Weaker Forms of Monotonicity for Declarative Networking: a More Fine-grained Answer to the CALM-conjecture

Tom J. Ameloot*
Hasselt University &
transnational University of Limburg
tom.ameloot@uhasselt.be

Bas Ketsman
Hasselt University &
transnational University of Limburg
bas.ketsman@uhasselt.be

Frank Neven
Hasselt University &
transnational University of Limburg
frank.neven@uhasselt.be

Daniel Zinn
LogicBlox, Inc
daniel.zinn@logicblox.com

ABSTRACT

The CALM-conjecture, first stated by Hellerstein [23] and proved in its revised form by Ameloot et al. [13] within the framework of relational transducer networks, asserts that a query has a coordination-free execution strategy if and only if the query is monotone. Zinn et al. [32] extended the framework of relational transducer networks to allow for specific data distribution strategies and showed that the non-monotone win-move query is coordination-free for domain-guided data distributions. In this paper, we complete the story by equating increasingly larger classes of coordination-free computations with increasingly weaker forms of monotonicity and make Datalog variants explicit that capture each of these classes. One such fragment is based on stratified Datalog where rules are required to be connected with the exception of the last stratum. In addition, we characterize coordination-freeness as those computations that do not require knowledge about *all* other nodes in the network, and therefore, can not globally coordinate. The results in this paper can be interpreted as a more fine-grained answer to the CALM-conjecture.

Categories and Subject Descriptors

H.2 [**Database Management**]: Languages; H.2 [**Database Management**]: Systems—*Distributed databases*; F.1 [**Computation by Abstract Devices**]: Models of Computation

Keywords

Distributed database, relational transducer, consistency, coordination, expressive power, cloud programming

*PhD Fellow of the Fund for Scientific Research, Flanders (FWO).

PODS'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2594538.2594541.

1. INTRODUCTION

Declarative networking is an approach where distributed computations are modeled and programmed using declarative formalisms based on extensions of Datalog. On a logical level, programs (queries) are specified over a global schema and are computed by multiple computing nodes over which the input database is distributed. These nodes can perform local computations and communicate asynchronously with each other via messages. The model operates under the assumption that messages can never be lost but can be arbitrarily delayed. An inherent source of inefficiency in such systems are the global barriers raised by the need for synchronization in computing the result of queries.

This source of inefficiency inspired Hellerstein [11] to formulate the *CALM-principle* which suggests a link between logical monotonicity on the one hand and distributed consistency without the need for coordination on the other hand.[1] A crucial property of monotone programs is that derived facts must never be retracted when new data arrives. The latter implies a simple coordination-free execution strategy: every node sends all relevant data to every other node in the network and outputs new facts from the moment they can be derived. No coordination is needed and the output of all computing nodes is consistent. This observation motivated Hellerstein [23] to formulate the CALM-conjecture which, in its revised form[2], states

> *"A query has a coordination-free execution strategy iff the query is monotone."*

Ameloot, Neven, and Van den Bussche [13] formalized the conjecture in terms of relational transducer networks and provided a proof. Zinn, Green, and Ludäscher [32] subsequently showed that there is more to this story. In particular, they obtained that when computing nodes are increasingly more knowledgeable on how facts are distributed, increasingly more queries can be computed in a coordination-free manner. Zinn et al. [32] considered two extensions of the original transducer model introduced in [13]. In the first extension, here referred to as the *policy-aware* model, every computing node is aware of the facts that should be assigned to it and can consequently evaluate negation over schema relations. In the second extension, referred to as the

[1]CALM stands for Consistency And Logical Monotonicity.
[2]The original conjecture replaced monotone by Datalog [13].

domain-guided model, data distribution is restricted as follows: each possible domain value d is assigned to at least one node; and, when an input fact contains value d, this fact is given to all nodes that d is assigned to. It was shown in [32] that the coordination-free computations within the original, policy-aware, and domain-guided models form a strict hierarchy and that the non-monotone win-move query can be computed by a coordination-free domain-guided transducer network. *The central objective of this paper is to characterize these increasingly larger classes of coordination-free computations in terms of increasingly weaker forms of monotonicity thereby obtaining a more fine-grained answer to the CALM-conjecture.*

Towards this goal, we introduce the set of *domain-distinct-monotone* and the set of *domain-disjoint-monotone* queries, which we denote by $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$, respectively. Recall that a query is monotone if the output does not decrease (w.r.t. set inclusion) when new facts are added. The classes of domain-distinct-monotone and domain-disjoint-monotone queries then correspond to queries with non-decreasing output (again w.r.t. set inclusion) when only facts are added that contain *at least one* and *only* new domain elements, respectively. While $\mathcal{M}_{distinct}$ is a reformulation of the class of queries preserved under extensions (c.f., Section 3.2), $\mathcal{M}_{disjoint}$ appears to be a new class. We semantically characterize the coordination-free computations within the policy-aware and domain-guided model in terms of $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$, respectively, to obtain the following answer to the CALM-conjecture: for a query Q,

(*i*) Q can be computed by a coordination-free (original) transducer network iff Q is monotone; [13]

(*ii*) Q can be computed by a coordination-free policy-aware transducer network iff $Q \in \mathcal{M}_{distinct}$; and,

(*iii*) Q can be computed by a coordination-free domain-guided transducer network iff $Q \in \mathcal{M}_{disjoint}$.

It is tricky to formally define coordination-freeness because ideally it should forbid communication for coordination purposes but it should allow communication to exchange data values, for instance, to compute joins. We employ the formalization of coordination-freeness as introduced in [13]. While we do not claim this notion of coordination-freeness to be the only possible one, the results in this paper imply that it is a sensible one. In particular, we show that coordination-free computations can not globally coordinate across *all* computing nodes. This is made precise by proving that every coordination-free transducer is equivalent to one that has no knowledge of all other nodes in the network. We refer to Section 4.1.5 and Section 4.3 for a more thorough discussion.

In its original formulation [23], the CALM-conjecture did not refer to the general class of monotone queries, but rather to the monotone queries definable in Datalog. Therefore, it is interesting to investigate subclasses of Datalog with negation that remain within $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$, respectively. As mentioned before, $\mathcal{M}_{distinct}$ corresponds to the well-known class of queries which are preserved under extensions, denoted by $\mathcal{E}$. Afrati et al. [6] obtained that semi-positive datalog, denoted SP-Datalog, is included in $\mathcal{E}$, while Cabibbo [18] showed that SP-Datalog extended with value invention captures $\mathcal{E}$ and therefore $\mathcal{M}_{distinct}$. We show that *semi-connected stratified Datalog*, denoted semicon-Datalog$^\neg$, a fragment of stratified Datalog where only 'connected' rules are allowed (except for the last stratum) and which contains SP-Datalog, is included in $\mathcal{M}_{disjoint}$ and that this fragment extended with value invention captures precisely $\mathcal{M}_{disjoint}$. Furthermore, all queries definable in SP-Datalog and semicon-Datalog$^\neg$ are coordination-free within the policy-aware and domain-guided transducer network model, respectively.

The results of this paper are summarized in Figure 2.

Outline. In Section 2, we introduce the necessary definitions. In Section 3, we investigate the classes $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$. In Section 4, we semantically characterize coordination-free transducer networks in the policy-aware and domain-guided models. In Section 5, we consider Datalog fragments for $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$. We discuss related work in Section 6 and conclude in Section 7.

2. DEFINITIONS

Queries and instances. We assume an infinite set **dom** of data values. A *database schema* σ is a collection of relation names R where every R has arity $ar(R)$. We call $R(\bar{d})$ a *fact* when R is a relation name and $\bar{d}$ is a tuple in **dom**. We say that a fact $R(d_1, \ldots, d_k)$ is *over* a database schema σ if $R \in \sigma$ and $ar(R) = k$. A *(database) instance* I over σ is simply a finite set of facts over σ. We denote by $adom(I)$ the set of all values that occur in facts of I. When $I = \{\mathbf{f}\}$, we simply write $adom(\mathbf{f})$ rather than $adom(\{\mathbf{f}\})$. By $|I|$ we denote the number of facts in I. A *query* over a schema σ to a schema σ' is a generic mapping Q from instances over σ to instances over σ'. Genericity means that for every permutation π of **dom** and every instance I, $Q(\pi(I)) = \pi(Q(I))$. For a set of facts I and a schema σ, we write $I|_\sigma$ to denote the maximal subset of I that is over σ.

For convenience, we restrict our attention to schemas for which all relations have arity greater than zero. In particular, this means that queries, and therefore also Datalog programs, can not define nullary relations. We address how the addition of nullary facts changes our results in Section 7.

Datalog with negation. Let **var** be the universe of variables, disjoint from **dom**. An *atom* is of the form $R(u_1, \ldots, u_k)$ where R is a relation name and each $u_i \in \mathbf{var}$. We call R the *predicate*. A *literal* is an atom or a negated atom and is called *positive* in the former case and *negative* in the latter case.

We recall Datalog with negation [4], abbreviated Datalog$^\neg$. Formally, a Datalog$^\neg$ *rule* φ is a quadruple $(head_\varphi, pos_\varphi, neg_\varphi, ineq_\varphi)$ where $head_\varphi$ is an atom; pos_φ and neg_φ are sets of atoms; $ineq_\varphi$ is a set of inequalities $(u \neq v)$ with $u, v \in \mathbf{var}$; and, the variables of φ all occur in pos_φ. The components $head_\varphi$, pos_φ and neg_φ are called respectively the *head*, the *positive body atoms* and the *negative body atoms*. We refer to $pos_\varphi \cup neg_\varphi$ as the *body atoms*. Note, neg_φ contains just atoms, not negative literals. Every Datalog$^\neg$ rule φ must have a head, pos_φ must be non-empty and neg_φ may be empty. If $neg_\varphi = \emptyset$ then φ is called *positive*. Of course, a rule φ may be written in the conventional syntax. For instance, if $head_\varphi = T(u, v)$, $pos_\varphi = \{R(u, v)\}$, $neg_\varphi = \{S(v)\}$, and $ineq_\varphi = \{u \neq v\}$, with $u, v \in \mathbf{var}$, then we can write φ as $T(u, v) \leftarrow R(u, v), \neg S(v), u \neq v$. The set of variables of φ is denoted $vars(\varphi)$. A rule φ is said to be *over schema* σ if for each atom $R(u_1, \ldots, u_k) \in$

$\{head_\varphi\} \cup pos_\varphi \cup neg_\varphi$, the arity of R in σ is k. A Datalog$^\neg$ *program* P *over* σ is a set of Datalog$^\neg$ rules over σ. We write $sch(P)$ to denote the (minimal) database schema that P is over. We define $idb(P) \subseteq sch(P)$ to be the database schema consisting of all relations in rule-heads of P. We abbreviate $edb(P) = sch(P) \setminus idb(P)$. As usual, the abbreviation "idb" stands for "intensional database schema" and "edb" stands for "extensional database schema" [4]. A *valuation for* a rule φ in P w.r.t. an instance I over $edb(P)$, is a total function $V : vars(\varphi) \to \mathbf{dom}$. The *application* of V to an atom $R(u_1, \ldots, u_k)$ of φ, denoted $V(R(u_1, \ldots, u_k))$, results in the fact $R(a_1, \ldots, a_k)$ where $a_i = V(u_i)$ for each $i \in \{1, \ldots, k\}$. This is naturally extended to a set of atoms, which results in a set of facts. The valuation V is said to be *satisfying for* φ *on* I if $V(pos_\varphi) \subseteq I$, $V(neg_\varphi) \cap I = \emptyset$, and $V(u) \neq V(v)$ for each $(u \neq v) \in ineq_\varphi$. If so, φ is said to *derive* the fact $V(head_\varphi)$.

Positive and semi-positive Datalog. A Datalog$^\neg$ program P is *positive* if all rules of P are positive. We say that P is *semi-positive* if for each rule $\varphi \in P$, the atoms of neg_φ are over $edb(P)$. We now give the semantics of a semi-positive Datalog$^\neg$ program P [4]. First, let T_P be the *immediate consequence operator* that maps each instance J over $sch(P)$ to the instance $J' = J \cup A$ where A is the set of facts derived by all possible satisfying valuations for the rules of P on J. Let I be an instance over $edb(P)$. Consider the infinite sequence I_0, I_1, I_2, etc, inductively defined as follows: $I_0 = I$ and $I_i = T_P(I_{i-1})$ for each $i \geq 1$. The *output of* P *on input* I, denoted $P(I)$, is defined as $\bigcup_j I_j$; this is the *minimal fixpoint* of the T_P operator.

We denote by Datalog, Datalog($\neq$), and SP-Datalog the the class of positive Datalog$^\neg$ programs without inequalities, the positive Datalog$^\neg$ programs, and the class of semi-positive Datalog$^\neg$ programs, respectively. Note that the last two classes may use inequalities.

Stratified semantics. We say that P is *syntactically stratifiable* if there is a function $\rho : sch(P) \to \{1, \ldots, |idb(P)|\}$ such that for each rule $\varphi \in P$, having some head predicate T, the following conditions are satisfied: (1) $\rho(R) \leq \rho(T)$ for each $R(\bar{u}) \in pos_\varphi \cap idb(P)$; and, (2) $\rho(R) < \rho(T)$ for each $R(\bar{u}) \in neg_\varphi \cap idb(P)$. For $R \in idb(P)$, we call $\rho(R)$ the *stratum number* of R. Intuitively, ρ partitions P into a sequence of semi-positive Datalog$^\neg$ programs P_1, ..., P_k with $k \leq |idb(P)|$ such that for each $i = 1, \ldots, k$, the program P_i contains the rules of P whose head predicate has stratum number i. This sequence is called a *syntactic stratification* of P. We can now apply the *stratified semantics* to P: for an input I over $sch(P)$, we first compute the fixpoint $P_1(I)$, then the fixpoint $P_2(P_1(I))$, etc. The *output of* P *on input* I, denoted $P(I)$, is defined as $P_k(P_{k-1}(\ldots P_1(I)\ldots))$. It is well known that the output of P does not depend on the chosen syntactic stratification (if more than one exists). Not all Datalog$^\neg$ programs are syntactically stratifiable. By *stratified Datalog* we refer to all Datalog$^\neg$ programs which are syntactically stratifiable.

Since we only consider syntactically stratifiable programs in this paper, we denote from now on the class of stratified Datalog$^\neg$ programs simply by Datalog$^\neg$.

Computing Queries. For a query Q with input schema σ and output schema σ', and a stratifiable Datalog$^\neg$ program P, we say that P *computes* Q if $Q(I) = P(I)|_{\sigma'}$ for all instances I over σ.

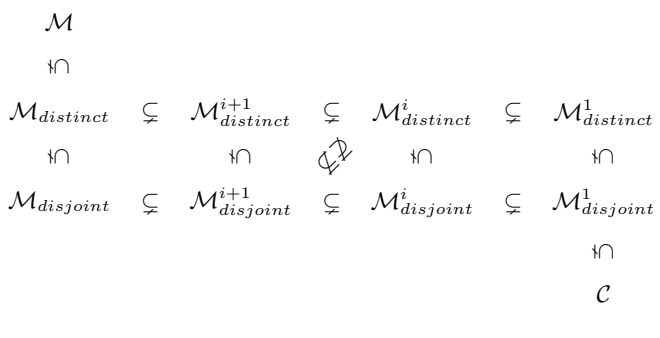

Figure 1: Monotonicity hierarchy

We assume that for each Datalog$^\neg$ program some *idb*-relations are marked as the intended output. In our example Datalog$^\neg$ programs, we use the convention that relation 'O' denotes that output. The input relations are recognizable as the *edb*-relations.

In the sequel, we overload notation and denote both the fragment of Datalog$^\neg$ programs as well as the queries expressed by programs in that class with the same notation. For instance, we use SP-Datalog to denote both the class of semi-positive Datalog$^\neg$ programs as well as the queries which are expressible by a semi-positive Datalog$^\neg$ program.

In examples, we use a unary *idb*-relation *Adom* that contains the active domain of the input. This predicate is computed as the union of the projections of all positions of all *edb*-relations. We omit the rules to compute *Adom*.

3. WEAKER FORMS OF MONOTONICITY

We introduce in Section 3.1 two weaker forms of monotonicity that will be used in Section 4 to characterize classes of coordination-free transducers. In Section 3.2, we relate these notions with the well-known classes of queries preserved under homomorphisms and extensions.

3.1 Domain distinct & disjoint monotonicity

We say that a fact $\mathbf{f}$ is *domain distinct* from instance I when $adom(\mathbf{f}) \setminus adom(I) \neq \emptyset$ (i.e., $\mathbf{f}$ should contain a new domain element not occurring in I); $\mathbf{f}$ is *domain disjoint* when $adom(\mathbf{f}) \cap adom(I) = \emptyset$. Furthermore, an instance J is domain distinct (resp., domain disjoint) from I, when every fact $\mathbf{f} \in J$ is domain distinct (resp., domain disjoint) from I.

We introduce two weaker forms of monotonicity by restricting the set of instances for which the monotonicity condition should hold:

Definition 1. Let Q be a query. Then,

- Q is *monotone* if $Q(I) \subseteq Q(I \cup J)$ for all database instances I and J;
- Q is *domain-distinct-monotone* if $Q(I) \subseteq Q(I \cup J)$ for all instances I and J for which J is domain distinct from I; and,
- Q is *domain-disjoint-monotone* if $Q(I) \subseteq Q(I \cup J)$ for all instances I and J for which J is domain disjoint from I.

We denote the class of monotone, domain-distinct-monotone, and domain-disjoint–monotone queries by $\mathcal{M}$, $\mathcal{M}_{distinct}$, and

$\mathcal{M}_{disjoint}$, respectively. Next, we restrict the monotonicity definitions to sets J of bounded size. More precisely, for $i \geq 1$, we say that Q is i-monotone, i-domain-distinct-monotone, and i-domain-disjoint-monotone when in the corresponding definition J is restricted to a size of at most i. We denote the respective classes by $\mathcal{M}^i$, $\mathcal{M}^i_{distinct}$, and $\mathcal{M}^i_{disjoint}$. By definition, $\mathcal{M} \subseteq \mathcal{M}_{distinct} \subseteq \mathcal{M}_{disjoint}$ and $\mathcal{M}^i \subseteq \mathcal{M}^i_{distinct} \subseteq \mathcal{M}^i_{disjoint}$.

The following theorem provides some insight in how the above classes are related. Separating examples can all be expressed in fragments of Datalog$^\neg$. A more visual representation is provided in Figure 1. We denote the class of all computable queries by $\mathcal{C}$.

THEOREM 3.1. *For every $i, j \geq 1$ with $i < j$,*

1. $\mathcal{M} \subsetneq \mathcal{M}_{distinct} \subsetneq \mathcal{M}_{disjoint} \subsetneq \mathcal{C}$;
2. $\mathcal{M} = \mathcal{M}^i$;
3. $\mathcal{M}_{distinct} \subsetneq \mathcal{M}^{i+1}_{distinct} \subsetneq \mathcal{M}^i_{distinct}$;
4. $\mathcal{M}_{disjoint} \subsetneq \mathcal{M}^{i+1}_{disjoint} \subsetneq \mathcal{M}^i_{disjoint}$;
5. $\mathcal{M}^i_{distinct} \subsetneq \mathcal{M}^i_{disjoint}$;
6. $\mathcal{M}^j_{disjoint} \not\subseteq \mathcal{M}^i_{distinct}$; *and,*
7. $\mathcal{M}^i_{distinct} \not\subseteq \mathcal{M}^j_{disjoint}$.

PROOF. We focus on the inequalities and sketch the separating examples. All queries are over directed graphs that are defined over the binary edge relation E.

For (1), first, $\mathcal{M} \subsetneq \mathcal{M}_{distinct}$ follows from SP-Datalog $\subseteq \mathcal{M}_{distinct}$. Second, we show $Q_{\overline{TC}} \in \mathcal{M}_{disjoint} \setminus \mathcal{M}_{distinct}$, where $Q_{\overline{TC}}$ is the query that computes the complement of the transitive closure of the edge relation. To see $Q_{\overline{TC}} \in \mathcal{M}_{disjoint}$, for all instances I with a path missing from vertex a to vertex b, i.e., $(a, b) \in Q_{\overline{TC}}$, the addition of domain-disjoint subgraphs will not create this path. To see $Q_{\overline{TC}} \notin \mathcal{M}_{distinct}$, the addition of domain-distinct subgraphs can create a path $E(a, c), E(c, b)$, where c is a new vertex. Third, for $\mathcal{M}_{disjoint} \subsetneq \mathcal{C}$, take the query that outputs all triangles on condition that no two disjoint triangles exist.

For (2), by definition, $\mathcal{M}^{i+1} \subseteq \mathcal{M}^i$. We show that $\mathcal{M}^i \subseteq \mathcal{M}^{i+1}$. Let $Q \in \mathcal{M}^i$ and let I and J be arbitrary instances such that $|J| \leq i + 1$. Pick an arbitrary fact $\mathbf{f} \in J$ and set $J' = J \setminus \{\mathbf{f}\}$. By assumption, $Q(I) \subseteq Q(I \cup J')$ and $Q(I \cup J') \subseteq Q((I \cup J') \cup \{\mathbf{f}\})$. Hence, $Q(I) \subseteq Q(I \cup J)$.

For (3), it suffices to show that $\mathcal{M}^i_{distinct} \setminus \mathcal{M}^{i+1}_{distinct} \neq \emptyset$. Ignoring the direction of edges, let Q^k_{clique} be the query that outputs the edge relation when no clique of k vertices exists and the empty relation otherwise. We show $Q^{i+2}_{\text{clique}} \in \mathcal{M}^i_{distinct} \setminus \mathcal{M}^{i+1}_{distinct}$. Let I be an input not containing $(i + 2)$-size cliques, upon which the output is thus nonempty. To expose the non-monotone behavior of Q^{i+2}_{clique}, we can try to extend any $(i + 1)$-size cliques in I to $(i + 2)$-size cliques by adding a domain-distinct instance J. For this to work, J needs to contain a star: one new value is the center and it points at old clique vertices of I, requiring $|J| \geq i + 1$. Such instances J are not considered in the definition of $\mathcal{M}^i_{distinct}$, but they are considered in the definition of $\mathcal{M}^{i+1}_{distinct}$.

For (4), again, we only show that $\mathcal{M}^i_{disjoint} \setminus \mathcal{M}^{i+1}_{disjoint} \neq \emptyset$. Let Q^k_{star} be the query that outputs the edge relation when there is no star with k spokes in the input and the empty relation otherwise. Clearly, $Q^{i+1}_{\text{star}} \notin \mathcal{M}^{i+1}_{disjoint}$ as $i+1$ domain-disjoint edges suffice to create an entirely new star with $i+1$ spokes. On the other hand, if there is not already a star with $i + 1$ spokes in the input, we can never create one by adding i domain-disjoint edges.

For (5), using similar arguments as for (3) above, we can see that $Q^{i+1}_{\text{clique}} \notin \mathcal{M}^i_{distinct}$ and $Q^{i+1}_{\text{clique}} \in \mathcal{M}^i_{disjoint}$.

For (6), we show $Q^{j+1}_{\text{star}} \in \mathcal{M}^j_{disjoint} \setminus \mathcal{M}^i_{distinct}$, where Q^{j+1}_{star} is the query that outputs the edge relation when there is no star with $j + 1$ spokes in the input and the empty relation otherwise. To see $Q^{j+1}_{\text{star}} \in \mathcal{M}^j_{disjoint}$, if we may only add j domain-disjoint edges, we can not extend a star with less than $j + 1$ spokes to a star with $j + 1$ spokes, and we can also not create a completely new star with $j+1$ spokes. To see $Q^{j+1}_{\text{star}} \notin \mathcal{M}^i_{distinct}$, when the input already contains a star with j spokes, we can increase the number of spokes to $j + 1$ by adding one additional edge containing the old central vertex and one new value.

Finally, for (7), we can only prove the stated result for a schema that grows with j. Therefore, let $R_1, \ldots, R_j$ be binary predicates and define $Q^j_{\text{duplicate}}$ as the query that gives as output the relation R_1 when the (global) intersection of all relations is empty, and the emptyset otherwise. We first argue $Q^j_{\text{duplicate}} \in \mathcal{M}^i_{distinct}$. Let I be an arbitrary instance where the intersection of all relations is empty. Instances J that are domain-distinct with respect to I can not replicate any existing tuples of I over all relations. Moreover, if $|J| \leq i$ with $i < j$ then J can not even replicate a completely new tuple over all relations. To see $Q^j_{\text{duplicate}} \notin \mathcal{M}^j_{disjoint}$, a domain-disjoint instance J with $|J| = j$ can replicate a new tuple over all relations. □

3.2 Correspondence with other classes

We relate the above classes to those defined in terms of preservation of properties. Let I and J be two instances over σ. A *homomorphism* from I to J is mapping h from $adom(I)$ to $adom(J)$ such that $R(\bar{d}) \in I$ implies $R(h(\bar{d})) \in J$ for every $R \in \sigma$ and $\bar{d} \in adom(I)^{ar(R)}$. We say that h is *injective* if $h(d) \neq h(d')$ whenever $d \neq d'$. An instance J is called an *induced subinstance* of I if $J = \{\mathbf{f} \in I \mid adom(\mathbf{f}) \subseteq adom(J)\}$.

Definition 2. Let Q be a query. Then,

- Q is *preserved under (injective) homomorphisms* if for all instances I and J, and for every (injective) homomorphism $h : adom(I) \to adom(J)$, $R(\bar{d}) \in Q(I)$ implies $R(h(\bar{d})) \in Q(J)$ for all facts $R(\bar{d})$.
- Q is *preserved under extensions* if for all instances I and J for which J is an induced subinstance of I, $R(\bar{d}) \in Q(J)$ implies $R(\bar{d}) \in Q(I)$ for all facts $R(\bar{d})$.

We denote by $\mathcal{H}$, $\mathcal{H}_{\text{inj}}$, and $\mathcal{E}$ the class of queries preserved under homomorphisms, injective homomorphisms, and extensions, respectively. Sometimes $\mathcal{H}$ is also referred to as the class of *strongly monotonic queries* (e.g., [6, 26]).

LEMMA 3.2. $\mathcal{H} \subsetneq \mathcal{H}_{inj} = \mathcal{M} \subsetneq \mathcal{E} = \mathcal{M}_{distinct}$.

PROOF. We only argue that $\mathcal{E} = \mathcal{M}_{distinct}$ as the rest is folklore (see, e.g., [6, 26, 29]). The equality follows immediately as J is an induced subinstance of I iff $I \setminus J$ is domain distinct from J. □

4. COORDINATION-FREENESS

A *relational transducer* is essentially a collection of queries that transforms a sequence of input facts to a sequence of output facts while maintaining a relational state [5, 20, 21]. In the distributed context, the functionality at each node of a network can be described via a relational transducer, giving rise to so-called relational transducer networks [13]. Subsequently, relational transducer networks have been extended in two ways to give nodes restricted access to *distribution policies* [32]. Such policies model deterministic input data distributions based on hashing. In this section, we review these two extensions, here referred to as *policy-aware* and *domain-guided* transducer networks, and characterize the corresponding classes of coordination-free queries. In particular, we show the following: the queries that are *coordination-free* in policy-aware transducer networks and domain-guided transducer networks correspond to the query classes $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$, respectively. It was shown in [13] that coordination-free queries in the original transducer networks correspond precisely to the set $\mathcal{M}$ of all monotone queries. The results in this section therefore provide a refinement of the CALM-conjecture in terms of weaker forms of monotonicity.

We formalize the transducer networks in Section 4.1 and then present the results in Section 4.2. We provide some additional discussion in Section 4.3.

4.1 Transducer models

We review the two extensions of Zinn et al. [32] to the original transducer network model [13]. Here, we refer to these extensions as *policy-aware* and *domain-guided* transducer networks, respectively.

4.1.1 Networks, data distribution, and policies

A *network* $\mathcal{N}$ is a nonempty finite set of values from **dom**, which we call *nodes*. Let σ be a database schema. A *distributed database instance* H over σ and $\mathcal{N}$ is a function that maps each $x \in \mathcal{N}$ to an instance over σ. A distributed database instance H over σ models a distribution with potential replication of data over the schema σ.

For a set X, let $\mathcal{P}^+(X) = \mathcal{P}(X) \setminus \{\emptyset\}$ denote the set of all non-empty subsets of X. We write $facts(\sigma)$ to denote the set of all possible facts over σ, i.e., using all possible values from **dom**. A *distribution policy* $\boldsymbol{P}$ *for* σ *and* $\mathcal{N}$ is a total function from $facts(\sigma)$ to $\mathcal{P}^+(\mathcal{N})$. Intuitively, $\boldsymbol{P}$ says how to distribute any instance I over σ to the nodes of $\mathcal{N}$, possibly with replication. Concretely, we define $dist_{\boldsymbol{P}}(I)$ to be the distributed database instance H over σ and $\mathcal{N}$ that satisfies $H(x) = \{\mathbf{f} \in I \mid x \in \boldsymbol{P}(\mathbf{f})\}$ for each $x \in \mathcal{N}$.

A *domain assignment* $\boldsymbol{\alpha}$ *for* $\mathcal{N}$ is a total function from **dom** to $\mathcal{P}^+(\mathcal{N})$. A distribution policy $\boldsymbol{P}$ for σ and $\mathcal{N}$ is called *domain-guided* if there exists a domain assignment $\boldsymbol{\alpha}$ for $\mathcal{N}$ such that for each $R(a_1, \ldots, a_k) \in facts(\sigma)$ we have $\boldsymbol{P}(R(a_1, \ldots, a_k)) = \bigcup_{i=1}^{k} \boldsymbol{\alpha}(a_i)$.[3] Intuitively, for each value $a \in$ **dom**, function $\boldsymbol{\alpha}$ says which nodes get input facts containing a.

EXAMPLE 4.1. *Suppose* **dom** *is the set* $\mathbb{N}$ *of natural numbers. Let* $\mathcal{N} = \{1, 2\}$ *be a two-node network and let the schema* σ *contain the single relation symbol* $\mathtt{E}$ *of arity* 2. *Consider the following distribution policy* $\boldsymbol{P}_1$ *for* σ *and* $\mathcal{N}$*:*

$$\boldsymbol{P}_1(\,\mathtt{E}(a,b)\,) = \begin{cases} \{1\} & \text{if } a \text{ is odd} \\ \{2\} & \text{otherwise} \end{cases}$$

for each $a, b \in \mathbb{N}$. *Note that* $\boldsymbol{P}_1$ *partitions any input* I *over* σ *based on its first attribute. If* $I = \{\mathtt{E}(1,3), \mathtt{E}(3,4), \mathtt{E}(4,6)\}$, *the distributed database instance* $dist_{\boldsymbol{P}_1}(I)$ *is*

$$\{1 \mapsto \{\mathtt{E}(1,3), \mathtt{E}(3,4)\}, 2 \mapsto \{\mathtt{E}(4,6)\}\}.$$

This input I *demonstrates that* $\boldsymbol{P}_1$ *is not a domain-guided policy: neither node is assigned all facts containing domain value* 4.

A domain-guided policy assigns domain values (rather than facts) to nodes in the network. Consider the domain assignment $\boldsymbol{\alpha}$ *for* $\mathcal{N}$ *that maps odd numbers to* $\{1\}$ *and even numbers to* $\{2\}$. *The corresponding domain-guided distribution policy* $\boldsymbol{P}_2$ *for* σ *and* $\mathcal{N}$ *allocates a fact* $\mathbf{f}$ *to node* 1 *if* any *of its two values is odd; to node* 2 *if* any *of its two values is even. For the specific input* I *from above, the instance* $dist_{\boldsymbol{P}_2}(I)$ *is*

$$\{1 \mapsto \{\mathtt{E}(1,3), \mathtt{E}(3,4)\}, 2 \mapsto \{\mathtt{E}(3,4), \mathtt{E}(4,6)\}\}.$$

Note that node identifiers can occur as data in relations. □

4.1.2 Policy-aware relational transducers

In the following, we write $R^{(k)}$ to denote a relation symbol R of arity k. A *(policy-aware) transducer schema* Υ is a tuple $(\Upsilon_{\text{in}}, \Upsilon_{\text{out}}, \Upsilon_{\text{msg}}, \Upsilon_{\text{mem}}, \Upsilon_{\text{sys}})$ of database schemas with disjoint relation names, with the additional restriction that

$$\Upsilon_{\text{sys}} = \{\mathtt{Id}^{(1)}, \mathtt{All}^{(1)}, \mathtt{MyAdom}^{(1)}\} \cup \{\mathtt{policy}_R^{(k)} \mid R^{(k)} \in \Upsilon_{\text{in}}\}.$$

These schemas are called respectively "input", "output", "message", "memory", and "system".

A *(policy-aware relational) transducer* Π over Υ is a quadruple $(Q_{\text{out}}, Q_{\text{ins}}, Q_{\text{del}}, Q_{\text{snd}})$ of queries having the input schema $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{msg}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$ and such that

- query Q_{out} has target schema Υ_{out};
- queries Q_{ins} and Q_{del} both have target schema Υ_{mem};
- query Q_{snd} has target schema Υ_{msg}.

These queries form the mechanism by which a node on a network produces output, updates its memory (through insertions and deletions), and sends messages.

We note that the transducers in [13] do not have the relations $\mathtt{MyAdom}$ and $\mathtt{policy}_R$ (with R in Υ_{in}).

4.1.3 Policy-aware transducer networks

A *(policy-aware) transducer network* $\boldsymbol{\Pi}$ is a quadruple $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ where $\mathcal{N}$ is a network, Υ is a transducer schema, Π is a transducer over Υ, and $\boldsymbol{P}$ is a distribution policy for Υ_{in} and $\mathcal{N}$. We say that $\boldsymbol{\Pi}$ is *domain-guided* if the distribution policy $\boldsymbol{P}$ is domain-guided. Note that domain-guided transducer networks are a special kind of policy-aware transducer networks.

Intuition. We will next give the semantics of $\boldsymbol{\Pi}$ on an input I over Υ_{in}. We start with the underlying intuition. Intuitively, we put a copy of transducer Π on each node of $\mathcal{N}$, and the policy $\boldsymbol{P}$ tells us how to initialize each node with a fragment of I. Then we choose an arbitrary node $x \in \mathcal{N}$ and make it "active": we execute the queries of Π on the

[3] Recall that we have assumed $k \geq 1$ throughout the paper.

union of the local input facts at x, the output and memory facts stored at x, any message facts (over Υ_{msg}) received by x, and also some facts over Υ_{sys}. The facts over Υ_{sys} consist of the following: relation `Id` provides the identifier of x (i.e., just 'x'); relation `All` provides the identifiers of all nodes in the network; relation `MyAdom` provides for convenience the active domain that x knows about (either from its local facts or from received messages); and, the relations $\texttt{policy}_R$ provide the set of facts over this active domain that x is responsible for (i.e., that are assigned to x) in policy $\boldsymbol{P}$.[4] Intuitively, by considering only $\texttt{policy}_R$-facts over the active domain at x, we provide "safe" access to the distribution policy, i.e., we prevent x from using values outside $adom(I) \cup \mathcal{N}$.[5] Next, the queries of Π update the output and memory at x, and generate new messages that are sent to the other nodes. Such an active moment of a node is called a *transition* of $\boldsymbol{\Pi}$. The semantics of $\boldsymbol{\Pi}$ is described by so-called *runs* which are infinite sequences of transitions.

EXAMPLE 4.2. *Recall* **dom**, $\mathcal{N}$, σ, *and* $\boldsymbol{P}_1$ *from Example 4.1. Consider a policy-aware transducer network* $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P}_1)$ *with* $\Upsilon_{\text{in}} = \sigma$. *We leave* Υ_{out}, Υ_{msg}, *and* Υ_{mem} *unspecified. Let us focus on the node* 1. *On input* $I = \{\texttt{E}(1,3), \texttt{E}(3,4), \texttt{E}(4,6)\}$, *at least the following facts will be exposed to node* 1 *during each transition: the local input facts* $\texttt{E}(1,3)$ *and* $\texttt{E}(3,4)$; *the system facts* $\texttt{Id}(1)$, $\texttt{All}(1)$, $\texttt{All}(2)$, $\texttt{MyAdom}(a)$ *for each* $a \in \{1,2,3,4\}$, *and* $\texttt{policy}_{\texttt{E}}(a,b)$ *with* $a \in \{1,3\}$ *and* $b \in \{1,2,3,4\}$. *If node* 1 *would later receive (and store) the value* 6, *then also* $\texttt{MyAdom}(6)$ *will be exposed, and the* $\texttt{policy}_{\texttt{E}}(a,b)$*-facts with* $a \in \{1,3\}$ *and* $b = 6$. *Also, note that node* 1 *can in principle deduce that* $\texttt{E}(3,2)$ *is not part of* I *since* $\texttt{policy}_{\texttt{E}}(3,2)$ *is present at node* 1 *but not* $\texttt{E}(3,2)$. □

Formal semantics. Let $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ be a policy-aware transducer network. A *configuration* of $\boldsymbol{\Pi}$ is a pair $\rho = (s, b)$ of functions s and b such that:

- s maps each $x \in \mathcal{N}$ to a set of facts over $\Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}$;
- b maps each $x \in \mathcal{N}$ to a multiset of facts over Υ_{msg}.

We call s and b respectively the *state* and *(message) buffer*. Intuitively, s specifies for each node what output and memory facts it has locally available, and b specifies for each node what messages have been sent to it and that are not yet delivered. The reason for having multisets for the buffers, is that the same message can be sent multiple times to the same recipient and thus multiple copies can be floating around in the network simultaneously. The *start configuration* of $\boldsymbol{\Pi}$ is the unique configuration $\rho = (s, b)$ that satisfies $s(x) = \emptyset$ and $b(x) = \emptyset$ for each $x \in \mathcal{N}$.

A *transition of* $\boldsymbol{\Pi}$ *on an input* I *over* Υ_{in} is a quadruple (ρ_1, x, m, ρ_2) where $\rho_1 = (s_1, b_1)$ and $\rho_2 = (s_2, b_2)$ are configurations of $\boldsymbol{\Pi}$, $x \in \mathcal{N}$, m is a submultiset of $b_1(x)$, and, letting

$$
\begin{aligned}
H &= dist_{\boldsymbol{P}}(I),\\
M &= m \text{ collapsed to a set},\\
J &= H(x) \cup s_1(x) \cup M,\\
A &= \mathcal{N} \cup \bigcup_{\mathbf{f} \in J} adom(\mathbf{f}),\\
S &= \{\texttt{Id}(x)\} \cup \{\texttt{All}(y) \mid y \in \mathcal{N}\} \cup \{\texttt{MyAdom}(a) \mid a \in A\} \cup\\
&\qquad \{\texttt{policy}_R(a_1, \ldots, a_k) \mid R^{(k)} \in \Upsilon_{\text{in}},\ \{a_1, \ldots, a_k\} \subseteq A,\\
&\qquad x \in \boldsymbol{P}(R(a_1, \ldots, a_k))\},\\
D &= J \cup S,
\end{aligned}
$$

for the state s_2 we have

$$
\begin{aligned}
s_2(x)|_{\Upsilon_{\text{out}}} &= s_1(x)|_{\Upsilon_{\text{out}}} \cup Q_{\text{out}}(D),\\
s_2(x)|_{\Upsilon_{\text{mem}}} &= \big[s_1(x)|_{\Upsilon_{\text{mem}}} \cup (Q_{\text{ins}}(D) \setminus Q_{\text{del}}(D))\big]\\
&\qquad \setminus (Q_{\text{del}}(D) \setminus Q_{\text{ins}}(D)),\\
s_2(y) &= s_1(y) \text{ for each } y \in \mathcal{N} \setminus \{x\},
\end{aligned}
$$

and for the buffer b_2 we have (using multiset difference and union)

$$
\begin{aligned}
b_2(x) &= b_1(x) \setminus m,\\
b_2(y) &= b_1(y) \cup Q_{\text{snd}}(D) \text{ for each } y \in \mathcal{N} \setminus \{x\}.
\end{aligned}
$$

We call ρ_1 and ρ_2 respectively the *source* and *target* configuration of the transition and we refer to x as the active node. If $m = \emptyset$, then we call the transition a *heartbeat*.

For an input I over Υ_{in}, a *run* $\mathcal{R}$ *of* $\boldsymbol{\Pi}$ *on* I is an infinite sequence of transitions of $\boldsymbol{\Pi}$ on I, such that the start configuration of $\boldsymbol{\Pi}$ is used as the source configuration of the first transition, and the target configuration of each transition is the source configuration of the next transition. Note that runs represent nondeterminism: each transition can choose what node becomes active and what messages to deliver from the buffer of that node. We consider only *fair* runs. These are the runs that satisfy the following additional conditions: *(i)* each node is the active node in an infinite number of transitions; and, *(ii)* if a fact occurs infinitely often in the message buffer of a node then this fact is infinitely often delivered to that node. Intuitively, the last condition demands that no sent messages are infinitely delayed.

4.1.4 Computing queries

We are interested in transducers that produce the same facts over Υ_{out} regardless of the network, the distribution policy, and the order of transitions. These transducers are said to compute a query.

Formally, let Q be a query with input schema σ_1 and output schema σ_2. Further, let $\boldsymbol{\Pi} = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ be a policy-aware transducer network. We define the output of a run $\mathcal{R}$ of $\boldsymbol{\Pi}$, denoted $out(\mathcal{R})$, to be the union of all output facts jointly produced by the nodes of $\mathcal{N}$ during $\mathcal{R}$, i.e., all facts over Υ_{out}. Note that once a fact is added to Υ_{out}, it can never be retracted. We say that $\boldsymbol{\Pi}$ *computes* Q if *(i)* $\Upsilon_{\text{in}} = \sigma_1$ and $\Upsilon_{\text{out}} = \sigma_2$; and, *(ii)* for each input I over σ_1, every (fair) run $\mathcal{R}$ of $\boldsymbol{\Pi}$ on I satisfies $out(\mathcal{R}) = Q(I)$.

Now, we say that a policy-aware transducer Π over transducer schema Υ *(distributedly) computes* Q *(for all policies)* if for all networks $\mathcal{N}$ and all distribution policies $\boldsymbol{P}$ for Υ_{in} and $\mathcal{N}$, the policy-aware transducer network $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ computes Q. We say that Π *(distributedly) computes* Q *un-*

[4] The relations $\texttt{policy}_R$ were previously called '$\texttt{local}_R$' [32]. Here, we have chosen a new predicate name to avoid confusion with local input facts at a node.

[5] This safety restriction also makes our model more realistic: in general, a node still needs to communicate with other nodes before it can draw global conclusions about the input or network.

der domain-guidance if for all networks $\mathcal{N}$ and all domain-guided policies $\boldsymbol{P}$ for Υ_{in} and $\mathcal{N}$, the transducer network $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ computes Q.

4.1.5 Coordination-freeness

We define coordination-freeness for policy-aware transducers similarly as for the original transducer model [13].

Definition 3. Let Π be a policy-aware transducer over a schema Υ. We say that Π is *coordination-free* if (1) Π distributedly computes a query Q, and (2) for all networks $\mathcal{N}$, for all inputs I for Q, there is a distribution policy $\boldsymbol{P}$ for Υ_{in} and $\mathcal{N}$ such that the policy-aware transducer network $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ has a run on input I in which $Q(I)$ is already computed in a prefix consisting of only heartbeat transitions.[6]

Similarly, we say that Π is *coordination-free under domain-guidance* if (1) Π distributedly computes a query Q under domain-guidance, and (2) if for all networks $\mathcal{N}$, for all inputs I for Q, there is a domain-guided policy $\boldsymbol{P}$ for $\mathcal{N}$, such that the transducer network $(\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P})$ has a run on input I in which $Q(I)$ is already computed in a prefix consisting of only heartbeat transitions.

We write $\mathcal{F}_1$ to denote the set of queries distributedly computed by coordination-free policy-aware transducers. We write $\mathcal{F}_2$ to denote the set of queries distributedly computed by policy-aware transducers that are coordination-free under domain-guidance.

It is useful to reflect on what it means to be coordination-free. Of course, one could prohibit any form of communication but that would be too drastic and unworkable when data is distributed and communication is already needed for the simple purpose of exchanging data (for instance, to compute joins). Therefore, the present formalization tries to separate the 'data'-communication (that can never be eliminated in a distributed setting) from the 'coordination'-communication by requiring that there is some 'ideal' distribution on which the query can be computed without any communication.[7] The intuition is as follows: because on the ideal distribution there is no coordination (as communication is prohibited), and the transducer network has to correctly compute the query on *all* distributions, communication is only used to transfer data on non-ideal distributions and is not used to coordinate. While we do not claim our notion of coordination-freeness to be the only possible one, the results in Section 4.3 confirm that the just described intuition is not too far off. Indeed, it follows that coordination-freeness corresponds precisely to those computations that do not require the knowledge about *all* other nodes in the network, and hence, can not globally coordinate. Specifically, we show that when a node has no complete overview of *all* the nodes in the network, which can be achieved by removing the relation `All`, then every transducer is coordination-free. More importantly, the converse holds true as well. That is, we show that every coordination-free transducer is equivalent to one that does not use the relation `All`.

[6]Technically, heartbeat transitions allow to send messages but not to read them. So, 'only heartbeat' transitions effectively means 'no communication'.

[7]We remark that in this ideal distribution it is not always sufficient to give the full input to all nodes (see, e.g., [13]). Furthermore, the network is not necessarily aware that the data is ideally distributed as it can not communicate.

4.2 Characterization

We characterize the classes $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$ by coordination-free transducers.

THEOREM 4.3. $\mathcal{F}_1 = \mathcal{M}_{distinct}$.

PROOF. We sketch the proof.

$\boxed{\mathcal{F}_1 \subseteq \mathcal{M}_{distinct}}$ Let Q be a query from σ to σ' distributedly computed by a coordination-free policy-aware transducer Π. Let I and J be two input instances for Q, such that J is domain-distinct from I. Let $\mathbf{f} \in Q(I)$. We show $\mathbf{f} \in Q(I \cup J)$. Because Π distributedly computes Q, transducer Π also computes Q on a network $\mathcal{N}$ with at least two nodes. By coordination-freeness, there is a distribution policy $\boldsymbol{P}_1$ for σ and $\mathcal{N}$ such that the transducer network $\boldsymbol{\Pi}_1 = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P}_1)$ when given input I, has a run $\mathcal{R}_1$ in which $Q(I)$ is already computed in a prefix consisting of only heartbeat transitions. Let $x \in \mathcal{N}$ be a node that outputs $\mathbf{f}$ in this prefix. We can make x output $\mathbf{f}$ on input $I \cup J$, so that $\mathbf{f} \in Q(I \cup J)$.

To do this, fix an arbitrary node $y \in \mathcal{N} \setminus \{x\}$. Consider the following distribution policy $\boldsymbol{P}_2$ for σ and $\mathcal{N}$: $\boldsymbol{P}_2(\mathbf{g}) = \{y\}$ for all $\mathbf{g} \in J$, and $\boldsymbol{P}_2(\mathbf{g}) = \boldsymbol{P}_1(\mathbf{g})$ for all $\mathbf{g} \in \mathit{facts}(\sigma) \setminus J$. Denote $\boldsymbol{\Pi}_2 = (\mathcal{N}, \Upsilon, \Pi, \boldsymbol{P}_2)$. It can be verified that $\boldsymbol{\Pi}_2$ on input $I \cup J$ gives to x the same local input facts as $\boldsymbol{\Pi}_1$ on input I. So, when running $\boldsymbol{\Pi}_2$ on input $I \cup J$, if we initially do only heartbeats with active node x, the node x goes through the same state changes as in the heartbeat-prefix of $\mathcal{R}_1$. So after a while, say after k heartbeats, node x outputs $\mathbf{f}$. This finite prefix can be extended to a full fair run $\mathcal{R}_2$ of $\boldsymbol{\Pi}_2$ on input $I \cup J$, for which $\mathit{out}(\mathcal{R}_2) = Q(I \cup J)$ holds by assumption on Π. Hence, $\mathbf{f} \in Q(I \cup J)$.

$\boxed{\mathcal{M}_{distinct} \subseteq \mathcal{F}_1}$ Let $Q \in \mathcal{M}_{distinct}$. Let I be an input for Q that is distributed over a network $\mathcal{N}$. We say that a subset $C \subseteq \mathit{adom}(I) \cup \mathcal{N}$ is *complete* at a node $x \in \mathcal{N}$, when x knows for every fact $\mathbf{f} \in \mathit{facts}(\sigma)$ with $\mathit{adom}(\mathbf{f}) \subseteq C$ whether $\mathbf{f} \in I$ or $\mathbf{f} \notin I$. If C is indeed complete at x, node x will output $Q(I')$ where $I' = \{\mathbf{f} \in I \mid \mathit{adom}(\mathbf{f}) \subseteq C\}$. Note that $Q(I') \subseteq Q(I)$ by domain-distinct-monotonicity of Q. We will construct a policy-aware transducer that postpones executing Q at a node x until the system relation `MyAdom` is complete at x.

To implement this idea, the nodes broadcast their locally given input facts. Each node x stores every received input fact. This way, the system relation `MyAdom` grows at x. During each transition, x checks for each input relation $R^{(k)}$ and each k-tuple $(a_1, \ldots, a_k)$ over $\texttt{MyAdom}^k$ whether the fact $\texttt{policy}_R(a_1, \ldots, a_k)$ is shown to x. If so, then x is reponsible for the fact $R(a_1, \ldots, a_k)$ under the distribution policy. In that case, if $R(a_1, \ldots, a_k)$ is absent in the local input at x, node x can conclude that $R(a_1, \ldots, a_k)$ is actually globally absent from the entire input. Then x broadcasts the absence of this fact. These absences are also accumulated at all nodes. Now, consider a transition of node x, and let $I' \subseteq I$ denote the set of collected input facts at x so far. For each potential input fact $\mathbf{f}$ over `MyAdom`, node x checks whether $\mathbf{f} \in I'$ or that node x knows $\mathbf{f} \notin I$ (by means of the explicit absences). If this is so then `MyAdom` is complete at x, and x subsequently computes Q on I'.

We have already ensured that no wrong outputs are produced. To show that at least $Q(I)$ is produced, we note that at some point, each node x has received all available input

facts and all absences of facts over $adom(I) \cup \mathcal{N}$. At that moment, x computes Q on I, causing at least $Q(I)$ to be output in each run.

Transducer Π is indeed coordination-free according to the formal definition: for all networks $\mathcal{N}$, and for all inputs I, the full output will be computed at some node x with only heartbeats when x is made responsible (under the distribution policy) for all facts over Υ_{in} made with the values $adom(I) \cup \mathcal{N}$; then x will immediately detect that relation `MyAdom` is complete. □

THEOREM 4.4. $\mathcal{F}_2 = \mathcal{M}_{disjoint}$.

PROOF. We sketch the proof. The proof of $\mathcal{F}_2 \subseteq \mathcal{M}_{disjoint}$ is similar to the proof of $\mathcal{F}_1 \subseteq \mathcal{M}_{distinct}$.

$\boxed{\mathcal{M}_{disjoint} \subseteq \mathcal{F}_2}$ Let $Q \in \mathcal{M}_{disjoint}$. Let I be an input for Q that is distributed over a network $\mathcal{N}$, by means of a domain-guided distribution policy. We call a subset $C \subseteq adom(I) \cup \mathcal{N}$ *complete* at a node x when x knows it has collected all facts $\mathbf{f} \in I$ for which $adom(\mathbf{f}) \cap C \neq \emptyset$. If C is indeed complete at x, node x will compute $Q(I')$ where $I' = \{\mathbf{f} \in I \mid adom(\mathbf{f}) \cap C \neq \emptyset\}$. Note that $Q(I') \subseteq Q(I)$ by domain-disjoint-monotonicity of Q. We will construct a policy-aware transducer that postpones executing Q at a node x until the system relation `MyAdom` is complete at x.

To implement this idea, the nodes broadcast the active domain of their local input fragment. These values are accumulated at each node. Note that when a node x has some value a in relation `MyAdom`, node x is responsible for a under the domain assignment if and only if $\texttt{policy}_R(a, \ldots, a)$ is shown to x for at least one input relation R. If x is indeed responsible for a then x is already locally given all facts of I containing a (because the distribution policy is domain-guided). Now, when x is not responsible for a, node x will send out the request (x, a). Any node y responsible for a under the domain assignment will then send all input facts containing a to x. When x has acknowledged all these facts to y, node y will send "OK(x, a)". Now, consider a transition of x, and let $I' \subseteq I$ denote the set of collected input facts at x so far. Node x checks for each value a in `MyAdom` whether x is responsible for a or that x has "OK" for a. If this is so, the protocol above implies that x has obtained all input facts containing values from `MyAdom`, i.e., that `MyAdom` is complete at x. Subsequently, x computes Q on I'.

We have already argued that no wrong outputs are produced. To see that at least $Q(I)$ is computed, we note that each node x eventually knows of the entire active domain (because the broadcasted domain will eventually arrive), and will thus at some point compute Q on the entire set of collected input facts.

The transducer is coordination-free according to the formal definition: for all networks $\mathcal{N}$, and all inputs I, the output will be computed with only heartbeats at some node x when x is made responsible (under the domain assignment) for all values $adom(I) \cup \mathcal{N}$; then x will immediately detect that relation `MyAdom` is complete. □

4.3 Discussion

We contrast the evaluation algorithms in the above proofs for $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$ with that for $\mathcal{M}$. It is important to realize that the formulated algorithms are naive in the sense that the whole database is sent to all nodes and every node computes the result of the query. It is the type of monotonicity that determines when a node can start producing output:

- $\mathcal{M}$: every node broadcasts all local input facts; output is generated for every newly received fact;
- $\mathcal{M}_{distinct}$: every node broadcasts all local input facts as well as non-facts (missing input facts that the node is responsible for); output is generated for every complete (cf. proof of Theorem 4.3) subset of facts; and,
- $\mathcal{M}_{disjoint}$: every node broadcasts the active domain of the local input facts; whenever a new domain element is received that the node is not responsible for, a coordination protocol is initiated with nodes responsible for this value; output is generated for every complete (cf. proof of Theorem 4.4) subset of facts.

While it might seem contradictory that the coordination-free evaluation of queries in $\mathcal{M}_{disjoint}$ requires the use of a coordination protocol, it is important to realize that this coordination is only determined by the way data is distributed and does not require *global* coordination between *all* nodes. Indeed, we show below that transducers that do not access the system relation `All`, containing the names of all the nodes in the network, are automatically coordination-free. Moreover, the classes of queries that are distributedly computed by policy-aware transducers without relation `All` still capture $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$. That is, in absence of relation `All`, we do not need the notion of coordination-freeness to characterize $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$.

Formally, to prevent usage of relation `All`, at a node x we modify the semantics of transitions from Section 4.1.3 as follows: we define the set A now as $A = \{x\} \cup \bigcup_{\mathbf{f} \in J} adom(\mathbf{f})$; and, the set S is defined as before, but now without the `All`-facts. For this resulting model, let $\mathcal{A}_1$ denote the set of queries distributedly computed by policy-aware transducers, and let $\mathcal{A}_2$ denote the set of queries distributedly computed by policy-aware transducers under domain-guidance.

THEOREM 4.5. $\mathcal{A}_1 = \mathcal{M}_{distinct}$ *and* $\mathcal{A}_2 = \mathcal{M}_{disjoint}$

PROOF. First, the transducers constructed for the proofs of directions $\mathcal{M}_{distinct} \subseteq \mathcal{F}_1$ and $\mathcal{M}_{disjoint} \subseteq \mathcal{F}_2$ can be used unmodified to respectively show that $\mathcal{M}_{distinct} \subseteq \mathcal{A}_1$ and $\mathcal{M}_{disjoint} \subseteq \mathcal{A}_2$, because they do not use `All`.

We sketch the proof of $\mathcal{A}_1 \subseteq \mathcal{M}_{distinct}$. Let Q be a query that is distributedly computed by a policy-aware transducer Π without `All`. By assumption, transducer Π also computes Q on a single-node network $\{x\}$. So, on input I, the single-node transducer network will produce $Q(I)$ with just heartbeats (no messages can be sent). Let J be an input instance for Q that is domain-distinct from I. To show $Q(I) \subseteq Q(I \cup J)$, we consider a two-node network $\{x, y\}$ on which we run Π, and consider the distribution policy that assigns J to just y and the other facts to x. Then x is still given precisely I when the two-node transducer network is given $I \cup J$ as input. Now, if we do only heartbeats at x, then x will behave the same as on the single-node network because it can not detect the difference with the two-node network in absence of relation `All`. So, x will produce $Q(I)$ after a finite number of heartbeats. This prefix can be extended to a full fair run of the two-node transducer network on input $I \cup J$.

The inclusion $\mathcal{A}_2 \subseteq \mathcal{M}_{disjoint}$ can be shown similarly, except that now we assign all values of $adom(J)$ to y and all other domain values to x. □

It is interesting to note what happens when transducers are not aware of the distribution policies, i.e., when we do not provide the $\texttt{policy}_R$-relations. In the resulting model, the set $\mathcal{F}_0$ of queries distributedly computed by coordination-free transducers is precisely the set $\mathcal{M}$ of monotone queries; and, relating to the above, the set $\mathcal{A}_0$ of queries distributedly computed by transducers without relation `All` is also $\mathcal{M}$ [13].

For completeness, we also mention the existence of so-called *oblivious* transducers in the original transducer model of Ameloot et al. [13]: these transducers may use neither relation `Id` nor relation `All`. The set of queries distributedly computed by oblivious transducers is again the set $\mathcal{M}$.

COROLLARY 4.6. $\mathcal{F}_0 = \mathcal{A}_0 = \mathcal{M}$.

5. DATALOG FRAGMENTS

5.1 Semi-connected Datalog$^\neg$

As mentioned in the introduction, the original formulation of the CALM-conjecture links coordination-free computation to Datalog. It is therefore interesting to investigate subclasses of Datalog with negation that remain within $\mathcal{M}_{distinct}$ and $\mathcal{M}_{disjoint}$. While SP-Datalog $\subseteq \mathcal{M}_{distinct} (= \mathcal{E})$ [6], it is not known whether SP-Datalog can be further extended while remaining within $\mathcal{E}$.

We next identify a fragment of Datalog$^\neg$ which is domain-disjoint-monotone. Let φ be a Datalog$^\neg$ rule. We define $graph^+(\varphi)$ as the graph where nodes are the variables in positive body atoms of φ, and there is an edge between two variables if they occur together in a positive body atom of φ. We say φ is *connected* if $graph^+(\varphi)$ is connected. We say that an SP-Datalog program is *connected* when all rules are connected.

Definition 4. Let P be a program in Datalog$^\neg$. Then P is a *connected stratified datalog program*, when there is a stratification for P such that all strata are connected SP-Datalog programs. We say that P is *semi-connected* when there is a stratification such that all strata except possibly the last one are connected SP-Datalog programs.

In the following, we denote the class of connected and semi-connected stratified Datalog$^\neg$ programs by con-Datalog$^\neg$ and semicon-Datalog$^\neg$, respectively. Then *(i)* SP-Datalog $\subsetneq$ semicon-Datalog$^\neg$, *(ii)* SP-Datalog $\not\subseteq$ con-Datalog$^\neg$, and *(iii)* con-Datalog$^\neg \subsetneq$ semicon-Datalog$^\neg$.

EXAMPLE 5.1. *Consider the following Datalog$^\neg$ program* P_1

$$\begin{array}{rcl} T(x) & \leftarrow & E(x,y), E(y,z), E(z,x), y \neq x, y \neq z, x \neq z \\ O(x) & \leftarrow & \neg T(x), Adom(x) \end{array}$$

Then, P_1 is in con-Datalog$^\neg$, but $P_1 \notin \mathcal{M}_{distinct}$. Indeed, $P_1(\{E(a,b)\}) \neq \emptyset$, while $P_1(\{E(a,b)\} \cup \{E(b,c), E(c,a)\}) = \emptyset$. Therefore, $P_1 \notin$ SP-Datalog.

Consider the program P_2 which is not a semicon-Datalog$^\neg$ program:

$$\begin{array}{rcl} T(x,y,z) & \leftarrow & E(x,y), E(y,z), E(z,x), y \neq x, y \neq z, x \neq z \\ D(x_1) & \leftarrow & T(x_1,x_2,x_3), T(y_1,y_2,y_3), \bigwedge_{1 \leq i,j \leq 3} x_i \neq y_j \\ O(x) & \leftarrow & \neg D(x), Adom(x) \end{array}$$

Note that the expressed query is not in $\mathcal{M}_{disjoint}$. As we will see shortly, this implies that the query itself can not be defined by any semicon-Datalog$^\neg$ program.

□

We next relate connected programs to distributed evaluation over components. An instance J is a *component* of an instance I when $J \subseteq I$, $J \neq \emptyset$, $adom(J) \cap adom(I \setminus J) = \emptyset$ and J is minimal with this property. That is, there is no strict nonempty subset J' of J for which $adom(J') \cap adom(I \setminus J') = \emptyset$. Intuitively, a component is complete w.r.t. its active domain as it either contains all the facts in I in which a certain domain element occurs, or it contains none of these facts. Denote by $co(I)$ the components of I.

Definition 5. A query Q *distributes over components* if for all instances I: (1) $Q(I) = \bigcup_{C \in co(I)} Q(C)$; and, (2) $adom(Q(C)) \cap adom(Q(C')) = \emptyset$ for all $C, C \in co(I)$ with $C \neq C'$.

The lemma below highlights an important property of con-Datalog$^\neg$ and forms a crucial part of the proof of Theorem 5.3. We omit the proof because of space limitations.

LEMMA 5.2. *Every query in con-Datalog$^\neg$ distributes over components.*

We are now armed to prove the following theorem:

THEOREM 5.3. *semicon-Datalog$^\neg \subseteq \mathcal{M}_{disjoint}$*

PROOF. Let P be a semicon-Datalog$^\neg$ program with stratification $P_1, \ldots, P_s$ where P_i is a connected SP-Datalog program for $i < s$. Define $P_{\leq i}$ as the composition $P_i \circ \cdots \circ P_1$ of the first i strata. Clearly, $P = P_{\leq s} = P_s \circ P_{\leq s-1}$. Let I and J be two domain disjoint instances. Then by Lemma 5.2, $P_{\leq s-1}(I)$ is domain disjoint from $P_{\leq s-1}(J)$. By domain-disjoint-monotonicity of SP-Datalog programs, it follows that

$$P_s(P_{\leq s-1}(I)) \subseteq P_s(P_{\leq s-1}(I) \cup P_{\leq s-1}(J)). \qquad (\dagger)$$

We next show that

$$P_{\leq s-1}(I) \cup P_{\leq s-1}(J) = P_{\leq s-1}(I \cup J), \qquad (\ddagger)$$

which together with (†) implies that $P(I) \subseteq P(I \cup J)$ and therefore $P \in \mathcal{M}_{disjoint}$.

It remains to prove (‡). Lemma 5.2 implies that $P_{\leq s-1}(I) = \bigcup_{C \in co(I)} P_{\leq s-1}(C)$, $P_{\leq s-1}(J) = \bigcup_{D \in co(J)} P_{\leq s-1}(D)$ and $P_{\leq s-1}(I \cup J) = \bigcup_{K \in co(I \cup J)} P_{\leq s-1}(K)$. Since I and J are domain disjoint, $co(I) \cup co(J) = co(I \cup J)$. Therefore,

$$\begin{array}{rcl} & & P_{\leq s-1}(I) \cup P_{\leq s-1}(J) \\ & = & \bigcup_{C \in co(I)} P_{\leq s-1}(C) \cup \bigcup_{D \in co(J)} P_{\leq s-1}(D) \\ & = & \bigcup_{K \in co(I \cup J)} P_{\leq s-1}(K) \\ & = & P_{\leq s-1}(I \cup J). \end{array}$$

This completes the proof. □

5.2 Datalog$^\neg$ with value invention

The classes $\mathcal{M}$ and $\mathcal{E}$ (and therefore $\mathcal{M}_{distinct}$) are captured by fragments of ILOG$^\neg$, a declarative formalism in the style of stratified Datalog$^\neg$ originally introduced in the context of object databases by Hull and Yoshikawa [24].

We follow Cabibbo [18] for the definition of ILOG$^\neg$ and assume the existence of *invention relations*. These are relations with a distinguished first position, called the *invention position* or the *invention attribute*. An *invention atom* is an atom of the form $R(*, u_1, \ldots, u_k)$, where R is an invention relation and each $u_i \in \mathbf{var}$. The symbol $*$ is called the *invention symbol*. An ILOG$^\neg$ program P over σ is a Datalog program with negation over σ where we allow head atoms to be either (ordinary) relation atoms or invention atoms.

Before defining the semantics of ILOG$^\neg$, we associate to each invention relation R a distinct *Skolem functor* f_R of arity $ar(R)-1$. The *Skolemization* of P, denoted by $Skol(P)$, is the set of rules in P where every occurrence of the invention symbol is replaced by appropriate Skolem functor terms. For example, the Skolemization of the rule

$$R(*, x_1, x_2) \leftarrow E(x_1, x_2)$$

is

$$R(f_R(x_1, x_2), x_1, x_2) \leftarrow E(x_1, x_2).$$

The semantics of ILOG$^\neg$ is defined similar to the semantics of Datalog with negation, but valuations are applied on the Skolemized rules instead of the rules itself and are w.r.t. the Herbrand universe $\mathcal{H}_P$ for P which is the set of all ground terms built using elements from **dom** and Skolem functors for invention relations in P. When the repeated application of the immediate consequence operator eventually gives rise to relations of infinite size, the output of the program is undefined, otherwise the output consists of the facts in the output relations.

An ILOG$^\neg$ program P is *safe* when the output contains no invented values. Although safety is an undecidable property, there are syntactical restrictions that ensure safe programs. One of these is called weakly safe which we define next. We consider the set of *unsafe positions* in atoms of P. This is the smallest set S containing pairs of the form (R, i), where R is a relation name and $i \leq ar(R)$, such that

- $(R, 1) \in S$ for every invention relation R; and,
- if $(R, i) \in S$ and there is a rule φ in P such that $R(x_1, \ldots, x_k) \in pos_\varphi$, $T(y_1, \ldots, y_\ell) = head_\varphi$, and x_i and y_j refer to the same variable, then $(T, j) \in S$.

An ILOG$^\neg$ program P is *weakly safe* when the output relations of P do not contain unsafe positions. Note that a weakly safe program is always safe. We denote weakly safe ILOG$^\neg$ by wILOG$^\neg$. Furthermore, wILOG$^\neg$ where negation is restricted to predicates in $edb(P)$ is denoted by SP-wILOG for semi-positive wILOG$^\neg$. Stratification can be defined for wILOG$^\neg$ in the same way as for Datalog$^\neg$. As before, we only consider stratified negation in this paper.

Cabibbo investigated the expressiveness of ILOG$^\neg$ over relational databases [18] and obtained the following results: stratified wILOG$^\neg$ programs with two strata capture the class of all computable queries; wILOG$^\neg$ programs where negation is restricted to inequalities and extensional database predicates captures $\mathcal{M}$ and $\mathcal{E}$, respectively.

We introduce the class of *semi-connected wILOG$^\neg$* programs and show that it captures precisely the domain-disjoint-monotone queries. Analogous to the definitions for Datalog$^\neg$, we say that a SP-wILOG program is *connected* when all rules are connected. A wILOG$^\neg$ program P is *semi-connected* when there is a stratification for P such that all strata, except possibly the last one, are connected SP-wILOG programs. Due to space restrictions, the proof of the next result is omitted.

THEOREM 5.4. *Semi-connected wILOG$^\neg$ computes precisely all queries in $\mathcal{M}_{disjoint}$.*

6. RELATED WORK

Declarative networking. Hellerstein [23] argues that the theory of declarative database query languages can provide a foundation for the next generation of parallel and distributed programming languages and formulates a number of related conjectures for the PODS community. Datalog has previously been considered for distributed systems, see e.g. [25, 1, 28].

Ameloot et al. [13] have introduced the framework of relational transducer networks to formalize and prove the CALM-conjecture. The formalism was then parameterized by Zinn et al. [32] to allow for specific data distribution strategies. These authors showed that the classes of coordination-free queries in the original transducer network model ($\mathcal{F}_0$), in the policy-aware transducer network model ($\mathcal{F}_1$), and in the domain-guided transducer network model ($\mathcal{F}_2$) form a strict hierarchy. In particular, they showed that the non-monotone win-move query is in $\mathcal{F}_2$. Some further work on relational transducer networks was done by Ameloot and Van den Bussche who studied decidability of confluence [14] and consistency [12]. The CRON-conjecture, which relates the causal delivery of messages to the nature of the computations that those messages participate in (like monotone versus non-monotone) is treated by Ameloot and Van den Bussche [15].

Zinn introduced in [31] the idea of domain-distinct- and domain-disjoint-monotone queries (albeit under a different name). He defined these as $\mathcal{M}^1_{distinct}$ and $\mathcal{M}^1_{disjoint}$, respectively, and used them to show the corresponding version of Theorem 4.3 and Theorem 4.4. Although the proof and statement of the results are incorrect, the proposed approach did already contain the ideas on which the proofs presented in this paper are based. In fact, it was the enthusiasm of the first three authors over the approach in [31] that in a collaborative effort with the fourth author led to the present paper. In this way, Section 4.2 of the present paper can be seen as an extension and correction of [31]. The results in Section 3, Section 4.3, and Section 5 are not discussed in [31].

The networked relational transducer model is just one paradigm for studying distributed query evaluation. Another notable model (or language) is WebdamLog [3], which is a Datalog-variant for declarative networking. This language supports *delegation*, where a node can send rules to another node instead of just facts; transmitted rules can then be locally evaluated at the addressee. In general, the distributed computations expressed by WebdamLog do not assume a fixed set of nodes, but they allow previously unseen nodes to participate.

Another model, called the massively parallel (MP) model, is introduced by Koutris and Suciu [27]. Here, computation proceeds in a sequence of parallel steps, each followed by global synchronization of all servers. In this model, evaluation of conjunctive queries [27, 17] as well as skyline queries [8] have been considered.

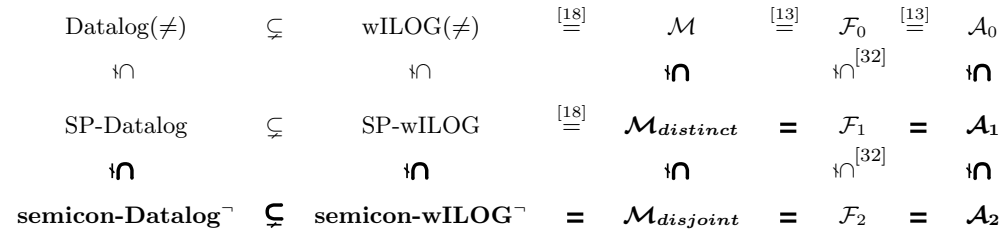

Figure 2: Main results of this work: query classes introduced and relationships obtained in this paper are shown in bold-face. Non-bold-face relationships without annotation are part of database folklore.

Related to the MP model, is the MapReduce framework where the evaluation of queries has been considered by a number of researchers. Afrati and Ullman [9] study the evaluation of join queries and take the amount of communication, calculated as the sum of the sizes of the input to reducers, as a complexity measure. Evaluation of transitive closure and datalog queries in MapReduce has been investigated in [7, 10].

Lastly, in Active XML [2], a distributed system is represented as a collection of XML documents in which some vertices contain calls to remote webservices. Any data returned by the calls is incorporated into the calling document.

Finite model theory. The expressiveness of (extensions of) Datalog and its relation to monotonicity have been previously investigated. We discuss some of the results relevant to the present paper. It is known that Datalog and Datalog($\neq$) are strictly included in $\mathcal{H}$ and $\mathcal{M}$, respectively (c.f., e.g., [6, 26]). As mentioned before, Afrati et al. [6] obtained that SP-Datalog $\subseteq \mathcal{E}$. Feder and Vardi [22] showed that all queries in SP-Datalog that are preserved under homomorphisms can already be expressed in Datalog. That is, SP-Datalog$\cap\mathcal{H}$ = Datalog. Dawar and Kreutzer [19] showed that the latter result can not be extended to least fixed-point logic (LFP): LFP $\cap\, \mathcal{H} \not\subseteq$ Datalog. The status of SP-Datalog w.r.t. $\mathcal{E}$ is less clear. It is, for instance, not known whether SP-Datalog $= \mathcal{E} \cap$ Datalog$^\neg$. A related result here is the one by Rosen [29], who showed that FO$[\exists^*\forall\exists]\cap\mathcal{E} \subseteq$ SP-Datalog, where FO$[\exists^*\forall\exists]$ denotes first-order logic formulas of the form $\exists\bar{x}\forall y\exists z\,\psi$ where ψ is quantifier-free. We note that Tait's counterexample [30] separating $FO[\exists]$ from $\mathcal{E}\cap FO$ is definable in SP-Datalog [29]. Atserias et al. [16] study $\mathcal{E}$ in relation to FO over restricted classes of structures.

7. CONCLUSION

Figure 2 summarizes the main findings of this paper. At the same time the figure formulates a more fine-grained answer to the CALM-conjecture which stipulates that a program has a coordination-free execution strategy if and only if the program is monotone. In particular, our results equate increasingly larger classes of coordination-free computations with increasingly weaker forms of monotonicity and make Datalog variants explicit for each of these classes. Furthermore, the last two columns, as already explained in Section 4.1.5 and Section 4.3, confirm that the notion of coordination-freeness as proposed in [13] is a sensible one. Indeed, the notion corresponds to the intended semantics in that coordination-freeness avoids global synchronization barriers through the absence of knowledge about *all* the nodes in the network. That said, we do not claim that our notion is the only possible one. Indeed, one could argue that, especially within $\mathcal{F}_1$ and $\mathcal{F}_2$, even though there is no global synchronization barrier, computing nodes are still prone to wait until complete subsets of the input data have been accumulated (cf. Section 4.2). Of course, the semantical characterizations in terms of weaker forms of monotonicity make precise that this waiting is determined by the way data is distributed. The query evaluation algorithms in the proofs in Section 4.2 are inefficient in that they require all data to be sent to all nodes, it remains to investigate how the insights obtained in this paper can lead to more practical algorithms.

Another contribution of this paper is the identification of (semi-)connected Datalog variants which to the best of our knowledge have not been considered before. We can prove that the connected variant of Datalog under the well-founded semantics, making use of the well-known "doubled program" approach, remains within $\mathcal{M}_{disjoint}$ which implies a simpler proof of the fact that win-move is in $\mathcal{M}_{disjoint}$ (one of the main results in [32]). It is open whether semi-connected Datalog under the well-founded semantics is in $\mathcal{M}_{disjoint}$. A direction for future work is to study the properties of connected Datalog.

As is to be expected, deciding whether a query class belongs to one of the monotonicity classes quickly turns undecidable. Still, it would be interesting to find decidable subclasses or identify sufficient conditions as this would provide insight on the way queries can be distributedly computed. In Section 3, we introduced the bounded classes $\mathcal{M}^i_{distinct}$ and $\mathcal{M}^i_{disjoint}$ mainly because of the mismatch with [31] as explained in Section 6. It remains to investigate their relationship with distributed evaluation of queries.

We avoided the use of nullary relations. This is not a fundamental restriction but a practical one. With general policies, all results in this paper carry over to programs with schemas that can derive nullary relations unaltered. For domain-guided policies, definitions need to be slightly adapted: a nullary fact is never domain disjoint from any relation. Furthermore, in a domain-guided distribution policy, all nullary facts always have to be assigned to all computing nodes, and the definition of a component has to be extended to include all nullary facts. We will provide all details in the journal version of this paper.

Acknowledgments.

We thank Georg Gottlob, Thomas Eiter, and Phokion Kolaitis for answering our questions on Datalog. We also thank Phokion Kolaitis for bringing [29] to our attention.

8. REFERENCES

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

[2] S. Abiteboul, O. Benjelloun, and T. Milo. The active xml project: An overview. *The VLDB Journal*, 17(5):1019–1040, 2008.

[3] S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for Web data management. In *PODS*, pages 293–304. ACM Press, 2011.

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

[5] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *J. Comput. Syst. Sci.*, 61(2):236–269, 2000.

[6] F. Afrati, S. S. Cosmadakis, and M. Yannakakis. On Datalog vs polynomial time. *Journal of computer and system sciences*, 51:177–196, 1995.

[7] F. N. Afrati, V. R. Borkar, M. J. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *ICDE*, pages 1–8, 2011.

[8] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012.

[9] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[10] F. N. Afrati and J. D. Ullman. Transitive closure and recursive Datalog implemented on clusters. In *EDBT*, pages 132–143, 2012.

[11] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[12] T. J. Ameloot. Deciding correctness with fairness for simple transducer networks. In *ICDT*, 2014.

[13] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *J. ACM*, 60(2):15, 2013.

[14] T. J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. In *ICDT*, pages 86–98, 2012.

[15] T. J. Ameloot and J. Van den Bussche. On the CRON conjecture. In *Datalog*, pages 44–55, 2012.

[16] A. Atserias, A. Dawar, and M. Grohe. Preservation under extensions on well-behaved finite structures. *SIAM J. Comput.*, 38(4):1364–1381, 2008.

[17] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In *PODS*, pages 273–284, 2013.

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

[19] A. Dawar and S. Kreutzer. On Datalog vs. LFP. In *ICALP*, pages 160–171, 2008.

[20] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven Web applications. *J. Comput. Syst. Sci.*, 73(3):442–474, 2007.

[21] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. In *PODS*, pages 90–99. ACM Press, 2006.

[22] T. Feder and M. Y. Vardi. Homomorphism closed vs. existential positive. In *LICS*, pages 311–320, 2003.

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

[24] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *VLDB*, pages 455–468, 1990.

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

[26] P. G. Kolaitis and M. Y. Vardi. On the expressive power of Datalog: Tools and a case study. In *PODS*, pages 61–71, 1990.

[27] P. Koutris and D. Suciu. Parallel evaluation of conjunctive queries. In *PODS*, pages 223–234, 2011.

[28] B. Loo, T. Condie, et al. Declarative networking: Language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 97–108. ACM, 2006.

[29] E. Rosen. *Finite Model Theory and finite Variable Logics*. PhD thesis, University of Pennsylvania, 1995.

[30] W. W. Tait. A counterexample to a conjecture of Scott and Suppes. *The journal of Symbolic Logic*, 24(1):15–16, 1959.

[31] D. Zinn. Weak forms of monotonicity and coordination-freeness. *CoRR*, abs/1202.0242, 2012.

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