Deciding Eventual Consistency for a Simple Class of Relational Transducer Networks

Deciding Eventual Consistency for a Simple Class of Relational Transducer Networks

Tom J. Ameloot*
Hasselt University &
Transnational University of Limburg
Diepenbeek, Belgium
tom.ameloot@uhasselt.be

Jan Van den Bussche
Hasselt University &
Transnational University of Limburg
Diepenbeek, Belgium
jan.vandenbussche@uhasselt.be

ABSTRACT

Networks of relational transducers can serve as a formal model for declarative networking, focusing on distributed database querying applications. In declarative networking, a crucial property is eventual consistency, meaning that the final output does not depend on the message delays and reorderings caused by the network. Here, we show that eventual consistency is decidable when the transducers satisfy some syntactic restrictions, some of which have also been considered in earlier work on automated verification of relational transducers. This simple class of transducer networks computes exactly all distributed queries expressible by unions of conjunctive queries with negation.

Categories and Subject Descriptors

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

General Terms

languages, theory

Keywords

distributed database, relational transducer, consistency, decidability, expressive power, cloud programming

1. INTRODUCTION

Declarative networking [15] is an approach by which distributed computations and networking protocols, as occurring in cloud computing, are modeled and programmed using formalisms based on Datalog. Recently, declarative networking formalisms are enjoying attention from the database theory community, so that now a number of models and languages are available with a formally defined semantics and initial investigations on their expressive power [5, 16, 12, 1, 6].

A major hurdle in using declarative methods for cloud computing is the nondeterminism inherent to such systems. This nondeterminism is typically due to the asynchronous communication between the compute nodes in a cluster or network. Accordingly, one of the challenges is to design distributed programs so that the same outputs can eventually be produced on the same inputs, no matter how messages between nodes have been delayed or received in different orders. When a program has this property we say it is *eventually consistent* [19, 13, 14, 4]. Of course, eventual consistency is undecidable in general, and there is much recent interest in finding ways to guarantee it [4, 1].

In the present paper, we view eventual consistency as a confluence notion. On any fixed input, let J be the union of all outputs that can be produced during any possible execution of the distributed program. Then in our definition of eventual consistency, we require that for any two different outputs $J_1 \subseteq J$ and $J_2 \subseteq J$ resulting from two (partial) executions on the same input, the same output J can be produced in an extension of either partial execution. So, intuitively, the prior execution of the program will not prevent outputs from being produced if those outputs can be produced with another execution (on the same input).

In this paper, we consider clusters of compute nodes modeled as *relational transducers*, an established formal model for data-centric agents [3, 18, 10, 9, 11]. In particular, we consider relational transducers where the rules used by the nodes to send messages, to update their state relations, and to produce output, are unions of conjunctive queries with negation. This setting yields a clear model of declarative networking, given the affinity between conjunctive queries and Datalog. We thus believe our results also apply to other declarative networking formalisms, although in this paper we have not yet worked out these applications.

Our first main result is the identification of a number of syntactic restrictions on the rules used in the transducers, *not* so that eventual consistency always holds, but so that checking it becomes decidable. Informally, the restrictions comprise the following.

1. The cluster must be recursion-free: the different rules among all local programs cannot be mutually recursive through positive subgoals. Recursive dependencies through negative subgoals are still allowed.

2. The local programs must be inflationary: deletions from state relations are forbidden.

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

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

3. The rules are message-positive: negation on message relations is forbidden.

4. The state-update rules must satisfy a known restriction which we call "message-boundedness". This restriction is already established in the verification of relational transducers: it was first identified under the name "input-boundedness" by Spielmann [18] and was investigated further by Deutsch et al. [10, 11].

5. Finally, the message-sending rules must be "static" in the sense that they cannot depend on state relations; they can still depend on input relations and on received messages.

The last two restrictions are the most fundamental; in fact, even if just the last restriction is dropped and all the others are kept in place, the problem is already back to undecidable. The first three restrictions can probably be slightly relaxed without losing decidability, and indeed we just see our work as a step in the right direction. Consistency is not an easy problem to analyze.

The second result of our paper is an analysis of the expressive power of clusters of relational transducers satisfying our above five restrictions; let us call such clusters "simple". Specifically, we show that simple clusters can compute *exactly* all distributed queries expressible by unions of conjunctive queries with negation, or equivalently, the existential fragment of first-order logic, without any further restrictions. So, this result shows that simple clusters form indeed a rather weak computational model, but not as weak as to be totally useless.

Related work.

The work most closely related to ours is that by Deutsch et al. on verification of communicating data-driven Web services [11]. The main differences between our works are the following. *(i)* In their setting, message buffers are ordered queues; in our setting, message buffers are unordered multisets. Unordered buffers model the asynchronous communication typical in cloud computing [14] where messages can be delivered out of order. *(ii)* In their setting, to obtain decidability, message buffers are bounded and lossy; in our setting, they are unbounded and not lossy. But actually, a crucial step in our proof of decidability will be to show that, under the restrictions we impose, even if buffers are not a priori bounded and are not lossy, they can be assumed to be bounded and lossy without loss (sic!) of generality. *(iii)* In their setting, transducers are less severely restricted than in our setting. *(iv)* In their setting, clusters of transducers are verified for properties expressed in (first-order) linear temporal logic;[1] in our setting, we are really focusing on the property of eventual consistency. It is actually not obvious whether eventual consistency (in the way we define it formally) is a linear-time temporal property, and if it is, whether it is expressible in first-order linear temporal logic.

Also, this paper is a follow-up on our previous paper [6]. In our previous paper, we did not consider the problem of deciding eventual consistency; we simply assumed eventual consistency and were focusing on expressiveness issues. Moreover, while the distributed computing model used in our previous paper is also based on relational transducers, there are differences in the models. In the previous model, we were focusing on standard queries to databases, computed in a distributed fashion by distributing the database in an arbitrary way over the nodes of the network. In the present model, we directly consider distributed queries, i.e., the input to the query is a distributed database, and different distributions of the same dataset may yield different answers to the query. Furthermore, in the previous model, transducer programs are considered to be network-independent, and nodes communicate in an epidemic manner by spreading messages to their neighbors, who read them one at a time; in the present model, the network is given, different nodes can run different programs, and nodes can directly address their messages to specified nodes. The perspective taken in our previous paper is equally interesting but different; we have simply chosen here to focus on the present perspective because it is the one mostly assumed by other authors in the area of declarative networking.

[1]Deutsch et al. can also verify branching-time temporal properties, but only when transducer states are propositional.

Organization.

In Section 2 we give some preliminaries. Next, in Section 3 we state our results about decidability and expressivity of the simple transducer networks. In Sections 4 and 5 we give the proof techniques used to obtain these results. We conclude in Section 6.

2. PRELIMINARIES

2.1 Overview and Design Choices

We start with preliminaries on database theory in Section 2.2. As stated in the introduction, we model computation by means of *(relational) transducers*, which are defined in Section 2.3. These transducers are given database facts as input and they can derive new facts by applying database queries to previously obtained facts. In Section 2.4 we introduce the notion of *distributed query*, which serves as an implementation independent specification for what a distributed program computes. Section 2.5 formalizes multisets.

The computation of a distributed program will be modeled using *networks of transducers*, and this is formalized in Section 2.6. We will next only consider transducers whose queries are implemented with unions of conjunctive queries with negation, see Section 2.7. This results in a *rule-based formalism* to express distributed computation, following the idea behind declarative networking [15]. For our decidability result we impose restrictions, which are given in Section 2.8. The restrictions result in so-called "simple" transducers.

2.2 Database Schema, Facts and Queries

We recall some basic notions from database theory [2], although the specific definitions below are slightly tailored to our technical use. 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 occurs at most once in every database schema. We define $names(\mathcal{D}) = \{R \mid \exists k : (R, k) \in \mathcal{D}\}$. We often write a pair $(R, k) \in \mathcal{D}$ as $R^{(k)}$.

We assume some infinite universe **dom** of atomic data values. A *fact* $\boldsymbol{f}$ is a pair $(R, \bar{t})$, often denoted as $R(\bar{t})$, where R is a relation name and $\bar{t}$ is a tuple of values over **dom**. A database *instance* I of (or over) a database schema $\mathcal{D}$ is a set of facts such that for $R(a_1, \ldots, a_k) \in I$ we have

$R^{(k)} \in \mathcal{D}$. A database instance over an empty database schema can only be the empty set. For a database instance I over $\mathcal{D}$ and for a relation name R in $\mathcal{D}$ we define the set $I(R) = \{\bar{t} \mid R(\bar{t}) \in I\}$, i.e, to extract the set of tuples from the facts.

We define some additional notation and terminology concerning facts. Let $\boldsymbol{f} = R(\bar{t})$ be a fact. Here, R is called the *predicate* and the number of components in $\bar{t}$ is called the *arity* of the fact. The former is denoted as $pred(\boldsymbol{f})$. If $\boldsymbol{f}$ has arity 0 we call $\boldsymbol{f}$ *nullary*. The *active domain* of a fact $\boldsymbol{f}$, denoted $adom(\boldsymbol{f})$, is the set of values occurring in its tuple. For a fact $\boldsymbol{f} = R(a_1, \ldots, a_k)$, a set A with $adom(\boldsymbol{f}) \subseteq A$ and a function $h : A \to \mathbf{dom}$, we define $h(\boldsymbol{f}) = R(h(a_1), \ldots, h(a_k))$. Let I be a set of facts. We define $adom(I) = \bigcup_{\boldsymbol{f} \in I} adom(\boldsymbol{f})$. For a function $h : adom(I) \to \mathbf{dom}$ we define $h(I) = \{h(\boldsymbol{f}) \mid \boldsymbol{f} \in I\}$. If h is injective, we say that $h(I)$ and I are *isomorphic*.

Let $C \subseteq \mathbf{dom}$. We call a fact $\boldsymbol{f}$ a *C-fact* if $adom(\boldsymbol{f}) \subseteq C$. For a set I of facts, we define I^C to be the subset of all C-facts in I (including nullary facts).

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 query $\mathcal{Q}$ is called *generic* if for all input instances I it holds that *(i)* $adom(\mathcal{Q}(I)) \subseteq adom(I)$, *(ii)* $\mathcal{Q}$ is also defined on the isomorphic instance $h(I)$, for each permutation h of $\mathbf{dom}$, and $\mathcal{Q}(h(I)) = h(\mathcal{Q}(I))$. It is well known that query languages like first-order logic (FO) can be used to express queries [2]. The most common kind of query are those where the output database schema contains just one relation name.

2.3 Transducers

We now formalize the computation on a single node of a network by means of relational transducers [3, 6, 9, 10, 11, 18]. A *transducer schema* Υ is a tuple $(\mathcal{D}_{in}, \mathcal{D}_{out}, \mathcal{D}_{msg}, \mathcal{D}_{mem}, \mathcal{D}_{sys})$ of database schemas, called "input", "output", "message", "memory" and "system" respectively. A relation name can occur in at most one database schema of Υ. We fix $\mathcal{D}_{sys}$ to always contain two unary relations Id and All. Let $\mathcal{D} \subseteq \mathcal{D}_{in} \cup \mathcal{D}_{out} \cup \mathcal{D}_{msg} \cup \mathcal{D}_{mem} \cup \mathcal{D}_{sys}$. For a database instance I over $\mathcal{D}$ and $e \subseteq \{in, out, msg, mem, sys\}$ we write $I|_e$ to denote the restriction of I to the facts whose predicate is a relation name in $\bigcup_{f \in e} \mathcal{D}_f$.

Let $\Upsilon = (\mathcal{D}_{in}, \mathcal{D}_{out}, \mathcal{D}_{msg}, \mathcal{D}_{mem}, \mathcal{D}_{sys})$ be a transducer schema. A *transducer state* for Υ is a database instance over $\mathcal{D}_{in} \cup \mathcal{D}_{out} \cup \mathcal{D}_{mem} \cup \mathcal{D}_{sys}$. A *relational transducer* Π (or just transducer for short) over Υ is a collection of queries

$$\{\mathcal{Q}^R_{out} \mid R^{(k)} \in \mathcal{D}_{out}\} \cup \{\mathcal{Q}^R_{snd} \mid R^{(k)} \in \mathcal{D}_{msg}\} \cup \{\mathcal{Q}^R_{ins} \mid R^{(k)} \in \mathcal{D}_{mem}\} \cup \{\mathcal{Q}^R_{del} \mid R^{(k)} \in \mathcal{D}_{mem}\}$$

that are all over the input database schema $\mathcal{D}_{in} \cup \mathcal{D}_{out} \cup \mathcal{D}_{msg} \cup \mathcal{D}_{mem} \cup \mathcal{D}_{sys}$ and for $R^{(k)} \in \mathcal{D}_{out}$ the output schema of $\mathcal{Q}^R_{out}$ consists of relation R with arity k, the same for $R^{(k)} \in \mathcal{D}_{mem}$ and the queries $\mathcal{Q}^R_{ins}$ and $\mathcal{Q}^R_{del}$, and finally for $R^{(k)} \in \mathcal{D}_{msg}$ the output schema of query $\mathcal{Q}^R_{snd}$ consist of relation R with arity $k+1$. The reason for the incremented arity of $\mathcal{Q}^R_{snd}$ is that the extra component will be used to indicate the addressee of messages. Importantly, at this point we have not said anything about how the queries of a transducer are actually implemented. The idea is that the above transducer model is parameterized by the query language used. If a query language $\mathcal{L}$ is used to express the queries of a transducer Π then we say that Π is an $\mathcal{L}$-transducer.

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

$$\begin{aligned} J|_{in,sys} &= I|_{in,sys} \\ J|_{out} &= I|_{out} \cup \bigcup_{R^{(k)} \in \mathcal{D}_{out}} \mathcal{Q}^R_{out}(I') \\ J|_{mem} &= \bigcup_{R^{(k)} \in \mathcal{D}_{mem}} update(R, I, I') \\ J_{snd} &= \bigcup_{R^{(k)} \in \mathcal{D}_{msg}} \mathcal{Q}^R_{snd}(I') \end{aligned}$$

with

$$\begin{aligned} update(R, I, I') = &\big(\mathcal{Q}^R_{ins}(I') \setminus \mathcal{Q}^R_{del}(I')\big) \\ &\cup \big(\mathcal{Q}^R_{ins}(I') \cap \mathcal{Q}^R_{del}(I') \cap I\big) \\ &\cup \big(I \setminus (\mathcal{Q}^R_{ins}(I') \cup \mathcal{Q}^R_{del}(I'))\big). \end{aligned}$$

Intuitively, on the receipt of message facts I_{rcv}, a local state transition updates the old transducer state I to new transducer state J and sends the facts in J_{snd}. The new transducer state J has unmodified input and system facts compared to I, has potentially produced more output facts, and for each memory relation R the update semantics basically adds all facts produced by the *additive* query $\mathcal{Q}^R_{ins}$, removes all facts produced by the *subtractive* query $\mathcal{Q}^R_{del}$ and in case the same fact is both added and removed at the same time there is no-op semantics [6, 18]. Note that local transitions are deterministic in the following sense: if $I, I_{rcv} \Rightarrow J, J_{snd}$ and $I, I_{rcv} \Rightarrow J', J'_{snd}$ then $J = J'$ and $J_{snd} = J'_{snd}$.

2.4 Distributed Databases and Queries

We now formalize how input data is distributed across a network and define a notion of queries over this data. A *distributed database schema* is a pair $(\mathcal{N}, \eta)$ with $\mathcal{N} \subseteq \mathbf{dom}$ a finite set of values called *nodes* and η a function that maps every node $x \in \mathcal{N}$ to a database schema $\eta(x)$ that does not contain the relation names Id and All. A *distributed database instance* I over a distributed database schema $(\mathcal{N}, \eta)$ is a function mapping every node $x \in \mathcal{N}$ to a (normal) database instance over schema $\eta(x)$.

We can view a distributed database schema $(\mathcal{N}, \eta)$ as a normal database schema

$$norm(\mathcal{N}, \eta) = \{(R^x, k) \mid x \in \mathcal{N}, (R, k) \in \eta(x)\},$$

thus by writing the node in the relation name. Intuitively, every node is a namespace in which relation names occur. Now, we can view a distributed database instance I over $(\mathcal{N}, \eta)$ unambiguously as a normal database instance I' over schema $norm(\mathcal{N}, \eta)$ as follows

$$I' = \bigcup_{x \in \mathcal{N}} \bigcup_{R^{(k)} \in \eta(x)} \{R^x(\bar{t}) \mid R(\bar{t}) \in I(x)\}.$$

In the other direction, a database instance J over schema $norm(\mathcal{N}, \eta)$ can be unambiguously viewed as a distributed database instance over schema $(\mathcal{N}, \eta)$.

For a distributed database schema $(\mathcal{N}, \eta)$ we define

$$norm^+(\mathcal{N}, \eta) = norm(\mathcal{N}, \eta) \cup \{(Id^x, 1) \mid x \in \mathcal{N}\} \cup \{(All^*, 1)\}.$$

Here relation name All^* has the superscript $*$ to differentiate it from the regular relation name All used in $\mathcal{D}_{sys}$ (Section 2.3). For a database instance I over $norm^+(\mathcal{N}, \eta)$ we say that I has *context* $\mathcal{N}$ if for $x \in \mathcal{N}$ we have $I(Id^x) = \{x\}$ and $I(All^*) = \mathcal{N}$. Like above, a distributed database instance I over schema $(\mathcal{N}, \eta)$ can be naturally viewed as a normal database instance I' over schema $norm^+(\mathcal{N}, \eta)$ as follows:

$$I' = \bigcup_{x \in \mathcal{N}} \bigcup_{R^{(k)} \in \eta(x)} \{R^x(\bar{t}) \mid R(\bar{t}) \in I(x)\} \cup \bigcup_{x \in \mathcal{N}} \{Id^x(x), All^*(x)\}.$$

A *distributed query* $\mathcal{Q}$ *for a nodeset* $\mathcal{N}$ *over distributed input database schema* $(\mathcal{N}, \eta_{in})$ *and distributed output database schema* $(\mathcal{N}, \eta_{out})$ is a (normal) query over input schema $norm^+(\mathcal{N}, \eta_{in})$ and output schema $norm(\mathcal{N}, \eta_{out})$ that is defined for the input instances with context $\mathcal{N}$.

Intuitively, in a distributed query we are not only interested in the set of output facts, but also on which node an output fact is produced.

The reason for including the relations Id^x with $x \in \mathcal{N}$ and All^* in the input of a distributed query is that the values $\mathcal{N}$ can in principle occur as values in the input facts and that query-results could meaningfully depend on this interaction. See for instance Example 1 in Section 2.8.

2.5 Multisets

The concept of multiset plays an important role in our modeling of message transmission in a network. Let $\mathcal{U}$ be a universe of elements. A *multiset* over $\mathcal{U}$ is a total function $m : \mathcal{U} \to \mathbb{N}$. Intuitively, for each element of the universe the multiset m says how many times that element "occurs" in the multiset. We define $set(m) = \{v \in \mathcal{U} \mid m(v) \geq 1\}$; these are the unique elements in m, without their precise counts. For $v \in \mathcal{U}$ we will also often write $cnt(v, m)$ to mean $m(v)$. This notation will be more readable when the multiset is not given by one symbol but by an expression. The *size* of a multiset m over $\mathcal{U}$, denoted $|m|$, is defined as $|m| = \sum_{v \in \mathcal{U}} m(v)$. This sum can be infinite, in which case we say that the multiset is infinite, otherwise it is called finite. We write $|m| = 0$ also as $m = \emptyset$.

A set m can be viewed as a special kind of multiset, where $img(m) = \{0, 1\}$. Likewise, a multiset m with $img(m) = \{0, 1\}$ can be viewed as a set.

Let m_1 and m_2 be two multisets over a universe $\mathcal{U}$. We say that m_1 is *contained* in m_2, denoted $m_1 \sqsubseteq m_2$, if for all $v \in \mathcal{U}$ it holds $m_1(v) \leq m_2(v)$. Multiset union ($\cup$), intersection ($\cap$) and difference ($\setminus$) are respectively defined as follows: $m_3 = m_1 \cup m_2$ if $\forall v \in \mathcal{U} : m_3(v) = m_1(v) + m_2(v)$, $m_3 = m_1 \cap m_2$ if $\forall v \in \mathcal{U} : m_3(v) = \min\{m_1(v), m_2(v)\}$, $m_3 = m_1 \setminus m_2$ if $\forall v \in \mathcal{U} : m_3(v) = \max\{0, m_1(v) - m_2(v)\}$. When at least one of the operands of the operators $\cup$, $\cap$ or $\setminus$ is a multiset (with the other operand a set or a multiset) then the multiset semantics just given can be applied.

Let $\mathcal{D}$ be a database schema. Let m be a multiset of facts. We say that m is of (or over) $\mathcal{D}$ if $set(m)$ is a database instance of $\mathcal{D}$.

2.6 Transducer Networks

We now formalize a network of compute nodes. A *transducer network* is a pair $(\mathcal{N}, \theta)$ where $\mathcal{N} \subseteq \mathbf{dom}$ is a nonempty finite set of values called *nodes* and θ is a function that maps every node $x \in \mathcal{N}$ to a pair (Υ^x, Π^x) with Υ^x a transducer schema and Π^x a transducer over Υ^x. $(\mathcal{N}, \theta)$. Denote $\Upsilon^x = (\mathcal{D}^x_{in}, \mathcal{D}^x_{out}, \mathcal{D}^x_{msg}, \mathcal{D}^x_{mem}, \mathcal{D}^x_{sys})$. We consider only transducer networks such that for $x, y \in \mathcal{N}$ it holds $\mathcal{D}^x_{msg} = \mathcal{D}^y_{msg}$, i.e., the transducers all share the same message relations. This is not really a restriction because a transducer on a node does not have to read all these message relations in its queries and for some message relations its sending query might always return the empty set of facts. For $x, y \in \mathcal{N}$ it automatically also holds that $\mathcal{D}^x_{sys} = \mathcal{D}^y_{sys}$ because we fixed the $\mathcal{D}_{sys}$-component in every transducer schema to consist of $Id^{(1)}$ and $All^{(1)}$. Except for the sharing of the message database schema and $\mathcal{D}_{sys}$ we make no further assumptions about how relation names for input, output and memory might be shared by several nodes. For $e \in \{in, out\}$ define the distributed database schema $(\mathcal{N}, \eta_e)$ as follows: for $x \in \mathcal{N}$ we have $\eta_e(x) = \mathcal{D}^x_e$. Denote $inSchema(\mathcal{N}, \theta) = (\mathcal{N}, \eta_{in})$ and $outSchema(\mathcal{N}, \theta) = (\mathcal{N}, \eta_{out})$. Intuitively, these distributed schemas are the joint input and output schema of the transducer network respectively.

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.

The *network transition system* describes how a transducer network performs computations. The *input* of a transducer network $(\mathcal{N}, \theta)$ is a distributed input database instance I over schema $inSchema(\mathcal{N}, \theta)$. A *configuration* of $(\mathcal{N}, \theta)$ on input I is a pair $\rho = (s^\rho, b^\rho)$ of functions where

- s^ρ maps each $x \in \mathcal{N}$ to a transducer state $J = s^\rho(x)$ for (Υ^x, Π^x) such that $J|_{in} = I(x)$, $J(Id) = \{x\}$ and $J(All) = \mathcal{N}$,
- b^ρ maps each $x \in \mathcal{N}$ to a finite multiset of facts over the shared message database schema.

We call s^ρ the "state function" and b^ρ the "buffer function". Intuitively, for $x \in \mathcal{N}$ the system relations Id and All in $\mathcal{D}_{sys}$ provide the local transducer Π^x the identity of the node x it is running on and the identities of the other nodes. Also, the buffer function maps $x \in \mathcal{N}$ to the multiset of messages that have been sent to x but that have not yet been delivered to x.

The *start configuration* of $(\mathcal{N}, \theta)$ on a distributed input database instance I, denoted $start(\mathcal{N}, \theta, I)$, is the configuration ρ of $(\mathcal{N}, \theta)$ on input I defined by:

- for $x \in \mathcal{N}$ we have $s^\rho(x)|_{out,mem} = \emptyset$,
- for $x \in \mathcal{N}$ we have $b^\rho(x) = \emptyset$.

A *(global) transition* of $(\mathcal{N}, \theta)$ on input I is a 4-tuple (ρ_1, x, m, ρ_2), also denoted as $\rho_1 \xrightarrow{x,m} \rho_2$, where $x \in \mathcal{N}$ and ρ_1 and ρ_2 are configurations of $(\mathcal{N}, \theta)$ on input I such that

- for $y \in \mathcal{N} \setminus \{x\}$ we have $s^{\rho_1}(y) = s^{\rho_2}(y)$,
- $m \sqsubseteq b^{\rho_1}(x)$ and there exists a message instance J_{snd} such that $s^{\rho_1}(x), set(m) \Rightarrow s^{\rho_2}(x), J_{snd}$ is a local transition of transducer Π^x,

- for $y \in \mathcal{N} \setminus \{x\}$ we have $b^{\rho_2}(y) = b^{\rho_1}(y) \cup J^y_{snd}$ (multiset union) and for x we have $b^{\rho_2}(x) = \big(b^{\rho_1}(x) \setminus m\big) \cup J^x_{snd}$ (multiset union and difference) where $J^z_{snd} = \{R(\bar{a}) \mid R(z, \bar{a}) \in J_{snd}\}$ for $z \in \mathcal{N}$.

If $m = \emptyset$ we call this global transition a *heartbeat* transition and otherwise we call it a *delivery* transition. Intuitively, during a global transition we select a random node $x \in \mathcal{N}$ and a random submultiset m of 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. We refer to x as the *recipient*. Consider how the message sending by node x is modeled. The first component z of a fact $R(z, \bar{a}) \in J_{snd}$ above is regarded as the *addressee* of that fact. The projected fact $R(\bar{a})$ is then transferred to the message buffer of the addressee z. Values for the addressee component outside $\mathcal{N}$ will result in the loss of the corresponding message fact. A heartbeat transition can be likened to the real life situation in which a node does a computation step when a local timer goes off and when no messages are received from the network.

A *run* $\mathcal{R}$ of a transducer network $(\mathcal{N}, \theta)$ on a distributed input database instance I is a *finite* sequence of global transitions $(\rho_{1,a}, x_1, m_1, \rho_{1,b})$, $\ldots$, $(\rho_{i,a}, x_i, m_i, \rho_{i,b})$, $\ldots$, $(\rho_{n,a}, x_n, m_n, \rho_{n,b})$ such that $\rho_{1,a} = \mathit{start}(\mathcal{N}, \theta, I)$ and for $i \in \{2, \ldots, n\}$ we have $\rho_{i,a} = \rho_{i-1,b}$. In other words, for $i \geq 2$ the i^{th} transition works on the resulting configuration of the previous transition. Note that $\mathcal{R}$ represents a sequence of configurations $\rho_1, \ldots, \rho_{n+1}$ such that $\rho_1 = \rho_{1,a}$ and for $i \in \{2, \ldots, n+1\}$ we have that $\rho_i = \rho_{i-1,b}$. We write $\mathit{last}(\mathcal{R})$ to denote ρ_{n+1}. We write $\mathit{length}(\mathcal{R})$ to denote the number of transitions in $\mathcal{R}$. We say that a run $\mathcal{R}'$ is an *extension* of run $\mathcal{R}$ if $\mathcal{R}$ is a prefix of $\mathcal{R}'$, i.e., $\mathcal{R}'$ is obtained by appending zero or more transitions after those of $\mathcal{R}$.

2.7 Conjunctive Queries and Literals

In this paper we will work with transducers whose queries are expressed as unions of conjunctive queries with (safe) negation (equivalently, the existential fragment of first-order logic). It will be convenient to use a slightly unconventional formalization of conjunctive queries.

A *conjunctive query with negation and inequality*, or conjunctive query for short, is a 4-tuple $\varphi = (\boldsymbol{h}, p, n, q)$ where $\boldsymbol{h}$ is a fact, p and n are finite sets of a facts such that $\mathit{adom}(\boldsymbol{h}) \cup \mathit{adom}(n) \subseteq \mathit{adom}(p)$ (safety condition) and q is a set of inequalities of the form $(x \neq y)$ with $x, y \in \mathit{adom}(p)$ and $x \neq y$. We call $\boldsymbol{h}$, p and n respectively the "head" fact, the set of "positive" facts and the set of "negative" facts. We denote $\mathit{head}(\varphi) = \boldsymbol{h}$, $\mathit{pos}(\varphi) = p$, $\mathit{neg}(\varphi) = n$, $\mathit{ineq}(\varphi) = q$, $\mathit{var}(\varphi) = \mathit{adom}(\mathit{pos}(\varphi))$, $\mathit{free}(\varphi) = \mathit{adom}(\boldsymbol{h})$ and $\mathit{bound}(\varphi) = \mathit{var}(\varphi) \setminus \mathit{free}(\varphi)$. The elements in $\mathit{var}(\varphi)$, $\mathit{free}(\varphi)$ and $\mathit{bound}(\varphi)$ respectively are called the "variables", "free variables" and "bound variables". Indeed, bound variables can be seen as being existentially quantified. For an arbitrary ordering $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k$ of $\mathit{pos}(\varphi)$, $\boldsymbol{g}_1, \ldots, \boldsymbol{g}_l$ of $\mathit{neg}(\varphi)$ and $q_1, \ldots, q_m$ of q one can syntactically write φ in the conventional form

$$\boldsymbol{h} \leftarrow \boldsymbol{f}_1, \ldots, \boldsymbol{f}_k, \neg \boldsymbol{g}_1, \ldots, \neg \boldsymbol{g}_l, q_1, \ldots, q_m.$$

The set $\{\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k, \neg \boldsymbol{g}_1, \ldots, \neg \boldsymbol{g}_l, q_1, \ldots, q_m\}$ is called the *body* of φ and is denoted $\mathit{body}(\varphi)$. In the notation above, the ordering of the elements of the body does not matter. The elements $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k, \ldots, \neg \boldsymbol{g}_1, \ldots, \neg \boldsymbol{g}_l$ are called *literals*. A literal is *negative* if it is a fact with an uneven number of $\neg$-symbols prepended, otherwise it is called *positive*. Literals will be denoted in boldface, just like facts. If we want to denote the fact contained in a literal $\boldsymbol{g}$ we write $+\boldsymbol{g}$.

Let φ be a conjunctive query. A *valuation* for φ is a function Val with $\mathit{var}(\varphi) \subseteq \mathit{dom}(\mathit{Val})$ and $\mathit{img}(\mathit{Val}) \subseteq \boldsymbol{dom}$. Let I be a set of facts. We say that a valuation Val for φ is *satisfying* on I if $\mathit{Val}(\mathit{pos}(\varphi)) \subseteq I$, $\mathit{Val}(\mathit{neg}(\varphi)) \cap I = \emptyset$ and for each $(x \neq y) \in \mathit{ineq}(\varphi)$ it holds $\mathit{Val}(x) \neq \mathit{Val}(y)$. We say that $\mathit{Val}(\mathit{pos}(\varphi))$ are the facts whose presence is *needed* by Val. The *(query) result* of φ when applied to an *input* set of facts I, denoted $\varphi(I)$, is the set of facts

$$\{\, \mathit{Val}(\mathit{head}(\varphi)) \mid \mathit{Val} \text{ is a satisfying valuation for } \varphi \text{ on } I\}.$$

We also say that φ "produces" these facts when applied to I. The conjunctive queries as defined here can only express generic queries. Let $CQ^\neg$ denote the resulting language.

Note that conjunctive queries φ with an empty body have $\mathit{free}(\varphi) = \emptyset$. These queries produce the (nullary) head fact on every input. For technical reasons, we assume that there is some relation name *false* that cannot be used as a predicate name in the sets of input facts for a conjunctive query. Conjunctive queries with an empty body can be simulated with this relation: give the query no positive facts and one negative fact $\mathit{false}()$. This special relation name can also be used to write conjunctive queries that cannot produce any fact on any database. Indeed, it suffices to include the fact $\mathit{false}()$ in the positive facts. This allows us to assume that all conjunctive queries φ in this text have either $\mathit{pos}(\varphi) \neq \emptyset$ or $\mathit{neg}(\varphi) \neq \emptyset$.

A *union of conjunctive queries with negation (and inequality)* is a finite nonempty set of queries in $CQ^\neg$ that have the same predicate and arity for their head facts. The resulting language is denoted $UCQ^\neg$. Let $\psi = \{\varphi_1, \ldots, \varphi_n\}$ be a $UCQ^\neg$ query, where each φ_i is a $CQ^\neg$ query. The *query result* of ψ when applied to a set of facts I is the set of facts $\bigcup_{i=1}^n \varphi_i(I)$.

We use the query language $UCQ^\neg$ for implementing the queries of a transducer. We will often speak of all the $CQ^\neg$ queries that appear in an $UCQ^\neg$-transducer Π as the *rules* of Π (so not the $UCQ^\neg$ queries themselves). If a rule appears in the $UCQ^\neg$ query for a relation name R of a transducer schema we call it a "rule for R". If R is a message relation name then we also call the rule a "send rule". Similarly for the other relation types in a transducer schema. This rule-based formalism can be used to write distributed applications in a relatively high-level manner, which characterizes declarative networking [1, 5, 14].

2.8 Transducer Restrictions

Let Π be an $UCQ^\neg$-transducer over transducer schema $\Upsilon = (\mathcal{D}_{in}, \mathcal{D}_{out}, \mathcal{D}_{msg}, \mathcal{D}_{mem}, \mathcal{D}_{sys})$. We can impose syntactic restrictions on Π.

Let φ be a $CQ^\neg$ rule used in Π.

- We say that φ is *message-positive* if there are no message facts in $\mathit{neg}(\varphi)$.
- We say that φ is *static* if $\mathit{pos}(\varphi) \cup \mathit{neg}(\varphi)$ contains only facts over $\mathcal{D}_{in} \cup \mathcal{D}_{msg} \cup \mathcal{D}_{sys}$, i.e., no output or memory relations are used.[2]

[2]The restrictions considered by Deutsch et al. [10] for "input-rules", which are closely related to our send rules, are a bit

- Denote $A = adom(pos(\varphi)|_{msg})$ and $B = adom\big((pos(\varphi) \cup neg(\varphi))|_{out,mem}\big)$. We say that φ is *message-bounded* if $bound(\varphi) \subseteq A$ and $bound(\varphi) \cap B = \emptyset$. In words: every bounded variable of φ must occur in a positive message fact and may not occur in output or memory literals (positive or negative). This is the application of the more general definition of "input-boundedness" [10, 11, 18] to $CQ^\neg$ queries.[3]

Let Π and Υ be as above. The *positive dependency graph* of Π is the graph in which the vertices are the relation names in $\mathcal{D}_{out} \cup \mathcal{D}_{msg} \cup \mathcal{D}_{mem}$ and there is an edge from relation name R to relation name S if S is the predicate of a positive fact in a rule for relation R in Π. We say that Π is *recursion-free* if there are no cycles in the positive dependency graph of Π.

We say that Π is *inflationary* if for $R^{(k)} \in \mathcal{D}_{mem}$ the subtractive query $\mathcal{Q}^R_{del}$ is empty (or formally, $\mathcal{Q}^R_{del}$ has in each of its $CQ^\neg$ queries a positive fact with the predicate *false* from Section 2.7). In that case Π can not delete facts from the memory relations.

To summarize, we call Π *simple* (for lack of a better name) if

- Π is recursion-free and inflationary,
- for $R^{(k)} \in \mathcal{D}_{msg}$ all the rules in $\mathcal{Q}^R_{snd}$ are message-positive and static,
- for $R^{(k)} \in \mathcal{D}_{out}$ and $S^{(l)} \in \mathcal{D}_{mem}$ respectively all the rules in $\mathcal{Q}^R_{out}$ and $\mathcal{Q}^S_{ins}$ are message-positive and message-bounded.

We would like to stress again that when Π is inflationary this transducer will not be able to delete any memory facts once they are produced. This makes memory and output relations basically behave in the same way. However, we still preserve the difference between these two kinds of relations to retain the connection to the unrestricted transducer model and because memory relations are useful as a separate construct, namely, as relations that can be used for computation but that don't belong to the final result.

As for notation, when we give an example of simple transducers we may give a set of rules and it will be clear by looking at their heads for what relation name of the transducer schema they are meant. We will also omit all rules for the subtractive queries because they are assumed never to produce any facts.

Let $(\mathcal{N}, \theta)$ be a transducer network. The *positive message dependency graph* of $(\mathcal{N}, \theta)$ is the graph in which the vertices are the message relation names used in $(\mathcal{N}, \theta)$ and there is an edge from relation name R to relation name S if S is the predicate of a positive fact in a rule for relation R in some transducer of $(\mathcal{N}, \theta)$.

We now call $(\mathcal{N}, \theta)$ *simple* if for each $x \in \mathcal{N}$ with $\theta(x) = (\Upsilon^x, \Pi^x)$ we have that Π^x is simple, and the positive message dependency graph contains no cycles. Note that this is a "global" recursion-freeness requirement, across the network. Note also that, because all the send rules of the transducers are static, at every node the send rules must always produce the same facts on receipt of the same messages, independently of what output or memory facts might have been derived.

Consider the following example of a simple transducer network.

Example 1. Let $\mathcal{N} = \{x, y\}$ be a set of two different nodes. We define a simple transducer network $(\mathcal{N}, \theta)$. Define message database schema $\mathcal{D}_{msg} = \{A^{(1)}_{msg}, B^{(2)}_{msg}\}$. Define $\mathcal{D}^x_{in} = \{A^{(1)}\}$, $\mathcal{D}^x_{out} = \{T^{(1)}\}$, $\mathcal{D}^x_{mem} = \emptyset$, $\mathcal{D}^y_{in} = \{B^{(2)}\}$, $\mathcal{D}^y_{out} = \{T^{(1)}\}$ and $\mathcal{D}^y_{mem} = \emptyset$. For transducer Π^x we have the following rules:

$$\begin{aligned} A_{msg}(n, v) &\leftarrow A(v), All(n), \neg Id(n). \\ T(v) &\leftarrow B_{msg}(u, v), Id(u). \end{aligned}$$

For transducer Π^y we have the rules

$$\begin{aligned} B_{msg}(n, u, v) &\leftarrow B(u, v), All(n), \neg Id(n). \\ T(v) &\leftarrow A_{msg}(v). \end{aligned}$$

On any distributed input database instance I for $(\mathcal{N}, \theta)$ the node x sends its local A-facts as A_{msg}-facts to y and y sends its local B-facts as B_{msg}-facts to x. Node x outputs for all received B_{msg}-facts the second component in its output relation T if the first component is its own node identifier. Node y simply outputs all received A_{msg}-facts as T-facts. This example illustrates that the occurrence of node-identifiers in the input database facts can be used to produce output. □

3. CONSISTENCY AND EXPRESSIVITY

Let $(\mathcal{N}, \theta)$ be a transducer network. Let I be a distributed input database instance for $(\mathcal{N}, \theta)$. Recall that output facts on a node can never be deleted at that node once they are derived. We call $(\mathcal{N}, \theta)$ *consistent on* I if for any two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $(\mathcal{N}, \theta)$ on input I, for every node $x \in \mathcal{N}$, for every fact $\boldsymbol{f} \in s^{last(\mathcal{R}_1)}(x)|_{out}$ there exists an extension $\mathcal{R}'_2$ of $\mathcal{R}_2$ such that $\boldsymbol{f} \in s^{last(\mathcal{R}'_2)}(x)|_{out}$. Thus, on input I, if during one run on some node an output fact can be produced, then for *any* run there exists an extension in which that fact can be produced on that node too. We call $(\mathcal{N}, \theta)$ *consistent* if $(\mathcal{N}, \theta)$ is consistent for all distributed input database instances for $(\mathcal{N}, \theta)$. Naturally, we will sometimes use the term "inconsistent" to mean "not consistent". Our definition of consistency is a formalization of the notion of "eventual consistency" [4, 14], but see also Section 6.

The transducer network given in Example 1 is consistent. Indeed, say, node x outputs a fact $T(a)$ during a run. This means that earlier in the run node y has sent the input fact $B(x, a)$ as $B_{msg}(x, a)$ to node x and x has received this message fact (possibly together with other message facts). Now consider any run on the same distributed input database instance where x has not yet output $T(a)$. We can extend the run as follows. If y has not yet performed any local transition we first do a global transition with recipient node y which makes that all input B-facts of y are sent to x as B_{msg}-facts. Then in a following global transition we deliver $B_{msg}(x, a)$ to x and x outputs $T(a)$. In a similar way we can argue that if the node y outputs a T-fact in one run, then any other run on the same distributed input database can be extended to output this fact as well. Therefore the transducer network is consistent.

less restrictive. Roughly speaking, they still allow the use of nullary memory facts. It seems plausible that our results can be similarly extended.

[3]We have replaced the word "input" by "message" because the former has a different meaning in our text, namely, as the input that a transducer is locally initialized with.

In contrast, consider the following example of a simple transducer network that is *not* consistent.

Example 2. Let $\mathcal{N} = \{x, y\}$ be a set of two different nodes. We define a simple transducer network $(\mathcal{N}, \theta)$ as follows. Define database schema $\mathcal{D}_{msg} = \{A^{(1)}_{msg}, B^{(1)}_{msg}\}$. Define $\mathcal{D}^x_{in} = \{A^{(1)}, B^{(1)}\}$, $\mathcal{D}^x_{out} = \emptyset$, $\mathcal{D}^x_{mem} = \emptyset$, $\mathcal{D}^y_{in} = \emptyset$, $\mathcal{D}^y_{out} = \{T^{(1)}\}$ and $\mathcal{D}^y_{mem} = \{B^{(1)}\}$. In particular, node x does not output anything. For transducer Π^x we have the following rules:

$$\begin{aligned} A_{msg}(n, v) &\leftarrow A(v), All(n), \neg Id(n). \\ B_{msg}(n, v) &\leftarrow B(v), All(n), \neg Id(n). \end{aligned}$$

For transducer Π^y we have the rules

$$\begin{aligned} B(v) &\leftarrow B_{msg}(v). \\ T(v) &\leftarrow A_{msg}(v), \neg B(v). \end{aligned}$$

Let I be the distributed input database instance over schema $inSchema(\mathcal{N}, \theta)$ such that $I(x) = \{A(1), B(1)\}$ and $I(y) = \emptyset$. There are two quite different runs possible, that we describe next. Suppose that in the first transition of both runs node x is the recipient. This causes x to send both $A_{msg}(1)$ and $B_{msg}(1)$ to y. In the second transition of the first run we deliver only $A_{msg}(1)$ to y, which causes y to output $T(1)$. In the second transition of the second run, we deliver only $B_{msg}(1)$ to y, which causes y to create only the memory fact $B(1)$. Now, in no extension of the second run can the output fact $T(1)$ be created because every time we deliver $A_{msg}(1)$ to y the presence of memory fact $B(1)$ makes the valuation $v \mapsto 1$ unsatisfying for the rule of T. These two different runs clearly show inconsistent behavior of $(\mathcal{N}, \theta)$. □

It is interesting to investigate whether the property of consistency can be automatically checked for any $UCQ^\neg$-transducer network. We have the following *consistency decision problem*: given a transducer network $(\mathcal{N}, \Pi)$, decide if $(\mathcal{N}, \Pi)$ is consistent. Consistency for $UCQ^\neg$-transducer networks is undecidable, even under several restrictions:

Proposition 1.
(i) Consistency is undecidable for $UCQ^\neg$-transducer networks that are simple except that send rules don't have to be static.
(ii) Consistency is undecidable for $UCQ^\neg$-transducer networks that are simple except that in the positive dependency graph of each transducer the message relations may participate in cycles.

Proof. The proofs for *(i)* and *(ii)* are an adaptation of the techniques used by Deutsch et al. to prove their Theorems 3.9 and 3.7 respectively [11, 8]. □

One of the difficulties of the problem is that we need to verify a property of an infinite state system. Intuitively, the transducer network must be consistent on infinitely many distributed input database instances and even for a fixed input instance there are infinitely many configurations because there is no bound on the sizes of the message buffers. By imposing more restrictions we obtain decidability:

Theorem 2. *Consistency for simple transducer networks is decidable.*

A problem related to the above consistency problem is as follows. Let S be a multiset of pairs of the form (Υ, Π) where Π is a simple transducer over transducer schema Υ. Intuitively the transducers of S represent an abstract transducer network, except that no particular node set has been chosen yet. We can transform S into a transducer network $(\mathcal{N}, \theta)$ such that $|\mathcal{N}| = |S|$ and for each (Υ, Π) we have $cnt((\Upsilon, \Pi), S) = |\{x \in \mathcal{N} \mid \theta(x) = (\Upsilon, \Pi)\}|$. In words: we make nodes for the pairs in S and the number of nodes associated to some unique pair is the multiplicity of that pair in S. The related consistency problem is now to decide if the abstract transducer network represented by S is consistent for all node sets $\mathcal{N}$ chosen like this. Decidability for this problem follows from the previously given theorem and the fact that all $UCQ^\neg$ queries are generic, so the particular node set $\mathcal{N}$ does not matter as long as the cardinality is right.

A natural question to ask is what computations can be "expressed" with simple transducer networks. Let $(\mathcal{N}, \theta)$ be a consistent $UCQ^\neg$-transducer network. For a distributed input database instance I over schema $inSchema(\mathcal{N}, \theta)$ we can uniquely define a distributed output database instance $out(\mathcal{N}, \theta, I) = J$ over schema $outSchema(\mathcal{N}, \theta)$ as follows. The instance J is the function that maps every node $x \in \mathcal{N}$ to a database instance $J(x)$ over $\mathcal{D}^x_{out}$ that consists of all output facts that can be derived at node x on input I (during any run). From this viewpoint, the consistent transducer network $(\mathcal{N}, \theta)$ *computes* a distributed query $\mathcal{Q}_{\mathcal{N},\theta}$ for node-set $\mathcal{N}$ over distributed input database schema $inSchema(\mathcal{N}, \theta)$ and distributed output database schema $outSchema(\mathcal{N}, \theta)$. The set of all consistent $UCQ^\neg$-transducer networks can thus be viewed as a distributed query language. We have the following result:

Theorem 3. *The query language of consistent simple transducer networks captures the class of distributed queries expressible in $UCQ^\neg$.*

4. ON THE PROOF OF THEOREM 2

4.1 Single-node Transducer Networks

A transducer network $(\mathcal{N}, \theta)$ is called *single-node* if $|\mathcal{N}| = 1$. In this case the transducer on the single node x can only send messages to x. For a single-node transducer network it is more natural to consider a configuration to be a pair (s, b) with s a single set of input, output, memory and system facts and with b a single multiset of message facts. Here, s represents the state of the node and b its message buffer. Let $\Upsilon = (\mathcal{D}_{in}, \mathcal{D}_{out}, \mathcal{D}_{msg}, \mathcal{D}_{mem}, \mathcal{D}_{sys})$ be the single transducer schema of $(\mathcal{N}, \theta)$. The input of $(\mathcal{N}, \theta)$ will be considered to be a normal (nondistributed) database instance over $\mathcal{D}_{in}$ and for a database instance I over $\mathcal{D}_{in}$ the output $out(\mathcal{N}, \theta, I)$ will be considered to be a normal database instance over $\mathcal{D}_{out}$.

4.2 Fibers

The concept of *fibers* allows us to represent database instances over a given database schema where a specific part has already been filled in.[4] Let $\mathcal{D}$ be a database schema. A *fiber-base* for $\mathcal{D}$ is a pair $\boldsymbol{T} = (S, J)$ with S a subset of the relation names in $\mathcal{D}$ and J a database instance over $\mathcal{D}$ that consists of facts with a predicate in S. Denote $adom(\boldsymbol{T}) = adom(J)$. Let I be an instance over $\mathcal{D}$. We

[4]The name "fiber" comes from algebraic geometry.

say that I *has* or *belongs to* fiber-base $\boldsymbol{T} = (S, J)$ if for $R \in S$ we have $I(R) = J(R)$. Define the set

$$\mathit{fiber}(\mathcal{D}, \boldsymbol{T}) = \{ \text{ instance } I \text{ over } \mathcal{D} \mid I \text{ has fiber-base } \boldsymbol{T}\}.$$

We call $\mathit{fiber}(\mathcal{D}, \boldsymbol{T})$ the *fiber* of $\mathcal{D}$ for fiber-base $\boldsymbol{T}$. We use the symbol $\emptyset$ for the "empty" fiber-base $(\emptyset, \emptyset)$. Now, the set $\mathit{fiber}(\mathcal{D}, \emptyset)$ contains all database instances over $\mathcal{D}$. Note that for two different fiber-bases $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ for $\mathcal{D}$ it is possible that $\mathit{fiber}(\mathcal{D}, \boldsymbol{T}_1) \cap \mathit{fiber}(\mathcal{D}, \boldsymbol{T}_2) \neq \emptyset$.

4.3 Outline

In this section we give an overview of the techniques used to show that consistency is decidable for simple transducer networks (Theorem 2).

Let $(\mathcal{N}, \theta)$ be a simple transducer network for which we want to decide consistency.

1. In Section 4.4 it is shown how to "simulate" the behavior of $(\mathcal{N}, \theta)$ on a simple single-node network $(\mathcal{N}', \theta')$, such that $(\mathcal{N}, \theta)$ is consistent iff $(\mathcal{N}', \theta')$ is consistent on a fiber. Although we focus on simple transducer networks, this simulation could also be applied to more general $UCQ^\neg$ transducer networks.

2. Next, the results from Section 4.5 are used to establish a small model property for $(\mathcal{N}', \theta')$: if $(\mathcal{N}', \theta')$ is inconsistent on an input database instance of the fiber then it is inconsistent on an input database instance of the fiber whose active domain size is limited by the syntactic properties of $(\mathcal{N}', \theta')$. For this result we use all of our syntactic restrictions.

3. Finally, in addition to the small model property we have to impose a bound on the message buffers. Indeed, without this bound there are still an infinite number of configurations of $(\mathcal{N}', \theta')$ on the same input database instance. In Section 4.6 we construct a finite transition system that is "consistent" iff $(\mathcal{N}', \theta')$ is consistent on the fiber. For this result we also use all of our syntactic restrictions.

We obtain that consistency can be decided for the single-node network $(\mathcal{N}', \theta')$ that simulates $(\mathcal{N}, \theta)$. This implies that consistency for $(\mathcal{N}, \theta)$ is decidable.

4.4 Simulation by Single-Node Network

In this section we reduce the consistency problem of a simple transducer network $(\mathcal{N}, \theta)$ to the consistency problem of a simple single-node transducer network $(\mathcal{N}', \theta')$ that "simulates" the original transducer network in a way we will make more precise below. The single transducer of $(\mathcal{N}', \theta')$ is called the *simulator* and $(\mathcal{N}', \theta')$ will also be called the *simulator transducer network*.

Let $(\mathcal{N}, \theta)$ be a simple transducer network with shared message database schema $\mathcal{M}$. Like before, for $x \in \mathcal{N}$ we denote $\theta(x) = (\Upsilon^x, \Pi^x)$ and $\Upsilon^x = (\mathcal{D}^x_{in}, \mathcal{D}^x_{out}, \mathcal{D}^x_{msg}, \mathcal{D}^x_{mem}, \mathcal{D}^x_{sys})$. We start by defining a new transducer schema $\Upsilon = (\mathcal{D}_{in}, \mathcal{D}_{out}, \mathcal{D}_{msg}, \mathcal{D}_{mem}, \mathcal{D}_{sys})$ for the simulator as follows:

- $\mathcal{D}_{in} = \mathit{norm}^+(\mathit{inSchema}(\mathcal{N}, \theta))$,
- $\mathcal{D}_{out} = \mathit{norm}(\mathit{outSchema}(\mathcal{N}, \theta))$,
- $\mathcal{D}_{msg} = \bigcup_{x \in \mathcal{N}} \mathcal{M}^x$ with
  $\mathcal{M}^x = \{(R^x, k+1) \mid (R, k) \in \mathcal{M}\} \cup \{(\mathit{heartbeat}^x, 0)\}$,
- $\mathcal{D}_{mem} = \bigcup_{x \in \mathcal{N}} \{(R^x, k) \mid (R, k) \in \mathcal{D}^x_{mem}\}$.

The $\mathit{heartbeat}^x$ relations with $x \in \mathcal{N}$ will also be referred to as the "heartbeat" relations. For a relation name R^x in $\mathcal{D}_{msg}$ we call the x in the superscript the "addressee".

For $\mathcal{D}_{in}$ we define fiber-base $\boldsymbol{T} = (S, J)$ with $S = \{\mathit{All}^*\} \cup \{\mathit{Id}^x \mid x \in \mathcal{N}\}$ and $J = \{\mathit{All}^*(x), \mathit{Id}^x(x) \mid x \in \mathcal{N}\}$. Let I be a distributed input database instance over schema $\mathit{inSchema}(\mathcal{N}, \theta)$. Like in Section 2.4 the instance I can be naturally and unambiguously transformed into an input database instance $\mathit{sim}(I) \in \mathit{fiber}(\mathcal{D}_{in}, \boldsymbol{T})$, where "sim" stands for "simulated".

Let $(\mathcal{N}, \theta)$ and Υ be as above. We next formalize what properties the simulator has to satisfy with respect to how it represents the simulated node states and message buffers of the original transducer network $(\mathcal{N}, \theta)$. The property for the message buffers is more involved and this is detailed next. Let $x \in \mathcal{N}$ and $R^{(k)} \in \mathcal{M}$. Let $\boldsymbol{f} = R^x(a_0, a_1, \ldots a_k)$ be a fact, where $(R^x, k+1) \in \mathcal{D}_{msg}$. We write $\boldsymbol{f}^\downarrow$ to denote the fact $R(a_1, \ldots, a_k)$. Intuitively, this corresponds to projecting $\boldsymbol{f}$ to a fact that could be found in the message buffer of node x in the original transducer network $(\mathcal{N}, \theta)$.

Now, let m be a multiset of facts over $\mathcal{D}_{msg}$. Let $\boldsymbol{g}$ be a fact with predicate R and arity k. Thus, m contains message facts of the new simulator transducer schema and $\boldsymbol{g}$ is an original message fact of $(\mathcal{N}, \theta)$. We define the following set

$$\omega(R, x, \boldsymbol{g}, m) = \{\boldsymbol{f} \in \mathit{set}(m) \mid \mathit{pred}(\boldsymbol{f}) = R^x, \boldsymbol{f}^\downarrow = \boldsymbol{g}\}.$$

Intuitively, we find in m all facts whose projection is $\boldsymbol{g}$.

Let m be as previously defined and let m_2 be a multiset of facts over the original message schema $\mathcal{M}$. Let $x \in \mathcal{N}$. We say that m *represents* m_2 *for addressee* x if for $R^{(k)} \in \mathcal{M}$ and for facts $\boldsymbol{g}$ with predicate R and arity k it holds that

$$\mathit{cnt}(\boldsymbol{g}, m_2) = \sum_{\boldsymbol{f} \in \omega(R, x, \boldsymbol{g}, m)} \mathit{cnt}(\boldsymbol{f}, m).$$

Note that the special heartbeat messages are ignored.

Let $(\mathcal{N}', \theta')$ be a simple single-node transducer network such that the single transducer, denoted Π_{sim}, is over the schema Υ from above. Let I be a distributed input database instance over schema $\mathit{inSchema}(\mathcal{N}, \theta)$. Let ρ and σ respectively be a configuration of $(\mathcal{N}, \theta)$ on input I and a configuration of the network $(\mathcal{N}', \theta')$ on input $\mathit{sim}(I)$. We say that ρ is *simulated* by σ if

- for each $x \in \mathcal{N}$ and for each relation name R in $\mathcal{D}^x_{in} \cup \mathcal{D}^x_{out} \cup \mathcal{D}^x_{mem}$ we have $s^\rho(x)(R) = s^\sigma(R^x)$,
- for each $x \in \mathcal{N}$ it holds that b^σ represents $b^\rho(x)$ for addressee x.

We say that $(\mathcal{N}', \theta')$ *simulates* $(\mathcal{N}, \theta)$ if for any distributed input database instance I over schema $\mathit{inSchema}(\mathcal{N}, \theta)$ the following holds:

- For any run $\mathcal{R}$ of $(\mathcal{N}, \theta)$ on input I there is a run $\mathcal{S}$ of $(\mathcal{N}', \theta')$ on input $\mathit{sim}(I)$ such that $\mathit{last}(\mathcal{R})$ is simulated by $\mathit{last}(\mathcal{S})$.
- For any run $\mathcal{S}$ of $(\mathcal{N}', \theta')$ on input $\mathit{sim}(I)$ there is a run $\mathcal{R}$ of $(\mathcal{N}, \theta)$ on input I such that $\mathit{last}(\mathcal{R})$ is simulated by $\mathit{last}(\mathcal{S})$.
- For any run $\mathcal{R}$ of $(\mathcal{N}, \theta)$ on input I, any run $\mathcal{S}$ of $(\mathcal{N}', \theta')$ on input $\mathit{sim}(I)$ with $\mathit{last}(\mathcal{R})$ being simulated

by $last(\mathcal{S})$, for any extension $\mathcal{R}_2$ of $\mathcal{R}$ there is an extension $\mathcal{S}_2$ of $\mathcal{S}$ such that $last(\mathcal{R}_2)$ is simulated by $last(\mathcal{S}_2)$.

- For any run $\mathcal{S}$ of $(\mathcal{N}', \theta')$ on input $sim(I)$, any run $\mathcal{R}$ of $(\mathcal{N}, \theta)$ on input I with $last(\mathcal{R})$ being simulated by $last(\mathcal{S})$, for any extension $\mathcal{S}_2$ of $\mathcal{S}$ there is an extension $\mathcal{R}_2$ of $\mathcal{R}$ such that $last(\mathcal{R}_2)$ is simulated by $last(\mathcal{S}_2)$.

It is possible to construct $(\mathcal{N}', \theta')$ such that it simulates $(\mathcal{N}, \theta)$ according to the definition just given. We omit the technical details but the essential idea is that the $UCQ^\neg$ queries of the single transducer Π_{sim} of $(\mathcal{N}', \theta')$ are unions of the original $UCQ^\neg$ queries of all transducers of $(\mathcal{N}, \theta)$ where the $UCQ^\neg$ query of an original transducer Π^x with $x \in \mathcal{N}$ is modified to only read and modify output or memory relations with superscript x. Similarly, by taking unions of modified original queries, the sending rules in Π_{sim} for a message relation R^y in $\mathcal{D}_{msg}$ model the sending behavior for all the original transducers of $(\mathcal{N}, \theta)$ specifically for message relation R in $\mathcal{M}$ and the addressee y. Because the original transducers are simple it is possible to construct Π_{sim} to be simple.

Let $(\mathcal{N}, \theta)$, $\boldsymbol{T}$ and $(\mathcal{N}', \theta')$ be as above. We say that $(\mathcal{N}', \theta')$ is *consistent on fiber*$(\mathcal{D}_{in}, \boldsymbol{T})$ if $(\mathcal{N}', \theta')$ is consistent on all input database instances $I \in$ *fiber*$(\mathcal{D}_{in}, \boldsymbol{T})$. We have the following result:

PROPOSITION 4. *For every simple transducer network* $(\mathcal{N}, \theta)$ *there exists a simple single-node transducer network* $(\mathcal{N}', \theta')$ *that simulates it; in particular,* $(\mathcal{N}, \theta)$ *is consistent iff* $(\mathcal{N}', \theta')$ *is consistent on fiber*$(\mathcal{D}_{in}, \boldsymbol{T})$.

Since the input schema of the transducer Π_{sim} above is $norm^+(inSchema(\mathcal{N}, \theta))$ and its output schema is $norm($ $outSchema(\mathcal{N}, \theta))$, the simulator network $(\mathcal{N}', \theta')$ can be seen as computing the same distributed query $\mathcal{Q}_{\mathcal{N},\theta}$ as computed by $(\mathcal{N}, \theta)$.

4.5 Small Model Property

Let $(\mathcal{N}, \theta)$ be a simple single-node transducer network with transducer Π over transducer schema $\Upsilon = (\mathcal{D}_{in}, \mathcal{D}_{out}, \mathcal{D}_{msg}, \mathcal{D}_{mem}, \mathcal{D}_{sys})$. Let $\boldsymbol{T}$ be a fiber-base for $\mathcal{D}_{in}$. We have the following syntactically defined quantities:

- the length l of the longest path in the positive dependency graph of Π (measured in the number of edges),
- the largest number b of positive facts in any $CQ^\neg$ query of Π,
- the total number c of different output and memory facts of Υ that can be made with as many domain values as the maximal arity of an output relation,
- the maximal arity k of any relation in Υ,
- $t = |adom(\boldsymbol{T})|$.

Define the expression $sizeDom(\Pi, \boldsymbol{T}) = 2kcb^{(l+1)} + t$. We have the following small model property for consistency:

PROPOSITION 5. *If there is an instance* $I \in$ *fiber*$(\mathcal{D}_{in}, \boldsymbol{T})$ *on which* $(\mathcal{N}, \theta)$ *is not consistent then there exists* $J \in$ *fiber*$(\mathcal{D}_{in}, \boldsymbol{T})$ *such that* $(\mathcal{N}, \theta)$ *is not consistent on* J *and* $|adom(J)| \leq sizeDom(\Pi, \boldsymbol{T})$.

PROOF. We sketch the proof. Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two runs of $(\mathcal{N}, \theta)$ on input I that show that $(\mathcal{N}, \theta)$ is not consistent on I: there is an output fact $\boldsymbol{f}$ in $last(\mathcal{R}_1)$ that is not present in $last(\mathcal{R}_2)$ and this fact can not be created in any extension of $\mathcal{R}_2$ either. Denote $C = adom(\boldsymbol{f})$. Below, we transform $\mathcal{R}_1$ and $\mathcal{R}_2$ to new runs $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively that run on an input instance $J \subseteq I$ such that for $i \in \{1, 2\}$ configurations $last(\mathcal{S}_i)$ and $last(\mathcal{R}_i)$ contain the same output and memory C-facts.[5]

Let $i \in \{1, 2\}$. The first step involves collecting some input and message facts "used" in $\mathcal{R}_i$. Let $n = length(\mathcal{R}_i)$. We start with the n^{th} transition of $\mathcal{R}_i$. Define $\mathcal{C}_n$ to be the set of all output and memory C-facts that are created during the n^{th} transition. Let $\boldsymbol{g} \in \mathcal{C}_n$ be a fact. Denote $R = pred(\boldsymbol{g})$. Let ψ_R be the query for R in Π. By definition of $\mathcal{C}_n$ there must be a $CQ^\neg$ query $\varphi \in \psi_R$ that produces $\boldsymbol{g}$ during the n^{th} transition under a satisfying valuation Val. We look at *just one* such pair (φ, Val). We then remember (or "mark") all positive input and message facts that are needed by Val, more formally the set $Val(pos(\varphi))|_{in,msg}$. This is done for all facts of $\mathcal{C}_n$. Next, we proceed to transition $n-1$ of $\mathcal{R}_i$ and we repeat the same process for all output/memory C-facts created during this transition. Moreover, suppose that one of the message facts $\boldsymbol{h}$ that we previously marked during transition n is created during transition $n-1$. Let S be its predicate and let ψ_S be the query for S in Π. Now similarly, we choose just one pair (φ, Val) such that $\varphi \in \psi_S$ produces the fact $\boldsymbol{h}$ during transition $n-1$ under satisfying valuation Val. We again mark all positive input and message facts needed by Val. We are thus recursively marking messages needed for the creation of other messages. Such marking of messages is originally only initiated when we notice that during a transition an output/memory C-fact is created. This process of marking needed facts recursively needed for the creation of other facts is repeated for all transitions of $\mathcal{R}_i$, from transition n to 1. Since Π is inflationary every output and memory C-fact is derived only once. Now also using that Π is recursion-free it can be shown that the number of input facts that we mark is upper bounded by $cb^{(l+1)}$. Let $K_i \subseteq I$ denote the union of all marked input facts. After both $\mathcal{R}_1$ and $\mathcal{R}_2$ have been processed as described above we have obtained two sets K_1 and K_2. Now denote $\mu = adom(K_1) \cup adom(K_2) \cup adom(\boldsymbol{T})$. Define $J = I^\mu$. Observe that $|adom(J)| \leq 2kcb^{(l+1)} + t$.

Let $i \in \{1, 2\}$. It is possible to construct a projected run $\mathcal{S}_i$ for $\mathcal{R}_i$ that receives input J such that $last(\mathcal{S}_i)$ and $last(\mathcal{R}_i)$ contain the same output and memory C-facts. The intuition is that in run $\mathcal{S}_i$ we deliver the marked message facts from above, in the same order as in $\mathcal{R}_i$. Because all rules are message-positive, when we deliver just these marked messages there are no side-effects: no other output or memory C-facts are created. In this proof we also rely on message-boundedness and static send rules.

Next, one can similarly show that for any extension $\mathcal{S}_i'$ of $\mathcal{S}_i$ (still on input J) there exists an extension $\mathcal{R}_i'$ of $\mathcal{R}_i$ such that $last(\mathcal{S}_i')$ and $last(\mathcal{R}_i')$ again contain the same output and memory C-facts. To obtain extension $\mathcal{R}_i'$ it is sufficient to deliver the same messages at the end of $\mathcal{R}_i$ as during every corresponding transition in the extension $\mathcal{S}_i'$. Note that since $\boldsymbol{f}$ is an output C-fact we have that $\boldsymbol{f}$ is present

[5]The technique is inspired by *pseudoruns* from [10], although it was adapted to deal with the consistency problem and to deal with message buffers (multisets).

in $last(\mathcal{S}_1)$ and *not* present in $last(\mathcal{S}_2)$. If there exists an extension $\mathcal{S}_2'$ of $\mathcal{S}_2$ such that $\boldsymbol{f}$ is present in $last(\mathcal{S}_2')$ this would imply that there is an extension $\mathcal{R}_2'$ of $\mathcal{R}_2$ such that $\boldsymbol{f}$ is present in $last(\mathcal{R}_2')$, a contradiction with our assumption about $\mathcal{R}_2$. We conclude that $\mathcal{S}_1$ and $\mathcal{S}_2$ show that $(\mathcal{N}, \theta)$ is not consistent on input J. □

4.6 Bounded Buffer Property

Let $(\mathcal{N}, \theta)$ be a simple single-node transducer network, with transducer Π over schema Υ. Denote $\Upsilon = (\mathcal{D}_{in}, \mathcal{D}_{out}, \mathcal{D}_{msg}, \mathcal{D}_{mem}, \mathcal{D}_{sys})$. Let $\boldsymbol{T}$ be a fiber-base for $\mathcal{D}_{in}$. We show how to construct a finite transition system that reflects the original runs of $(\mathcal{N}, \theta)$ and that is "consistent" iff $(\mathcal{N}, \theta)$ is consistent on input I. This will be made more precise below. The crucial step here is imposing a bound on the message buffers. First, in Section 4.6.1 we show how to construct a finite transition system for an arbitrary bound on the message buffers. In Section 4.6.2 some additional definitions are given. In Section 4.6.3 the result is proven.

4.6.1 Finite Transition System

Let $(\mathcal{N}, \theta)$, Π and Υ be as above. Let $I \in fiber(\mathcal{D}_{in}, \boldsymbol{T})$ and let $B \in \mathbb{N}$. We describe the construction of a finite transition system $\mathcal{F}_{I,B}$ for I that represents the runs of $(\mathcal{N}, \theta)$ on input I in which the message buffer can contain at most B messages. The configurations of $\mathcal{F}_{I,B}$ are the configurations τ of $(\mathcal{N}, \theta)$ on input I where $|b^\tau| \leq B$. The unique start configuration is denoted $start(\mathcal{F}_{I,B})$. We redefine global transitions to respect the buffer bound B by using *lossy* message buffers. Because we are specifically working with a single-node network (Section 4.1), a global transition of $\mathcal{F}_{I,B}$ is a 3-tuple (τ_1, m, τ_2) with τ_1 and τ_2 configurations and $m \sqsubseteq b^{\tau_1}$ such that there is a message instance J_{snd} for which *(i)* $s^{\tau_1}, set(m) \Rightarrow s^{\tau_2}, J_{snd}$ is a local transition of Π (state condition) and *(ii)* $b^{\tau_2} \sqsubseteq (b^{\tau_1} \setminus m) \cup J_{snd}$ (buffer condition). We do not require that $b^{\tau_2} = (b^{\tau_1} \setminus m) \cup J_{snd}$ because we allow the dropping of messages, in order to satisfy the buffer bound B. Messages that are dropped in the finite transition system $\mathcal{F}_{I,B}$ express the indefinite delay of the corresponding message in the original transducer network. Intuitively, we don't really "lose" messages because the send-rules are static, which implies that if a message can be sent it can be sent forever.

For a path $\mathcal{T}$ in $F_{I,B}$ we write $last(\mathcal{T})$ to denote the last configuration in $\mathcal{T}$. Naturally, we call $\mathcal{F}_{I,B}$ *consistent* if for all paths $\mathcal{T}_1$ and $\mathcal{T}_2$ in $\mathcal{F}_{I,B}$ starting from $start(\mathcal{F}_{I,B})$, for all output facts $\boldsymbol{f}$ in $last(\mathcal{T}_1)$ there is an extended path $\mathcal{T}_2'$ of $\mathcal{T}_2$ such that $\boldsymbol{f}$ is present in $last(\mathcal{T}_2')$. Deciding if $\mathcal{F}_{I,B}$ is consistent can be done in time polynomial in function of the size of $\mathcal{F}_{I,B}$.

4.6.2 Derivation Trees

Let $(\mathcal{N}, \theta)$, Π, Υ and $\boldsymbol{T}$ be as above. We now present definitions that help us reason about which rules of Π are used to create an output or memory fact during a run of $(\mathcal{N}, \theta)$. Let $\boldsymbol{f}$ be a fact with predicate in $\mathcal{D}_{out} \cup \mathcal{D}_{mem}$. A *full derivation tree* T in Π for $\boldsymbol{f}$ is a tuple (N, E, φ, l, V) such that:

- N and $E \subseteq N \times N$ form the nodes and parent-to-child edges of an unordered tree. We write r_T to denote the root of T. We write $int(T)$ to denote the set of internal nodes of T (nodes with children). For $x \in int(T)$ we define $child_T(x) = \{y \in N \mid (x, y) \in E\}$. We abbreviate $child(T) = N \setminus \{r_T\}$, the nodes that are a child of another node.
- φ is a function that maps every $x \in int(T)$ to a $CQ^\neg$ query $\varphi(x)$ used in Π.
- l is a function that labels every node $x \in child(T)$ with parent y with a literal in the body of $\varphi(y)$. We require that for every literal in the body of $\varphi(y)$ there is precisely one child of y labelled with that literal.
- Let $x \in child(T)$. If $l(x)$ has its predicate in $\mathcal{D}_{in}$ or $l(x)$ is negative then x must be a leaf. For $x \in int(T) \cap child(T)$ it must be that $head(\varphi(x))$ has the same predicate (and arity) as $l(x)$.
- V is a function that maps $x \in int(T)$ to a valuation $V(x)$ for rule $\varphi(x)$ such that the inequalities $ineq(\varphi(x))$ are satisfied under $V(x)$.
- For $x \in int(T)$, denote $f_T(x) = V(x)(head(\varphi(x)))$. For a leaf node x with parent y we denote $f_T(x) = V(y)(+l(x))$. Thus for all $x \in N$ we have that $f_T(x)$ is a fact. For $x \in child(T) \cap int(T)$ with parent y we require that $f_T(x) = V(y)(l(x))$. Lastly, we require that $f_T(r_T) = \boldsymbol{f}$.

For a derivation tree $T = (N, E, \varphi, l, V)$ we write $e_T = e$ for $e \in \{N, E, \varphi, l, V\}$. We define

$$\alpha(T) = \{x \in int(T) \mid pred(f_T(x)) \in names(\mathcal{D}_{out} \cup \mathcal{D}_{mem})\}.$$

Let $\boldsymbol{f}$ be an output or memory fact. Intuitively, a full derivation tree T for $\boldsymbol{f}$ represents one possible way in which the fact $\boldsymbol{f}$ can be derived using the rules of Π. Intuitively, according to T the facts $\{f_T(x) \mid x \in int(T)\}$ must be produced. Let $R = pred(\boldsymbol{f})$. We also say that T is a derivation tree for relation R.

Let T_1 and T_2 be two full derivation trees. We say that trees T_1 and T_2 are *structurally equivalent* if they are isomorphic when ignoring the valuations V_{T_1} and V_{T_2}.

Let T be a full derivation tree in Π for a fact $\boldsymbol{f}$. A function $\kappa : int(T) \to \mathbb{N} \setminus \{0\}$ is called a *scheduling* of T if for any two nodes $x, y \in int(T)$ if y is an ancestor of x then $\kappa(x) < \kappa(y)$. Intuitively, for $x \in int(T)$ the number $\kappa(x)$ represents the transition in which the fact $f_T(x)$ must be produced. For $x \in int(T)$ we write $height(x, T)$ to denote the height of the subtree of T rooted at node x. For $x \in int(T) \setminus \alpha(T)$ define $anc(x)$ to be the closest ancestor of x in $\alpha(T)$ and define $edgeCnt(x)$ to be the number of edges on the simple path from x to $anc(x)$. We define a *canonical scheduling* κ_T of T as follows. For every $x \in \alpha(T)$ we define $\kappa_T(x) = height(x, T)$ and for $x \in int(T) \setminus \alpha(T)$ we define $\kappa_T(x) = height(anc(x), T) - edgeCnt(x)$. Note that $\max(img(\kappa_T)) \leq height(T)$.

4.6.3 Buffer Bound

Let $(\mathcal{N}, \theta)$, Π, Υ and $\boldsymbol{T}$ be as above. We specify a number $B(\Pi, \Upsilon)$ that will be used as the message buffer bound for the finite transition system. Referring to the proof sketch of Proposition 5, it can be shown that the number of messages that have to be present in the message buffers of a projected run $\mathcal{S}_i$ is bound by some natural number B_1 that only depends on the syntactic properties of Π and Υ. Now, let F be a maximal set of full derivation trees for all output relations in $\mathcal{D}_{out}$ such that no two trees are structurally equivalent.

Because Π is recursion-free we know that the size of F and the size of each tree in F are dependent only on the syntactic properties of Π. For a tree $T \in F$ define $msgWidth(T)$ as follows:

$$msgWidth(T) = \max \bigcup_{i=1}^{height(T)} \{ \sum_{\substack{x \in int(T),\\ \kappa_T(x) = i}} |pos(\varphi(x))|_{msg}| \}.$$

Intuitively, $msgWidth(T)$ is the maximum number of messages in T that have to be delivered during the same transition (as specified by κ_T). Let $B_2 = \max\{msgWidth(T) \mid T \in F\}$ and define $B(\Pi, \Upsilon) = \max\{B_1, B_2\}$. Overall, note that $B(\Pi, \Upsilon)$ is defined only in terms of the syntactical properties of Π and Υ. We have the following result:

PROPOSITION 6. *Let $I \in fiber(\mathcal{D}_{in}, \boldsymbol{T})$. The finite transition system $\mathcal{F}_{I,B(\Pi,\Upsilon)}$ is consistent iff $(\mathcal{N}, \theta)$ is consistent on input I.*

PROOF. We sketch the proof. Abbreviate $B = B(\Pi, \Upsilon)$.

First, we show that if $(\mathcal{N}, \theta)$ is not consistent on input I then $\mathcal{F}_{I,B}$ is not consistent. Because $(\mathcal{N}, \theta)$ is not consistent on input I there are two runs $\mathcal{R}_1$ and $\mathcal{R}_2$ that show the inconsistency: without loss of generality we may assume that there is an output fact $\boldsymbol{f}$ in $last(\mathcal{R}_1)$ such that $\boldsymbol{f}$ is not present in $last(\mathcal{R}_2)$ and there is no extended run of $\mathcal{R}_2$ in which that fact can be output. Denote $C = adom(\boldsymbol{f})$. Using the same techniques mentioned in the proof of Proposition 5, we can project runs $\mathcal{R}_1$ and $\mathcal{R}_2$ to runs $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively such that for $i \in \{1, 2\}$ the configurations $last(\mathcal{R}_i)$ and $last(\mathcal{S}_i)$ contain the same set of output and memory C-facts. Let $i \in \{1, 2\}$. It can be shown that the choice of the bound B allows us to trace $\mathcal{S}_i$ in $\mathcal{F}_{I,B}$ as path $\mathcal{T}_i$ starting from $start(\mathcal{F}_{I,B})$ such that $last(\mathcal{T}_i)$ and $last(\mathcal{S}_i)$ (or $last(\mathcal{R}_i)$) contain the same set of output and memory C-facts. We use message-boundedness, message-positiveness and static send rules for this proof. Next using similar techniques, we can show that if $\mathcal{T}_2$ can be extended in $\mathcal{F}_{I,B}$ to a path $\mathcal{T}_2'$ such that $\boldsymbol{f}$ is present in $last(\mathcal{T}_2')$ then $\mathcal{R}_2$ can be extended to a run $\mathcal{R}_2'$ in which $\boldsymbol{f}$ can be output, a contradiction. We find that paths $\mathcal{T}_1$ and $\mathcal{T}_2$ together are proof that $\mathcal{F}_{I,B}$ is not consistent.

Next, we show that if $(\mathcal{N}, \theta)$ is consistent on input I then $\mathcal{F}_{I,B}$ is consistent. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be paths in $\mathcal{F}_{I,B}$ starting from $start(\mathcal{F}_{I,B})$. Let $\boldsymbol{f}$ be an output fact in $last(\mathcal{T}_1)$ that is not present in $last(\mathcal{T}_2)$. We show that $\mathcal{T}_2$ can be extended to $\mathcal{T}_2'$ such that $\boldsymbol{f}$ is present in $last(\mathcal{T}_2')$. Denote $C = adom(\boldsymbol{f})$. Using message-boundedness, message-positiveness and static send rules, we can show that there are runs $\mathcal{R}_1$ and $\mathcal{R}_2$ of $(\mathcal{N}, \theta)$ on input I such that for $i \in \{1, 2\}$ the configurations $last(\mathcal{R}_i)$ and $last(\mathcal{T}_i)$ contain the same output and memory C-facts. This implies that $\boldsymbol{f}$ is in $last(\mathcal{R}_1)$ and not in $last(\mathcal{R}_2)$. Now, since $\boldsymbol{f}$ can be output by $(\mathcal{N}, \theta)$ on input I, we can consider a maximal set G of concrete derivation trees for $\boldsymbol{f}$ that can be extracted from all possible runs of $(\mathcal{N}, \theta)$ on input I such that no two trees in G are structurally equivalent. Because Π is recursion-free we know that G is finite and the size of each tree in G is dependent only on some syntactic properties of Π. We fix some random order on G: $T_1, \ldots, T_p$. Let $T_i \in G$ and $h_i = height(T_i)$. For $j \in \{1, \ldots, h_i\}$ define the following set of message facts (using canonical scheduling κ_{T_i}):

$$M_j^i = \bigcup_{\substack{x \in int(T_i),\\ \kappa_{T_i}(x) = j}} V_{T_i}(x)\big(pos(\varphi_{T_i}(x))|_{msg}\big).$$

Using inflationarity, message-boundedness, message-positiveness and static send rules we can show that $\mathcal{R}_2$ can be extended to a run $\mathcal{R}_2'$ such that $\boldsymbol{f}$ is present in $last(\mathcal{R}_2')$, by delivering the following message sets in order: $M_1^1, \ldots, M_{h_1}^1$, $\ldots$, $M_1^p, \ldots, M_{h_p}^p$. Next, we can show that by definition of message buffer bound B it is possible to extend path $\mathcal{T}_2$ to path $\mathcal{T}_2'$ by delivering the same message sets in order such that configurations $last(\mathcal{T}_2')$ and $last(\mathcal{R}_2')$ have the same output and memory C-facts. This implies that $\boldsymbol{f}$ is present in $last(\mathcal{T}_2')$. □

4.7 Decision Procedure

COROLLARY 7. *The consistency problem for simple transducer networks is in EXPSPACE.*

PROOF. (Sketch.) We actually show membership in EXPSPACE of the *in*consistency problem. By Proposition 4, we first translate to a single-node network. The resulting simulating transducer is of polynomial size and the translation can be done in polynomial time. By Proposition 5, a single-node simple transducer network is inconsistent iff it is inconsistent on some instance I of at most exponential size. Since non-deterministic EXPSPACE equals EXPSPACE, we may guess such an instance I and check inconsistency on I. By Proposition 6, inconsistency on I is equivalent to inconsistency of a finite transition system $\mathcal{F}$. Since I is of at most exponential size, for each configuration ρ of $\mathcal{F}$, the database instance part s^ρ of ρ is of at most exponential size as well. Moreover, by the buffer bound provided on $\mathcal{F}$, the buffer part b^ρ may be restricted to be of at most exponential size as well. We now check inconsistency of $\mathcal{F}$ by guessing two reachable configurations ρ_1 and ρ_2 and an output fact $\boldsymbol{f}$ in ρ_1. Using the standard nondeterministic algorithm for reachability, these elements can be guessed in exponential space. We then proceed to check that there does not exist a configuration ρ_3 reachable from ρ_2 that contains the output fact $\boldsymbol{f}$. Using the Immerman-Szelepscényi algorithm [17], this is again possible in nondeterministic exponential space. Whenever the reachability algorithm needs to check if there is an edge in $\mathcal{F}$ between two given configuration, this check can be performed by guessing a valuation for each produced message, memory, and output fact, so that an appropriate rule can fire. This is again possible in nondeterministic time polynomial in the size of the transducer and the sizes of the configurations, hence again in exponential space. □

5. ON THE PROOF OF THEOREM 3

Let $\mathcal{N}$ be a nodeset. Let $\mathcal{Q}$ be a distributed query for $\mathcal{N}$ over distributed input database schema $(\mathcal{N}, \eta_{in})$ and distributed output database schema $(\mathcal{N}, \eta_{out})$ that is expressible in $UCQ^\neg$, namely, as a set Ψ of $UCQ^\neg$ queries: one $UCQ^\neg$ query for each output relation in $norm(\mathcal{N}, \eta_{out})$ and all having the same input database schema $norm^+(\mathcal{N}, \eta_{in})$. We can show in general that there is a simple and consistent transducer network $(\mathcal{N}, \theta)$ that computes $\mathcal{Q}$. We illustrate this with an example:

Example 3. Consider the nodeset $\mathcal{N} = \{x, y\}$. The distributed input database schema $(\mathcal{N}, \eta_{in})$ and distributed output database schema $(\mathcal{N}, \eta_{out})$ are defined as: $\eta_{in}(x) = \{A^{(1)}\}$, $\eta_{in}(y) = \{C^{(1)}\}$, $\eta_{out}(x) = \{S^{(1)}\}$, $\eta_{out}(y) = \{T^{(1)}\}$. The distributed query $\mathcal{Q}$ for $\mathcal{N}$ over distributed input schema $(\mathcal{N}, \eta_{in})$ and distributed output database schema $(\mathcal{N}, \eta_{out})$ is specified in $UCQ^\neg$ as follows:

$$\{S^x(v) \leftarrow A^x(v), \neg C^y(v)\},$$

$$\{T^y(v) \leftarrow A^x(v), C^y(v)\}.$$

We now specify the simple transducer network $(\mathcal{N}, \theta)$ that computes $\mathcal{Q}$, with the following shared message database schema $\mathcal{D}_{msg}$:

$$\mathcal{D}_{msg} \quad = \quad \{(A^x_{msg}, 1), (C^{y,?}_{msg}, 1), (C^{y,\neg}_{msg}, 1)\}.$$

The transducer Π^x has the following $UCQ^\neg$ queries

$$\begin{aligned} \{S(v) &\leftarrow A(v), C^{y,\neg}_{msg}(v)\}, \\ \{A^x_{msg}(n, v) &\leftarrow A(v), All(n)\}, \\ \{C^{y,?}_{msg}(n, v) &\leftarrow A(v), All(n)\}. \end{aligned}$$

The transducer Π^y has the following $UCQ^\neg$ queries

$$\begin{aligned} \{T(v) &\leftarrow A^x_{msg}(v), C(v)\}, \\ \{C^{y,\neg}_{msg}(n, v) &\leftarrow C^{y,?}_{msg}(v), \neg C(v), All(n)\}. \end{aligned}$$

Intuitively, node x broadcasts the contents of its local input relation A using the message relation A^x_{msg}. Node y simply uses these messages to compute the intersection of relation A on x and its own relation C. Next, node x wants to ensure the absence of C-facts at y. In order to that, node x broadcasts the *probe* message $C^{y,?}_{msg}$. For instance, when y receives $C^{y,?}_{msg}(a)$ from x and indeed the fact $C(a)$ is not specified at node y, then y broadcasts $C^{y,\neg}_{msg}(a)$. When node x then receives $C^{y,\neg}_{msg}(a)$ it knows that y does not have the fact $C(a)$. Broadcasting is used because a node does not know the value used to identify the other node. In general, negation on the input (or system) relations of another node can be implemented in the manner shown by this example. □

The converse direction of Theorem 3, that the query $\mathcal{Q}_{\mathcal{N},\theta}$ computed by a consistent simple transducer network $(\mathcal{N}, \theta)$ is always expressible in $UCQ^\neg$, is quite intricate to prove. We only provide a rough sketch omitting the details. The crucial insight is that all output facts can already be produced in a canonical run of fixed length. This canonical run is formed by the set of all possible derivation trees for output facts, which is finite up to isomorphism by recursion-freeness. Messages are delivered according to the canonical schedulings (Section 4.6.2). Indeed, consider a fact $\boldsymbol{f}$ output by $\mathcal{Q}_{\mathcal{N},\theta}$. By consistency, we know that $\boldsymbol{f}$ can be produced in an extension of the canonical run. The concrete derivation tree T that produces $\boldsymbol{f}$ also occurs in the canonical run. The negative literals in T succeed, which means that certain memory facts are absent in the extended run. Since the transducer is inflationary, these memory facts are also absent in the canonical run; hence, $\boldsymbol{f}$ will already be output there. The second step of the proof then consists of observing that runs of fixed length can be expressed by existential first-order formulas (and then, equivalently, in $UCQ^\neg$).

6. CONCLUSION AND DISCUSSION

We have shown that under five restrictions: recursion-freeness, inflationarity; message-positivity; static message sending; and message-boundedness, one obtains decidability in EXPSPACE of eventual consistency for networks of relational transducers with unions of conjunctive queries with negation. As already mentioned in the Introduction, a topic for further work is to investigate whether decidability can be retained while lifting or relaxing the restrictions of recursion-freeness, inflationarity, and message-positivity. We also do not yet know if the problem is EXPSPACE-complete.

There seem to be several reasonable ways to formalize the intuitive notion of eventual consistency. In contrast to our current formalization, a stronger view of eventual consistency [1, 6] is to require that on every input, all infinite fair runs produce the same set of output facts. Again, a number of reasonable fairness conditions could be considered here; a rather standard one would be to require that every node performs a transition infinitely often, and that every message fact that is sent is eventually delivered. When a transducer network is consistent in this stronger sense, it is also in the confluence sense, but the other implication is not obvious. Since eventual consistency is indeed meant to be a very weak guarantee [19], it deserves further research to better understand different consistency and fairness requirements.

Acknowledgments

We would like to thank Joseph M. Hellerstein and William R. Marczak (UC Berkeley) for an inspiring email correspondence.

7. 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 283–292. ACM Press, 2011.

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

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

[4] 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.

[5] P. Alvaro, W. Marczak, et al. Dedalus: Datalog in time and space. Technical Report EECS-2009-173, University of California, Berkeley, 2009.

[6] 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.

[7] T.J. Ameloot and J. Van den Bussche. Deciding eventual consistency of a simple class of relational transducer networks (technical report). `http://alpha.uhasselt.be/tom.ameloot/tom/doc/dp7@3c4/icdt2012report.pdf`, 2011.

[8] A. Deutsch. Personal communication.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

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

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

[16] J.A. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In A. Gill and T. Swift, editors, *Proceedings 11th International Symposium on Practical Aspects of Declarative Languages*, volume 5419 of *Lecture Notes in Computer Science*, pages 76–90, 2009.

[17] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

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

[19] W. Vogels. Eventual consistency. *Communications of the ACM*, 52(1):40–44, 2009.