Relational transducers for declarative networking

Relational Transducers for Declarative Networking

Tom J. Ameloot, Frank Neven, Jan Van den Bussche

Abstract

Motivated by a recent conjecture concerning the expressiveness of declarative networking, we propose a formal computation model for "eventually consistent" distributed querying, based on relational transducers. A tight link has been conjectured between coordination-freeness of computations, and monotonicity of the queries expressed by such computations. Indeed, we propose a formal definition of coordination-freeness and confirm that the class of monotone queries is captured by coordination-free transducer networks. Coordination-freeness is a semantic property, but the syntactic class of "oblivious" transducers we define also captures the same class of monotone queries. Transducer networks that are not coordination-free are much more powerful.

1 Introduction

Declarative networking [28] is a recent approach by which distributed computations and networking protocols are modeled and programmed using formalisms based on Datalog. In his keynote speech at PODS 2010 [24, 25], Hellerstein made a number of intriguing conjectures concerning the expressiveness of declarative networking. In the present paper, we are focusing on the CALM conjecture (Consistency And Logical Monotonicity). This conjecture suggests a strong link between, on the one hand, "eventually consistent" and "coordination-free" distributed computations, and on the other hand, expressibility in monotonic Datalog (without negation or aggregate functions). The conjecture was not fully formalized, however; indeed, as Hellerstein notes himself, a proper treatment of this conjecture requires crisp definitions of eventual consistency and coordination, which have been lacking so far. Moreover, it also requires a formal model of distributed computation.

In the present paper, we investigate the CALM conjecture in the context of a model for distributed database querying. In the model we propose, the computation is performed on a network of relational transducers. The relational transducer model, introduced by Abiteboul and Vianu, is well established in database theory research as a model for data-centric agents reacting to inputs [7, 17, 18, 19, 30]. Relational transducers are firmly grounded in the theory of

database queries [5, 6] and also have close connections with Abstract State Machines [16]. It thus seems natural to consider networks of relational transducers, as we will do here. We give a formal operational semantics for such networks, formally define "eventual consistency", and formally define what it means for a network to compute a conventional database query, in order to address the expressiveness issues raised by Hellerstein.

It is less clear, however, how to formalize the intuitive notion of "coordination". We do not claim to resolve this issue definitively, but we propose a new, non-obvious definition that appears workable. Distributed algorithms requiring coordination are viewed as less efficient than coordination-free algorithms. Hellerstein has identified *monotonicity* as a fundamental property connected with coordination-freeness. Indeed, monotonicity enables "embarrassing parallelism" [25]: agents working on parts of the data in parallel can produce parts of the output independently, without the need for coordination.

One side of the CALM conjecture now states that any database query expressible in monotonic Datalog can be computed in a distributed setting in an eventually consistent, coordination-free manner. This is the easy side of the conjecture, and indeed we formally confirm it in the following broader sense: any monotone query $\mathcal{Q}$ can be computed by a network of "oblivious" transducers. Oblivious transducers are unaware of the network extent (in a sense that we will make precise), and every oblivious transducer network is coordination-free. Here, we should note that the transducer model is parameterized by the query language $\mathcal{L}$ that the transducer can use to update its local state. Formally, the monotone query $\mathcal{Q}$ to be computed must be expressible in the while-closure of $\mathcal{L}$ for the above confirmation to hold. If $\mathcal{Q}$ is defined in Datalog, for example, then $\mathcal{L}$ can just be unions of conjunctive queries.

The other side of the CALM conjecture, that the query computed by an eventually consistent, coordination-free distributed program is always expressible in Datalog, is false when taken literally, as we will point out. Nevertheless, we do give an extended version of the conjecture that holds. More importantly, we confirm the conjecture in the following more general form: coordination-free networks of transducers can compute only monotone queries. Note that here we are using our newly proposed formal definition of coordination-free.

This paper is organized as follows. Preliminaries are in Section 2. Section 3 introduces networks of transducers. Section 4 investigates the use of transducer networks for expressing conventional database queries in a distributed fashion. Section 5 discusses the issue of coordination, and looks into the CALM conjecture and related results. Section 6 contains results about the expressiveness of transducer networks. Section 7 shortly looks into a variation of the transducer model, and Section 8 is the conclusion.

This paper is the extended version of our conference paper [11].

1.1 Related Work

The desire to better understand coordination in the field of declarative networking is evidenced by the steadily growing literature on this subject. First,

Alvaro et al. [9] look at coordination as a quantitative property that can be minimized. They describe a program analysis technique to spot syntactical locations in the code where coordination is not needed. The goal then, is to help the programmer iteratively reduce the number of locations where coordination is used.

The conference paper of this work has also inspired follow-up work by others. In particular, Zinn et al. [32] have generalized our results to also include a "partitioning policy", which is a strategy to initialize every node of a network with input data before the computation starts. The basic idea is that each node is given local relations that provide information about how data is distributed, and in particular what data each node can autonomously reason about, i.e., without coordination with other nodes. This allows a node to sometimes perform nonmonotone operations without the need for communication. It even turns out that in some variations of the model considered, all database queries are "coordination-free". It can be expected however, that such a partitioning policy is quite expensive in terms of how much additional data each node should have.

One of our results is that a monotone query can in principle be computed without coordination, but it remains open in what exact way the best performance can be achieved in a practical scenario. The work of Loo et al. [27] and Nigam et al. [29], however, provides concrete algorithms for the case of distributed Datalog programs. They want to efficiently update the state (i.e., the derivations) on nodes of the network whenever some input facts change. It would be too costly to completely recompute the state of every node when an update happens. Instead, they propose a technique that propagates only the incremental changes that have to be distributedly applied. This allows for sending less messages around the network, and does not require needless recomputations of data. Their algorithms require no coordination, can handle recursive Datalog rules, and can tolerate messages that are delivered out of order by the network.

2 Preliminaries

2.1 Basic Notions

We first recall some basic notions from database theory [2]. A *database schema* is a finite set $\mathcal{D}$ of pairs (R, k) where R is a *relation name* and $k \in \mathbb{N}$ is the associated *arity* of R. A relation name is allowed to occur only once in a database schema. We often write a pair $(R, k) \in \mathcal{D}$ as $R^{(k)}$. We assume some infinite universe **dom** of atomic data values. An arity of zero is also called *nullary*. We write ( ) to denote the nullary tuple.

A fact $\boldsymbol{f}$ is a pair $(R, \bar{a})$, often denoted as $R(\bar{a})$, where R is a relation name and $\bar{a}$ is a tuple of values over **dom**. A *database instance* I over a database schema $\mathcal{D}$ is a finite set of facts such that for each $R(a_1, \ldots, a_k) \in I$ we have $R^{(k)} \in \mathcal{D}$. Let Z be a subset of relation names in $\mathcal{D}$. We write $I|_Z$ to denote the restriction of I to the facts whose predicate is a relation name in Z. For a func-

tion $h : \mathbf{dom} \to \mathbf{dom}$ we define $h(I) = \{R(h(a_1), \ldots, h(a_k)) \mid R(a_1, \ldots, a_k) \in I\}$. The *active domain* of I, denoted $adom(I) \subseteq \mathbf{dom}$, is the set of atomic data values that occur in I.

A *query $\mathcal{Q}$ over input database schema $\mathcal{D}$ and output database schema $\mathcal{D}'$* is a partial function mapping database instances of $\mathcal{D}$ to database instances of $\mathcal{D}'$. A special but common kind of query are those where the output database schema contains just one relation. A query $\mathcal{Q}$ is called *generic* if for all input instances I and all permutations h of **dom** the query $\mathcal{Q}$ is also defined on the isomorphic instance $h(I)$ and $\mathcal{Q}(h(I)) = h(\mathcal{Q}(I))$. We recall that a generic query $\mathcal{Q}$ is *domain-preserving*, in the sense that $adom(\mathcal{Q}(I)) \subseteq adom(I)$ for all input instances I. We use the word "query" in this text to mean generic query, unless explicitly specified otherwise.

We recall the following query languages [2]:

- UCQ: unions of conjunctive queries,
- UCQ$^\neg$: UCQ with negation in the body,
- FO: first order logic (relational calculus),
- While: FO with iteration,
- Datalog,
- NrDatalog: nonrecursive Datalog,
- NrDatalog$^\neg$: NrDatalog with negation on body atoms.

The weakest query language among these is UCQ.

2.2 Transducers

The computation on a single node of a network is formalized by means of relational transducers [7, 11, 17, 18, 19, 30]. A *transducer schema* Υ is a tuple $\langle \Upsilon_{\text{in}}, \Upsilon_{\text{out}}, \Upsilon_{\text{msg}}, \Upsilon_{\text{mem}}, \Upsilon_{\text{sys}} \rangle$ of database schemas, called respectively "input", "output", "message", "memory" and "system". A relation name can occur in at most one database schema of Υ. We fix Υ_{sys} to always contain two unary relations `Id` and `All`. A *transducer state* for Υ is a database instance over $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$. For a transducer state I we use notations of the form $I|_{(\text{in,sys})}$ to denote the facts in I whose predicates are in Υ_{in} or Υ_{sys}. This notation is extended to other identifiers among in, out, msg, mem, and sys.

An *(epidemic) relational transducer* Π over Υ is a collection of queries:

- for each $R^{(k)} \in \Upsilon_{\text{out}}$ there is a query $\mathcal{Q}^R_{\text{out}}$ having output schema $\{R^{(k)}\}$;
- for each $R^{(k)} \in \Upsilon_{\text{mem}}$ there are queries $\mathcal{Q}^R_{\text{ins}}$ and $\mathcal{Q}^R_{\text{del}}$ both having output schema $\{R^{(k)}\}$;
- for each $R^{(k)} \in \Upsilon_{\text{msg}}$ there is a query $\mathcal{Q}^R_{\text{snd}}$ having output schema $\{R^{(k)}\}$;

where each of these queries has the input schema $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{msg}} \cup \Upsilon_{\text{mem}} \cup \Upsilon_{\text{sys}}$. These queries will form the internal mechanism that a node uses to update its local storage and to send messages. The transducer model is parameterized by a generic query language $\mathcal{L}$: this language is used to concretely specify the above queries, in which case we call Π an $\mathcal{L}$-transducer. We will often abbreviate "epidemic relational transducer" simply as "transducer". The term "epidemic" will become clear in Section 3.1, where the transducer model is used on a network.

Let Π be a transducer over schema Υ. A *message instance* for Υ is a database instance over Υ_{msg}. A *local transition* of Π is a 4-tuple $(I, I_{\text{rcv}}, J, J_{\text{snd}})$, also denoted as $I, I_{\text{rcv}} \to J, J_{\text{snd}}$, where I and J are transducer states for Υ, I_{rcv} and J_{snd} are message instances for Υ such that (denoting $I' = I \cup I_{\text{rcv}}$):

$$\begin{aligned}
J|_{(\text{in,sys})} &= I|_{(\text{in,sys})}; \\
J|_{(\text{out})} &= I|_{(\text{out})} \cup \bigcup_{R^{(k)} \in \Upsilon_{\text{out}}} \mathcal{Q}^R_{\text{out}}(I'); \\
J|_{(\text{mem})} &= \bigcup_{R^{(k)} \in \Upsilon_{\text{mem}}} (I|_R \cup R^+) \setminus R^- \\
J_{\text{snd}} &= \bigcup_{R^{(k)} \in \Upsilon_{\text{msg}}} \mathcal{Q}^R_{\text{snd}}(I'),
\end{aligned}$$

where, following the presentation in [32],

$$\begin{aligned}
R^+ &= \mathcal{Q}^R_{\text{ins}}(I') \setminus \mathcal{Q}^R_{\text{del}}(I'); \text{ and,} \\
R^- &= \mathcal{Q}^R_{\text{del}}(I') \setminus \mathcal{Q}^R_{\text{ins}}(I').
\end{aligned}$$

Intuitively, on the receipt of message facts I_{rcv}, a local transition updates the old transducer state I to new transducer state J and sends the facts in J_{snd}. When compared to I, in J potentially more output facts are produced; and the update semantics for each memory relation R adds the facts produced by *insertion* query $\mathcal{Q}^R_{\text{ins}}$, removes the facts produced by *deletion* query $\mathcal{Q}^R_{\text{del}}$, and there is no-op semantics in case a fact is both added and removed at the same time [30]. Output facts can not be removed. Note that local transitions are deterministic in the following sense: if $I, I_{\text{rcv}} \to J, J_{\text{snd}}$ and $I, I_{\text{rcv}} \to J', J'_{\text{snd}}$ then $J = J'$ and $J_{\text{snd}} = J'_{\text{snd}}$.

3 Transducer Networks

We now formalize a network of computing nodes. In Section 4 we give example programs.

A *network* $\mathcal{N}$ is a finite, connected, and undirected graph whose nodes are all in **dom**. We write *nodes*($\mathcal{N}$) and *edges*($\mathcal{N}$) to denote the nodes and edges of $\mathcal{N}$ respectively. For $x \in$ *nodes*($\mathcal{N}$), we write *neighbor*$(x, \mathcal{N})$ to denote the set $\{y \mid (x, y) \in \textit{edges}(\mathcal{N})\}$.

A *(homogeneous) transducer network* is a triple $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ where $\mathcal{N}$ is a network, Υ is a transducer schema, and Π is a transducer over Υ whose copies will be running at each node of the network. One could also consider non-homogeneous transducer networks, where each node contains a different transducer, possibly over a different schema, but these types of networks are not considered in the present paper. We discuss our design choices in Section 3.4.

For a query language $\mathcal{L}$, we say that a transducer network is an $\mathcal{L}$-transducer network if all its transducers are $\mathcal{L}$-transducers.

We now formalize how data is distributed across a network. A *distributed database instance over a network $\mathcal{N}$ and a database schema $\mathcal{D}$* is a total function that assigns to each node of $\mathcal{N}$ an ordinary database instance over $\mathcal{D}$.

3.1 Operational Semantics

Let $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ be a transducer network. Any distributed database instance over $\mathcal{N}$ and Υ_{in} can be given as input to $\mathcal{T}$. Let H be such an instance. A *configuration of $\mathcal{T}$ on H* is a pair $\rho = (s, b)$ of functions where

- s maps each $x \in \mathit{nodes}(\mathcal{N})$ to a transducer state $J = s(x)$ such that $J|_{(\text{in})} = H(x)$ and $J|_{(\text{sys})} = \{\texttt{Id}(x)\} \cup \{\texttt{All}(y) \mid y \in \mathit{nodes}(\mathcal{N})\}$; and,
- b maps each $x \in \mathit{nodes}(\mathcal{N})$ to a finite multiset of facts over Υ_{msg}.

We call s the *state function* and b the *buffer function*. Intuitively, the instance H is used to initialize each node, and for each $x \in \mathit{nodes}(\mathcal{N})$ the system relations `Id` and `All` in Υ_{sys} provide the local transducer at x the identity of the node x it is running on and the identities of the other nodes. Next, the buffer function maps each $x \in \mathit{nodes}(\mathcal{N})$ to the multiset of messages that have been sent to x but that have not yet been delivered to x. A multiset allows us to represent duplicates of the same message (sent at different times).

The *start configuration of $\mathcal{T}$ on H*, denoted $\mathit{start}(\mathcal{T}, H)$, is the configuration $\rho = (s, b)$ of $\mathcal{T}$ on H that for each $x \in \mathit{nodes}(\mathcal{N})$ defines $s(x)|_{(\text{out,mem})} = \emptyset$ and $b(x) = \emptyset$.

We now describe the actual computation of the transducer network. A *global transition* of $\mathcal{T}$ on input H is a 4-tuple (ρ_1, x, m, ρ_2), also denoted as $\rho_1 \xrightarrow{x,m} \rho_2$, where $x \in \mathit{nodes}(\mathcal{N})$, and $\rho_1 = (s_1, b_1)$ and $\rho_2 = (s_2, b_2)$ are configurations of $\mathcal{T}$ on H such that

- m is a submultiset of $b_1(x)$ and there exists a message instance J_{snd} such that
$$s_1(x),\ \mathit{set}(m) \to s_2(x),\ J_{\text{snd}}$$
is a local transition of transducer Π;
- for each $y \in \mathit{nodes}(\mathcal{N}) \setminus \{x\}$ we have $s_1(y) = s_2(y)$;
- $b_2(x) = b_1(x) \backslash m$; for each $y \in \mathit{neighbor}(x, \mathcal{N})$ we have $b_2(y) = b_1(y) \cup J_{\text{snd}}$; and for all other nodes y we have $b_2(y) = b_1(y)$.

We call x the *active node* (or *recipient*) and $set(m)$ the *delivered messages*. Intuitively, in a global transition we select an arbitrary node x and allow it to receive some arbitrary submultiset m from its message buffer. The messages in m are then delivered at node x (as a set, i.e., without duplicates) and x performs a local transition, in which it updates its memory and output relations, and possibly sends some new messages to all its neighbors.[1] The node does not send messages to itself. If $m = \emptyset$, we call this global transition a *heartbeat* transition and otherwise we call it a *delivery* transition. A heartbeat transition corresponds to the real life situation in which a node does a computation step when a local timer goes off and no messages have been received from the network.

A *run* $\mathcal{R}$ of a transducer network $\mathcal{T}$ on a distributed input database instance H is an *infinite* sequence of global transitions $\rho_i \xrightarrow{x_i, m_i} \rho_{i+1}$ for $i = 1, 2, 3, \ldots$, where $\rho_1 = start(\mathcal{T}, H)$, and the i^{th} transition with $i \geq 2$ operates on the resulting configuration of the previous transition. It follows from the semantics of local transitions that when a node during one global transition changes its output or memory, then these changes are visible to that node only starting from the next global transition in which that node is active. Note also that several facts can be delivered together during a transition, regardless of whether they were sent during different earlier transitions or during the same earlier transition.

We have not defined global transitions that are concurrent, i.e., global transitions in which multiple nodes receive a message multiset and do local transitions at the same time. The reason for not including this kind of global transition is that it can be simulated by multiple sequential global transitions: this is done by letting the previously concurrent nodes become active in some arbitrary order, and in each of those single-node transitions, the active node just reads its own message buffer like in the concurrent transition. Because local transitions are deterministic, the nodes will update their state and send out messages in the same way as during the concurrent transition.

3.2 Fairness

In the literature on process models it is customary to require certain "fairness" conditions on runs [21, 14, 26]. Let $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ be a transducer network. In this paper, a run of $\mathcal{T}$ on some input distributed database instance is called *fair* if *(i)* every node of $\mathcal{N}$ is active in an infinite number of transitions and *(ii)* if for some node a fact occurs in its message buffer in an infinite number of configurations, then this fact is delivered to that node during an infinite number of transitions. Intuitively, the last condition demands that no sent messages are infinitely delayed. We consider only fair runs.

Note that every transducer network has a fair run for every input because heartbeat transitions are still possible even when the message buffers have become empty.

[1] Hence the name "epidemic" [20].

3.3 Message Delivery Constraints

We may want to impose a size-constraint on the delivered message multisets. Indeed, for a transducer network $\mathcal{T}$ and a natural number $k \geq 1$, we can restrict our attention to runs of $\mathcal{T}$ where the sizes of the delivered message *multisets* are of size at most k. This is the *k-delivery* semantics for $\mathcal{T}$. In a previous paper [11], we restricted attention to 1-delivery semantics. In this paper we assume no such bound, unless explicitly mentioned.

3.4 Discussion

In this section we want to motivate the usefulness of our transducer model by comparing it with the literature. First, some of the main characteristics of our model are that *(i)* the same transducer program is replicated at all nodes; *(ii)* a node can only send messages to its neighbors; and *(iii)* messages are never lost. These three characteristics occur commonly in declarative networking literature [28, 23, 29].

The aspect where our model deviates, however, is that the sender of a message cannot address the message to a particular neighbor. But this can be simulated in our model by designating a specific component of each message as the addressee. When a node receives a message, it can verify that it is the valid addressee by using local relation `Id`, in which case it will process the message, and otherwise the node forwards the message to (all) its neighbors. This can be seen as less efficient: depending on the network topology, some nodes might receive many messages not addressed to them, and this could lead to some messages being forwarded forever. But we will not be concerned with such efficiency issues in this paper. Instead, we believe that sending to all neighbors is sufficient to reason about distributed algorithms. For completeness, however, in Section 7 we also briefly consider a transducer variant in which a node can send messages to a specific neighbor, and we relate this model to our epidemic model.

As a last remark, although the network topology is fixed at the start of a run, we note that our model can be used to simulate a dynamic network, in which nodes and edges can be temporarily offline. Indeed, offline nodes can be simulated by having some nodes not doing local transitions for a while, and offline edges can be simulated by delaying the messages sent through them. Our fairness condition however requires that eventually every node keeps on doing local transitions and that message delays are bounded.

4 Expressing Queries

What does it take for a transducer network to compute some global query? Here we propose a formal definition based on the two properties of *consistency* and *network-independence*. This is discussed in the following subsections.

4.1 Transducer Kinds

We will use the following terminology for transducers. We call a transducer *oblivious* if it does not read the relations `Id` and `All` in its queries. Intuitively, this means that the transducer is unaware of the network context, because it does not know about the node it is running on, and it does not know about the other nodes. A transducer is called *inflationary* if it never deletes facts from its memory relations. That is, the deletion queries for the memory relations return the empty set of facts on all inputs. A transducer is called *monotone* if all its queries are monotone. The later Example 4.4 describes a transducer that is at the same time oblivious, inflationary, and monotone.

4.2 Input and Output

Let $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ be a transducer network. Let I be an ordinary (non-distributed) database instance over schema Υ_{in}. This instance can be given as *input* to $\mathcal{T}$ by partitioning it across the nodes, where the same fact can be given to multiple nodes. Formally, a distributed database instance H over $\mathcal{N}$ and Υ_{in} is said to be a *horizontal partition* of I if $I = \bigcup_{x \in nodes(\mathcal{N})} H(x)$.

Let $\rho = (s, b)$ be a configuration of $\mathcal{T}$ on input H. Naturally, ρ defines an output database instance $out(\rho)$ over the schema Υ_{out} as follows:

$$out(\rho) = \bigcup_{x \in nodes(\mathcal{N})} s(x)|_{(\text{out})}.$$

Let $\mathcal{R}$ be a run of $\mathcal{T}$ on some input. We denote the sequence of configurations of $\mathcal{R}$ as ρ_1, ρ_2, etc. If there is a number $i \geq 1$ such that $out(\rho_i) = out(\rho_j)$ for all $j > i$, then we call i a *quiescence point* for $\mathcal{R}$. We call a configuration ρ_i of $\mathcal{R}$ a *quiescence configuration* if i is a quiescence point. Only quiescence configurations can follow a quiescence configuration, and all quiescence configurations define the same output database instance. Only a finite number of distinct output facts are possible because we only consider finite input instances, and because the queries of transducers are generic, and hence domain-preserving. The following property is now clear:

Proposition 4.1. For every transducer network, on every input, every run contains a quiescence configuration.

The *output* of run $\mathcal{R}$ is now defined as $out(\rho_i)$ where ρ_i is a quiescence configuration of $\mathcal{R}$. Our notion of output does not specify the output at individual nodes and does not prevent messages from being sent once a quiescence point is reached.

4.3 Consistency

We say that a transducer network $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$ is *consistent* if for all database instances I over Υ_{in}, on all horizontal partitions of I over $\mathcal{N}$, all fair runs of $\mathcal{T}$ have the same output, denoted $\mathcal{T}(I)$.

When $\mathcal{T}$ is consistent, this function $\mathcal{T}(.)$ has the characteristic of a query, except that it need not be generic. For example, the "query" that asks for all data elements in the input that are not node identifiers, can be computed by a consistent transducer network. We mainly focus on the computation of generic queries. Naturally, $\mathcal{T}$ is said to *compute a query* $\mathcal{Q}$ over input schema Υ_{in} and output schema Υ_{out} if $\mathcal{T}$ is consistent and $\mathcal{T}(I) = \mathcal{Q}(I)$ for every database instance I over Υ_{in} on which $\mathcal{Q}$ is defined.

Because the individual queries that make up a transducer are generic, we can make the following observation:

Proposition 4.2. The function $\mathcal{T}(.)$ is generic for each consistent transducer network $\mathcal{T}$ in which the transducer is oblivious.

4.4 Examples

First, we explain our notational conventions for specifiying concrete transducers. Because FO is equivalent to NrDatalog$^\neg$ [2], we will frequently use the more readable rule-based syntax of NrDatalog$^\neg$ to specify FO-transducers. The answer relations of NrDatalog$^\neg$ programs will be clearly marked. For example, for a memory-insertion query $\mathcal{Q}^R_{\text{ins}}$, the answer relation of the corresponding NrDatalog$^\neg$ program is R_{ins}; for an output query $\mathcal{Q}^T_{\text{out}}$, the answer relation is T_{out}; for a message-sending query $\mathcal{Q}^S_{\text{snd}}$, the answer relation is S_{snd}. We leave a blank line between the NrDatalog$^\neg$ rules that belong to different queries. Unmentioned queries of a transducer are assumed to always return the empty set of facts.

We now give some examples of transducer networks.

Example 4.3. For a simple example of a consistent transducer network, let the input be a binary relation R. Each node outputs the identical pairs from its part of the input. No messages are sent. This network computes the equality selection $\sigma_{\$1=\$2}(R)$. This is easily programmed on an FO-transducer, which is specified as follows. The transducer schema is $\Upsilon_{\text{in}} = \{R^{(2)}\}$, $\Upsilon_{\text{out}} = \{T^{(2)}\}$, $\Upsilon_{\text{msg}} = \emptyset$, $\Upsilon_{\text{mem}} = \emptyset$, and the single transducer rule is:

$$T_{\text{out}}(\mathtt{u}, \mathtt{u}) \leftarrow R(\mathtt{u}, \mathtt{u}).$$

□

Example 4.4. To give an example of a consistent transducer network that involves communication, we compute the transitive closure of a binary relation R in a well-known distributed manner [28]. We present here a naive, unoptimized version. Each node sends its part of the input to its neighbors. Specifically, each node also forwards all messages it receives to its neighbors. This way, the input is flooded to all nodes. Each node accumulates the input facts it receives in a binary memory relation S. Finally, each node has an output relation T in which we repeatedly insert $R \cup S \cup (T \circ T)$, where $\circ$ stands for relational composition. Thanks to the monotonicity of the transitive closure,

this transducer network is consistent. We can implement this idea with an UCQ-transducer. The transducer schema is $\Upsilon_{\text{in}} = \{R^{(2)}\}$, $\Upsilon_{\text{out}} = \{T^{(2)}\}$, $\Upsilon_{\text{msg}} = \{U^{(2)}\}$, $\Upsilon_{\text{mem}} = \{S^{(2)}\}$, and the transducer rules are:

$$U_{\text{snd}}(\mathtt{u}, \mathtt{v}) \leftarrow R(\mathtt{u}, \mathtt{v}).$$
$$U_{\text{snd}}(\mathtt{u}, \mathtt{v}) \leftarrow U(\mathtt{u}, \mathtt{v}).$$

$$S_{\text{ins}}(\mathtt{u}, \mathtt{v}) \leftarrow U(\mathtt{u}, \mathtt{v}).$$

$$T_{\text{out}}(\mathtt{u}, \mathtt{v}) \leftarrow R(\mathtt{u}, \mathtt{v}).$$
$$T_{\text{out}}(\mathtt{u}, \mathtt{v}) \leftarrow S(\mathtt{u}, \mathtt{v}).$$
$$T_{\text{out}}(\mathtt{u}, \mathtt{v}) \leftarrow T(\mathtt{u}, \mathtt{w}), T(\mathtt{w}, \mathtt{v}).$$

Note that the transducer is oblivious. There is no need to reason explicitly about node identifiers, because all we need is let the nodes steadily accumulate all input facts across the network and incrementally produce chunks of output. The transducer is also inflationary and monotone, reflecting the essential nature of the transitive closure computation. □

Example 4.5. Let us see a simple example of a transducer network that is *not* consistent. Consider a network with at least two nodes. The input is a unary relation R. Each node sends its part of R to its neighbors. Next, each node outputs the received messages on condition that the output is still empty. When there are at least two nodes and at least two different R-facts, different runs may deliver the messages in different orders, so different outputs can be produced, even for the same input distributed database instance. We can write an FO-transducer Π to implement this idea. The transducer schema Υ is $\Upsilon_{\text{in}} = \{R^{(1)}\}$, $\Upsilon_{\text{out}} = \{T^{(1)}\}$, $\Upsilon_{\text{msg}} = \{U^{(1)}\}$, $\Upsilon_{\text{mem}} = \emptyset$, and the transducer rules are:

$$U_{\text{snd}}(\mathtt{u}) \leftarrow R(\mathtt{u}).$$

$$\mathtt{block}(\,) \leftarrow T(\mathtt{u}).$$
$$T_{\text{out}}(\mathtt{u}) \leftarrow \neg\mathtt{block}(\,), U(\mathtt{u}).$$

□

Undecidability for testing consistency of a transducer network readily follows from undecidability of satisfiability of FO. The proof is in Appendix A.

4.5 Network-Independence

We are mainly interested in the case where a query can be correctly computed by a transducer regardless of the network.

Let Π be a transducer over a schema Υ. We say that Π is *network-independent* if for all networks $\mathcal{N}$, the transducer networks $\langle \mathcal{N}, \Upsilon, \Pi \rangle$ are consistent and compute the same query $\mathcal{Q}$. We say that $\mathcal{Q}$ is the query *distributedly computed* by

Π. The transducer from Example 4.4 is network-independent. Now consider the following example.

Example 4.6. We give a simple example of a transducer that gives rise to consistent transducer networks but that is *not* network-independent. Suppose we have a unary input relation R. Each node sends its own identifier to its neighbors. This way the edges of the network can be discovered. The discovered edges are forwarded to every node, and when a node detects that the collected edges together form a complete graph, then the node outputs its local input for relation R. If the network is indeed a complete graph, by fairness eventually all nodes will detect this, and then the transducer network computes the identity query. But on other network topologies the empty query is computed.

For completeness, we specify an FO-transducer Π to implement this idea. We define the transducer schema Υ as $\Upsilon_{\text{in}} = \{R^{(1)}\}$, $\Upsilon_{\text{out}} = \{T^{(1)}\}$, $\Upsilon_{\text{msg}} = \{A^{(1)}, B^{(2)}\}$, $\Upsilon_{\text{mem}} = \{S^{(2)}\}$. The rules of the transducer are:

$$A_{\text{snd}}(\mathtt{u}) \leftarrow \mathtt{Id}(\mathtt{u}).$$

$$B_{\text{snd}}(\mathtt{u},\mathtt{v}) \leftarrow A(\mathtt{u}),\ \mathtt{Id}(\mathtt{v}).$$
$$B_{\text{snd}}(\mathtt{u},\mathtt{v}) \leftarrow B(\mathtt{u},\mathtt{v}).$$

$$S_{\text{ins}}(\mathtt{u},\mathtt{v}) \leftarrow B(\mathtt{u},\mathtt{v}).$$

$$\mathtt{missing}(\,) \leftarrow \mathtt{All}(\mathtt{u}),\ \mathtt{All}(\mathtt{v}),\ \mathtt{u} \neq \mathtt{v},\ \neg S(\mathtt{u},\mathtt{v}).$$
$$T_{\text{out}}(\mathtt{u}) \leftarrow R(\mathtt{u}),\ \neg\mathtt{missing}(\,).$$

□

Testing network-independence for FO-transducers is undecidable. See Appendix A for the proof.

4.6 Preliminary Observations

We now give several preliminary results about expressing queries with transducers, that are important for deriving later results.

First, we present two lemmas which show that at each node a transducer can always assemble a local copy of all input facts available on the network.

Lemma 4.7. Let $\mathcal{D}$ be a database schema. There is a transducer schema Υ with $\Upsilon_{\text{in}} = \mathcal{D}$ and an oblivious, inflationary, monotone UCQ-transducer Π over Υ such that for every transducer network for Π, for every instance I of $\mathcal{D}$, on every horizontal partition of I, every fair run reaches a configuration where every node has a local copy of the entire instance I in its memory.

Proof. Because the construction is straightforward, we only provide a sketch. The idea is that all nodes will send out their local input facts and forward any

message they receive. The local inputs, together with the received inputs, are accumulated in local memory relations. In any fair run, eventually all nodes will have received all input facts. Relations `Id` and `All` are not needed. We also do not need deletions on the memory relations. This technique has already been illustrated by Example 4.4. □

We can refine the technique of Lemma 4.7 to let a node know *when* it has collected every input fact in memory:

Lemma 4.8. Let $\mathcal{D}$ be a database schema. There is a transducer schema Υ with $\Upsilon_{\text{in}} = \mathcal{D}$ and an UCQ$^\neg$-transducer Π over Υ such that for every transducer network for Π, for every instance I of $\mathcal{D}$, on every horizontal partition of I, every 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`' is true (implemented by a nullary memory relation). Moreover, the flag '`ready`' does not become true at a node before that node has the entire instance I in its memory.

The transducer Π is not oblivious, but can be made inflationary when using locally the language NrDatalog$^\neg$ instead of UCQ$^\neg$.[2]

Proof. We provide a sketch, and the full construction can be found in Appendix A.2. The idea is that a node x will send its local input facts over relation $R^{(k)} \in \mathcal{D}$ to every other node, with an additional last component that contains the identifier of x, to indicate the origin of the fact. We call this last component the "tag". Next, when a node y receives a tagged input fact, it removes the tag and stores the fact in its memory. This already lets each node incrementally accumulate all inputs across the network. Now, for each fact that y receives from x, node y also sends an acknowledgment back to x. The node x checks whether y has (eventually) acknowledged all the input facts of x. If yes, then x sends out `done`(x, y). From the viewpoint of y, if y has received `done`(x, y) from all other nodes x then it knows that it has accumulated all the input facts on the network, and the `ready`-flag is created at y. The relations `Id` and `All` are used heavily in this protocol. □

The following theorem indicates that our transducer model has enough expressive power to study queries in the distributed context:

Theorem 4.9. Let $\mathcal{L}$ be a language containing UCQ$^\neg$. Then every query expressible in $\mathcal{L}$ can be distributedly computed by an $\mathcal{L}$-transducer. In particular, if $\mathcal{L}$ is a computationally complete query language, every partial computable query can be distributedly computed by an $\mathcal{L}$-transducer.

Proof. Let $\mathcal{Q}$ be a query expressible in $\mathcal{L}$. Let $\mathcal{D}$ and $\mathcal{D}'$ be respectively the input and output schema of $\mathcal{Q}$. We construct an $\mathcal{L}$-transducer Π to compute $\mathcal{Q}$ in two steps. In the first step, we use the partial specification of Π from Lemma 4.8 to obtain the entire input instance at every node. The language UCQ$^\neg$ suffices for this. This transducer has input schema $\Upsilon_{\text{in}} = \mathcal{D}$ but does

[2]This is because a NrDatalog$^\neg$ program allows auxiliary relations to be declared, to which negation can be applied.

not produce any output yet. In the second step, we define the output schema of this transducer to be $\Upsilon_{\text{out}} = \mathcal{D}'$. Now, because $\mathcal{Q}$ is expressible in $\mathcal{L}$, once the flag `ready` becomes true, we can output $\mathcal{Q}$ in the next local transition, by implementing for each output relation an $\mathcal{L}$-query that reads only the collected input facts. □

In the context of the CALM conjecture, monotone queries will play an important role. For now, we observe that oblivious transducers are sufficient to compute them:

Theorem 4.10. Let $\mathcal{L}$ be a query language containing UCQ. Then every *monotone* query expressible in $\mathcal{L}$ can be distributedly computed by an *oblivious* $\mathcal{L}$-transducer. In particular, if $\mathcal{L}$ is computationally complete, every partial computable monotone query can be distributedly computed by an oblivious $\mathcal{L}$-transducer. Moreover, these oblivious transducers can be made inflationary and monotone.

Proof. Let $\mathcal{Q}$ be a monotone query expressible in $\mathcal{L}$. The idea is the same as in the proof of Theorem 4.9, but we now use the oblivious, inflationary, monotone transducer from Lemma 4.7, to let every node gradually collect all inputs facts available on the network. Now, because $\mathcal{Q}$ is expressible in $\mathcal{L}$, in every local transition we can execute $\mathcal{L}$-queries for the output relations that read the part of the input already accumulated in memory. Since $\mathcal{Q}$ is monotone, no incorrect tuples are output this way. Eventually, all nodes have accumulated all the input across the network, and no new outputs will be produced. □

5 The CALM Conjecture

The following was conjectured by Hellerstein:

Conjecture 1 (CALM Conjecture [25])**.** A program has an eventually consistent, coordination-free execution strategy if and only if it is expressible in (monotonic) Datalog.

Before we can rigorously investigate this conjecture, we want to formalize the notion of "coordination-freeness". This is presented in Section 5.1. Next, we will present our formal CALM conjecture and its associated results in Section 5.2. Additional results are in Section 5.3.

5.1 Coordination-free

The CALM conjecture hinges on an intuitive notion of "coordination" of certain distributed computations. We illustrate this notion with a few examples.

In the well-known two-phase commit protocol [22], each node is responsible for executing some part of a distributed transaction. To keep the distributed database consistent in the face of runtime crashes, either all parts should be committed or none is. To this purpose, after executing its part of the distributed

transaction, but before actually committing the results, a node checks that *every* node can commit its results. This way, the distributed commit can proceed only if all individual nodes can commit. Naturally, the nodes have to exchange messages to determine if they can commit or not.

As another example, the multicast protocol of Lemma 4.8 also relies on heavy coordination: the nodes exchange many messages, including acknowledgments, before they all obtain the flag '`ready`'.

Generalizing both examples, the main idea behind coordination is that a large set of nodes needs to obtain a consensus. For two-phase commit this is the global decision whether all nodes should commit or not, and for Lemma 4.8 the consensus is that all nodes have the same data. Reaching a consensus is known to be difficult in the distributed context [15]. Because of the complexity of consensus, the involved nodes sometimes have to wait relatively long before they can continue with the actual computation. This is called a "global barrier" [25].

It should be clear that coordination typically decreases the efficiency of distributed computations, because while the coordination is under way, the nodes are just waiting. So, it seems useful to understand precisely when coordination can be avoided, for which we will use the term "coordination-freeness". This is what the CALM conjecture is all about. It seems hard to give a definitive formalization of coordination-freeness. Still, we offer here a nontrivial definition that appears interesting. A very drastic, too drastic, definition of coordination-free would be to disallow any communication. Our definition is much less severe and only requires that the computation can succeed without communication on "suitable" horizontal partitions. It actually does not matter what a suitable partition is, as long as it exists. Even under this liberal definition, the only-if direction of (our formalization of) the CALM conjecture will turn out to hold.[3]

Formally, let Π be a transducer over a schema Υ. Let $\mathcal{T}$ be a transducer network for Π. We call $\mathcal{T}$ *coordination-free* if for every database instance I over Υ_{in}, there exists a horizontal partition H of I and a run of $\mathcal{T}$ on H in which a quiescence configuration is already reached by performing only heartbeat transitions (zero or more). Intuitively, if the horizontal partition is right, then no communication is required to correctly compute the query. The property of coordination-freeness is mainly interesting for consistent transducer networks, because then at the quiescence configuration that was reached with only heartbeat transitions, the produced output is the same as produced by any other fair run. We call transducer Π *coordination-free* if for every network its corresponding transducer network is coordination-free.

Example 5.1. Consider again the transitive closure computation from Example 4.4. When every node already has the full input, they can each individually compute the transitive closure with only heartbeats. Hence, this transducer is coordination-free. □

[3]Of course, under the drastic definition of coordination-freeness, the if-direction of the CALM conjecture (which is the easy direction) as formulated below in Proposition 5.2, will no longer hold.

The transitive closure query is monotone and this example can actually be generalized in the following proposition. This proposition is implicit in the literature on embarrasingly parallel computation [25, 27, 29], and our main result (Theorem 5.9) will provide a converse to it.

Proposition 5.2. Let $\mathcal{L}$ be a query language containing UCQ. Every monotone query $\mathcal{Q}$ expressible in $\mathcal{L}$ can be distributedly computed by a coordination-free $\mathcal{L}$-transducer.

Proof. Recall from the proof of Theorem 4.10 that there is an oblivious $\mathcal{L}$-transducer that distributedly computes $\mathcal{Q}$. Using the same intuition as in Example 5.1, this transducer is coordination-free. □

The reader should not be lulled into believing that with a coordination-free program it is always sufficient to give the full input at all nodes, as the following example shows:

Example 5.3. Consider the following query $\mathcal{Q}$, having as input two nullary relations A and B, and a nullary output relation T: create the non-empty output (representing "true") if at least one of A and B is nonempty. This query is monotone. Consider the following (contrived) transducer Π to compute $\mathcal{Q}$. If the network has only one node (which can be tested by looking at the relation `All`), the transducer simply outputs the answer to the query. Otherwise, it first tests if its local input fragments of A and B are *both* nonempty. If this is the case, nothing is output locally yet, but a nullary fact C is sent out. Any node that receives the message C will output it. When precisely one of A and B is nonempty locally, the transducer simply outputs the correct output directly. The transducer is network-independent. Also, the transducer is coordination-free, because on networks with at least two nodes there always is a partition of the data under which no node has both A and B locally nonempty, and the query can be computed without communication. Moreover, when A and B are both nonempty, and every node has the entire input, no run will reach a quiescence configuration without communication. □

The following two examples show that network-independence for a transducer does not guarantee coordination-freeness, and vice versa.

Example 5.4. We provide an example of a transducer that is network-independent but not coordination-free, i.e., requires communication. Let $\mathcal{Q}$ be the following "emptiness" query, having a nullary input relation R, and a nullary output relation T: create the non-empty output (representing "true") iff R is empty. This query is nonmonotone. We now describe a transducer to distributedly compute $\mathcal{Q}$. Since every node can have a part of the input, the nodes coordinate with each other to be certain that R is empty at every node. Every node sends out its identifier (using the relation `Id`) on condition that its local relation R is empty. Received messages are forwarded, so that if R is globally empty, eventually all nodes will have received the identifiers of all nodes, which can be checked using

the relation `All`. When this happens, the transducer at each node outputs a nullary fact.

For completeness, we specify an FO-transducer Π to implement this idea. The transducer schema Υ is as follows: $\Upsilon_{\text{in}} = \{R^{(0)}\}$, $\Upsilon_{\text{out}} = \{T^{(0)}\}$, $\Upsilon_{\text{msg}} = \{U^{(1)}\}$ and $\Upsilon_{\text{mem}} = \{S^{(0)}\}$. The rules are:

$$U_{\text{snd}}(\mathtt{u}) \leftarrow \mathtt{Id}(\mathtt{u}),\ \neg R(\,).$$
$$U_{\text{snd}}(\mathtt{u}) \leftarrow U(\mathtt{u}).$$

$$S_{\text{ins}}(\mathtt{u}) \leftarrow \mathtt{Id}(\mathtt{u}),\ \neg R(\,).$$
$$S_{\text{ins}}(\mathtt{u}) \leftarrow U(\mathtt{u}).$$

$$\mathtt{missing}(\,) \leftarrow \mathtt{All}(\mathtt{u}),\ \neg S(\mathtt{u}).$$
$$T_{\text{out}}(\,) \leftarrow \neg\mathtt{missing}(\,).$$

□

Example 5.5. We give a transducer that is coordination-free, and that is consistent on every network, but is not network-independent. The transducer has two unary input relations R and S, and it has a unary output relation T. Using relations `Id` and `All`, the transducer can detect if there is only one node, or if there are more nodes. If there is just one node, the single node outputs the union of R and S. If there are at least two nodes, then all nodes will copy their local inputs into their memory; they also broadcast their input facts to each other, so that all nodes accumulate all inputs of the network; and, the nodes will continuously output the intersection of the accumulated R-facts with the accumulated S-facts.

First, we see that on each network this transducer is consistent. Indeed, on a single-node network the union of R and S is output, and on a multi-node network the intersection of R and S is output. This different output behavior prevents the transducer from being network-independent. Finally, the transducer is coordination-free because on a single-node network the output is always computed with only heartbeats, and on a multi-node network we can consider the partition where each node has the entire input, and then the intersection of R and S can already be computed with only heartbeats. □

Coordination-freeness seems a useful property for a transducer to have. However, it cannot be decided automatically in general:

Proposition 5.6. Coordination-freeness is undecidable for FO-transducers.

Proof. We reduce the finite satisfiability problem for FO to deciding coordination-freeness for FO-transducers. Let φ be an FO-sentence over a database schema $\mathcal{D}$. We construct an FO-transducer Π that is coordination-free iff φ is *not* finitely satisfiable.

Consider the transducer Π in Example 5.4, that is over schema Υ. We may assume without loss of generality that the relation names of Υ do not occur in $\mathcal{D}$. We obtain a new transducer schema Υ' from Υ by adding $\mathcal{D}$ to Υ_{in}; by adding new message relations $\{(C^{\text{msg}}, k) \mid C^{(k)} \in \mathcal{D}\}$; and, by adding new memory relations $\{(C^{\text{mem}}, k) \mid C^{(k)} \in \mathcal{D}\}$. We obtain a new transducer Π' over Υ' by modifying Π to let all nodes gradually accumulate all input facts by means of message forwarding. Moreover, besides keeping the old output condition "$\neg$`missing`", we will only produce an output if additionally φ is satisfied on the accumulated $\mathcal{D}$-facts so far (in memory).

For the first direction, suppose that φ is finitely satisfiable on a database instance I over $\mathcal{D}$. We show that Π' is not coordination-free. We can regard I as a database instance over Υ'_{in}, where relation R is empty. Let $\mathcal{N}$ be a network containing two nodes x and y. Let $\mathcal{T}$ denote the transducer network based on $\mathcal{N}$ and Π'. Suppose that there is some horizontal partition H of I over $\mathcal{N}$ and a run $\mathcal{R}$ of $\mathcal{T}$ on input H in which a first quiescence configuration is already reached by doing only heartbeat transitions. Because I does not contain $R(\,)$, the nodes send the messages $U(x)$ and $U(y)$. Because of fairness, these messages must be delivered to y and x respectively, which can happen only after the first quiescence point because before the quiescence point there are only heartbeat transitions. Eventually, every node will find $\neg$`missing`$(\,)$ to be true. The same reasoning can be applied to the relations of $\mathcal{D}$: whether I is empty or not, there must be a configuration after the first quiescence point, in which all nodes have accumulated I in the memory relations. Then φ also becomes true, and thus we know that every node eventually outputs $T()$. Note that this fact cannot be in the first quiescence configuration because it requires the delivery of at least one of the messages $U(x)$ or $U(y)$. So, the initial quiescence configuration that was reachable by only heartbeat transitions cannot exist. Thus, the network $\mathcal{N}$ and input I are a proof that Π' is not coordination-free.

For the other direction, suppose that φ is not finitely satisfiable. Then no transducer network based on Π' can produce output, no matter what the input instance over Υ'_{in} or horizontal partition of that instance is. Hence, the start configuration of every run is already a quiescence configuration, and Π' is coordination-free. □

Although coordination-freeness is undecidable for FO-transducers (and by extension more powerful transducers), we can identify a syntactic class of transducers that is guaranteed to be coordination-free, and that will prove to have the same expressive power as the class of coordination-free transducers. Importantly, the syntactic restriction does not guarantee network-independence. Recall from Section 4.1 that an *oblivious* transducer does not read the system relations `Id` and `All`. For now we observe:

Proposition 5.7. Let $\mathcal{L}$ be a query language. Every network-independent, oblivious $\mathcal{L}$-transducer is coordination-free.

Proof. Let Π be a network-independent, oblivious $\mathcal{L}$-transducer over a schema Υ. Let $\mathcal{Q}$ be the query distributedly computed by Π.

First, on a single-node network, the single node is always given the entire input and there can only be heartbeat transitions. Then, for an input instance I over Υ_{in}, a quiescence configuration containing $\mathcal{Q}(I)$ is always reached by doing only heartbeat transitions.

Now consider any other network $\mathcal{N}$, any instance I over Υ_{in}, and the horizontal partition H that places the entire instance I at every node. Since Π is oblivious, nodes cannot detect that they are on a network with multiple nodes unless they receive a message. So, by doing only heartbeat transitions initially, every node will act the same as if in a single-node network and will already output the entire $\mathcal{Q}(I)$. Because Π is network-independent, the nodes cannot output more than $\mathcal{Q}(I)$ when they receive messages afterwards. □

5.2 Main Results

Now we can formalize the original Conjecture 1. We will take the terms "program" and "to have an execution strategy" to mean "query" and "to be distributedly computed by a transducer", respectively. The term "eventually consistent" is then formalized by our notions of consistency and network-independence. Under this interpretation, the conjecture becomes:

Conjecture 2. A query can be distributedly computed by a coordination-free transducer if and only if it is expressible in Datalog.

Let us immediately get the if-side of this conjecture out of the way. It holds, because a query in Datalog is monotone, and then by Theorem 4.10 there exists an oblivious transducer to compute the query, but we have seen in Proposition 5.7 that oblivious transducers are coordination-free.

As to the only-if side, the explicit mention of Datalog is a bit of a nuisance because Datalog is limited to polynomial time whereas there certainly are monotone queries outside PTIME. We also mention the celebrated paper [8] where Afrati, Cosmadakis and Yannakakis show that even within PTIME there exist queries that are monotone but not expressible in Datalog.

But Datalog aside, however, the true emphasis of the CALM Conjecture clearly lies in the monotonicity aspect. Indeed, we confirm it in this sense:

Theorem 5.8. Let $\mathcal{L}$ be a query language. Every query that is distributedly computed by a coordination-free $\mathcal{L}$-transducer is monotone.

Proof. Let Π be a coordination-free $\mathcal{L}$-transducer over a schema Υ that distributedly computes a query $\mathcal{Q}$. Let I and J be two database instances over the schema Υ_{in} such that $I \subseteq J$. We must show that $\mathcal{Q}(I) \subseteq \mathcal{Q}(J)$. Consider a fact $\boldsymbol{f} \in \mathcal{Q}(I)$. Consider a network $\mathcal{N}$ with at least two nodes. Let $\mathcal{T}$ denote the transducer network based on $\mathcal{N}$ and Π. Since Π is coordination-free and network-independent, there exists a horizontal partition H of I and a run $\mathcal{R}$ of $\mathcal{T}$ on input H in which a quiescence configuration, containing the facts $\mathcal{Q}(I)$, is already reached by letting the nodes do only heartbeat transitions. Let x be a node where $\boldsymbol{f}$ is output in the quiescence configuration. Let y be a node different from x and consider a horizontal partition H' of J where $H'(x) = H(x)$

and $H'(y) = H(y) \cup (J \setminus I)$. Let n be the number of initial heartbeat transitions with recipient x in run $\mathcal{R}$ that were needed to output $\boldsymbol{f}$ at x. Consider a prefix of a run of $\mathcal{T}$ on input H' in which we initially do n heartbeat transitions, all with active node x. Because local transitions are deterministic, the node x goes through the same state changes as in run $\mathcal{R}$ before $\boldsymbol{f}$ is output and therefore $\boldsymbol{f}$ is output again in this prefix. The prefix can be extended to a full fair run $\mathcal{R}'$ of $\mathcal{T}$ on input H'. Since $\mathcal{T}$ is consistent, the fact $\boldsymbol{f}$ will be output on any partition of J, during any fair run. Hence, $\boldsymbol{f}$ belongs to the query computed by $\mathcal{T}$ applied to J. Moreover, Π is network-independent, so $\boldsymbol{f}$ belongs to $\mathcal{Q}(J)$. □

We can now obtain the following result:

Theorem 5.9. Let $\mathcal{L}$ be a query language containing UCQ. For every query $\mathcal{Q}$ that is expressible in $\mathcal{L}$, the following are equivalent:

1. $\mathcal{Q}$ can be distributedly computed by a coordination-free $\mathcal{L}$-transducer;
2. $\mathcal{Q}$ can be distributedly computed by an oblivious $\mathcal{L}$-transducer; and,
3. $\mathcal{Q}$ is monotone.

Proof. Theorem 4.10 yields *(3)* ⇒ *(2)*; Proposition 5.7 yields *(2)* ⇒ *(1)*; Theorem 5.8 yields *(1)* ⇒ *(3)*. □

In particular, if $\mathcal{L}$ is computationally complete, then the previous equivalences hold for any computable query. As a small remark, now it is of no surprise that Example 5.4 required coordination; indeed, there we distributedly compute a non-monotone query.

5.2.1 Discussion

Theorem 5.9 can be used as follows in practice. Essentially, by restricting a language, its execution can in general be optimized more thoroughly than the unrestricted language. A well-known example is SQL versus a Turing-complete programming language. For our situation, the programmer of a distributed (query) algorithm can write a program in a high-level declarative formalism, like the transducer model presented in this paper, or a Datalog-variant like e.g. [27, 10, 1]. Suppose that the query is monotone. Then we know by Theorem 5.9 that it can be implemented in a coordination-free manner. Moreover, we can prevent the programmer from abusing coordination using the syntactic restriction of obliviousness. The main idea is that the programmer is given only a few communication primitives, like sending a message to its neighbors, and a syntactic restriction is imposed to prevent the programmer from using network relations like `Id` or `All` (or equivalent information). Next, the programmer, or a software tool, needs to assert that the program is network-independent, i.e., on every network, all fair runs produce the desired outcome. Then, using Theorem 5.9, if the runtime is told that the program is oblivious and network-independent, the runtime can execute the program without any coordination. By contrast, if the programmer uses `Id` and `All`, then this semantic property

is no longer guaranteed, and one would have to resort to a general execution strategy that has built-in coordination, which seems a waste if the program expresses a monotone computation. This way, obliviousness could be a useful guiding principle for distributed query evaluation. The works of Loo et al. [27] and Nigam et al. [29] provide coordination-free distributed execution engines for Datalog.

5.3 Further Results

It is natural to wonder about variations of our model. One question may be about the system relations `Id` and `All`. Without them (the oblivious case), we know that we are always coordination-free and thus monotone.

What if we would read precisely one system relation; only `Id` or only `All`? As to coordination-freeness, the argument given in the proof of Proposition 5.7 still works when the transducer reads only `Id`, because then nodes still cannot detect that they are on a network with multiple nodes. However, the argument fails when the transducer reads only `All`, and indeed we have the following counterexample.

Example 5.10. We describe a transducer that is network-independent, reads only `All`, but that is not coordination-free. The query expressed is simply the identity query on a unary relation R. The transducer can observe the difference between a single-node and a multi-node network by looking at the relation `All`. If it is a single-node network, the node simply outputs the result directly. If it is a multi-node network, every node sends out a message. Only upon receiving a message will a node then output the result. Thus on a multi-node network, regardless of the horizontal partition, communication is needed for the transducer network to produce the required output. □

So, coordination-freeness is not guaranteed when reading only `All`, but yet, monotonicity is not lost.

Theorem 5.11. Let $\mathcal{L}$ be a query language. Every query distributedly computed by an $\mathcal{L}$-transducer that reads only relation `All`, is monotone.

Proof. Let Π be a network-independent transducer that reads only `All`. As a technical convenience, we assume that runs can use *concurrent* global transitions, in which multiple nodes can be active at the same time, each receiving messages from their own message buffer. At the end of such a concurrent global transition, for each node, its message buffer is extended with the multiset union of all messages sent to it by its neighbors. These concurrent transitions can be simulated by a sequence of ordinary single-node transitions, as remarked at the end of Section 3.1.

Let Υ be the schema of Π. Let $\mathcal{Q}$ be the query distributedly computed by Π. Let I and J be two database instances over Υ_{in} such that $I \subseteq J$. Let $\boldsymbol{f} \in \mathcal{Q}(I)$. We have to show that $\boldsymbol{f} \in \mathcal{Q}(J)$. The main trick used in this proof is that although Π can count the number of nodes of a network (using relation `All`),

it cannot directly observe the *edges* of the network. So, when $\boldsymbol{f}$ is output on input I in one network, we can fool the transducer to output $\boldsymbol{f}$ on input J in another network that has only slightly different edges.

Run on I Consider a network $\mathcal{N}_1$ in the form of a ring, containing at least four nodes. See Figure 1 for an example. Let $\mathcal{T}$ denote the transducer network based on $\mathcal{N}_1$ and Π. Let H_1 be the horizontal partition of I that places I on every node of $\mathcal{N}_1$.

We show now that there exists a run $\mathcal{R}_1$ of $\mathcal{T}$ on input H_1 with sequence of configurations $\rho_1 = (s_1, b_1)$, $\rho_2 = (s_2, b_2)$, ..., such that for each $i \geq 1$ and each $x, y \in \mathit{nodes}(\mathcal{N}_1)$ we have $s_i(x) = s_i(y)$ and $b_i(x) = b_i(y)$. In words: in every configuration, all nodes have the same transducer state and the same message buffer. We inductively construct $\mathcal{R}_1$. For the base case ($i = 1$), configuration ρ_1 satisfies the property because it is the start configuration: all nodes are given the entire input I, and all message buffers are empty. For the induction hypothesis, assume that the property holds for i. For the inductive step, we show how to continue the partially constructed run $\mathcal{R}_1$ so that the property holds for $i+1$. Denote $m = b_i(x)$ for some node x. Possibly $m = \emptyset$. We next do a concurrent global transition in which *each* node is the recipient of delivered message multiset m. This is possible by using the induction hypothesis. So, we are delivering the entire message buffer at once to each node. Again by the induction hypothesis, all nodes have the same state in configuration ρ_i, and since local transitions are deterministic, all nodes will have the same state in configuration ρ_{i+1}. Also, if one node sends a message set J_{snd} on delivery of m, then all nodes will send this set on delivery of m. Hence, because $\mathcal{N}_1$ is a ring, for each node, the messages of J_{snd} will have been added *twice* to its message buffer at the end of the concurrent global transition. Since all nodes emptied their message buffer at the beginning of the concurrent transition, we see that in ρ_{i+1} the nodes again have the same message buffer.

The run $\mathcal{R}_1$ can be converted to a fair run $\mathcal{R}'_1$ with only non-concurrent global transitions and that produces the same output as $\mathcal{R}_1$. Moreover, because Π is network-independent, we know that $\mathcal{R}'_1$ outputs $\mathcal{Q}(I)$, and thus $\mathcal{R}_1$ outputs $\mathcal{Q}(I)$. Therefore, we can consider a node u of $\mathcal{N}_1$ and an index $k \geq 1$ such that u outputs $\boldsymbol{f}$ during the k^{th} concurrent global transition of $\mathcal{R}_1$.

Run on J Let u be the node as previously defined. Let z be a node of $\mathcal{N}_1$ that is not a neighbor of u. We obtain a new network $\mathcal{N}_2$ from $\mathcal{N}_1$ by adding an edge between the two neighbors of z. Because $\mathcal{N}_1$ is a ring with at least four nodes, we know that this edge was not previously there and thus $\mathcal{N}_2$ contains a smaller ring without node z. Let $\mathcal{T}'$ denote the transducer network based on $\mathcal{N}_2$ and Π. Importantly, note that $\mathcal{N}_1$ and $\mathcal{N}_2$ have precisely the same nodes. Let H_2 be the horizontal partition of J that places I on every node except z and that places $J \setminus I$ on z.

Let us abbreviate $N = \mathit{nodes}(\mathcal{N}_2) \setminus \{z\}$. Recall the sequence of configurations ρ_1, ρ_2, ..., of run $\mathcal{R}_1$ from above. We now show that there exists an (unfair) run

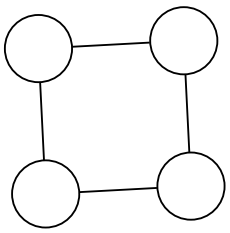

Figure 1: Ring network topology

$\mathcal{R}_2$ of $\mathcal{T}'$ on input H_2 with sequence of configurations $\rho'_1 = (s'_1, b'_1)$, $\rho'_2 = (s'_2, b'_2)$, ..., such that for each $i \geq 1$ and each $y \in N$ we have $s'_i(y) = s_i(y)$ and $b'_i(y) = b_i(y)$. In words: the smaller ring of nodes N follows exactly the states and message buffers of run $\mathcal{R}_1$. We inductively construct $\mathcal{R}_2$. For the base case ($i = 1$), the property is satisfied because input partition H_2 initializes the nodes of N in the same way as input partition H_1. For the induction hypothesis, we assume that the property holds for index i. For the inductive step, we show that the property holds for index $i + 1$. As in the construction of $\mathcal{R}_1$, we next do a concurrent global transition in which we deliver to every node of N the contents of its entire message buffer. Using the induction hypothesis, this causes each node of N to send the same message instance J_{snd} to their neighbors. This message instance was also sent during the corresponding global transition of $\mathcal{R}_1$. Let y_1 and y_2 denote the two neighbors of node z in $\mathcal{N}_1$. We have $\{y_1, y_2\} \subseteq N$. Because we have added the extra edge between y_1 and y_2 in $\mathcal{N}_2$, node y_1 sends J_{snd} to z *and* to y_2. This is similar for y_2. Node z does not send anything because it is ignored. So, like in $\mathcal{R}_1$, both y_1 and y_2 have J_{snd} added precisely twice to their message buffer at the end of the concurrent global transition. The rest of the reasoning is the same as in the inductive step for constructing $\mathcal{R}_1$. We obtain that the nodes of N have the same state and message buffers in configuration ρ'_{i+1} as in configuration ρ_{i+1}.

Consider again the run $\mathcal{R}_2$. Because $u \in N$, the fact $\boldsymbol{f}$ is eventually output at u during $\mathcal{R}_2$, during some global transition k. But $\mathcal{R}_2$ is clearly not fair because the node z is ignored. However, we can make a new run $\mathcal{R}'_2$ by copying only the first k global transitions of $\mathcal{R}_2$, converting each of them to a sequence of ordinary (non-concurrent) global transitions and then extending this prefix arbitrarily to a full fair run. Thus, we obtain that $\boldsymbol{f}$ is output in a fair run of $\mathcal{T}'$ on input H_2. Since Π is network-independent, we obtain that $\boldsymbol{f} \in \mathcal{Q}(J)$, as desired. □

As a corollary, we can add two more statements to the three equivalent statements of Theorem 5.9:

Corollary 5.12. Let $\mathcal{L}$ be a query language containing UCQ. The following statements are equivalent for any query $\mathcal{Q}$ expressible in $\mathcal{L}$:

1. $\mathcal{Q}$ can be distributedly computed by an oblivious $\mathcal{L}$-transducer;

2. $\mathcal{Q}$ can be distributedly computed by an $\mathcal{L}$-transducer that is given only `Id`; and,

3. $\mathcal{Q}$ can be distributedly computed by an $\mathcal{L}$-transducer that is given only `All`.

Proof. The directions *(1)* $\Rightarrow$ *(2)* and *(1)* $\Rightarrow$ *(3)* are immediate because an oblivious transducer is given neither of `Id` or `All`. For *(2)* $\Rightarrow$ *(1)*, when only `Id` is read, the query $\mathcal{Q}$ is monotone as argued above. Then, by also using that $\mathcal{Q}$ is expressible in $\mathcal{L}$, we can apply Theorem 4.10 to know that $\mathcal{Q}$ is computable by an oblivious $\mathcal{L}$-transducer. The direction *(3)* $\Rightarrow$ *(1)* is similar, but this time Theorem 5.11 is used. □

To conclude this section, we note that distributed algorithms involving a form of coordination typically require the participating nodes to have some knowledge about the other participating nodes [15]. This justifies our modeling of this knowledge in the form of the system relations `Id` and `All`. Importantly, we have shown that these relations are only necessary if one wants to compute a nonmonotone query in a distributed fashion.

6 Expressiveness Analysis

In this section we want to better understand the transducer model itself. The main question we would like to address is how the transducer model can be combined with a local query language to express a global query. It is not obvious what peculiarities of the model can be exploited in the local queries, and how. It will turn out actually that the global query language expressed by the transducer is the while-closure of its local query language. Intuitively, this is because each node can do multiple local transitions in a run, which can be seen as iterations of an implicit while-loop. This is very natural, and we believe this shows that our (distributed) transducer model is relatively elegant, because it respects previous results about well-known query languages [2].

Table 1 summarizes the expressiveness results.

6.1 While versus FO

We first show the following property, and although the result might not sound very surprising, writing out the details turned out to be rather intricate.

Lemma 6.1. A query is expressible in While if and only if it is computable by an FO-transducer on a single-node network.

(sketch). For the only-if direction, we have to simulate a While-program on a single-node FO-transducer network. A While-program can be simulated by iterated heartbeats using well-known techniques [4]. The main idea is that the loops in the while-program are rewritten with explicit "goto" statements. The statements of this rewritten program can then be simulated by an FO-transducer

Queries expressible in While = queries computable by FO-transducers = queries computable by UCQ$^{\neg}$-transducers
Monotone queries expressible in While = queries computable by oblivious FO-transducers
Queries expressible in Datalog = queries computable by inflationary NrDatalog-transducers
Queries in PSPACE = queries computable by multi-node FO-transducer networks under 1-delivery semantics

Table 1: Expressiveness Summary

that keeps track of which statement is to be executed next, and goto-statements can make the simulation jump back to a previous statement (simulating a loop). We illustrate this technique in Appendix B.1.

For the if-direction, let Π be an FO-transducer over a schema Υ that computes a query $\mathcal{Q}$ on a single-node network $\mathcal{T}$. A While-program that computes the query $\mathcal{Q}$ has to use exactly the same input and output schema as Π, namely, Υ_{in} and Υ_{out} respectively. The While-program is however allowed to declare any number of temporary relations. We may assume that Π does not read message relations in its internal queries, because no messages can be received on a single-node network. As a first case, let us additionally assume that the internal FO-queries of Π do not read relations `Id` and `All` (the oblivious case). Now, because the memory relations of Π start empty, and temporary relations declared in the While-program also start empty, we can easily construct a While-program P that consists of one big loop, of which one iteration performs the same state changes as Π during one heartbeat transition. We provide an example in Appendix B.2. In order to terminate, P must detect repetition of transducer states, because this implies that Π has repeated a state and will output no new output facts. Detecting such a repetition is possible by using the technique of Abiteboul and Simon [3].

Let us now consider the second case where Π reads `Id` or `All` (or both) in its internal queries. These relations can not be simulated by the While-program. Indeed, these relations are always non-empty from the perspective of Π, and a While-program can not create temporary relations to represent them: when the input is empty, the While-program can not invent a value to store in `Id` and `All`, and when the input is nonempty, the While-program can in general not choose one value to store in `Id` and `All`. Therefore, we will first eliminate the use of `Id` and `All` from the queries of Π. Once this is done, we can apply the above translation for the oblivious case.

Remove relation `All` Note that in the FO-queries of Π we can replace the use of relation `All` by `Id` because, on a single-node network, both relations have the same contents. Formally, in a transducer state there is a fact $\texttt{Id}(a)$ iff there is a fact $\texttt{All}(a)$.

Remove relation `Id` Assume that relation `All` is not used in Π. Next, we remove the use of relation `Id` from Π. We will only sketch the approach, and the details can be found in Appendix B.3. We use the work of Van den Bussche and Cabibbo [31], who have shown how to convert an ordinary (untyped) FO-formula to a *typed* formula that computes the same query. In typed formulas, each variable is of a specific *sort*, meaning that it ranges over an isolated domain of values. In our case, we distinguish between two sorts: *(i)* values in the active domain of an input database instance over Υ_{in}; and, *(ii)* the identifier x of the single node in $\mathcal{T}$ (with $\mathcal{T}$ as defined above). We will denote these sorts as respectively *adom* and *id*. A *type* τ is a tuple of sort symbols, like $(adom, id, id)$.

Based on Π, we construct a second transducer Π^2 as follows. For each relation $R^{(k)} \in \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}}$ of Π and each type τ of arity k, transducer Π^2 has a relation ${R_\tau}^{(k)}$. Transducer Π^2 also has a memory relation `Adom` in which it stores all values from its input. We now describe how Π^2 updates such a relation ${R_\tau}^{(k)}$. Let φ denote the FO-formula used by Π to insert tuples in relation R (deletion is similar). If for example $\tau = (adom, adom, \ldots, id)$, then transducer Π^2 will use a formula of the form

$$\psi(\mathtt{u_1}, \ldots, \mathtt{u_k}) \wedge \texttt{Adom}(\mathtt{u_1}) \wedge \texttt{Adom}(\mathtt{u_2}) \wedge \ldots \wedge \texttt{Id}(\mathtt{u_k})$$

to insert into R_τ the tuples of type τ that are computed by φ. The formula ψ is basically the formula φ, but modified to cope with the separation of tuples by their type: each time φ reads a tuple from a relation $S^{(l)}$, formula ψ reads a tuple from the union $\bigcup_{\tau \in \alpha} {S_\tau}^{(l)}$, where α is all types of arity l. This way, Π^2 also computes the same query as Π.

Now, we can apply Proposition 1 of [31] to the formulas in Π^2 to obtain new formulas in which there is no explicit reference to relations `Adom` and `Id`. Instead, the converted formulas use variables of two sorts (the *adom* and *id* sorts). In a last step, we can syntactically eliminate any reference to *id* variables, and obtain back normal FO-formulas. These can be used in a new transducer Π^3, which is oblivious, to compute the same query as Π. □

Now we can obtain the following result:

Theorem 6.2. A query is expressible in While if and only if it can be distributedly computed by an FO-transducer.

Proof. For the if-direction, let Π be an FO-transducer that distributedly computes a query $\mathcal{Q}$. Because Π is network-independent, the query $\mathcal{Q}$ must also be computed when executing Π on a single-node network. Then, by using Lemma 6.1, there is a While-program that computes $\mathcal{Q}$.

For the only-if direction, let $\mathcal{Q}$ be a query that can be computed by a While-program. We specify an FO-transducer to compute $\mathcal{Q}$ in two steps. First we

use Lemma 4.8 to obtain the entire input instance at every node. Every node can then act as if it was alone, ignoring any further messages, and simulate the While-program again using Lemma 6.1. □

For monotone queries we have the following, more specific result:

Theorem 6.3. Every *monotone* query expressible in While can be distributedly computed by an *oblivious* FO-transducer.

Proof. Let $\mathcal{Q}$ be a monotone query expressible in While. We construct an oblivious FO-transducer to compute $\mathcal{Q}$. Note that Theorem 4.10 is not applicable, because that would give us an oblivious While-transducer, and not an oblivious FO-transducer. But the proof idea of the theorem can still be used.

First, we use the simple UCQ-protocol of Lemma 4.7 to let all nodes accumulate all input facts in memory. This does not require `Id` or `All`. Next, every time a node receives a new input fact, it starts or restarts a simulation of the While-program for $\mathcal{Q}$. The simulation uses the techniques of the proof of Lemma 6.1 (only-if direction), where specifically the output facts are first computed in temporary memory relations before being officially output. Checking whether a new input fact is received is done by comparing a received input fact with the previously accumulated input facts in memory. The restarting of the simulation of the While-program is done by emptying all memory relations, and restarting the program counter. The restart can happen at the moment a simulation is busy, in which case the temporary output is discarded. The restart can also happen after a simulation was already successfully ended. Since the query $\mathcal{Q}$ is monotone, no incorrect facts were output by previous simulations.

Eventually, every node will have accumulated all input facts, so the simulation can surely run to completion on all input facts. We also do not need relations `Id` and `All` to simulate the While-program. Hence, the transducer is oblivious. □

Note that the converse of Theorem 6.3, to the effect that every query distributedly computed by an oblivious FO-transducer is monotone and expressible in While, holds by combining Theorems 5.9 and 6.2 that give respectively the monotonicity of the query and the expressibility in While.

For our next result, we will use that FO is equivalent to NrDatalog$^\neg$ [2]. Basically, a program in NrDatalog$^\neg$ is a sequence of UCQ$^\neg$ statements. The following proposition shows that transducers can simulate this sequential composition of simpler statements:

Proposition 6.4. (i) Every query that can be distributedly computed by an FO-transducer can be distributedly computed by an UCQ$^\neg$-transducer.

(ii) Every *monotone* query that can be distributedly computed by an FO-transducer can be distributedly computed by an *oblivious* UCQ$^\neg$-transducer.

Proof. First, we make a general observation. For every query $\mathcal{Q}$ that is distributedly computed by an FO-transducer, we can apply Theorem 6.2 to know

that $\mathcal{Q}$ is expressible with a While-program P. Moreover, since the language FO is equivalent to NrDatalog$^\neg$ [2], every FO-statement in P can be replaced by a sequence of UCQ$^\neg$-statements, to obtain a new program P'. Then, it is clear that program P' can be simulated by an UCQ$^\neg$-transducer on a single-node network using iterated heartbeats, very similar to the proof of the only-if direction for Lemma 6.1.

For result *(i)*, we let each node first collect a local copy of the entire input by using the protocol of Lemma 4.8, which can be done with a UCQ$^\neg$-transducer. After collecting the input, each node can simulate the program P' is isolation.

For result *(ii)*, where $\mathcal{Q}$ is monotone, we use instead Lemma 4.7 to let each node gradually accumulate all input, and we restart the simulation of P' when new inputs arrive. □

6.2 Datalog versus NrDatalog

What if we are only interested in Datalog? Between the languages Datalog and NrDatalog, a similar relation exists as between While and FO:

Theorem 6.5. A query is expressible in Datalog if and only if it can be distributedly computed by an inflationary NrDatalog-transducer.

Proof. First we consider the only-if direction. We construct an oblivious, inflationary transducer to simulate a Datalog program. The basic idea is the same as in the proof of Theorem 4.10. The input tuples are sent out and accumulated on every node. During every transition, we apply the immediate consequence operator of the Datalog program [2], that can be expressed by NrDatalog. The relations `Id` and `All` are not needed, and the transducer can be made oblivious. Also, by the monotone nature of Datalog evaluation, deletions are never needed, and the transducer can be made inflationary.

Now we consider the if-direction. Let $\mathcal{Q}$ be a query distributedly computed by an inflationary NrDatalog-transducer Π over a schema Υ. We show that $\mathcal{Q}$ can be expressed in Datalog. Because of network-independence, it is sufficient to look at the behavior of Π on a single-node network. We simulate this behavior with a Datalog program P as follows. We assume that the logical "and" and the universal quantifier are not core primitives of FO, since these can be simulated by negation together with respectively the logical "or" and the existential quantifier. We call an FO-formula *positive* if each atom and existential quantifier occurs under an even number of negation symbols. The language NrDatalog is equivalent to positive FO. So, Π is just an inflationary FO-transducer, in which the internal FO-queries are positive. Now, the same transformation as in the proof of the if-direction for Lemma 6.1 can be applied to transform Π into a new FO-transducer Π' that computes $\mathcal{Q}$ without reading relations `Id` and `All`. Moreover, this transformation preserves the positivity of the formula. Hence, Π' can be immediately seen as an inflationary NrDatalog transducer that does not read `Id` and `All`. Next, we unite all NrDatalog rules of Π' in a Datalog program P. Because P by the nature of Datalog can only accumulate its generated facts, P has at least the opportunities of Π' to join facts, and P outputs at least the

output of Π'. Moreover, because Π' is inflationary itself, Π' eventually has the same opportunities to join facts as P. In conclusion, P computes exactly the original query $\mathcal{Q}$. □

It remains open if we can drop the word "inflationary" from Theorem 6.5.

6.3 Restrict Delivery

It is well-known that providing an order on the active domain increases the expressiveness of a query language [2]. This result transfers nicely to our transducer model. By guaranteeing that only one message is delivered during every global transition, referred to as 1-*delivery semantics* (cf. Section 3.3), an order can be established on each node:

Proposition 6.6. Under 1-delivery semantics, every PSPACE query can be computed by an FO-transducer network with at least two nodes.

Proof. In a network with at least two nodes, under 1-delivery semantics, each node can establish a linear order on the active domain by cooperating with the other nodes as follows. When a node has collected all inputs of the network (by means of Lemma 4.8), it sends out the elements of the active domain, that get forwarded by other nodes. Eventually, all these elements arrive back at the node, and the order can be established because at most one value is received at once. Then, each node can simulate a While-program on the collected input, that uses the established order. The transducer involved is not truly network-independent, as this only works when there are at least two nodes. □

6.4 Specialized CALM Properties

Using our previous results about expressivity, we obtain the following variants of Theorem 5.9. Especially, the second variant, which deals with Datalog, may come closest to the CALM conjecture as originally imagined by Hellerstein [25].

Corollary 6.7. Within each of the following two groups, the statements are equivalent, for any query $\mathcal{Q}$:

1. (a) $\mathcal{Q}$ can be distributedly computed by a coordination-free FO-transducer.
   (b) $\mathcal{Q}$ can be distributedly computed by an oblivious FO-transducer.
   (c) $\mathcal{Q}$ is monotone and expressible in the language While.

2. (a) $\mathcal{Q}$ can be distributedly computed by a coordination-free, inflationary NrDatalog-transducer.
   (b) $\mathcal{Q}$ can be distributedly computed by an oblivious, inflationary NrDatalog-transducer.
   (c) $\mathcal{Q}$ is expressible in Datalog.

Proof. Regarding *(1)*, for $(c) \Rightarrow (b)$ use Theorem 6.3; for $(b) \Rightarrow (a)$ use Proposition 5.7; for $(a) \Rightarrow (c)$ use Theorems 5.8 and 6.2 to obtain respectively the properties of "$\mathcal{Q}$ is monotone" and "expressible in the language While".

Regarding *(2)*, for $(c) \Rightarrow (b)$ use (proof of if-direction in) Theorem 6.5; for $(b) \Rightarrow (a)$ use Proposition 5.7; for $(a) \Rightarrow (c)$ use Theorem 6.5. □

7 Variation of the Model

In the literature on declarative networking, a seemingly common language feature seems to be that nodes do not simply send each message to all of their neighbors, but instead to a specifically addressed neighbor [28, 23, 10, 29]. We call this the *addressing model*. One could argue that this model lies closer to how real networks operate, and that is why we devote a small section to this model.

7.1 Addressing Transducers

Recall our original epidemic transducer model that was presented in Sections 2.2 and 3. An *addressing transducer* Π over a transducer schema Υ is the same as an epidemic transducer over Υ with the only difference that for a message relation $R^{(k)} \in \Upsilon_{\text{msg}}$, the sending query will produce facts of arity $k+1$ instead of k. The extra component will contain the addressee of each message, which is by convention the first component. Now we look at how the operational semantics must be changed accordingly. With Π as above, consider a transducer network $\mathcal{T} = \langle \mathcal{N}, \Upsilon, \Pi \rangle$. We define how an active node $x \in \mathcal{N}$ does a global transition. Similarly to the original definition of global transition (in Section 3), we let x receive some messages from its message buffer. Then, x does a local transition in which it generates a set of newly sent messages J_{snd}, each having the addressee specified as their first component. Now, the messages that are effectively added to the message buffer of another node y, denoted $K^{\to y}$, is defined as: if y is a neighbor of x then $K^{\to y} = \{R(\bar{a}) \mid R(y, \bar{a}) \in J_{\text{snd}}\}$ and otherwise $K^{\to y} = \emptyset$, i.e., we select precisely the messages that are sent to y when y is a neighbor. An addressee value that is not a neighbor of x will result in the loss of the corresponding message. Note that the message buffers contain facts without an explicit addressee-component, like in the operational semantics for epidemic transducers.

As a special case, if $\mathcal{N}$ forms a complete graph, every node can send a message to every individual other node.

7.2 Properties

First, all our previous results that do not explicitly restrict `Id` or `All` still hold for addressing transducers because, when there is no restriction on using `Id` or `All`, addressing transducers and epidemic transducers are equivalent in terms of what queries they can compute. Indeed, it was already noted in Section 3.4

that the epidemic model can simulate the addressing model by manually adding an addressee-component to every message relation in the transducer schema, and by comparing for each received message the addressee component with the value in the local relation `Id`. The other direction is also possible, namely that an addressing transducer can simulate an epidemic one. It suffices for the addressing transducer to send each message explicitly to *every* neighbor.

Interestingly, a notion of obliviousness can also be defined for addressing transducers. Formally, we say that an addressing transducer is *oblivious* if the relations `Id` and `All` are only used in the message sending queries.[4]

Now, most of our results involving oblivious epidemic transducers also hold for oblivious *addressing* transducers, because of the following reasons. First, the proof techniques frequently use that every node sends out its local input facts, and these are forwarded so that eventually all nodes accumulate all inputs. This can be done with an oblivious addressing transducer as well. Second, these results are mostly about *network-independent* transducers, and a frequently occurring idea in those proofs is that we only focus on the behaviour of a single node: an oblivious epidemic transducer can not distinguish between a single-node network and a multi-node network unless it receives a message, so on a single-node network it should exhibit predictable behaviour if it wants to be network-independent. This trick is also applicable to oblivious addressing transducers, because they too can not distinguish between single-node and multi-node networks unless they receive a message. We now explicitly give the results that are not transferable to addressing transducers, and why this is the case.

First, Proposition 4.2 does not hold for oblivious addressing transducer networks, because this result talks about a concrete transducer network. The transducer may now exploit the number of nodes. In particular, if there are multiple nodes, the transducer may assume messages are eventually delivered. So, it is possible to construct a multi-node transducer network in which the oblivious addressing transducer smuggles node identifiers in the sent messages (by reading `All`), and when these arrive, it is possible to only output the input facts whose active domain is contained in the set of node identifiers. This would prevent the transducer network from computing a generic query.

Although not purely about obliviousness, the result of Theorem 5.11 is also not transferable to addressing transducers, as illustrated by the following example, where relation `All` is used to make the nodes dependent on message arrival.

Example 7.1. We give an addressing transducer that reads only relation `All` and that computes the nonmonotone emptiness query on a nullary input relation R (see also Example 5.4). Reading relation `All` in output or memory queries, a node can know from the start if it is alone or not. If the node is alone, then it can immediately output the desired result by looking at the local relation R. But if there are multiple nodes, every node x sends each local fact $\texttt{All}(y)$ as a

[4]Note, we can not completely forbid the use of node relations because we need to indicate addressees.

message $A(y)$ to node y. Although the operational semantics drops the message when y is not a neighbor of x, because each network is connected, y has at least one neighbor from which it will receive $A(y)$. This way, each node can establish its own identity. Next, the same protocol as in Example 5.4 can be followed. □

8 Conclusion

Encouraged by Hellerstein [24, 25], we have tried in this paper to formalize and prove the CALM Conjecture. We do not claim that our approach is the only one that works. Yet, we believe our approach is natural because it is firmly grounded in previous database theory practice, and delivers solid results. Much further work is possible; we list a few obvious topics:

- Look at Hellerstein's other conjectures (e.g. the CRON conjecture [13]);
- Investigate the expressiveness of variations or extensions of the basic distributed computation model presented here; and,
- Identify special cases where essential semantic notions such as monotonicity, consistency [12], network-independence, coordination-freeness, etc, are decidable.

APPENDIX

A Expressing Queries

A.1 Undecidability

Proposition A.1. Consistency for FO-transducer networks is undecidable.

Proof. We reduce the finite satisfiability problem for FO to deciding consistency for FO-transducer networks. Let φ be an FO-sentence over a database schema $\mathcal{D}$. We construct a transducer network $\mathcal{T}$ that is consistent iff φ is *not* finitely satisfiable.

Consider the transducer Π from Example 4.5, that is over transducer schema Υ. We may assume without loss of generality that the relation names of Υ do not occur in $\mathcal{D}$. We obtain a new transducer schema Υ' from Υ by adding $\mathcal{D}$ to Υ_{in}. We obtain a new transducer Π' from Π by modifying the send rule for relation U to only send facts when φ is satisfied on the local input over $\mathcal{D}$.

Suppose that φ is finitely satisfiable. Consider a network $\mathcal{N}$ with two (connected) nodes x and y. Let $\mathcal{T}$ denote the transducer network with Π on both nodes. Let I be a database instance over $\mathcal{D}$ on which φ is true. Consider the input distributed database instance H with $H(x) = I \cup \{R(1), R(2)\}$ and $H(y) = \emptyset$. Consider the prefix of a fair run of $\mathcal{T}$ on H where first x does a heartbeat transition: because φ is true on I, node x sends messages $U(1)$ and $U(2)$ to y. In the second transition, we can deliver message $U(1)$ to y or message

$U(2)$, or both. Each choice results in a different output at y, which results in a different global output because x never outputs anything (x does not receive messages from y). Hence, the transducer network is not consistent.

For the other direction, suppose that φ is not finitely satisfiable. Then there is no input distributed database instance for $\mathcal{T}$ on which messages will be sent, in which case $\mathcal{T}$ is consistent because on every input, every run produces the empty output. □

Proposition A.2. Network independence for FO-transducers is undecidable.

Proof. We reduce the finite satisfiability problem of FO to deciding network-independence of FO-transducers. Let φ be an FO-sentence over a database schema $\mathcal{D}$. We construct a transducer Π that is network-independent iff φ is *not* finitely satisfiable.

Consider the transducer Π in Example 4.6, that is over schema Υ. We may assume without loss of generality that the relation names of Υ do not occur in $\mathcal{D}$. We obtain a new transducer Π' by modifying Π as follows:

- using the protocol of Lemma 4.8, we let all nodes collect all the input facts of $\mathcal{D}$ available on the network; and,
- the output query is modified so that output can only be produced if the formula φ is satisfied on the (fully) collected instance over $\mathcal{D}$ in memory, in addition to detecting a complete network-topology (as before).

It is possible to construct Π' so that its output, message, and memory relations are not in $\mathcal{D}$. Note that on any network, the transducer network resulting from Π' is consistent: this is because before the output can be produced at a node, it should have obtained the *entire* input over $\mathcal{D}$, and it should have detected that the network topology is complete.

Suppose that φ is finitely satisfiable. Let I be a database instance over $\mathcal{D}$ on which φ is true. Denote $I' = I \cup \{R(1)\}$. Then, on a complete network-topology, the transducer network resulting from Π' outputs $T(1)$ on any horizontal partition of I'. Indeed, the nodes forward all facts of I to each other and in any run there will be a moment when all nodes have these facts *and* have detected that the network-topology is complete. On any other network-topology, for every input partition of I', the resulting transducer network outputs nothing. Therefore Π' is not network-independent.

For the other direction, suppose that φ is not finitely satisfiable. Then every transducer network for Π' computes the empty query and Π' is therefore network-independent. □

A.2 Proof of Lemma 4.8

First, we specify the parts of the transducer-schema Υ that we need: $\Upsilon_{\text{in}} = \mathcal{D}$; $\Upsilon_{\text{msg}} = \{(R^{\text{msg}}, k+1), (R^{\text{ack}}, k+2) \mid R^{(k)} \in \mathcal{D}\} \cup \{\texttt{done}^{(2)}\}$; and,

$$\Upsilon_{\text{mem}} = \{(R^{\text{mem}}, k), (R^{\text{ackMem}}, k+2) \mid R^{(k)} \in \mathcal{D}\} \cup \{\texttt{doneMem}^{(2)}, \texttt{notDone}^{(1)}, \texttt{missing}^{(0)}\} \cup \{\texttt{started}^{(0)}, \texttt{ready}^{(0)}\}.$$

We have not specified the database schema Υ_{out} because this schema is not used in the transducer construction. The idea is that a node x will send its local input facts over relation $R^{(k)} \in \Upsilon_{\text{in}}$ as facts with predicate R^{msg}, with as the last component its own node identifier to indicate the origin of the fact (hence the increased arity of $k+1$). We call this last component the "tag". Next, when a node y receives a tagged R^{msg}-fact, it removes the tag and stores the fact in its relation R^{mem} *and* it sends an R^{ack}-fact to acknowledge the receipt of it. These acknowledgments have the contents of the received R^{msg}-fact (including the tag), with an additional last component containing the identifier of y. The node x checks whether y has (eventually) acknowledged all input facts of x. If yes, then x sends out $\texttt{done}(x, y)$. From the viewpoint of y, if y has received $\texttt{done}(x, y)$ from all other nodes x then it knows that it has accumulated all the input facts on the network.

Some further details are as follows. Let x be a node. For a relation R in Υ_{in}, node x uses the relation R^{ackMem} to store all received acknowledgments for its local input facts over relation R. The relation `notDone` is used by x to remember all the other nodes that have not yet acknowledged all input facts of x. The relation `doneMem` is used by x to remember all `done`-messages having as the second component its own identifier. The relation `missing` is nonempty at x as long as x has not received a `done`-message from all other nodes. The relation `started` helps x to differentiate between its first local transition and all the following local transitions. This makes sure that other memory relations have been correctly initialized before they are read.

Now we specify the transducer Π over Υ. We describe the queries with the language $\text{UCQ}^{\neg}$, which is contained in FO. As usual, unions are then expressed by having multiple rules with the same head. As a general remark about message sending, for each message relation, we always have a "forwarding" rule that just resends all received messages, so that eventually all nodes can receive those messages.

First, for each $R^{(k)} \in \mathcal{D}$ we have the following rules to let all nodes forward their (tagged) input R-facts to each other, and to store the received facts in memory (including acknowledgments):

$$R^{\text{msg}}_{\text{snd}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}) \leftarrow R(\mathtt{u}_1, \ldots, \mathtt{u}_k),\ \mathtt{Id}(\mathtt{x}).$$
$$R^{\text{msg}}_{\text{snd}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}) \leftarrow R^{\text{msg}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}).$$

$$R^{\text{mem}}_{\text{ins}}(\mathtt{u}_1, \ldots, \mathtt{u}_k) \leftarrow R(\mathtt{u}_1, \ldots, \mathtt{u}_k).$$
$$R^{\text{mem}}_{\text{ins}}(\mathtt{u}_1, \ldots, \mathtt{u}_k) \leftarrow R^{\text{msg}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}).$$

$$R^{\text{ack}}_{\text{snd}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}, \mathtt{y}) \leftarrow R^{\text{msg}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}),\ \mathtt{Id}(\mathtt{y}).$$
$$R^{\text{ack}}_{\text{snd}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}, \mathtt{y}) \leftarrow R^{\text{ack}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}, \mathtt{y}).$$

$$R^{\text{ackMem}}_{\text{ins}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}, \mathtt{y}) \leftarrow R^{\text{ack}}(\mathtt{u}_1, \ldots, \mathtt{u}_k, \mathtt{x}, \mathtt{y}),\ \mathtt{Id}(\mathtt{x}).$$

We also specify rules for the other relations in Υ. For convenience, let us denote $\mathcal{D} = \{R_1^{(k_1)}, R_2^{(k_2)}, \ldots, R_n^{(k_n)}\}$. First, we need to remember that the first local transition has already happened, using the following rule:

$$\mathtt{started}_{\text{ins}}(\,) \leftarrow .$$

Next, on each node x, the relation `notDone` contains all nodes that have not yet acknowledged the receipt of all local input facts of x. A node does not have to acknowledge its own input facts, so `notDone` will not contain x itself. This relation is recomputed during every local transition, using the following rules:

$$\begin{aligned}
\mathtt{notDone}_{\text{ins}}(\mathtt{y}) \leftarrow & R_1(\mathtt{u}_1, \ldots, \mathtt{u}_{k_1}),\ \mathtt{Id}(\mathtt{x}),\ \mathtt{All}(\mathtt{y}),\ \neg\mathtt{Id}(\mathtt{y}),\\
& \neg R_1^{\text{ackMem}}(\mathtt{u}_1, \ldots, \mathtt{u}_{k_1}, \mathtt{x}, \mathtt{y}).\\
\vdots\\
\mathtt{notDone}_{\text{ins}}(\mathtt{y}) \leftarrow & R_n(\mathtt{u}_1, \ldots, \mathtt{u}_{k_n}),\ \mathtt{Id}(\mathtt{x}),\ \mathtt{All}(\mathtt{y}),\ \neg\mathtt{Id}(\mathtt{y}),\\
& \neg R_n^{\text{ackMem}}(\mathtt{u}_1, \ldots, \mathtt{u}_{k_n}, \mathtt{x}, \mathtt{y}).
\end{aligned}$$

$$\mathtt{notDone}_{\text{del}}(\mathtt{y}) \leftarrow \mathtt{notDone}(\mathtt{y}).$$

Note that if a node was initialized with no local input tuples over some relation $R_i^{(k_i)} \in \mathcal{D}$, then the corresponding insertion rule for `notDone` will not fire. In that case, the node will not consider any other nodes "responsible" for acknowledging the receipt of its input facts over relation R_i. This is the desired behavior. Also, the deletion rule for `notDone` allows for the recomputation of `notDone` during every local step: only the nodes that have not confirmed every local input fact are reinserted again. Thus, after a while, relation `notDone` will become (and remain) empty.

When a node x notices that another node y has acknowledged all local input facts of x, node x sends out $\mathtt{done}(x, y)$. This is accomplished by the following rules:

$$\mathtt{done}_{\text{snd}}(\mathtt{x}, \mathtt{y}) \leftarrow \mathtt{started}(\,),\ \mathtt{Id}(\mathtt{x}),\ \mathtt{All}(\mathtt{y}),\ \neg\mathtt{Id}(\mathtt{y}),\ \neg\mathtt{notDone}(\mathtt{y}).$$
$$\mathtt{done}_{\text{snd}}(\mathtt{x}, \mathtt{y}) \leftarrow \mathtt{done}(\mathtt{x}, \mathtt{y}).$$

These done-messages are stored at the addressed node:

$$\texttt{doneMem}_{\text{ins}}(\texttt{x},\texttt{y}) \leftarrow \texttt{done}(\texttt{x},\texttt{y}),\ \texttt{Id}(\texttt{y}).$$

Finally, when a node y has received done from all other nodes , it can output the ready flag. This is accomplished by the following rules:

$$\texttt{missing}_{\text{ins}}(\,) \leftarrow \texttt{Id}(\texttt{y}),\ \texttt{All}(\texttt{z}),\ \neg\texttt{Id}(\texttt{z}),\ \neg\texttt{doneMem}(\texttt{z},\texttt{y}).$$

$$\texttt{missing}_{\text{del}}(\,) \leftarrow \texttt{missing}(\,).$$

$$\texttt{ready}_{\text{ins}}(\,) \leftarrow \texttt{started}(\,),\ \neg\texttt{missing}(\,).$$

Note that in a single-node network, the missing-fact is never created. In that case, the ready-fact is already produced in the second transition (thus after started( ) is created).

The transducer Π above can actually be made inflationary as well. In particular, using the equivalence FO = NrDatalog$^\neg$, we can write a NrDatalog$^\neg$-transducer where relations notDone and missing do not appear in the memory schema Υ_{mem} but are computed locally: one would locally compute notDone in the sending query for the done-relation, and one would locally compute missing in the insertion query for the ready-relation.

B Expressiveness Analysis

B.1 Proof of Lemma 6.1 (While to FO-transducer)

We show how to simulate a While-program on a single-node FO-transducer network. A While-program can be simulated by iterated heartbeats using well-known techniques [4]. Because this is not entirely obvious, we will illustrate the technique. Consider the simple While-program in Algorithm 1 over input schema $\mathcal{D} = \{R^{(2)}, S^{(1)}\}$ and output schema $\mathcal{D}' = \{T^{(2)}\}$. Intuitively, if R represents a graph then the While program collects all edges that are reachable from the nodes in relation S. By introducing a temporary relation U, we can rewrite this program so that T is only modified at the very end. See Algorithm 2. Next, any While-program can be translated to a list of statements in which explicit control flow is represented by conditional and unconditional "goto" statements. This also works for nested while-loops. When translating Algorithm 2 to this form, we obtain Algorithm 3. There, V holds the result of the expression that is tested for non-emptiness by the while-loop condition. Let us refer to this form as a "list program".

We will now simulate Algorithm 3 with an FO-transducer. First, we define a transducer schema Υ with $\Upsilon_{\text{in}} = \mathcal{D}$, $\Upsilon_{\text{out}} = \mathcal{D}'$, $\Upsilon_{\text{msg}} = \emptyset$ and

$$\Upsilon_{\text{mem}} = \{(U,2),\ (U^{\text{prev}},2),\ (\texttt{step}^1,0),\ \ldots,\ (\texttt{step}^8,0)\}.$$

Algorithm 1 A While-program.

$T := \{T(\mathtt{u},\mathtt{v}) \mid R(\mathtt{u},\mathtt{v}) \wedge S(\mathtt{u})\};$
$T^{\mathrm{prev}} := \emptyset;$
while $(T \setminus T^{\mathrm{prev}} \neq \emptyset)$ do
    $T^{\mathrm{prev}} := T;$
    $T := T \cup \big\{T(\mathtt{v},\mathtt{w}) \mid \exists \mathtt{u}\big(T(\mathtt{u},\mathtt{v}) \wedge R(\mathtt{v},\mathtt{w})\big)\big\};$
end

Algorithm 2 Rewritten version of Algorithm 1.

$U := \{U(\mathtt{u},\mathtt{v}) \mid R(\mathtt{u},\mathtt{v}) \wedge S(\mathtt{u})\};$
$U^{\mathrm{prev}} := \emptyset;$
while $(U \setminus U^{\mathrm{prev}} \neq \emptyset)$ do
    $U^{\mathrm{prev}} := U;$
    $U := U \cup \big\{U(\mathtt{v},\mathtt{w}) \mid \exists \mathtt{u}\big(U(\mathtt{u},\mathtt{v}) \wedge R(\mathtt{v},\mathtt{w})\big)\big\};$
end
$T := U;$

Algorithm 3 List program for Algorithm 2.

$\mathtt{step}^1$: $U := \{U(\mathtt{u},\mathtt{v}) \mid R(\mathtt{u},\mathtt{v}) \wedge S(\mathtt{u})\};$
$\mathtt{step}^2$: $U^{\mathrm{prev}} := \emptyset;$
$\mathtt{step}^3$: $V := \{V(\,) \mid U(\mathtt{u},\mathtt{v}) \wedge \neg U^{\mathrm{prev}}(\mathtt{u},\mathtt{v})\};$
$\mathtt{step}^4$: **if** $\langle V = \emptyset \rangle$ **goto** $\langle \mathtt{step}^8 \rangle;$
$\mathtt{step}^5$: $U^{\mathrm{prev}} := U;$
$\mathtt{step}^6$: $U := U \cup \big\{U(\mathtt{v},\mathtt{w}) \mid \exists \mathtt{u}\big(U(\mathtt{u},\mathtt{v}) \wedge R(\mathtt{v},\mathtt{w})\big)\big\};$
$\mathtt{step}^7$: **goto** $\langle \mathtt{step}^3 \rangle;$
$\mathtt{step}^8$: $T := U;$

Relations $\texttt{step}^1$ to $\texttt{step}^8$ model the program counter of the list program. The idea is that at any moment in time at most one of these relations is active (nonempty) and that they activate each other in the correct way in order to represent the desired control flow. We now specify a transducer Π over Υ that simulates Algorithm 3 on a single-node network. As a general remark, some $\texttt{step}^i$-relations with $i \in \{1, \ldots, 8\}$ are read inside the queries that update relations U and U^{prev}, to make sure that U and U^{prev} are updated only at the moment when the original list program updates them. We will also specify the deletion queries for U and U^{prev} because assignment in a While-program is destructive in the sense that previous facts can only stay in the relation if they are on the right hand side of the assignment. The queries are as follows:

$$\texttt{step}^1_{\text{ins}}() \leftarrow \neg\texttt{step}^1(), \neg\texttt{step}^2(), \ldots, \neg\texttt{step}^8().$$

$$U_{\text{ins}}(\texttt{u}, \texttt{v}) \leftarrow \texttt{step}^1(), R(\texttt{u}, \texttt{v}), S(\texttt{u}).$$
$$U_{\text{ins}}(\texttt{u}, \texttt{v}) \leftarrow \texttt{step}^6(), U(\texttt{u}, \texttt{v}).$$
$$U_{\text{ins}}(\texttt{v}, \texttt{w}) \leftarrow \texttt{step}^6(), U(\texttt{u}, \texttt{v}), R(\texttt{v}, \texttt{w}).$$

$$U_{\text{del}}(\texttt{u}, \texttt{v}) \leftarrow \texttt{step}^6(), U(\texttt{u}, \texttt{v}).$$

$$\texttt{step}^2_{\text{ins}}() \leftarrow \texttt{step}^1().$$

$$\texttt{step}^3_{\text{ins}}() \leftarrow \texttt{step}^2().$$
$$\texttt{step}^3_{\text{ins}}() \leftarrow \texttt{step}^7().$$

$$V_{\text{ins}}() \leftarrow \texttt{step}^3(), U(\texttt{u}, \texttt{v}), \neg U^{\text{prev}}(\texttt{u}, \texttt{v}).$$

$$V_{\text{del}}() \leftarrow \texttt{step}^4().$$

$$\texttt{step}^4_{\text{ins}}() \leftarrow \texttt{step}^3().$$

$$\texttt{step}^5_{\text{ins}}() \leftarrow \texttt{step}^4(), V().$$

$$U^{\text{prev}}_{\text{ins}}(\texttt{u}, \texttt{v}) \leftarrow \texttt{step}^5(), U(\texttt{u}, \texttt{v}).$$

$$U^{\text{prev}}_{\text{del}}(\texttt{u}, \texttt{v}) \leftarrow \texttt{step}^5(), U^{\text{prev}}(\texttt{u}, \texttt{v}).$$

$$\texttt{step}^6_{\text{ins}}() \leftarrow \texttt{step}^5().$$

$$\texttt{step}^7_{\text{ins}}() \leftarrow \texttt{step}^6().$$

$$\texttt{step}^8_{\text{ins}}() \leftarrow \texttt{step}^4(), \neg V().$$

$T_{\text{out}}(\mathtt{u},\mathtt{v}) \leftarrow \mathtt{step}^8(\,), U(\mathtt{u},\mathtt{v})$.

For $i \in \{1,\ldots,7\}$: $\mathtt{step}^i_{\text{del}}(\,) \leftarrow \mathtt{step}^i(\,)$.

We never delete $\mathtt{step}^8(\,)$ because we do not want to accidentally restart the simulation of the While program: indeed, when $\mathtt{step}^i(\,)$ for each $i \in \{1,\ldots,8\}$ is missing, a new $\mathtt{step}^1(\,)$ fact is created. Note that the queries for inserting into relations $\mathtt{step}^5$ and $\mathtt{step}^8$ together simulate the if-goto statement at line 4 of Algorithm 3.

B.2 Proof of Lemma 6.1 (FO-transducer to While)

Let $\mathcal{T}$ be a single-node transducer network, running an oblivious FO-transducer Π that does not read message relations. Let $\mathcal{Q}$ denote the query computed by $\mathcal{T}$. We describe a While-program P to compute $\mathcal{Q}$. Intuitively, P consists of one big loop that during one iteration performs the same state changes as Π during one heartbeat transition. Also, in order to terminate, the While-program must detect repetition of transducer states.

To illustrate, consider the transducer given in Algorithm 4, that computes the query of Section B.1 intentionally in a more complex way. Specifically, the memory relation A continuously alternates between being empty and nonempty. The insertion queries for relations U and T only produce a nonempty output when A is nonempty.

For two sets S_1 and S_2, let $\mathit{diff}(S_1, S_2)$ abbreviate the expression $\big((S_1 \setminus S_2) \cup (S_2 \setminus S_1)\big)$. Consider now the While-program P in Algorithm 5 to explicitly simulate the transducer of Algorithm 4. The program P keeps simulating the updates to the relations A, U and T until no more changes occur to all of them. Surely, if the transducer state stops changing, no more output facts can be produced because only heartbeat transitions can occur and because local transitions are deterministic. However, because of the alternating behavior of relation A, the program P never stops if it uses only this test. For the transducer itself, the alternating behavior of relation A is no problem because its output is defined on infinite runs anyway. But program P needs to halt because otherwise its output is undefined. Using the technique of Abiteboul and Simon [3], however, P can be modified to detect that it is in an infinite loop. This implies that the transducer has repeated a state and will output no new output facts. After detecting the infinite loop, the program P then breaks the loop and the final contents of relation T is the output.

B.3 Proof of Lemma 6.1 (eliminate `Id`)

Let $\mathcal{T}$ be a single-node transducer network, with FO-transducer Π over transducer schema Υ. Let x denote the single node. Let $\mathcal{Q}$ denote the query computed by $\mathcal{T}$. We assume that Π does not read relation `All` and does not read message

Algorithm 4 A simple FO-transducer.

Schema: $\Upsilon_{\text{in}} = \{R^{(2)}, S^{(1)}\}$; $\Upsilon_{\text{out}} = \{T^{(2)}\}$; $\Upsilon_{\text{msg}} = \emptyset$; $\Upsilon_{\text{mem}} = \{A^{(0)}, U^{(2)}\}$.
Queries:

$$A_{\text{ins}}() \leftarrow \neg A().$$

$$A_{\text{del}}() \leftarrow A().$$

$$U_{\text{ins}}(\mathtt{v}, \mathtt{w}) \leftarrow A(), T(\mathtt{u}, \mathtt{v}), R(\mathtt{v}, \mathtt{w}).$$

$$U_{\text{del}}(\mathtt{u}, \mathtt{v}) \leftarrow A(), U(\mathtt{u}, \mathtt{v}).$$

$$T_{\text{out}}(\mathtt{u}, \mathtt{v}) \leftarrow A(), R(\mathtt{u}, \mathtt{v}), S(\mathtt{u}).$$
$$T_{\text{out}}(\mathtt{u}, \mathtt{v}) \leftarrow A(), U(\mathtt{u}, \mathtt{v}).$$

relations. Here we will show how to rewrite Π to eliminate the use of relation `Id` as well. We thus obtain an oblivious transducer.

B.3.1 Definitions and Notations

Van den Bussche and Cabibbo [31] have shown how to convert an *untyped* FO-formula to a *typed* one that computes the same query (over a typed relation schema). We will use that result here. We distinguish between two *sorts* of values:

- values in the active domain of an input over Υ_{in}; and,
- the single node x of $\mathcal{T}$.

For these sorts we use the symbols *adom* and *id* respectively. The technique of [31] requires that each sort has a completely separated domain of values. Hence, we will assume that x does not occur in the *adom* values. We will see later that this assumption has no undesired consequences.

The definitions below are specifically tailored for the two sorts above and therefore less general as in the paper [31]. A *type* τ is a tuple of sort symbols. An example is $(adom, id, adom, adom)$. A *k-type* is a type with arity k.

A *typed database schema* $\mathcal{F}$ is a finite set of pairs (R, τ) with R a relation name and τ the associated type of R, such that no relation name occurs twice. This corresponds to an ordinary database schema *untyped*($\mathcal{F}$) that consists of

- a relation $R^{(k)}$ for each $(R, \tau) \in \mathcal{F}$ with k the arity of τ; and,
- the relations $\mathtt{Id}^{(1)}$ and $\mathtt{Adom}^{(1)}$ (assumed not to be in $\mathcal{F}$ already).

We define a *typed* database instance I over $\mathcal{F}$ as a normal database instance over *untyped*($\mathcal{F}$) such that

Algorithm 5 While-program based on the transducer of Algorithm 4.

[All auxiliary relations start empty]

while $\big(\neg\texttt{started}() \vee (\mathit{diff}(A, A^{\text{prev}}) \cup \mathit{diff}(U, U^{\text{prev}}) \cup \mathit{diff}(T, T^{\text{prev}}) \neq \emptyset)\big)$ do

$\texttt{started} := \{\texttt{started}()\}$;

$A^{\text{prev}} := A$;

$U^{\text{prev}} := U$;

$T^{\text{prev}} := T$;

$A_{\text{ins}} := \{A() \mid \neg A^{\text{prev}}()\}$;

$A_{\text{del}} := \{A() \mid A^{\text{prev}}()\}$;

$U_{\text{ins}} := \big\{U(\texttt{v},\texttt{w}) \mid \exists\texttt{u}\big(A^{\text{prev}}() \wedge T^{\text{prev}}(\texttt{u},\texttt{v}) \wedge R(\texttt{v},\texttt{w})\big)\big\}$;

$U_{\text{del}} := \{U(\texttt{u},\texttt{v}) \mid A^{\text{prev}}() \wedge U^{\text{prev}}(\texttt{u},\texttt{v})\}$;

[We explicitly simulate the updating of memory relations]

$A := \big(A^{\text{prev}} \cup (A_{\text{ins}} \setminus A_{\text{del}})\big) \setminus (A_{\text{del}} \setminus A_{\text{ins}})$;

$U := \big(U^{\text{prev}} \cup (U_{\text{ins}} \setminus U_{\text{del}})\big) \setminus (U_{\text{del}} \setminus U_{\text{ins}})$;

$T := \ T \cup \{T(\texttt{u},\texttt{v}) \mid A^{\text{prev}}() \wedge R(\texttt{u},\texttt{v}) \wedge S(\texttt{u})\}$
$\qquad \cup \{T(\texttt{u},\texttt{v}) \mid A^{\text{prev}}() \wedge U^{\text{prev}}(\texttt{u},\texttt{v})\}$;

end

- $I|_{\texttt{Id}} = \{\texttt{Id}(x)\}$,
- $I|_{\texttt{Adom}} = \{\texttt{Adom}(a) \mid a \in \mathit{adom}(I),\ a \neq x\}$,
- for each fact $R(a_1, \ldots, a_k) \in I$, where R is not `Id` or `Adom`, and $\tau = (s_1, \ldots, s_k)$ is the type of R in $\mathcal{F}$, we require for each $i \in \{1, \ldots, k\}$ that $a_i \neq x$ if $s_i = \mathit{adom}$ and $a_i = x$ if $s_i = \mathit{id}$. We say that this fact *has* type τ.

Also, we will specify the queries of FO-transducers with *relational calculus* to make a better connection with the previous work [31].[5] We assume that the active domain semantics is used to evaluate these queries [2].

B.3.2 Split tuples by type

We now construct an intermediate transducer Π^2 to compute on the single-node network x the same query $\mathcal{Q}$ as Π, when we restrict attention to input instances

[5] For easier technical presentation, we assume that relational calculus queries produce tuples instead of facts. These tuples can be easily turned into facts by prepending the correct predicate.

I over Υ_{in} with $x \notin adom(I)$. We define the schema Υ^2 of Π^2 as follows. First, $\Upsilon^2_{\text{in}} = \Upsilon_{\text{in}}$; $\Upsilon^2_{\text{out}} = \Upsilon_{\text{out}}$; $\Upsilon^2_{\text{msg}} = \emptyset$; and Υ^2_{mem} consists of

- relation $\texttt{Adom}^{(1)}$;
- relation $\texttt{started}^{(0)}$; and,
- the relations ${R_\tau}^{(k)}$ for each $R^{(k)} \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$ and *each* k-type τ.

Concerning Υ^2_{mem}, the idea is that in Π^2 the facts over relation $R^{(k)} \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$ of Π are represented by all the disjoint relations $\{R^{(k)}_\tau \mid \tau \text{ is a } k\text{-type}\}$.

We now describe the FO-queries of Π^2. To start, in the first transition, we initialize relation $\texttt{Adom}$ to contain the active domain of the input over Υ_{in}, and we also compute the fact $\texttt{started}()$ so that other queries can know that relation $\texttt{Adom}$ has been initialized. We omit the details of these relatively simple queries.

Next, we define the queries for the other memory relations of Π^2. Consider $R^{(k)}_\tau \in \Upsilon^2_{\text{mem}}$, with $R^{(k)} \in \Upsilon_{\text{mem}}$. Let the insertion query for R in Π be the following relational calculus query:

$$\{(\mathtt{u}_1, \ldots, \mathtt{u}_k) \mid \varphi(\mathtt{u}_1, \ldots, \mathtt{u}_k)\}.$$

The FO-formula φ is over $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}} \cup \{\texttt{Id}^{(1)}\}$. We write φ^{split} to denote the modification of φ that is obtained by replacing for each $R^{(k)} \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$, each occurrence of an atomic subformula $R(\mathtt{u}_1, \ldots, \mathtt{u}_k)$ by the non-atomic formula $\big(\bigvee_{\tau \in \alpha} R_\tau(\mathtt{u}_1, \ldots, \mathtt{u}_k)\big)$ where α is the set of all k-types. Now, denoting $\tau = (s_1, \ldots, s_k)$, we define the insertion query for relation R_τ in Π^2 to be the following relational calculus query:

$$\{(\mathtt{u}_1, \ldots, \mathtt{u}_k) \mid \varphi^{\text{split}}(\mathtt{u}_1, \ldots, \mathtt{u}_k) \wedge S_1(\mathtt{u}_1) \wedge \ldots \wedge S_k(\mathtt{u}_k) \wedge \texttt{started}()\}$$

where for each $i \in \{1, \ldots, k\}$ we define $S_i = \texttt{Adom}$ if $s_i = adom$ and $S_i = \texttt{Id}$ if $s_i = id$. The deletion query for R_τ in Π^2 can be defined in a similar way, based on the deletion query of R in Π. Again, this is similar for a relation ${R_\tau}^{(k)}$ with $R^{(k)} \in \Upsilon_{\text{out}}$, but with the difference that the deletion query in Π^2 will always return empty (because output only accumulates).

Finally, for $R^{(k)} \in \Upsilon^2_{\text{out}} = \Upsilon_{\text{out}}$ we define the output query of R in Π^2 to copy the contents of memory relation R_τ where τ is the k-type $(adom, \ldots, adom)$. This way, the value x can not be output on input instances that do not contain x in their active domain (see our earlier assumption). This completes the description of transducer Π^2.

As for notation, for a fact $\boldsymbol{f} = R_\tau(a_1, \ldots, a_k)$ we write $\widehat{\boldsymbol{f}}$ to denote $R(a_1, \ldots, a_k)$, i.e., to drop the type subscript. The following lemma can now be shown by induction on the structure of FO-formulas.

Lemma B.1. Let I be an input instance over $\Upsilon_{\text{in}} = \Upsilon^2_{\text{in}}$ such that $x \notin adom(I)$. On input I, on the single-node network x, let J_1 and J_2 be transducer states

for Π and Π^2 respectively. This implies $\mathtt{Id}(x) \in J_1|_{(\text{sys})}$ and $\mathtt{Id}(x) \in J_2|_{(\text{sys})}$. Suppose for each $R^{(k)} \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$ that

$$J_1|_R = \bigcup_{\tau \in \alpha} \{\widehat{\boldsymbol{f}} \mid \boldsymbol{f} \in J_2|_{R_\tau}\}$$

where α is the set of all k-types. In words: the relation R in J_1 is represented exactly by the split R-relations in J_2.

Consider some relational calculus query over input schema $\Upsilon_{\text{in}} \cup \Upsilon_{\text{out}} \cup \Upsilon_{\text{mem}} \cup \{\mathtt{Id}^{(1)}\}$:

$$\mathcal{C}_1 = \{(\mathtt{u}_1, \ldots, \mathtt{u}_k) \mid \varphi(\mathtt{u}_1, \ldots, \mathtt{u}_k)\}.$$

Consider the modified relational calculus query:

$$\mathcal{C}_2 = \{(\mathtt{u}_1, \ldots, \mathtt{u}_k) \mid \varphi^{\text{split}}(\mathtt{u}_1, \ldots, \mathtt{u}_k)\}.$$

We have $\mathcal{C}(J_1) = \mathcal{C}_2(J_2)$. □

Let $\mathcal{T}^2$ denote the transducer network where we run Π^2 on x. Now we have the following property:

Lemma B.2. *$\mathcal{T}^2$ computes the query $\mathcal{Q}$ when restricted to inputs not containing x.*

Proof. Let I be a database instance over Υ_{in} with $x \notin \mathit{adom}(I)$. Let $\mathcal{R}$ and $\mathcal{R}^2$ denote the unique runs of $\mathcal{T}$ and $\mathcal{T}^2$ on input I respectively.[6] Let ρ_1, ρ_2, …, and ρ'_1, ρ'_2, …, denote the sequences of configurations of $\mathcal{R}$ and $\mathcal{R}^2$ respectively. For $i \geq 1$ we denote $\rho_i = (s_i, b_i)$ and $\rho'_i = (s'_i, b'_i)$. Using Lemma B.1, it can be shown by induction on $i \geq 1$ that for each relation $R^{(k)} \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$ we have

$$s_i(x)|_R = \bigcup_{\tau \in \alpha} \{\widehat{\boldsymbol{f}} \mid \boldsymbol{f} \in s'_{i+1}(x)|_{R_\tau}\}.$$

In words: although in Π^2 the tuples are divided by their type, they still represent exactly the same tuples as in Π. In this expression, the configuration-index for run $\mathcal{R}^2$ is offset by 1 because the queries of Π^2 first have to wait until relation `started` becomes nonempty.

We are left to show that Π^2 outputs $\mathcal{Q}(I)$. Let $R^{(k)} \in \Upsilon_{\text{out}}$. Let $\boldsymbol{f} = R(a_1, \ldots, a_k)$ be an output fact produced in run $\mathcal{R}$. Because by assumption $x \notin \mathit{adom}(I)$ and because the query $\mathcal{Q}$ is generic, we must have that $\boldsymbol{f}$ has k-type $\tau = (\mathit{adom}, \ldots, \mathit{adom})$. Using the above property, the memory fact $R_\tau(a_1, \ldots, a_k)$ is produced in run $\mathcal{R}^2$ (and is never deleted). By specification of the output queries in Π^2, the fact $R(a_1, \ldots, a_k)$ is output in $\mathcal{R}^2$ as well. The other direction is similar. □

[6] The runs are unique because there are only heartbeat transitions.

B.3.3 Well-typed formulas

Consider the *typed* database schema $\mathcal{E}$ that consists of:

- the relation (R, τ) for each $R^{(k)} \in \Upsilon_{\text{in}}$ where τ is k-type $(adom, \ldots, adom)$; and,
- the relation (R_τ, τ) for each $R^{(k)} \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$ and *each* k-type τ.

Let φ be an FO-formula used in a query of Π^2. Formula φ is over the schema $\Upsilon^2_{\text{in}} \cup \Upsilon^2_{\text{mem}} \cup \{\texttt{Id}^{(1)}\}$. When we would ignore the (simple) usage of relation `started` in φ, formula φ is over the schema *untyped*$(\mathcal{E})$. Now we can apply Proposition 1 of [31] to φ, to obtain φ^{well}, which is a *well-typed* formula over $\mathcal{E}$. Formula φ^{well} computes the same query as φ when applied to a typed database instance over $\mathcal{E}$, but importantly, φ^{well} does not read the relations `Adom` and `Id` directly. Instead, it has variables with the sort *adom* or *id* that range over the active domain of the input instance and $\{x\}$, respectively. Since φ^{well} does not read relation `Adom` anymore, we can also safely remove the occurrence of relation `started` from it.

Because formula φ^{well} uses two sorts of variables, it is not directly usable for a (normal) FO-transducer. We now explain how to remove the *id*-variables from φ^{well} so that only the *adom*-variables remain, giving us again a (normal) formula with a single sort of variable. So, let φ be an FO-formula used for a memory relation in Π^2, either for insertion or deletion. Abbreviate $\psi = \varphi^{\text{well}}$. Let us also define the following sentences: *true* $:= \forall \mathtt{u}(\mathtt{u} = \mathtt{u})$ and *false* $:= \exists \mathtt{u}(\mathtt{u} \neq \mathtt{u})$, where $\mathtt{u}$ is an *adom*-variable not yet occurring in ψ. By structural induction, we now convert ψ to a normal FO-formula $\psi^{\downarrow}$, by keeping only the *adom* variables as follows:

- Suppose ψ is $(\mathtt{u} = \mathtt{v})$ with $\mathtt{u}$ an *adom*-variable and $\mathtt{v}$ an *id*-variable. We define $\psi^{\downarrow}$ as $(\mathtt{u} = \mathtt{u}) \wedge$ *false*, because *adom*- and *id*-variables can never point to the same value (using our assumption that inputs do not contain value x).
- Suppose ψ is $(\mathtt{v}_1 = \mathtt{v}_2)$ with $\mathtt{v}_1$ and $\mathtt{v}_2$ being *id*-variables. We define $\psi^{\downarrow}$ as *true*, because *id*-variables can only point to the same value x.
- Suppose ψ is $(\mathtt{u}_1 = \mathtt{u}_2)$ with $\mathtt{u}_1$ and $\mathtt{u}_2$ being *adom*-variables. We define $\psi^{\downarrow}$ as $(\mathtt{u}_1 = \mathtt{u}_2)$.
- Suppose ψ is $R(\mathtt{w}_1, \ldots, \mathtt{w}_k)$ with $R^{(k)} \in \Upsilon_{\text{in}}$. Then R has type $(adom, \ldots, adom)$ in $\mathcal{E}$. Hence, each variable $\mathtt{w}_i$ is an *adom*-variable. We define $\psi^{\downarrow}$ as $R(\mathtt{w}_1, \ldots, \mathtt{w}_k)$.
- Suppose ψ is $R_\tau(\mathtt{w}_1, \ldots, \mathtt{w}_k)$ with $R^{(k)}_\tau \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}$. Denote $\tau = (s_1, \ldots, s_k)$. Let $\mathtt{u}_1, \ldots, \mathtt{u}_n$ be the *adom*-variables of $\mathtt{w}_1, \ldots, \mathtt{w}_k$ in order. By construction of ψ, for each $i \in \{1, \ldots, k\}$, the sort of $\mathtt{w}_i$ is s_i. We define $\psi^{\downarrow}$ as $R_\tau(\mathtt{u}_1, \ldots, \mathtt{u}_n)$.

- Suppose ψ is $\psi_1 \vee \psi_2$. Let ${\psi_1}^{\downarrow}$ and ${\psi_2}^{\downarrow}$ denote the conversions of ψ_1 and ψ_2 respectively. We define $\psi^{\downarrow}$ as ${\psi_1}^{\downarrow} \vee {\psi_2}^{\downarrow}$.
- Suppose ψ is $\exists \mathtt{w}(\psi_1)$. If $\mathtt{w}$ is an *adom*-variable then we define $\psi^{\downarrow}$ as $\exists \mathtt{w}({\psi_1}^{\downarrow})$ and otherwise we define $\psi^{\downarrow}$ as ${\psi_1}^{\downarrow}$.
- Suppose $\psi = \neg\psi_1$. We define $\psi^{\downarrow}$ as $\neg({\psi_1}^{\downarrow})$.

In $\psi^{\downarrow}$, there are no *id*-variables and all *adom*-variables have been preserved. In conclusion, to remove relation `Id` from an FO-formula φ of Π^2, we use the transformation $(\varphi^{\text{well}})^{\downarrow}$. We will use this below.

B.3.4 Construct new transducer

We construct a third and last FO-transducer Π^3 that computes the query $\mathcal{Q}$.

For a type τ, we write $\#\tau$ to denote the number of *adom*-components. We now define the schema Υ^3 of Π^3 as

- $\Upsilon^3_{\text{in}} = \Upsilon_{\text{in}}$; $\Upsilon^3_{\text{out}} = \Upsilon_{\text{out}}$; $\Upsilon^3_{\text{msg}} = \emptyset$; and,
- $\Upsilon^3_{\text{mem}} = \{R_\tau^{(\#\tau)} \mid R^{(k)} \in \Upsilon_{\text{mem}} \cup \Upsilon_{\text{out}}, \tau \text{ is a } k\text{-type}\}$.

Note that Υ^3 is very similar to schema Υ^2, with the difference that *(i)* the memory relations in Υ^3 in general have a lower arity to store just the *adom*-values, and *(ii)* relation `started` is omitted because the relation `Adom` does not need to be computed anymore.

Now we define the queries of Π^3. First, let ${R_\tau}^{(l)} \in \Upsilon^3_{\text{mem}}$. By definition, $l = \#\tau$. Let k be the arity of τ. We have ${R_\tau}^{(k)} \in \Upsilon^2_{\text{mem}}$. Let the insertion query for ${R_\tau}^{(k)}$ in Π^2 be

$$\{(\mathtt{w_1}, \ldots, \mathtt{w_k}) \mid \varphi(\mathtt{w_1}, \ldots, \mathtt{w_k})\}.$$

We define the insertion query for ${R_\tau}^{(l)}$ in Π^3 to be:

$$\{(\mathtt{u_1}, \ldots, \mathtt{u_n}) \mid (\varphi^{\text{well}})^{\downarrow}(\mathtt{u_1}, \ldots, \mathtt{u_n})\}$$

where $\mathtt{u_1}, \ldots, \mathtt{u_n}$ are the *adom*-variables of $\mathtt{w_1}, \ldots, \mathtt{w_k}$ (in order). The deletion query for ${R_\tau}^{(l)}$ in Π^3 is defined similarly. For $R^{(k)} \in \Upsilon^3_{\text{out}}$, we define the output query in Π^3 as the one that copies the memory relation R_τ to R with τ the k-type $(adom, \ldots, adom)$.

Before we can look at the properties of Π^3, we need some additional notation. Consider $R_\tau^{(l)} \in \Upsilon^3_{\text{mem}}$. Let k be the arity of τ. We have $R_\tau^{(k)} \in \Upsilon^2_{\text{mem}}$. Let $A \subseteq \{1, \ldots, k\}$ be the component indices of τ corresponding to an *adom*-variable. Let $f : A \to \mathbb{N}$ be the strictly increasing function that maps A to contiguous integers starting at 1. For example, if $A = \{3, 6, 7\}$ then $f = \{3 \mapsto 1,\ 6 \mapsto 2,\ 7 \mapsto 3\}$. Let $\bar{t}$ be a tuple $(a_1, \ldots, a_l)$. We write $\bar{t}^{\uparrow(\tau)}$ to denote the tuple $(b_1, \ldots, b_k)$ where for each $i \in \{1, \ldots, k\}$ we have $b_i = a_{f(i)}$ if $i \in A$ and $b_i = x$ otherwise.

Intuitively, we insert the single *id*-value x back to obtain a k-tuple. We use this notation for facts as well.

Let $\mathcal{T}^3$ denote the transducer network obtained by putting Π^3 at node x. The following lemma can again be shown by structural induction on the (well-typed) FO-formulas.

Lemma B.3. Let I be a database instance over $\Upsilon_{\text{in}} = \Upsilon^2_{\text{in}} = \Upsilon^3_{\text{in}}$ with $x \notin adom(I)$. On input I, on the single-node network x, let J_2 and J_3 be transducer states for Π^2 and Π^3 respectively. Suppose for each ${R_\tau}^{(k)} \in \Upsilon^2_{\text{mem}}$ that

$$J_2|_{R_\tau} = \{\boldsymbol{f}^{\uparrow(\tau)} \mid \boldsymbol{f} \in J_3|_{R_\tau}\}.$$

Let σ be a type. Consider some relational calculus query over input schema $\Upsilon_{\text{in}} \cup \Upsilon^2_{\text{mem}} \cup \{\mathtt{Id}^{(1)}\}$ that produces only tuples of type σ:

$$\mathcal{C}_2 = \{(\mathtt{w}_1, \ldots, \mathtt{w}_k) \mid \varphi(\mathtt{w}_1, \ldots, \mathtt{w}_k)\}.$$

Consider the following transformed query, which is over input schema $\Upsilon_{\text{in}} \cup \Upsilon^3_{\text{mem}}$:

$$\mathcal{C}_3 = \{(\mathtt{u}_1, \ldots, \mathtt{u}_n) \mid (\varphi^{\text{well}})^{\downarrow}(\mathtt{u}_1, \ldots, \mathtt{u}_n)\}$$

where $\mathtt{u}_1, \ldots, \mathtt{u}_n$ are the free *adom*-variables of $(\varphi^{\text{well}})^{\downarrow}$, in the same relative order as they occur in $\mathtt{w}_1, \ldots, \mathtt{w}_k$. We have $\mathcal{C}_2(J_2) = \{\bar{t}^{\uparrow(\sigma)} \mid \bar{t} \in \mathcal{C}_3(J_3)\}$. □

We now have the following property:

Lemma B.4. $\mathcal{T}^3$ computes the query $\mathcal{Q}$.

Proof. Similarly to the proof of Lemma B.2, we can show by using Lemma B.3 that $\mathcal{T}^3$ computes the same query as $\mathcal{T}^2$ when restricted to inputs not containing x, which (by Lemma B.2) is the original query $\mathcal{Q}$ restricted to those inputs. At this point, we cannot say yet that $\mathcal{T}^3$ computes the full query $\mathcal{Q}$, i.e., for all inputs on which $\mathcal{Q}$ is defined. We will show now that this is actually the case.

First, observe that transducer Π^3 is oblivious. Indeed, Π does not use `All`, and we have further eliminated the use of `Id` from transducer Π to obtain Π^3. Then, Lemma 4.2 tells us that $\mathcal{T}^3$ computes a *generic* query $\mathcal{Q}'$. Let I be an instance over Υ_{in} with possibly $x \in adom(I)$. There is another instance J over Υ_{in} and a permutation h of **dom** such that $h(J) = I$ and $x \notin adom(J)$. As seen above, we have $\mathcal{Q}'(J) = \mathcal{Q}(J)$ and thus $h(Q'(J)) = h(\mathcal{Q}(J))$. By genericity of both $\mathcal{Q}'$ and $\mathcal{Q}$ we then have $\mathcal{Q}'(h(J)) = \mathcal{Q}(h(J))$ and thus $\mathcal{Q}'(I) = \mathcal{Q}(I)$. In conclusion, the transducer network $\mathcal{T}^3$ computes the same query $\mathcal{Q}$ as the original transducer network $\mathcal{T}$ but without reading the relations `Id` and `All`. □

References

[1] S. Abiteboul, M. Bienvenu, A. Galland, et al. A rule-based language for Web data management. In *Proceedings 30th ACM Symposium on Principles of Database Systems*, pages 293–304. ACM Press, 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. *Theoretical Computer Science*, 78:137–158, 1991.

[4] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[5] S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings 23rd ACM Symposium on the Theory of Computing*, pages 209–219, 1991.

[6] S. Abiteboul and V. Vianu. Computing with first-order logic. *Journal of Computer and System Sciences*, 50(2):309–335, 1995.

[7] S. Abiteboul, V. Vianu, et al. Relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 61(2):236–269, 2000.

[8] F.N. Afrati, S.C. Cosmadakis, and M. Yannakakis. On Datalog vs polynomial time. *Journal of Computer and System Sciences*, 51(2):177–196, 1995.

[9] P. Alvaro, N. Conway, J. Hellerstein, and W.R. Marczak. Consistency analysis in Bloom: A CALM and collected approach. In *Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260. www.cidrdb.org, 2011.

[10] P. Alvaro, W.R. Marczak, et al. Dedalus: Datalog in time and space. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded: First International Workshop, Datalog 2010*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281, 2011.

[11] T.J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Proceedings 30th ACM Symposium on Principles of Database Systems*, pages 283–292. ACM Press, 2011.

[12] T.J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducers. In *Proceedings of the 15th International Conference on Database Theory*, pages 86–98. ACM Press, 2012.

[13] T.J. Ameloot and J. Van den Bussche. On the cron conjecture. In P. Barceló and R. Pichler, editors, *Datalog in Academia and Industry*, volume 7494 of *Lecture Notes in Computer Science*, pages 44–55. Springer, 2012.

[14] K.R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.

[15] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, 2004.

[16] A. Blass, Y. Gurevich, and J. Van den Bussche. Abstract state machines and computationally complete query languages. *Information and Computation*, 174(1):20–36, 2002.

[17] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *Proceedings 12th International Conference on Database Theory*, 2009.

[18] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven Web applications. *Journal of Computer and System Sciences*, 73(3):442–474, 2007.

[19] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. In *Proceedings 25th ACM Symposium on Principles of Database Systems*, pages 90–99. ACM Press, 2006.

[20] P.T. Eugster, R. Guerraoui, A.M. Kermarrec, and L. Massoulié. From epidemics to distributed computing. *IEEE Computer*, 37:60–67, 2004.

[21] N. Francez. *Fairness*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.

[22] J. Gray. Notes on data base operating systems. In M.J. Flynn et al., editors, *Operating Systems—An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.

[23] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In M. Carro and R. Peña, editors, *Proceedings 12th International Symposium on Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 88–103, 2010.

[24] J.M. Hellerstein. Datalog redux: experience and conjecture. Video available (under the title "The Declarative Imperative") from `http://db.cs.berkeley.edu/jmh/`, 2010. PODS 2010 keynote.

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

[26] L. Lamport. Fairness and hyperfairness. *Distributed Computing*, 13:239–245, November 2000.

[27] B.T. Loo, T. Condie, et al. Declarative networking: language, execution and optimization. In S. Chaudhuri, V. Hristidis, and N. Polyzotis, editors, *SIGMOD Conference*, pages 97–108. ACM, 2006.

[28] B.T. Loo et al. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[29] V. Nigam, L. Jia, B.T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. *Computer Languages, Systems & Structures*, 38(2):158–180, 2012.

[30] M. Spielmann. Verification of relational transducers for electronic commerce. *Journal of Computer and System Sciences*, 66(1):40–65, 2003.

[31] J. Van den Bussche and L. Cabibbo. Converting untyped formulas into typed ones. *Acta Informatica*, 35(8):637–643, 1998.

[32] D. Zinn, T.J. Green, and B. Ludaescher. Win-move is coordination-free. In *Proceedings of the 15th International Conference on Database Theory*, pages 99–113. ACM Press, 2012.