathological cases, we first introduce a notion of *fairness*, which informally requires that each derivable update is eventually derived, and each derived update is eventually applied.

Definition 4.4 Given a Datalog$^{\neg}_{\forall}$ program P. A sequence of states $(I^0, \emptyset), (I_1, U_1), \ldots$ for which (I^{i+1}, U^{i+1}) is an immediate successor using a rule in P is a *fair trace* if (1) for any state (I^i, U^i) for which there is a rule $r = (head \leftarrow body) \in P$ and valuation ν with $I^i \models \nu(body)$, there are states $S^j = (I^j, U^j)$ and $S^{j+1} = (I^{j+1}, U^{j+1})$ with $j \geq i$ such that S^{j+1} is obtained from S^j by requesting the update $\nu(head)$ (i.e., $U^{j+1} = U^j \cup \{\nu(head)\}$). And also, (2) if $A \in U^i$ ($\neg A \in U^i$), then there are states $S^j = (I^j, U^j)$ and $S^{j+1} = (I^{j+1}, U^{j+1})$ with $j \geq i$ such that S^{j+1} is obtained from S^j by inserting (deleting) A.

Having ruled out these pathological traces, we can now define termination[6] desired:

Definition 4.5 A program P is *terminating* under a given semantics if every fair trace reaches a terminal state.

Finally, combining termination and determinism yields our desired property of Datalog$^{\neg}_{\forall}$ programs:

Definition 4.6 A program P is *eventually consistent* under a given semantics, if it is functional and terminating under this semantics.

It was shown in [4] that for Datalog$^{\neg}_{\forall}$ under the non-deterministic semantics, functionality is undecidable. As might be expected, this property (along with termination and eventual consistency) is undecidable for the disorderly semantics as well:

[6]This property is also often called quiesence, e.g., [9, 15].

Theorem 4.7 *Under the disorderly semantics, functionality, termination, and eventual consistency are undecidable for Datalog$^{\neg}_{\forall}$, even for programs without universal quantification or eidb predicates.*

The proof is by reduction from the undecidable problem of checking containment of (positive) Datalog programs [21], and can be found in the Appendix.

4.2 Semi-monotone Datalog$^{\neg}_{\forall}$

Consider again the win-move program from Section 3, which we saw there compiled into the Datalog$^{\neg}_{\forall}$ program P_{wm}. Observe that the updates in the program follow a certain, regular form: in particular, we only insert into the `won` relation, while we only delete from the `good_move` and `may_win` relations. Moreover, `won` occurs only positively in the bodies of rules, while `good_move` and `may_win` occur only negatively. We are therefore motivated to define the following fragment of Datalog$^{\neg}_{\forall}$:

Definition 4.8 A Datalog$^{\neg}_{\forall}$ program P is *semi-monotone* if the relation names in $idb(P)$ occur only positively, while the relation names in $eidb(P)$ occur only negatively.

Note that in the course of computation for such a program, the $idb(P)$ relations only grow, while the $eidb(P)$ relations only shrink. This justifies the choice of terminology "semi-monotone."

Theorem 4.9 *Every semi-monotone Datalog$^{\neg}_{\forall}$ program is eventually consistent under the disorderly semantics.*

The proof can be found in the Appendix.

We conclude the subsection by noting the following:

Corollary 4.10 *For semi-monotone Datalog$^{\neg}_{\forall}$ programs, the deterministic, non-deterministic, and disorderly semantics coincide.*

Proof. Let P be a semi-monotone Datalog$^{\neg}_{\forall}$ program, and let I be a source instance. Suppose that J is the result of evaluating P on I under the deterministic semantics. By results of Abiteboul and Vianu [5], J is in the result set of P applied to I under the non-deterministic semantics. By Proposition 4.2, J is also in the result set of P applied to I under the disorderly semantics. But since P is semi-monotone, it is functional under that semantics, by Theorem 4.9. It follows that the result under any of the three semantics is exactly J. □

4.3 Correctness of the Transformed Win-Move

Next, we return to the Datalog$^{\neg}_{\forall}$ version P_{wm} of the win-move game presented in Section 3. Since P_{wm} is semi-monotone, Theorem 4.9 tells us that it is eventually consistent. We now show that it also correctly computes the result of the original Datalog$^{\neg}$ under the well-founded semantics:

Lemma 4.11 *Let P be the Datalog$^{\neg}$ version of the win-move game, let (P_{init}, P_{wm}) be its semi-monotone Datalog$^{\neg}_{\forall}$ translation, and let I be a source instance. Denote by J the result of applying first P_{init} to I obtaining I', and then P_{wm} to I', both under the disorderly semantics. Let P' denote the extension of P to include the facts of I. Then for any ground fact* `win(a)` *we have the following:*

$$W_{P'}(\texttt{win}(a)) = \begin{cases} \textit{true} & \textit{iff} \quad \texttt{won}(a) \in J \\ \textit{false} & \textit{iff} \quad \texttt{may\_win}(a) \notin J \\ \textit{undefined} & \textit{iff} \quad \texttt{won}(a) \notin J \text{ and} \\ & \qquad \texttt{may\_win}(a) \in J \end{cases}$$

PROOF SKETCH. The basic idea is to show that the alternating fixpoint computation is simulated by a certain disorderly computation. (Since P_{wm} is semi-monotone, it is functional by Theorem 4.9, hence any disorderly computation will produce the same result.) The initialization stage computes the set of first overestimates. The proof is then done by induction on the length of the alternating fixpoint computation Γ^i using a strengthened induction hypothesis (which includes an added `good_move` relation to the alternating fixpoint sequence). The chosen execution for the disorderly semantics, computes all immediate consequences of the first rule in P_{wm}(after which the underestimates agree), then the secon and the third (after which the overestimates agree). Induction is done by proving the claim for $i = 0, 1$ as a base. Proving for $i = 2k + 2$ assuming $i = 2k$ and $i = 2k + 1$, i.e., correctness of a new underestimate es easy to show, since applying rule 1 to the earlier state naturally computes the new state. Correctness of the overestimates $i = 2k+1$ is a little trickier and requires applying induction hypothesis for $i = 2k, 2k-1, 2k-2$. The key insight here is that the `good_move` from both programs agree in the odd i, i.e., in the over-estimations. □

5. TRANSDUCER NETWORKS

In this section we develop four closely-related models for distributed computations.

The first model, denoted $\mathcal{N}_0$, is that of relational transducer networks as defined by Ameloot et al. [9]. The other three models are new and vary from $\mathcal{N}_0$ along two dimensions: how the input data is distributed across the network; and how much a transducer node "knows" about the distribution process. In the model $\mathcal{N}_1$, input facts are distributed according to a distribution policy which assigns each fact of the Herbrand base over the extensional schema to one or more nodes. Further, the distribution policy is known to each node in the transducer network, in a sense we shall make precise shortly. In model $\mathcal{N}_2$, the facts of a k-ary relation are distributed with a replication factor of at most k, determined by their attribute values. Finally, model $\mathcal{N}_3$ differs from $\mathcal{N}_1$ in that each transducer also has information about the active domain of the global input instance.

In the following, we describe these models in detail, beginning with a review of transducers and networks of relational transducers (model $\mathcal{N}_0$).

5.1 Background

We follow the paper by Ameloot et al. [9] in presenting the background notions of relational transducers [6] and relational transducer networks [9].

Relational Transducers

A *transducer schema* is a tuple $(\mathcal{S}_{\text{in}}, \mathcal{S}_{\text{sys}}, \mathcal{S}_{\text{msg}}, \mathcal{S}_{\text{mem}}, k)$ of four disjoint database schemas along with an arity k. (The subscripts stand for 'input', 'system', 'message', and 'memory', respectively.) An *abstract relational transducer* (*transducer* for short) over this schema is a collection of queries $\{Q^R_{\text{snd}} \mid R \in \mathcal{S}_{\text{msg}}\} \cup \{Q^R_{\text{ins}} \mid R \in \mathcal{S}_{\text{mem}}\} \cup \{Q^R_{\text{del}} \mid R \in \mathcal{S}_{\text{mem}}\} \cup \{Q_{\text{out}}\}$, where

- every query is over the combined schema $\mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{sys}} \cup \mathcal{S}_{\text{msg}} \cup \mathcal{S}_{\text{mem}}$;
- the arity of each Q^R_{snd}, each Q^R_{ins}, and each Q^R_{del} equals the arity of R; and
- the arity of Q_{out} equals the output arity k.

Here, 'snd' stands for 'send'; 'ins' stands for 'insert'; 'del' stands for 'delete'; and 'out' stands for 'output'. A *state* of the transducer is an instance of the combined schema $\mathcal{S}_{\text{in}} \cup \mathcal{S}_{\text{sys}} \cup \mathcal{S}_{\text{mem}}$. A *message instance* is an instance of $\mathcal{S}_{\text{msg}}$. Such a message instance can stand for a set of messages (facts) received by the transducer, or a set of messages sent by the transducer; the intended interpretation will always be clear in context.

Let $\mathcal{T}$ be a transducer. A *transition* of $\mathcal{T}$ is a five-tuple $(I, I_{\text{rcv}}, J_{\text{snd}}, J_{\text{out}}, J)$, also denoted $I, I_{\text{rcv}} \xrightarrow{J_{\text{out}}} J, J_{\text{snd}}$, where I and J are states, I_{rcv} and J_{snd} are message instances, and J_{out} is a k-ary relation such that

- every query of $\mathcal{T}$ is defined on $I' = I \cup I_{\text{rcv}}$;
- J agrees with I on $\mathcal{S}_{\text{in}}$ and $\mathcal{S}_{\text{sys}}$;
- $J_{\text{snd}}(R)$, for each $R \in S_{\text{msg}}$, equals $Q^R_{\text{snd}}(I')$;
- J_{out} equals $Q_{\text{out}}(I')$;
- $J(R)$, for each $R \in S_{\text{mem}}$, equals R' as follows. Let the set of insertions for R be $R^+ := Q^R_{\text{ins}}(I') \setminus Q^R_{\text{del}}(I')$, the set of deletions for R be $R^- := Q^R_{\text{del}}(I') \setminus Q^R_{\text{ins}}(I')$. Then, $R' := (R \cup R^+) \setminus R^-$. (That is, conflicting updates are ignored.) Note that this allows assignment $R' := Q$ to be implemented.

The intuition behind the instance I' is that $\mathcal{T}$ sees its input, system and memory relations, plus its received messages. The transducer does not modify the input and system relations. The transducer computes new tuples that can be sent out as messages; this is the instance J_{snd}. The transducer also outputs some tuples (which cannot later be retracted); this is the relation J_{out}. Finally the transducer updates its memory by inserting and deleting some tuples in its memory relations.

Transducers are parameterized by the language $\mathcal{L}$ in which the queries are expressed. A UCQ-transducer, for example, uses unions of conjunctive queries. Note that transducer transitions are deterministic, in constrast to those of transducer networks, discussed next.

Transducer Networks

Next we recall transducer networks (model $\mathcal{N}_0$), in which relational transducers are placed on nodes in a communicating network. Here a *network* is a connected (not necessarily complete), directed graph of *nodes* $N \subsetneq dom$. By insisting that the graph is connected, we ensure that information can (eventually) flow between any two nodes. A *transducer network* is a pair $(N, \mathcal{T})$ where N is a network and $\mathcal{T}$ is a relational transducer.

Operationally, each node in the network has a relational transducer and a *receive buffer* of incoming messages. In the initial state, all memory relations and receive buffers of all nodes are empty. The system relations contain useful information about the transducer network and input distribution (we will formally describe what this entails). The input I of schema $\mathcal{S}_{\text{in}}$ to the transducer network is partitioned (possibly with replication) across the input relations via a partitioning function H that maps every node n to a subset of I, such that $I = \bigcup_{n \in N} H(n)$.

The (global) *state* of a transducer network is a function $\mathcal{S}$ mapping each node $n \in N$ to a pair (I, B) where I is a (local) transducer state and B, the *receive buffer*, is a bag of facts

over the schema $\mathcal{S}_{\text{msg}}$. The state of a transducer network evolves via two kinds of transitions. In a *delivery transition*, a node reads and removes one fact over the schema $\mathcal{S}_{\text{msg}}$ from its input buffer B, adds the fact to the appropriate memory relation, makes a local transducer transition transforming the local state I to J, and sends the resulting message instance J_{snd} to its neighbors. A *heartbeat transition* is the same as a delivery transition, but no input is read from the input buffer. Sending J_{snd} to neighbors means that after the transition, J_{snd} is added to the input bags of the neighboring nodes.

Non-Determinism and Desired Properties

To define desired properties of network transducers, we first require the notion of a run. A *run* of a transducer network $(N, \mathcal{T})$ on input I according to a partitioning function H is an infinite sequence $(\tau_n)_n$ of transitions starting from an input configuration with empty network buffers, empty memory relations, and input relations populated according to the partitioning function H. The *result* of a run is defined as the union over all J_{out} produced during the transitions of the run. A run is *fair* if every node does heartbeat transitions infinitely often, and every fact in every message buffer is eventually taken out by a delivery transition.

Network transducer transitions are non-deterministic in several respects: from a given state, many transitions are possible in general depending on the choice of node where the transition occurs, the choice of transition type (heartbeat or delivery), and, for delivery transitions, the choice of fact delivered. These correspond to the kinds of non-determinism found in real distributed systems.

It is desirable nevertheless for transducer networks to produce the same output regardless of the network topology, partitioning strategy, or non-deterministic choices of the run. We formalize this notion below.

Definition 5.1 A transducer $\mathcal{T}$ *computes* the query Q if for any input I, network N, and distribution H, the result of any fair run of $(N, \mathcal{T})$ on I according to H is $Q(I)$.

5.2 Variations on Transducer Networks

In basic relational transducer networks (model $\mathcal{N}_0$), the horizontal partitioning of the input data is done arbitrarily and, without communication, nodes know only which part of the input data was assigned to them. By varying these assumptions, we derive three natural variations on the basic model.

In all of our variations, we derive the horizontal partition function for a particular input I from an instance-independent *partitioning policy*. A *partitioning policy* for a schema $\mathcal{SP}$ and network N is a computable function $\mathcal{P}$ that associates with each ground atom in the Herbrand base of $\mathcal{SP}$ a non-empty subset of the nodes of N. The domain of the function $\mathcal{P}$ is infinite (the policy is independent of a particular input instance), covering all "potential" tuples. Given a partitioning policy $\mathcal{P}$ and an input instance I, we define the horizontal partition function $H_{\mathcal{P},I}$ used for distributing the input instance data as follows:

$$H_{\mathcal{P},I}(i) := \{f \mid f \in I \text{ and } i \in \mathcal{P}(f)\}$$

Note that for any horizontal partioning H of an input instance I, there is a partitioning policy $\mathcal{P}$ such that $H_{\mathcal{P},I} = H$. (For one of the models to be described, however, we will restrict the allowed partitioning policies such that this no longer holds.)

Next, we allow each transducer *restricted access* to the partitioning policy $\mathcal{P}$ by adding a relation $\texttt{Local}_R$ for each $R \in \mathcal{S}_{\text{in}}$ to its system relations. $\texttt{Local}_R$ has the same arity as R, and a tuple $\bar{x}$ is in $\texttt{Local}_R^n$ (the copy of $\texttt{Local}_R$ at node n) iff $n \in \mathcal{P}(R(\bar{x}))$. Intuitively, on a node n, there is a tuple $\bar{x} \in \texttt{Local}_R$ if n is "responsible" for this tuple. If $R(\bar{x})$ is in the global input I, then $R(\bar{x})$ will be distributed to node n (and possibly others). Conversely, if node n finds $R(\bar{x})$ absent from its local input, then n "knows" that $R(\bar{x})$ is *not* in the global input I.

Note that the $\texttt{Local}_R^i$ are in general infinite relations; in practice each node would be equipped with a decision procedure to check whether an arbitrary tuple is contained in $\texttt{Local}_R^i$ or not. Also, while the queries of a transducer at node i may access $\texttt{Local}_R^i$, we require that the queries still produce finite results. (For UCQ-transducers, for instance, this can be ensured by extending the notion of safety to require all variables occuring in $\texttt{Local}_R$ atoms to also occur positively in normal atoms.) This requirement could be lifted by restricting $\texttt{Local}_R^i$ to the active domain of the global input instance, but we prefer not to do this as we shall see in Lemma 6.3 that providing nodes knowledge of the active domain has a dramatic impact on the notion of coordination-freeness.

We are now ready to define our first variation on relational transducer networks.

Definition 5.2 An $\mathcal{N}_1$-$\mathcal{L}$-*transducer network* is an $\mathcal{L}$-transducer network along with a partitioning policy $\mathcal{P}$, in which each transducer is additionally provided system relations $\texttt{Local}_R$ for all $R \in \mathcal{S}_{\text{in}}$ as described above.

In our second model, we restrict the allowed partitioning policies to those which, intuitively, map domain elements (rather than ground facts) to nodes. This captures the style of distribution used in our construction for the win-move game in the earlier sections. More precisely, a partitioning policy $\mathcal{P}$ is called *element-determined* if there exists a (unique) mapping $F : \mathit{dom} \to 2^N$ of domain elements to sets of nodes such that

$$\mathcal{P}(\, p(x_1, \ldots, x_n)\,) = \bigcup_{i=1,\ldots,n} F(x_i).$$

for every ground fact $p(x_1, \ldots, x_n)$ in the Herbrand base of the transducer's input schema $\mathcal{S}_{\text{in}}$.

Definition 5.3 An $\mathcal{N}_2$-$\mathcal{L}$-*transducer network* is an $\mathcal{N}_1$-$\mathcal{L}$-transducer network whose associated partitioning policy is element-determined.

We do not impose any restriction on $\mathcal{P}$ for nullary predicates beyond what was done for type $\mathcal{N}_2$ transducer networks. Note that the nodes of an $\mathcal{N}_2$-$\mathcal{L}$-transducer network have, essentially, full knowledge of the underlying mapping $F : \mathit{dom} \to 2^N$ for the partitioning policy via their $\texttt{Local}_R$ relations, since $F(a) = \texttt{Local}_R(a, \ldots, a)$ for any domain value a and predicate R.

Finally, our third variation on transducer networks exposes global knowledge of the active domain to nodes.

Definition 5.4 An $\mathcal{N}_3$-$\mathcal{L}$-*transducer network* is an $\mathcal{N}_1$-$\mathcal{L}$-transducer network in which the transducers additionally have access to a system relation $\texttt{adom}$ containing the active domain of the global input instance.

For any of these models, we also require that a transducer is "oblivious" to the network and partitioning policy, in the sense that the same transducer should produce the correct result on any fair run, regardless of the choice of network and partitioning policy.

Definition 5.5 A $\mathcal{L}$-transducer $\mathcal{T}$ *computes* a query Q in model $X \in \{\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3\}$ if for every network N, every distribution policy $\mathcal{P}$ compatible with X, and every input instance I, every run of $\mathcal{T}$ in which the input is distributed according to $H_{\mathcal{P},I}$ results in the output $Q(I)$. In this case we say that $\mathcal{T}$ is *consistent*.

5.3 Disorderly Semantics via Transducer Networks

In this section we will show how certain syntactic classes of semi-monotone programs can be "compiled down" to equivalent transducer networks of various kinds. The semi-monotone program we have given for computing the win-move game will be seen to obey the syntactic restrictions of one of these classes.

Lemma 5.6 *Let Q be a query computed under the disorderly semantics by a semi-monotone, $\forall$-free Datalog$^{\neg\neg}_{\forall}$ program P in which each rule has at most one negated eidb atom. Then there exists a type $\mathcal{N}_1$ UCQ$^{\neg}$-transducer network which computes Q.*

A similar result holds for type $\mathcal{N}_2$ transducer networks. Here, we are allowed to use multiple negated eidb in the body of rules and $\forall$-quantification. A Datalog$^{\neg\neg}_{\forall}$ rule r is *friendly* if any negated atoms in the body of r share a common variable x which is not universally-quantified.

Lemma 5.7 *Let Q be a query computed by a semi-monotone Datalog$^{\neg\neg}_{\forall}$ program P under the disorderly semantics. If all rules in P are friendly, then there exists a type $\mathcal{N}_2$ FO-transducer network which computes Q.*

Even certain kinds of pre-processing are allowed: A Datalog program consisting only of constant-free projection rules, i.e., of the form $\texttt{R'}(\bar{\texttt{Z}}) \leftarrow \texttt{R}(\bar{\texttt{X}})$ with $\emptyset \neq \bar{Z} \subseteq \bar{X}$, and all $X_i \in \bar{X}$ being variables, is called a *projection program*.

Lemma 5.8 *Let P be a projection program, and let Q be a query as defined in Lemma 5.7. Then there exists a type $\mathcal{N}_2$ FO-transducer network which computes $Q \circ P$.*

As is customary, we mean query compostion with $\circ$, i.e., for any input instance I, $Q \circ P := Q(P(I))$. We point out that the programs P_{init} and P_{wm} for computing win-move presented in Section 3 satisfy these requirements.

Proofs. All proofs for these lemmata are constructive. The intuition is that the created transducer networks essentially perform a distributed computation of the disorderly semantics. Each transition of the transducer implements the FO-query denoted by the body of the semi-monotone program applied to its *local state*. Derived updates are locally applied as well as broadcasted to all nodes in the network. Eventual consistency for the disorderly semantics of semi-monotone programs guarantees the consistency of the transducer. The syntactic restrictions and access to the `Local` relations are necessary to allow the evaluation of rule bodies in a distributed manner: Even though each transducer only has partial *local knowledge* of the virtual *global database state*, the restrictions guarantee that (1) each conclusion drawn from the local state could also have been drawn from the global state, and (2) each conclusion that can be drawn from the global state can also be drawn on at least one node with only its local knowledge.

The transducer $\mathcal{T}$ constructed for Lemma 5.8 is a modification of the one constructed for Lemma 5.7 that first computes the projection-program locally. Since the partitioning policy is element-determined, all necessary `Local` system relations for the $idb(P)$, which are the edb and eidb for the transducer, can be emulated.

While the proofs provide valuable insights in how to program the transducer to guarantee consistency and correctness, the details are somewhat technical. We thus decide to skip them here and refer the interested reader to Appendix A.2.

6. COORDINATION

Again following Ameloot et al. [9], we say that an $\mathcal{L}$-transducer $\mathcal{T}$ that computes Q is *coordination-free with respect to model $\mathcal{N}_0$* if for every input I and every network N, there exists a distribution H for which when the transducer network $(N, \mathcal{T})$ is run with only heartbeat transitions, it already produces the correct result. Note that since local heartbeat transitions are deterministic, the result is deterministic for a given input I, network N, and partitioning H. A query Q is *coordination-free in $\mathcal{N}_0$* if there exists a coordination-free $\mathcal{L}$-transducer $\mathcal{T}$ that computes Q, for some query language $\mathcal{L}$.

A main result of the paper by Ameloot et al. [9] relates coordination-freeness and monotonicity:

Theorem 6.1 **[9]** *A query is coordination-free in model $\mathcal{N}_0$ if and only if it is monotone.*

The notion of coordination-free queries extends naturally to models $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$ (replace horizontal distributions with distribution policies in the definitions above for $\mathcal{N}_0$). For $X \in \{\mathcal{N}_0, \mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3\}$, denote by $\mathcal{F}[X]$ the class of queries that are coordination-free with respect to model X. Denote by $\mathcal{C}$ the class of all computable queries, by $\mathcal{M}$ the class of monotone queries, and by $\mathcal{S}$ the class of all queries computable by a semi-positive Datalog$^{\neg}$ program.

The main result of this section is that these classes form the following hierarchy:

Theorem 6.2

$$\mathcal{M} = \mathcal{F}[\mathcal{N}_0] \subsetneq \mathcal{SP} = \mathcal{F}[\mathcal{N}_1] \subsetneq \mathcal{F}[\mathcal{N}_2] \subsetneq \mathcal{F}[\mathcal{N}_3] = \mathcal{C}$$

The first equality is just a restatement of Theorem 6.1; the first strict inclusion is evident; and it is clear from the construction of the models that $\mathcal{F}[\mathcal{N}_0] \subseteq F[\mathcal{N}_1]$, $F[\mathcal{N}_1] \subseteq F[\mathcal{N}_2]$, and $F[\mathcal{N}_1] \subseteq F[\mathcal{N}_3]$. The remainder of the section is devoted to a proof that these inclusions are proper, and indeed that the above hierarchy holds.

6.1 Proof of Main Theorem

First, we show that endowing the nodes with knowledge of the global active domain has a dramatic impact on the class of coordination-free queries:

Lemma 6.3 *Every computable query is coordination-free in model $\mathcal{N}_3$, i.e., $\mathcal{F}[\mathcal{N}_3] = \mathcal{C}$.*

This implies in particular that $\mathcal{F}[\mathcal{N}_2] \subseteq \mathcal{F}[\mathcal{N}_3]$.

PROOF. First, we claim that there is a coordination-free Datalog$^\neg$-transducer $\mathcal{T}$ of type $\mathcal{N}_3$ such that, on any network, and any input I distributed according to any policy, any fair run reaches a configuration where every node has a local copy of the entire instance I in its memory, and an additional flag `ready` (implemented by a nullary memory relation) is true. Moreover, the flag `ready` does not become true at a node before that node has the entire instance in its memory.

We use the programming technique *broadcast* as described in Appendix A.2 to let transducers send messages (containing update requests) that will eventually be delivered to any node in the system; besides sending the updates out, they are also locally applied. The transducer does the following on heartbeat transitions: for each edb-relation `R`, it uses `adom` and $\texttt{Local}_R$ to broadcast existence and non-existence of tuples which is then stored in the memory relations $\texttt{R}_m$, and $\texttt{R}^c_m$, respectively. With `@all` denoting broadcast:

$\texttt{R}_m\texttt{@all}(\bar{\texttt{X}}) \leftarrow \texttt{R}(\bar{\texttt{X}})\texttt{.}$
$\texttt{R}^c_m\texttt{@all}(\bar{\texttt{X}}) \leftarrow \texttt{adom}(\texttt{X}_1)\texttt{,...,adom}(\texttt{X}_n)\texttt{,Local}_R(\bar{\texttt{X}})\texttt{,}\neg\texttt{R}(\bar{\texttt{X}})\texttt{.}$

Then, each edb relation `R` also has a flag `readyR` that is set as follows:

```
knownR(X̄) ← R_m(X̄).
knownR(X̄) ← R^c_m(X̄).
not_readyR ← adom(X_1),...,adom(X_n),¬knownR(X̄).
readyR ← ¬not_readyR.
ready ← readyR, readyS, ...
```

Finally, `ready` is implemented as a conjunction of all `readyR` for each edb relation `R`. This Datalog$^\neg$ program is run in each transition by $\mathcal{T}$.

Next, for any query Q expressible in a query language $\mathcal{L}$, let $\mathcal{L}'$ be a query language at least as expressible as both $\mathcal{L}$ and stratified Datalog$^\neg$. The $\mathcal{L}'$-transducer $\mathcal{T}$, which distributedly computes Q, does now first use the sub-routine from above to distribute I to all nodes, and then, on each node, when `ready` is true, computes Q in one step.

$\mathcal{T}$ is coordination-free. For any network, consider the policy that allocates all data to a single node n_0. Even though none of the sent messages arrive, n_0 will reach the "`ready`-state" and output the complete result. □

Lemma 6.4 *In model $\mathcal{N}_1$, for each semi-positive Datalog$^\neg$ program there exists an equivalent coordination-free UCQ$^\neg$-transducer. As a consequence, $\mathcal{SP} \subseteq \mathcal{F}[\mathcal{N}_1]$.*

PROOF. Each semi-positive Datalog$^\neg$ program P is also a semi-monotone program, which additionally satisfies the syntactic restrictions identified in Lemma 5.7. Let $\mathcal{T}$ be the type $\mathcal{N}_1$ transducer as constructed in the proof for Lemma 5.7. We observe that $\mathcal{T}$ is coordination-free. Consider again the partitioning policy mapping all data to a single node n_0. $\mathcal{T}$ on 1 will run the complete semi-monotone program already with only heartbeat transitions. □

Lemma 6.5 *The query computing the won positions of the win-move game is $\mathcal{N}_2$-coordination free.*

PROOF. Lemma 4.11 states that $P_{wm} \circ P_{init}$ computes the won positions for win-move. P_{wm} satisfies the restrictions of Lemma 5.7; and P_{wm} is a projection program. Now, consider the $\mathcal{N}_2$-FO-transducer $\mathcal{T}$ that was constructed in the proof for Lemma 5.8. $\mathcal{T}$ is coordination-free: consider the partitioning policy that puts all data on a single node n_0. □

Lemma 6.6 *$\mathcal{F}[\mathcal{N}_1]$ is strictly contained in $\mathcal{F}[\mathcal{N}_2]$.*

PROOF. Considering Lemma 6.5, we assume towards a contradiction that there is a coordination-free $\mathcal{N}_1$-transducer $\mathcal{T}$ computing the won positions of the win-move game. Let $I = \{\texttt{move(a,b)}\}$. Since $\mathcal{T}$ is coordination-free it computes the correct result `win(a)` on all networks N and a partitioning policy $\mathcal{P}$ with only performing k heartbeat transitions for some $k \in \mathbb{N}$. Choose the network that contains only a single node n_0. We observe that this implies $\mathcal{P} \equiv \{n_0\}$. Now, consider input $I' = \{\texttt{move(a,b)}, \texttt{move(b,c)}\}$ on the network $N' = \{n_0, n_1\}$. We choose a partitioning policy $\mathcal{P}'$ that assigns all ground atoms to n_1 except `move(a,b)`, which is assigned to n_0. Consider stepping $\mathcal{T}$ on n_1 with k heartbeat transitions. $\mathcal{T}$ will output `win(a)` since its input and system relations `move` and $\texttt{Local}_{move}$ have the same contents as above (remember, $\mathcal{T}$ is deterministic if run only with heartbeat transitions). This is a contradiction since `win(a)` is *not* part of the correct result for I'. □

This argument can be generalized as follows:

Lemma 6.7 *$\mathcal{F}[\mathcal{N}_1] \subseteq \mathcal{SP}$.*

PROOF. Assume towards a contradiction that $\mathcal{T}$ is a coordination-free type $\mathcal{N}_1$ transducer computing Q and $Q \notin \mathcal{SP}$. Clearly, there exist inputs I and I' such that there is a ground fact f with $f \in Q(I)$ and $f \notin Q(I \cup I')$ (otherwise Q would be monotone, contradicting $Q \notin \mathcal{SP}$). Now, run $\mathcal{T}$ on the network containing one node on input $Q(I)$ with only heartbeat transitions. Since $\mathcal{T}$ is coordination-free it will output f after k heartbeat transitions for a $k \in \mathbb{N}$. Now, consider running $\mathcal{T}$ on $I \cup I'$ on the network with two nodes n_0 and n_1 with

$$\mathcal{P}(x) = \{n_0 | x \in I\} \cup \{n_1 | x \in I'\} \cup \{n_0, n_1 | x \notin I \cup I'\}$$

Step $\mathcal{T}$ on node n_0 with k heartbeat transitions. Note that the content of $\mathcal{S}_{\text{mem}}$ and $\mathcal{S}_{\text{sys}}$ are the same in both scenarios on the node n_0. Since heartbeat-transitions are deterministic, $\mathcal{T}$ will output f and does thus not correctly compute $Q(I \cup I')$ on $N = \{v_0, v_1\}$ with policy $\mathcal{P}$. □

We finish the proof of Theorem 6.2 by showing that the last containment in the hierarchy is strict.

Lemma 6.8 *Not all computable queries are coordination-free in model $\mathcal{N}_2$, i.e., $\mathcal{F}[\mathcal{N}_2] \subsetneq \mathcal{F}[\mathcal{N}_3] = \mathcal{C}$.*

PROOF. Assume towards a contradiction that the query that decides whether a unary relation R is empty is in $\mathcal{F}[\mathcal{N}_2]$. Following the argument from the proof for Lemma 6.6, consider $I = \emptyset$ on the network with one node n_0; and then $I' = \{\texttt{R(a)}\}$ together with $\mathcal{P}' \equiv \{n_1\}$ on the network $N' = \{n_0, n_1\}$. □

We point out that although win-move *is* coordination-free in $\mathcal{N}_2$, this cannot be used to show that all well-founded Datalog$^\neg$ is in $\mathcal{F}[\mathcal{N}_2]$, even though "win-move" is a normal-form for well-founded Datalog [13]. The normalform result for win-move as described in [13] requires a recursion-free semi-positive pre-processing step, as in the following program (which can easily be shown to not be $\mathcal{N}_2$-coordination-free).

```
move(x,y) ← r(x,z), s(z,y).
win(x,y) ← move(x,y), ¬win(y).
```

7. RELATED WORK

Our work is inspired by Hellerstein's quest for logic-based foundations of parallel, distributed, data-intensive computing [15], and the search for suitable models of computation and parallel complexity. Hellerstein's model incorporates non-determinism via the choice construct with an explicit representation of time within Datalog; we use instead a bag of pending updates. The paper by Alvaro et al. [7] also studies methods to ensure eventual consistency of distributed computations. In particular, the authors analyze Dedalus/Bloom/Bud programs (a version of distributed Datalog) to identify points of non-monotonicity that require coordination.

Much of our technical approach is inspired by the paper by Ameloot et al. [9], which introduced the framework of networks of relational transducers and was the first to venture a formal notion of coordination complexity. We have shown that slight variations on the model have a dramatic impact on the ensuing notion of coordination complexity.

A more recent paper by Ameloot et al. [8] investigates syntactic restrictions under which eventual consistency of networks of relational transducers can be decided. Our paper complements that work by introducing disorderly evaluation of semi-monotone Datalog$_\forall^{\neg}$ programs and a compilation procedure for such programs guaranteed to create eventually consistent transducer networks.

Our disorderly semantics is closely related to the so-called "production-rule" semantics of Datalog$^\neg$ [5, 23]. Recent work by Abiteboul et al. [1] also investigates confluence and distributed systems, however without the focus on coordination freeness.

The paper by Koutris and Suciu [18] presents a "massively parallel" computation model (MP) for conjunctive queries that takes data skew into account. They show that in the MP model, the so-called "tall-fat queries" are precisely the ones that can be executed in a single stage. It would be interesting to investigate formal notions of coordination that take into account data skew.

Finally, the paper by Brass et al. [10] describes transformation based strategies for incremental, bottom-up evaluation of Datalog$^\neg$ programs under the well-founded semantics [14]. Some of their conditional rewriting techniques resemble ours, but the paper does not address distributed or coordination-free computations.

8. CONCLUSION

We have presented a syntactic fragment of Datalog$_\forall^{\neg}$, semimonotone Datalog$_\forall^{\neg}$, that lends itself to a disorderly, eventually consistent evaluation strategy, while permitting certain kinds of negation. We have shown that the win-move game, the canonical example of a non-stratifiable Datalog$^\neg$ program, can be solved using a semimonotone Datalog$_\forall^{\neg}$ program, and that the program can also be compiled into a relational transducer network for distributed evaluation without resorting to "global synchronization barriers."

We have also introduced several natural variations on relational transducer networks [9], in which various degrees of partitioning policy information are made available to participating network nodes, and we used these models to study the notion of coordination-freeness. We showed that the classes of queries which are coordination-free under these various models form a strict hierarchy, highlighting the sensitivity of the formalization of coordination-freeness by Ameloot et al. to the precise details of how paritioning policy information is made available to nodes. The win-move game, and our semimonotone Datalog$_\forall^{\neg}$ program for solving it, play a starring role in separating two of these classes.

Like Ameloot et al. [9], we do not claim that the notions of coordination-freeness developed here are definitive. (Indeed, current events in American campaign finance law [20] suggest that confusion over the precise meaning of "coordination" is not limited to the database theory community!) However, we feel that our results may usefully inform the research community in the ongoing effort to find a robust notion of coordination-freeness.

9. REFERENCES

[1] S. Abiteboul, M. Bienvenu, A. Galland, and É. Antoine. A rule-based language for web data management. In *Proceedings of the 30th symposium on Principles of database systems of data*, pages 293–304. ACM, 2011.

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

[3] S. Abiteboul and E. Simon. Fundamental properties of deterministic and nondeterministic extensions of datalog. *Theor. Comput. Sci.*, 78(1):137–158, 1991.

[4] S. Abiteboul, E. Simon, and V. Vianu. Non-deterministic languages to express deterministic transformations. In *PODS*, 1990.

[5] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci.*, 43:62–124, August 1991.

[6] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *PODS*, pages 179–187. ACM, 1998.

[7] P. Alvaro, N. Conway, J. Hellerstein, and W. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, 2011.

[8] T. J. Ameloot and J. V. den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. In *ICDT*, 2012.

[9] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *PODS*, 2011.

[10] S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *TPLP*, 1(5):497–538, 2001.

[11] F. Bry. Logic programming as constructivism: a formalization and its application to databases. In *PODS*, 1989.

[12] P. M. Dung and K. Kanchanasut. A natural semantics for logic programs with negation. In *Foundations of Software Technology and Theoretical Computer Science*, LNCS 405, pages 78–88, 1989.

[13] J. Flum, M. Kubierschky, and B. Ludäscher. Total and partial well-founded datalog coincide. In *ICDT*, 1997.

[14] A. V. Gelder. The alternating fixpoint of logic programs with negation. *JCSS*, 47(1):185 – 221, 1993.

[15] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39, March 2010.

[16] D. B. Kemp, D. Srivastava, and P. J. Stuckey.

Bottom-up evaluation and query optimization of well-founded models. *TCS*, 146:145–184, 1995.

[17] P. Kolaitis. The expressive power of stratified logic programs. *Information and Computation*, 90(1):50–66, Jan. 1991.

[18] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, 2011.

[19] B. T. Loo, T. Condie, M. N. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *CACM*, 52(11), 2009.

[20] P. Overby. SuperPACs, candidates: Dancing solo or together? In *National Public Radio*. January 6, 2012. http://www.npr.org/2012/01/06/144801659/a-look-at-super-pacs-and-political-coordination.

[21] O. Shmueli. Equivalence of datalog queries is undecidable. *J. Logic Programming*, 15(3), 1993.

[22] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38:619–649, July 1991.

[23] V. Vianu. Rule-based languages. *Annals of Mathematics and Artificial Intelligence*, 19(1-2):215–259, 1997.

[24] U. Zukowski and B. Freitag. The differential fixpoint of general logic programs. In *DDLP*, pages 45–56, 1996.

APPENDIX

A. PROOFS

A.1 Proofs for Section 4

PROOF. (of Theorem 4.7.) We show the undecidability of all three problems via reduction from the problem of checking containment of (positive) Datalog programs, which is known to be undecidable [21], to deciding termination of Datalog$^{\neg\neg}$. Let P_1 and P_2 be two Datalog programs with $edb(P_1) = edb(P_2)$, assume w.l.o.g. that $idb(P_1)$ and $idb(P_2)$ are disjoint, and let p_1 (resp. p_2) denote the distinguished output predicate of P_1 (resp. P_2).

Termination and eventual consistency. Define P to be the Datalog$^{\neg\neg}_{\forall}$ program that is the union of the rules of P_1 and P_2, along with

```
toggle ← p1(X̄), ¬p2(X̄).
¬toggle ← toggle.
```

and with $eidb(P) = \emptyset$. It is easy to see that P is terminating under the disorderly semantics iff P_1 is contained in P_2. Since P is also functional—the result being empty for any input database instance—it follows that P is also eventually consistent iff P_1 is contained in P_2.

Functionality. Introduce a disjoint copy P_1' of P_1 (with distinguished output predicate p_1'), and define P to be the Datalog$^{\neg\neg}_{\forall}$ program that is the union of the rules of P_1, P_1', and P_2, along with

```
q(X̄) ← p1(X̄), ¬p1'(X̄).
¬q(X̄) ← p2(X̄).
```

where again $eidb(P) = \emptyset$. If P_1 is contained in P_2, then the result of P is empty on any source database, hence P is functional. On the other hand, if P_1 is not contained in P_2, then the order of firing the rules of P_1 and P_1' matters, and P is not functional. □

PROOF. (of Theorem 4.9.) Termination immediately follows from the fact that each ground atom from $ground(P, I)$ can only be changed (inserted into I or deleted from I) once. A finite number of changes implies termination since any (fair) trace has to reach a fixpoint.

It remains to establish functionality. First, it will be convenient to assume that use of the **request** case of the immediate successor relation is restricted to disallow requesting "useless" updates as follows: an insertion of A' is requested in a state (I, U) only if $A' \notin I$ and $A' \notin U$; conversely, a deletion of $\neg A'$ is requested only if $A' \in I$ and $\neg A' \notin U$. Using the semi-monotonicity of P, one can check that this restriction does not affect the result sets for P under the disorderly semantics.

Now, fix a semi-monotone Datalog$^{\neg\neg}_{\forall}$ program P and input instance I. Under the amended definition of immediate successor given above, we claim (1) that any sequence $(I_0, U_0), (I_1, U_1), \ldots$ of states such that for each i, (I_{i+1}, U_{i+1}) is an immediate successor of (I_i, U_i) using some rule in P must be finite. Indeed, this is a straightforward consequence of our earlier observation that the $idb(P)$ relations only grow during computation, while the $eidb(P)$ relations only shrink (and the finiteness of I).

Next, observe (2) that if J is in the result of applying P to I under the disorderly semantics, then $(J, \emptyset)$ is an eventual successor of $(I, \emptyset)$. To see this, suppose that (J, U) is an eventual successor of $(I, \emptyset)$. Let $(J', \emptyset)$ be the eventual successor of (J, U) reached by repeated application of the pending updates. By Definition 4.1, since J is in the result set, it must be the case that $J' = J$. Since $(J', \emptyset)$ is also an eventual successor of $(I, \emptyset)$, we conclude that $(J, \emptyset)$ is an eventual successor of $(I, \emptyset)$ as required.

Then, we show (3) the following weak confluence property: any distinct immediate successors (I_1, U_1) and (I_2, U_2) of a state (I, U) have a common immediate successor (J, V). Intuitively, whatever the change from (I, U) to (I_1, U_1), we can show that it still applies to (I_2, U_2), and commutes with the change from (I, U) to (I_2, U_2). We show this by an analysis of the nine possible cases. First, suppose that (I_1, U_1) and (I_2, U_2) follow as immediate successors of (I, U) using just the **insert** or **delete** changes. Then it is clear from inspecting the definition of immediate successor that the changes commute. Next we consider the five remaining cases involving some use of **request**. Suppose that (I_1, U_1) immediately succeeds (I, U) using **request**, with a rule $H \leftarrow \forall \bar{X} B_1, \ldots, B_n$ and valuation ν. It is easy to see that if (I_2, U_2) also succeeds (I, U) using **request**, then the changes commute; and that the same is true for **insert** or **delete**, using the semi-monotonicity of P. The remaining cases are symmetric.

Finally, we observe that (1) and (3) imply that the immediate consequence relation is strongly confluent. Together with (2), this establishes the theorem. □

A.2 Proofs for Section 5.3

A.2.1 *Proofs for Lemmata 5.6 and 5.7*

∗ The proof for Lemma 5.6 below assumes a stronger syntactic restriction requiring each rule to contain only a single negative atom. The follow lemma shows that this can safely be assumed:

Lemma A.1 *Let Q be a query computed under the disorderly semantics by a semi-monotone, $\forall$-free Datalog$^{\neg\neg}$ program P in which each rule has at most one negated eidb atom. Then Q can be computed under the disorderly semantics by a semi-monotone, $\forall$-free Datalog$^{\neg\neg}$ program P' in which each rule has at most one negated atom.*

PROOF SKETCH. Let $r = (H \leftarrow \mathit{body}, \neg A) \in P$ be the rule with more than one negated atoms. Replace r by $(H' \leftarrow \mathit{body})$ and $(H \leftarrow H', \neg A)$ where $H' = R(\bar{x})$ for R being a fresh relation symbol and $\bar{x}$ being all variables occuring in *body*. □

All proofs are constructive. Starting from a semi-monotone program for Q, we construct a coordination-free $\mathcal{N}_1$-FO-transducer-network $\mathcal{T}$. We will see that the policy restriction to be *element-determined* (restricting $\mathcal{T}$ to be a type $\mathcal{N}_2$ transducer) is important to allow multiple negated body atoms and $\forall$-quantification while still guaranteeing that $\mathcal{T}$ computes Q, i.e., is deterministic under all allowed circumstances. The differences for the proofs of Lemma 5.6 and Lemma 5.7 are only required in the proofs of Lemma A.3 and Lemma A.5.

Given Q; let P be the semi-monotone program computing Q. As a first step modify P to P' by adding one idb relation R' for each $R \in \mathit{edb}(P)$ via the rule:

$$\texttt{R'(X)} \leftarrow \texttt{R(X)}.$$

Also, in all other rules of P, replace $\texttt{R(X)}$ by $\texttt{R'(X)}$. Note that it is easy to see that P' computes the same query as P; also P' is semi-monotone and satisfies the requirements from Lemma 5.6 (resp. Lemma 5.7) if P did.

We then enrich P' with the system-relations $\texttt{Local}_R$ for each $R \in \mathit{eidb}(P)$ to obtain P'' in the following way: Consider each rule r of P' in turn. For each body atom $R(\bar{x})$, add an additional atom $\texttt{Local}_R\ (\bar{x})$ to the body of r if R is an eidb relation of P'; if $\texttt{Local}_R\ (\bar{x})$ contains a $\forall$-quantified variable x, replace x with the variable z as in the requirement of Lemma 5.7.

Example A.2 Applied to the semi-monotone win-move program P_{wm}, this two-step transformation yields:

```
0  move'(X,Y) ← move(X,Y).
1  won(X) ← move'(X,Y),
            ¬may_win(Y), Local_may_win(Y).
2  ¬good_move(X,Y) ← won(Y), move'(X,Y).
3  ¬may_win(X) ← ∀Y ¬good_move(X,Y),
            Local_good_move(X,X), move'(X,_).
```

(P''_{wm})

Before describing the operations in $\mathcal{T}$, we develop some programming techniques for transducers. We first need to emulate *update-able eidb relations* in $\mathcal{N}_1$-FO-transducers, since the relations in $\mathcal{S}_{\text{in}}$ are read-only for the transducers. To do so, we essentially program the transducer to—as a first action—copy the data of eidb relations into the memory. During normal operation, we refer to the memory version of the eidb relations. The construction is a little tedious because we cannot control whether the transducer performs heartbeat or network transitions in the beginning. However, with a number of boolean flags (e.g., to indicate when the copy had happened), and buffers for to-early-read network tuples, this can be done. To simplify notation later on, we also copy the edb relations into the memory of the transducers; these will not be changed during the local computation.

We then need another programming technique: *broadcast*. We *emulate* broadcast by always including the received fact in a network transition in the facts to be sent out to the neighbors. Since the network is connected every fact that is output on any transducer will eventually be read by any other transducer in the network.

We now describe the operations of the $\mathcal{N}_1$-FO-transducer $\mathcal{T}$. The FO queries that implement that behavior are easy to envision. $\mathcal{T}$ has the idb, eidb and edb relations of P' in his memory; the $\texttt{Local}_R$ relations are system relations. Note that edb and eidb relations are initially partitioned according to $H_{\mathcal{P},I}$. All updates to idb and eidb relations will be broadcasted during the run of the transducer network. During each transition, $\mathcal{T}$ evaluates the FO-queries denoted by the *bodies* of the semi-monotone program P'' over its memory to determine a set of updates. These updates are (1) locally applied (via the appropriate Q^R_{ins} and Q^R_{del} relations); (2) also broadcasted (via appropriate Q^R_{snd}); and (3) output (via Q_{out}) if they are insertions for the designated idb output predicate. Since the relation-name R already determines whether an update is an insertion or a deletion the update ground atom $R(\bar{x})$ can be sent. On the receiver side, it is then put into appropriate Q^R_{ins} and Q^R_{del} relation based on whether R is an idb or eidb.

Fix a network N and an arbitrary partitioning policy $\mathcal{P}$. Next, we will show a correspondence between (1) any run $\mathit{run}(I, \mathcal{T}, N, H)$ of the constructed transducer $\mathcal{T}$ in network N on an arbitrary input I over the schema $\mathit{edb}(P) \cup \mathit{eidb}(P)$ with H determined by $\mathcal{P}$ and (2) a sequence of eventual successors under the disorderly semantics of running P' on I (note that the disorderly program we compare to is the $\texttt{Local}_R$-free version P').

The main difference between the transducer network and the trace of the disorderly program is that in the transducer network the program P'' is run with *local knowledge* whereas in the disorderly semantics, an applicable body atom is evaluated based on the "global" database state. The next lemma states that, *at least*, the transducer network is not performing any *wrong* updates, provided the syntactic restrictions from Lemma 5.6 and Lemma 5.7 are met for the respective transducer types. We will use this lemma to subsequently show that the transducer is deterministic; which will allow us to choose any order of transitions. Which is then sufficient to show that the network transducer is not *missing* any derivations, i.e., each fair run of the transducer corresponds to a fair trace of the disorderly semantics.

We first relate the global state $\mathcal{S}$ of a transducer network to a single, database instance $M(\mathcal{S})$ over $\mathcal{S}_{\text{mem}}$. For any $R \in \mathit{edb}(P'') \cup \mathit{idb}(P'')$, $M(\mathcal{S})$ contains the *union* of the memory relations R across the network. For $R \in \mathit{eidb}(P'')$ we take the *intersection modulo scope*, that is a fact $R(\bar{x})$ is in $M(\mathcal{S})$ iff it is present on *all* nodes v with $v \in \mathcal{P}(R(\bar{x}))$, i.e., that have $R(\bar{x})$ in scope. We ommit the obvious (but somewhat tedious) formal definition. Furthermore, let $N(\mathcal{S})$ be the *bag* containing the bag-union of all local network buffers.

Lemma A.3 *Let $\mathcal{S}$ be an arbitrary global state of the transducer $\mathcal{T}$. Let $S_D := (M(\mathcal{S}), N(\mathcal{S}))$ be a state for the disorderly semantics as defined in Section 4.1. Let v be an arbitrary node in N with $\mathcal{S}(v) = (J, B)$, i.e., the contents of its memory relations being J. Then, $R(\bar{x}) \in Q^R_{\text{snd}}(J)$ implies that an insert/delete-update $\nu(H) = R(\bar{x})/\neg R(\bar{x})$ can be derived in S_D according to P'.*

PROOF. Let $R(\bar{x})$ be a ground atom with $R(\bar{x}) \in Q^R_{\text{snd}}(J)$; Since Q^R_{snd} is the FO-query obtained from the bodies of P'', the needs to be a valuation ν such that $J \models \nu(body'')$ for a rule

$$(\neg)R(\bar{x}) \leftarrow body'' \quad \in P''.$$

We need to show that not only $J \models \nu(body'')$ but also $M(\mathcal{S}) \models \nu(body')$ with $body'$ being the body of the corresponding rule in P'. Individually, consider all body atoms $A \in body'$. If A is a positive atom, then clearly $M(\mathcal{S}) \models \nu(A)$. Now consider a negative body atom $\neg R(\bar{x})$ not containing a $\forall$-quantified variable. According to our transformation from P' to P'', also the atom $\texttt{Local}_R\ (\bar{x})$ is in $body''$; thus, v has $R(\bar{x})$ in scope but does not contain $R(\bar{x})$, which implies $R(\bar{x}) \notin M(\mathcal{S})$. Without $\forall$-quantification, this completes the proof. Note, that for this lemma the restriction to one negative body atom in Lemma 5.6 is not required. The existence of $\forall$-quantified body atoms requires the restrictions given in Lemma 5.7 and a restriction to type $\mathcal{N}_2$ transducers. W.l.o.g., consider the single "ground" body atom with only one $\forall$-quantified variable. (It is easy to see that the argument holds for the other cases too). $\forall \texttt{Z}\ \neg\texttt{R}(\texttt{c}_1..\texttt{c}_k,\texttt{Z},\texttt{c}_{k+1}..\texttt{c}_n)$ with $\texttt{c}_i$ being constants from the domain. Since $body''$ is *friendly*, the exists at least one $\texttt{c}_i$ such that the non-ground atom had a variable $x \neq \texttt{Z}$ with $\nu(x) = \texttt{c}_i$. Since $body''$ is guarded by $\texttt{Local}_R$, $J \models \texttt{Local}_R(\texttt{c}_1..\texttt{c}_k,\texttt{c}_i,\texttt{c}_{k+1}..\texttt{c}_n)$, thus for the node v under consideration $v \in \mathcal{P}(\texttt{R}(\texttt{c}_1..\texttt{c}_k,\texttt{c}_i,\texttt{c}_{k+1}..\texttt{c}_n))$. But since $\mathcal{P}$ is *element-determined*, we can conclude that for all z ranging over the universe, $v \in \mathcal{P}(\texttt{R}(\texttt{c}_1..\texttt{c}_k,z,\texttt{c}_{k+1}..\texttt{c}_n))$. Thus, v is responsible for all relevant tuples which implies the goal $M(\mathcal{S}) \models \forall \texttt{Z}\ \neg\texttt{R}(\texttt{c}_1..\texttt{c}_k,\texttt{Z},\texttt{c}_{k+1}..\texttt{c}_n)$. □

We now show that any transition taken by the relational transducer can be emulated by the disorderly semantics, in the following sense:

Lemma A.4 *If the transducer network $\mathcal{T}$ can transition from a state $\mathcal{S}$ to $\mathcal{S}'$, then $(M(\mathcal{S}'), N(\mathcal{S}'))$ is an eventual successor of $(M(\mathcal{S}'), N(\mathcal{S}'))$ under the disorderly semantics for P'.*

PROOF SKETCH. The proof is technical, but straightforward using Lemma A.3. Using multiple request and update rule invocations, the disorderly semantics can emulate the set-based FO-queries of a single transducer. A key ingredient is the fact that if a node v in the transducer network is the first (amongst all nodes) that derives a new idb fact (or delete an eidb fact), then $M(\mathcal{S})$ changes; if it was not the first then $M(\mathcal{S})$ does not change even though $\mathcal{S}$ does. In this case, the change in pending network messages can be accomodated by the disorderly semantics since P' is semi-monotone. □

It is important to *also* show that the trace computed by the transducer network corresponds to a *fair trace* of the disorderly semantics according to Definition 4.4.

Lemma A.5 *Every fair run of $\mathcal{T}$ corresponds to a fair trace of P' under the disorderly semantics.*

PROOF SKETCH. Being a fair run ensures that every sent update is eventually applied. The more interesting part of the proof is to show condition (1) from Definition 4.4. This is somewhat dual to Lemma A.3. We need to show that being able to derive an update in the global state implies that there is (*or will be in a later state*) a node that can derive the same update based on its local knowledge. That is, we need to show if $M(\mathcal{S}) \models \nu(body')$ then there exists an eventual successor state $\mathcal{S}'$ of $\mathcal{S}$ that will be reached by any fair run of $\mathcal{T}$, for which there is a node $v \in N$ with memory contents J such that $J \models \nu(body'')$. Fix a $\nu(body')$ with $M(\mathcal{S}) \models \nu(body')$. If $body'$ does not contain any negated atoms than $body'' = body'$. But now, the claim is easy to show since either $body'$ is a single edb atom (in which case we are done according to the definition of $M(\mathcal{S})$) or $body'$ is a join of *only* idb relations; the claim then follows from the fact that we broadcast all idb-updates and the fact that P' is semi-monotone.

The more interesting cases are when $body'$ contains negated subgoals. According to the same argument as above, we can w.l.o.g., assume that on all nodes v the positive body atoms are satisfied. Next, a differentiated analysis of (1) Lemma 5.6 and (2) Lemma 5.7 is necessary. **(1)** $body'$ contains only one negated atom $\neg R(\bar{x})$. Since $M(\mathcal{S}) \models \neg R(\bar{x})$, none of the nodes $v \in \mathcal{P}(R(\bar{x}))$ contain $R(\bar{x})$ in their memory; consider one of these nodes v: both $\texttt{Local}_R\ (\bar{x})$ and $\neg R(\bar{x})$ are true on v. **(2)** Following the argument above, we see that for each $\neg R(\bar{x})$,$\texttt{Local}_R(\bar{x})$ pair in $body''$ there exists a node $v \in N$ for which $\nu(body'')$ makes both true. We would like one single node v' on which all of the pairs are true. Now, because the rule is friendly, all $\texttt{Local}_R(...)$ have one variable x in common. Since Lemma 5.7 requires a type $\mathcal{N}_2$ transducer and thus $\mathcal{P}$ is element-determined, we can conclude that $v' = F(\nu(x))$ is the desired node. □

Since an initialization of the transducer network $\mathcal{T}$ corresponds to the initial disorderly state, Lemmata A.4 and A.5 prove Theorem 5.6 and 5.7.

A.2.2 *Proof for Lemma 5.8*

Consider the semi-monotone program P_Q for Q. Replace occurrences of $R \in edb(P_Q)$ by their definition in P; the resulting program P'_Q is still semi-monotone. So, w.l.o.g., assume P only defines projections for relations R that are used as eidb in P_Q. Also, for later, let T be part of $edb(P)$. To simplify the exposition, consider only one of such R. We now modify the transducer constructed in the proof for Lemma 5.7. First, we add R to the eidb relations of the transducer (changing $\mathcal{S}_{\text{mem}}$ and $\mathcal{S}_{\text{msg}}$). We now change the procedure that initializes the memory relations from the input relation and compute the projection query during the copy process. While doing so, we also take the partioning policy $\mathcal{P}$ into consideration. Here, we use $\texttt{Local}_T$ to get access to the function H as defined just above Definition 5.3. In particular, for a projection rule $\texttt{R}(\bar{\texttt{Z}}) \leftarrow \texttt{S}(\bar{\texttt{X}})$, we use a UCQ as follows to decide what fragment of $\texttt{S}$ to copy into $\texttt{R}$.

$$\begin{aligned}
&\texttt{R}(\texttt{Z}_1,..,\texttt{Z}_n) = \texttt{S}(\bar{\texttt{X}}),\ \texttt{Local}_T(\texttt{Z}_1,\texttt{Z}_1,..,\texttt{Z}_1).\\
&\texttt{R}(\texttt{Z}_1,..,\texttt{Z}_n) = \texttt{S}(\bar{\texttt{X}}),\ \texttt{Local}_T(\texttt{Z}_2,\texttt{Z}_2,..,\texttt{Z}_2).\\
&...\\
&\texttt{R}(\texttt{Z}_1,..,\texttt{Z}_n) = \texttt{S}(\bar{\texttt{X}}),\ \texttt{Local}_T(\texttt{Z}_n,\texttt{Z}_n,..,\texttt{Z}_n).
\end{aligned}$$

The second modification in the process of creating $\mathcal{T}$ is a change to the program P''_Q (as by the proof for Lemma 5.7). We emulate the (non-existing system) relation $\texttt{Local}_R(\bar{\texttt{Z}})$ with $\texttt{Local}_T$ as done above. It is easy to see that the emulated $\texttt{Local}_R$ and the contents of $\texttt{R}$ corresponds to an allowed distribution policy for a type $\mathcal{N}_2$ transducer.