Datalog in Time and Space, Synchronously

Matteo Interlandi, Letizia Tanca, and Sonia Bergamaschi

Università degli Studi di Modena and Reggio Emilia,
{matteo.interlandi,sonia.bergamaschi}@unimore.it
Politecnico di Milano,
tanca@elet.polimi.it

Abstract. Motivated by recent developments in the formalization of Datalog-based languages for highly distributed systems [16], in this paper we introduce the operational and declarative semantic of a version of Datalog$^{\neg}$ augmented with a notion of time and space [5] in order to formally define how computation can be performed in distributed synchronous systems.

1 Introduction

Nowadays we are living the "*end of the Moores low as we know it*": increasing performance cannot be achieved just increasing hardware speed; instead, concurrent and parallel computation must be exploited. Thus, in the last few years, new paradigms such as cloud computing, frameworks – MapReduce [15] for example, and systems – NoSQL databases – are put at work to help programmers in developing scalable applications which exploit distributed systems. We think that Datalog is in a very favorable position to provide an important contribution to the development of the aforementioned technologies: (i) its intrinsically (embarrassingly) parallel nature spontaneously lends itself as a solid basis to develop more complex languages for modern parallel and distributed applications [12, 11]; (ii) its logical foundation provides the theoretical background that permits to formally specify and analyze complex distributed systems [6]; (iii) Datalog can act both as *Data Definition Language* (DDL) and as *Data Manipulation Language* (DML) therefore providing a valuable tool for designing flexible data-driven applications as the ones that are currently running on top of NoSQL key-values stores.

We are however aware that limitations still exist in the current literature concerning the formal models headed by the above discussed new trends. Our aims is, starting from the research on active and deductive databases developed during the '90s, to revise it in order to develop new techniques that can be employed to solve the challenges of today's highly distributed systems. The role of our present work is then to precisely define what a *synchronous distributed system* is and provides the semantics for such kind of systems for a version of Datalog specifically tailored for distributed programming. Indeed, in today settings, asynchronous systems are highly preferable compared to synchronous systems, however multiple models can be embedded over the latter, thanks to its

assumptions which are able to considerably simplify the analysis of such models. Our work is motivated by the following three use cases:

1. Active deductive databases [30, 19] were developed in centralized settings in which updates must be issued to a unique central database. Our aim is to enhance this model in order to develop the semantics of synchronous networks of partitioned and/or replicated databases.
2. With the advent of multicore architectures, programs that before were running sequentially on a single CPU, now must be able to exploit the parallelism provided by the new architecture. Indeed multicore machines can be seen as synchronous shared-memory computers, and by formalizing such systems we would then be able to characterize how to implement parallel programs on multicore computers.
3. In the *Massive Parallel* (MP) model introduced in [18] computation proceeds by *steps* that are performed in parallel by cluster of machines on which the same program is running. Each step can be divided into three phases, namely a *broadcast phase*, a *communication phase* and a *computation phase*. This model actually is a particular instantiation of a synchronous system. In addition, the MP model is stickily related with the MapReduce model (MR): the *map* and the *shuffle jobs* are the communication steps, while the *reduce jobs* are the computation steps [18]. As a consequence, by developing a semantics for synchronous systems we would be able to seamlessly embed in our framework the MP and the MR computation models.

Related Work In these years a renew interest in Datalog is arising, especially pushed by new emerging trends such as declarative packet routing [21], declarative overlay networks [20], network provenance [31], Map-Repuce [4], web data management [1], and cyber physical systems [29]. Our work in particular is motivated by [16], which discusses how Datalog$^\neg$ programs are suited to express logically distributed systems and their properties. The versions of Datalog illustrated in this paper, for what concern the time dimension is heavily based on Dedalus [5] and Statelog [22], while for the space dimension we are more close to the partitioned relations prospective of NDlog [21] then to the channels view of Dedalus [5]. As for the operational and model-theoretic semantics of synchronous systems, we largely take inspiration from [1, 6, 7] where relational transducers [3] have been employed to specify the semantics of reliable asynchronous systems. Our concept of DSR is heavily based on the *Distributed Shared Memory* (DSM) [8, 25, 24].

Contributions and Organization Our main contributions are the followings: (i) inspired by *Distributed Shared Memory* (DSM) we provide a prospective on how tuples can be communicated between distributed databases by employing *Distributed Shared Relation*s (DSRs); (ii) we introduce a new type of *Relational Transducer* [3] and *Transducer network* [6] specifically tailored for synchronous

distributed systems and, (iii) we present the model-theoretic semantics of distributed Datalog programs for such systems by showing the equivalence betweens the centralized and the distributed execution.

The paper is organized as follows: in Section 2 we introduce some preliminaries on Datalog$^\neg$ and Datalog$^\neg$ augmented with a notion of time. In Section 3 we enhance the syntax of the language by introducing the concept of *distributed shared relations*. In Section 4 we describe what a synchronous distributed systems is, and we provide the operational and declarative semantics of Datalog$^\neg$ augmented with time and space in such setting.

2 Preliminaries

Given a finite set of relation names **relname**, we denote with **R** a *database schema* composed by a set of relation names $R \in$ **relname**. In addition, we denote with **dom** a countable infinite set of *constants*, and with **var** an infinite set of *variables* used to range over the elements of **dom**. We consider **dom**, **var** and **relname** as disjoined from one another. We associate to each relation name R a function $arity : R \to \mathbb{N}_0$. Given a relation name R and related arity k, a *free tuple* over R is an ordered k-tuple composed of only constants and variables, or, more precisely, an element of the Cartesian product $(\mathbf{var} \cup \mathbf{dom})^k$. If $\bar{u}$ denotes a free tuple over a relation R with arity k, we use $\bar{u}(i)$ to refer to the i-th coordinate of $\bar{u}$, with $i \leq k$. An *atom* over R is an expression in the form $R(u_1, \ldots, u_k)$ where each u_i is referred to as a *term*. We sometimes write $R(\bar{u})$ to refer to an atom and $\bar{u}(i)$ to refer to the i-th term. In the following we will use interchangeably the terms *relations* and *predicates*. A *literal* is an atom – in this case we refer to it as *positive* – or the negation of an atom. Now, a *ground atom* is an atom containing only constant terms – i.e., $u_i \in$ **dom**. A *database instance* is a finite set of facts **I** over the relations of **R**, while a *relation instance* $I_R \subseteq \mathbf{I}$ is a set of facts defined over R, with $R \in \mathbf{R}$. The set of all constant appearing in a given database instance **I** is called *active domain*, and is represented by $adom(\mathbf{I})$.

A Datalog$^\neg$ rule is an expression in the form:

$$H(\bar{w}) \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n), \neg C_1(\bar{v}_1), \ldots, \neg C_m(\bar{v}_m). \tag{1}$$

where H, B_i, C_j are relation names and $\bar{w}, \bar{u}_i, \bar{v}_j$ are tuples of appropriate arity. As usual $H(\bar{w})$ is referred to as the *head*, and $B_1(\bar{u}_1), \ldots, C_m(\bar{v}_m)$ as the *body*. If $m = 0$ the rule is *positive* and expresses a *definite clause*, while if $m = n = 0$ the rule is expressing a *fact* (*clause*). Note that predicates appearing in the body of a rule can also be used in the head of the same rule to produce *recursive* computation. We allow *built-in* predicates to appear in the bodies of rules, i.e., we allow relation names such as $=, \neq, \leqslant, <, \geqslant$, and $>$, each of them having their natural meaning. To be consistent with the standard notation, we use the infix notation and write $X \geq Y$ instead of $\geq(X, Y)$. We also allow *aggregate relations* in positive rule-heads in the form $R(\bar{u}, \Lambda < w >)$, with Λ one of the usual aggregate functions, w a variable from the body, and $\bar{u}$ is a list thereof,

called the grouping variables [27]. In this paper we assume each rule to be *safe*, i.e., every variable occurring in a rule-head, negative literal or built-in input variable, appears in at least one positive literal of the rule body [13]. As a consequence, we assume all facts to be ground because all non-ground facts are not safe. A *Datalog¬ program* Π is then a set of safe rules. The *intensional* part of the database schema, namely $idb(\mathbf{R}, \Pi)$, is the subset of the database schema $\mathbf{R}$ containing all relations that appear in at least one rule-head in Π, while with $edb(\mathbf{R}, \Pi) = \mathbf{R} - idb(\mathbf{R}, \Pi)$ we refer to as the *extensional database*. If the program is composed by only positive rules, we refer to it as *positive*, while if negation appears only in front of *edb* relations – i.e., $C_j \subseteq edb(\mathbf{R}, \Pi)$ – we will denote it as *semi-positive*. Positive and semi-positive programs are special and very favorable cases of Datalog¬ programs [2].

As an introductory example, we use the program depicted in Listing 1.1 where we employ an *edb* relation `link` containing tuples in the form `(X,Y,D)` to specify the existence of a directed link with length `D` between a source node `X` and a destination node `Y`. In addition, we employ two intensional relations `path` and `shortest` respectively defining the transitive closure of the `link` relation, and the shortest(s) among the paths existing between each pair of source-destination nodes. To identify the shortest(s) path we make use of the `shortestDistance` aggregate relations which stores the minimum distance for each source-destination pair. These four relations `link`, `path`, `shortest` and `shortestDistance` will remain the same during the entire paper.

```
r1:   path(X,Y,D):-link(X,Y,D).
r2:   path(X,Z,D):-link(X,Y,D1),path(Y,Z,D2),D=D1+D2.
r3:   shortestDistance(X,Y,min<D>):-path(X,Y,D).
r4:   shortest(X,Y,D):-path(X,Y,D),shortestDistance(X,Y,D).
```

Listing 1.1. Simple Recursive Datalog¬ Program

2.1 Adding Time

The purpose of this paper is to define a language able to model programs for synchronous distributed systems. These systems are not static, but evolving with time. Therefore it will be useful to enrich our schema with a notion of time in order to logically reason on how database instances change with the progress of time. Considering time as isomorphic to the set of natural numbers, we add a new set of time variables **tvar** disjoined from **var** and ranging over $\mathbb{N}_0$. Then, a new database schema $\mathbf{R}_T$ is defined starting from $\mathbf{R}$ and incrementing the arity of each relation $R \in \mathbf{R}$ by one. By convention, we use symbols t, t_0, ..., t_n for elements in $\mathbf{tvar} \cup \mathbb{N}_0$ to represent the new extra term called *time-step identifier*. As a consequence, a tuple over $\mathbf{R}_T$ has now the (reified) form $R(u_1, \ldots, u_k, t)$ – where the time-step identifier occupies always the $(k+1)$-th position in a tuple – or equivalently the (suffix) form $R(u_1, \ldots, u_k)@t$. What we are basically doing by incorporating the time-step identifier term in the schema definition is to assign to each value t in $\mathbb{N}_0$ an instance $\mathbf{I}[t]$ composed by all the facts over $\mathbf{R}_T$ having the time-step value t, i.e., the set of ground atoms $R(u_1, \ldots, u_k, t)$ with

$R \in \mathbf{R}_T$ a relation of arity $k+1$. On the other hand, also the inverse relation holds: for each fact we can consider the time-steps in which it is assumed to be true. This last perspective is usually referred to as the *timestamp view*, while the one previously introduced is named *snapshot view*, which also defines how the notion of time can be related to the notion of *state* of a database: $\mathbf{I}[t]$ identifies the instance at time t or in an equivalent way the t-th state of the database [22]. Note the two different notations: we employ the *bracket notation* [] when we mean a complete instance at a certain time-step, while we use the *timestamp notation* @ when we want to refer to the time-step in which a certain fact is true.

In order to characterize the set of ground atoms initially given as input to a program, in the followings we will refer to **Init** as the *finite initial database instance* of the schema. We consider **Init** as containing facts possibly spanning multiple time-steps, or, more formally $\mathbf{Init} = \bigcup_{i \in \mathbb{N}_0} \mathbf{Init}[i]$. Thus by **Init** we refer to initial state of the (temporal) database, where facts belonging to the initial time-step 0, but also to the future, may be known. Thus for example:

```
Init′:{link(a,b)@0,link(b,c)@0}.
Init″:{link(c,b)@0,link(a,h)@1,link(c,b)@7}.
```

Listing 1.2. Examples of Initial Instances

are two valid initial database instances. Note that, because at the initial system time we are able to refer also to future time steps, the semantics we are using is that of *valid time* [28]. In the following, we will use the signature $\pi_{\mathbf{R}}(\mathbf{I}[t])$ to denote an instance over the schema $\mathbf{R}_T$ *restricted* to – or projected over using the algebraic prospective – the $\mathbf{R}$ schema. More formally, $\pi_{\mathbf{R}}(\mathbf{I}[t]) = \{R'(\bar{u}(1), \ldots, \bar{u}(k)) | R(\bar{u}) \in \mathbf{I}[t]$ and R' is equivalent to R but with $arity(R') = arity(R) - 1\}$. In the opposite way, with $\mathbf{I} \times t$ we express an instance defined over $\mathbf{R}$ but augmented with the time-step term $t \in \mathbb{N}_0$, or, more precisely, $\mathbf{I} \times t = \{R'(\bar{u}(1), \ldots, \bar{u}(k), t) |\ R(\bar{u}) \in \mathbf{I}$ and R' is equivalent to R except that $arity(R') = arity(R) + 1\}$.

Many versions of Datalog with a notion of time exist [30, 9]. Among all, our version, named $\text{Datalog}_T^{\neg}$, follows the road traced by Dedalus_0 [5] and Statelog [22]. We define $\text{Datalg}_T^{\neg}$ starting from $\text{Datalog}^{\neg}$ and employing the database schema $\mathbf{R}_T$ enhanced with two new built-in relations having $\mathbb{N}_0$ as domain for all the terms: `succ` (with arity two) and `time` (with arity one). The interpretation of $\texttt{succ}(t, t')$ is $t' = t+1$, while $\texttt{time}(t)$ is used to bind the evaluation step t with the tuples that are valid at that precise time-step. In $\text{Datalog}_T^{\neg}$, in fact, we consider tuples to be *immutable* – once instantiated they cannot be retracted nor changed – and *ephemeral* by default, i.e., they are valid only for the assigned time-step. For example, the facts composing the second initial instance `Init″` defined in Listing 1.2 are considered valid only for time-steps 0, 1 and 7 respectively, and are not allowed to change.

We can rewrite rule (1) in the new time-aware form (using the suffix notation):

$$H(\bar{w})@t \leftarrow B_1(\bar{u}_1)@t \ldots, B_n(\bar{u}_n)@t, \texttt{time}(t). \tag{2}$$

where with $B_i(\bar{u}_i)$ we now denote positive or negated literals. Note that built-in relations such as `time` and $\geq$, thanks to they "predefined" nature, do not need to be postfixed by the time-step identifier. Adopting the same notation of [5] we call this type of rules *deductive*, and they are used to instantaneously derive – once fixed a time-step specified by $\texttt{time}(t)$ – new facts given the information currently available at that point in time. These newly derived facts will be valid just for that time-step and cannot be altered.

Nevertheless, in Datalog$_T^\neg$ *mutable* tuples can be *emulated* using time and by explicitly stating how tuples evolve with the progress of time. In fact, if a tuple, let's say $R(a, b)@t$ is valid at time t, by employing *inductive* rules [5] we are able to specify a new immutable version which will be valid at time-step $t + 1$, thus even if the initial object is not actually modified, a different version of the tuple will exist at time $t + 1$:

$$H(\bar{w})@t' \leftarrow B_1(\bar{u}_1)@t, \ldots, B_n(\bar{u}_n)@t, \texttt{time}(t), \texttt{succ}(t, t'). \tag{3}$$

where predicates related to the next state can be specified only in rules' head. Starting from the tuples related to a single time-step t, by employing inductive rules we are able to build the next instance $\mathbf{I}[t+1]$ given the information currently available at the present time t. From another point of view, inductive rules can be seen as *production rules* used to form future instances [22].

If derivable from inductive rules, tuples from ephemeral become *persistent*: once derived, for example at time-step t, they will eventually last for every $t' \geq t$. For example a tuple in a relation $R \in \mathbf{R}_T$ can be "persisted" using the following rule:

$$R(\bar{u})@t' \leftarrow R(\bar{u})@t, \texttt{time}(t), \texttt{succ}(t, t'). \tag{4}$$

Persistent relations can only grow because in accordance with rule (4), once added a tuple exists forever. The permanency state of a tuple can be "broken" by defining, for example, an *idb* relation *del_R* and by modifying the previous rule as follows:

$$R(\bar{u})@t' \leftarrow R(\bar{u})@t, \neg del\_R(\bar{u})@t, \texttt{time}(t), \texttt{succ}(t, t'). \tag{5}$$

Rule (5) basically is stating that at a given time t, a tuple will continue to be valid also at the successive time-step $t + 1$ if it is true at time t and at the same point in time an explicit deletion for the same tuple does not exist. Since we want to avoid the specification of *edb* relations in the head of inductive rules to make them persistent, for each extensional relation $R \in edb(\mathbf{R}_T, \Pi)$, a new predicate *R_idb* is added to the $\mathbf{R}_T$ schema. In addition the following rule is added:

$$R\_idb@t \leftarrow R@t, \texttt{time}(t). \tag{6}$$

In this way, for each extensional predicate R one intensional relation exists that contains at least the tuples originally stored in R. We can now operate directly on such *idb* relations by means of inductive rules – *R_idb* relations are allowed to appear has heads only in inductive rules, except for rules of type (6) obviously – instead of modifying the extensional instance. The just introduced rules are the

only ones permitted to involve extensional predicates. This property of the *edb* is called *guarded edb* [5].

Some syntactic sugar is adopted in order to better manipulate rules and relations: all the time-step suffixes are omitted together with the `succ` and `time` relations – thus deductive rules appear as usual Datalog$^\neg$ rules – while a `next` suffix is introduced in the head relations of inductive rules.

$$deductive : H(\bar{w}) \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n). \tag{7}$$

$$inductive : H(\bar{w})@next \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n). \tag{8}$$

In Listing 1.3 the program of the previous section is rewritten introducing the new formalism.

```
r1:   link_idb(X,Y,D)@next:-link_idb(X,Y,D).
r2:   link_idb(X,Y,D):-link(X,Y,D)

r3:   path(X,Y,D):-link_idb(X,Y,D).
r4:   path(X,Z,D):-link_idb(X,Y,D1),path(Y,Z,D2),D=D1+D2.
r5:   shortestDistance(X,Y,min<D>):-path(X,Y,D).
r6:   shortest(X,Y,D):-path(X,Y,D), shortestDistance(X,Y,D).
```

Listing 1.3. Inductive and Deductive Rules

From here on, for the sake of conciseness we omit to write the rules of type (6) transmitting tuples from *edb* to *idb* relations (such as rule `r2`), and, instead, we employ R to express directly R_idb.

2.2 Model-Theoretic Semantics

In this section, to derive the proper semantics for Datalog$^\neg_T$, we take inspiration from the model-theoretic semantics of Statelog [22] . Given a program Π and a finite input instance $\pi_{\mathbf{R}}(\mathbf{Init})$, we define $\Sigma = \Pi \cup \pi_{\mathbf{R}}(\mathbf{Init})$, and we construct the *Herbrand universe* U_Σ in the usual way, by creating a set encompassing all the ground terms that can be constructed starting from constant symbols appearing in the clauses constituting the set Σ. Then we define the *Herbrand base* B_Σ as the set of all ground atoms which can be formed by using predicate symbols from Σ with constant terms from U_Σ as arguments. A Datalog$^\neg$ *interpretation* $\mathcal{I}_\Sigma$ is then a subset of $B_\Sigma \times \mathbb{N}_0$, i.e., a (potentially infinite if a final state does not exist) sequence $\mathcal{I}_\Sigma = (\mathcal{I}_\Sigma[0], \ldots, \mathcal{I}_\Sigma[n])$ where each $\mathcal{I}_\Sigma[i]$ with $i \in \mathbb{N}_0$ is an *Herbrand interpretation* – that is, a subset of the Herbrand base B_Σ – at the i-th time-step. With $\mathcal{I}_\Sigma[*]$ we name the database *final state* if it exists. Now, given a variable assignment function v which maps each variable to an element in **dom**, and an interpretation $\mathcal{I}_\Sigma$, the definition of *satisfiability* for a deductive

clause with no aggregate relations in $\Pi \cup \textbf{Init}$ is:

$$(\mathcal{I}_\Sigma, v) \models H(\bar{w})@t \leftarrow B_1(\bar{u}_1)@t, \ldots, B_n(\bar{u}_n)@t, \texttt{time}(t).$$
$$\text{iff whenever}$$
$$(\mathcal{I}_\Sigma[t], v) \models B_1(\bar{u}_1), \ldots, \ (\mathcal{I}_\Sigma[t], v) \models B_n(\bar{u}_n), \ (\mathcal{I}_\Sigma[t], v) \models \texttt{time}(t)$$
$$\text{then}$$
$$(\mathcal{I}_\Sigma[t], v) \models H(\bar{w})$$

The same applies for inductive rules. For what concerns instead aggregate predicates, they constitute the head of *aggregation rules* having the following form (here we show a deductive aggregation rule but the same applies for inductive aggregation rules):

$$R(\bar{u}, \Lambda < w >)@t \leftarrow B_1(\bar{u}_1)@t, \ldots, B_n(\bar{u}_n)@t, \texttt{time}(t). \tag{9}$$

where Λ is an aggregate function, $\bar{u}$ is a list of variables from the body, and w is also a body variable possibly belonging to the list $\bar{u}$. Now, if we denote with $\bar{u}'$ a ground assignment for $\bar{u}$, and with W the finite multi-set containing all the existing ground assignments of w, which, with $\bar{u}'$, satisfy the body of the rule, we have that $R(\bar{u}', a)$ is true, where $a = \Lambda < W >$. That is, a is the result of the application of Λ to the multi-set W [10, 17]. Our *canonical model* $\mathcal{M}_\Sigma$ is then selected from the set of possible models, i.e., interpretations satisfying all the clauses $r \in \Pi \cup \textbf{Init}$. We will usually denote with *model snapshot* $\mathcal{M}_\Sigma$[t] the model at time-step t, while with $\mathcal{M}_\Sigma$ we name the (potentially infinite if a final state does not exist) sequence of model snapshots.

If program Π is positive its semantics is a straightforward extension of classical Datalog programs semantics: the sequence of least Herbrand models where **Init** is true – that is, $\forall t \in \mathbb{N}_0$, $\mathcal{M}_\Sigma[t] \models \textbf{Init}[t]$ – and satisfying Π. It turns out that also for semi-positive programs the semantics coincides with the sequence of unique minimal models satisfying **Init** and Π. This is because, by employing the *Closed World Assumption* (*CWA*) w.r.t. the active domain, given an *edb* relation R, which, by its nature, has a stable content, the negative literal $\neg R(\bar{u})$ is satisfied if the ground tuple $\bar{u}$ is in the active domain and $\bar{u} \notin R$ [2]. If, instead, we allow negated *idb* atoms into programs, we consider the *stratified semantics*. This semantics is too restrictive for Datalog$_T^\neg$ programs: in fact, it turns out that certain types of negative dependency cycles can be allowed, i.e., negative cycles that are not within one time-step, but "across" different time-steps. The *temporally-stratified* semantics [5] is therefore introduced.

Definition 1. *A program Π is called temporally stratifiable if negative dependency cycles among deductive rules do not exist.*

It has been shown [5] – an analogous statement has been proved for Statelog [22] and Datalog XY-programs [30] – that given a *rectified* program Π – i.e., a program that does not contain any constant – Π is temporally stratifiable if and only if it is locally stratifiable. Since it is well-known that every locally stratified program has a unique *perfect model* [26], we can assume that every temporally

stratifiable program has a unique perfect model $\mathcal{M}_\Sigma$, and this is the intended canonical model.

Definition 2. *A temporally stratifiable Datalog$_T^\neg$ program Π has a unique perfect model $\mathcal{M}_\Sigma$.*

Even though local stratification is in general undecidable [14], temporal stratification is decidable and can be easily computed similarly as for normal stratification in Datalog$^\neg$ [22]. Thanks to the program stratification, we induce a partial order over rules: first deductive rules are evaluated according to the stratification until a stable perfect model $\mathcal{M}_{\Sigma_{ded}}[t]$ is reached for the current time-step t. Then the rules defining $\mathcal{M}_\Sigma[t+1]$ will be executed. The operational semantics of $\mathcal{M}_\Sigma$ follows just this approach. To note that $\mathcal{M}_\Sigma$ does not always terminate, i.e., the final state $\mathcal{M}_\Sigma[*]$ might not exist if the program encodes a periodic behavior, as shown for the example in Listing 1.4.

```
r1:   r(Y,X)@next:-r(X,Y).
r2:   r(a,b)@0
```

Listing 1.4. Example of a non terminating program

However, even if not terminating, $\mathcal{M}_\Sigma[t]$ exists for all $t > 0$ and $\mathcal{M}_\Sigma$ can be finitely represented due to its periodic behavior [22].

2.3 Operational Semantics

The operational semantics follows the usual bottom-up evaluation algorithm, although the semi-naive algorithm has been slightly modified in order to address the two different behaviors of deductive and inductive rules. To this purpose each computation round is divided into two steps: a *deductive step* where rules have to be executed in the order induced by the stratification; and an *inductive step* where facts are transferred from the t-th to the $(t+1)$-th state and where rules can be evaluated in any order because they do not depends on each other. We start the description of our algorithm by partitioning a program Π into the two sets of deductive and inductive rules, respectively Π_{ded} and Π_{ind}. Then a pre-processing step orders the deductive rules following the dependency graph stratification. After this pre-processing step, the model $\mathcal{M}_\Sigma$ is computed by the Algorithm 1.
The function *deduction* evaluates the stratified program Π_{ded} by employing the Algorithm 2 [30], where $T_{\Pi_{ded}}$ is the *immediate consequence* operator for program Π_{ded} and Δp^i is the set containing the newly derived facts at round i. To note that $\mathcal{M}_{\Sigma_{ded}}[t]$ is defined over $\mathbf{R}_T$ while $\mathcal{M}_\Sigma[t]$ is defined over $\mathbf{R}$.

3 Distributed Logic Programming

The language introduced so far allows us to logically model an evolving system thanks to the notion of time. But this formalism is not able to describe how computation can be distributed and performed simultaneously on multiple

Algorithm 1 Bottom-up evaluation

Input: a program Π; the initial database instance **Init**
Output: the model $\mathcal{M}_\Sigma$
t := 0;
$\mathcal{M}_\Sigma[0] \leftarrow \pi_{\mathbf{R}}(\mathbf{Init}[0]) \cup \texttt{time}(0)$;
repeat
  $\mathcal{M}_{\Sigma_{ded}}[t] \leftarrow deduction(\Pi_{ded})(\mathcal{M}_\Sigma[t] \times t)$;
  $\mathcal{M}_\Sigma[t+1] \leftarrow induction(\Pi_{ind})(\mathcal{M}_{\Sigma_{ded}}[t]) \cup \mathbf{Init}[t+1] \cup \texttt{time}(t+1)$;
  t := t + 1;
until $\mathcal{M}_\Sigma[t+1] = \mathcal{M}_\Sigma[t]$
return $\{\mathcal{M}_\Sigma[0], \ldots, \mathcal{M}_\Sigma[t]\}$

Algorithm 2 Function *deduction*

Input: a stratified deductive program $\Pi_{ded} = \Pi^0_{ded} \dot\cup ... \dot\cup \Pi^k_{ded}$;
  $\mathcal{M}_\Sigma[t]$
Output: the perfect model $\mathcal{M}_{\Sigma_{ded}}[t]$
$p^0 \leftarrow \mathcal{M}_\Sigma[t]$;
$\Delta p^0 \leftarrow T^0_{\Pi_{ded}}(p^0)$;
$p^1 \leftarrow \Delta p^0 \cup p^0$;
for $i := 1$ to k **do**
  repeat
    $\Delta p^i \leftarrow T^i_{\Pi_{ded}}(p^i)$ / p^i;
    $p^{i-1} \leftarrow p^i$;
    $\Delta p^{i-1} \leftarrow \Delta p^i$;
    $p^i \leftarrow p^{i-1} \cup \Delta p^{i-1}$;
  until $\Delta p^{i-1} = 0$
end for
return p^i

processing units. In order to reach this goal, in this section we are going to introduce the notion of *distributed shared relation* (DSR), and we will describe how this high-level abstraction can be employed to obtain a logical language for distributed systems. The discussion that follows is based on two assumptions: the network topology is *fully connected* and the system is *synchronous* and *reliable*. While the first assumption is made in order to simplify the treatment and indeed can be easily relaxed, for example by assuming a network layer or a special purpose program computing routing paths – as the example programs of previous sections – the second directly affects the syntax and semantics of the language.

3.1 Preliminary

In the following we define a *distributed system* to be composed by a non-empty finite set of nodes identifiers $\mathcal{L} = \{l^1, l^2, \ldots, l^n\}$. Each node identifier l^i has value in the domain **dom**, that we consider to be the same for every node – but, for simplicity here we assume a node l^i to be identified by its apex i. Thus, in the

following we use the set $L = \{1, \ldots, n\}$ as the set of node identifiers, where n is the total number of nodes in the system. We assume each node i to maintain its own database schema $\mathbf{R}_T^i$. Therefore starting from the relational schema $\mathbf{R}_T = \bigcup_{i \in L} \mathbf{R}_T^i$ we define the new schema $\mathbf{R}_{ST}$ obtained by adding a new set of (*distributed*) *shared relations* (DSR) defining the *sdb* schema. As usual, given a program Π, $sdb(\mathbf{R}_{ST}, \Pi)$ indicates the DSR relations in Π. The database schema $\mathbf{R}_S$ is obtained in a similar way. We assume *sdb* relations to be completely disjoined from both *edb* and *idb* relations. Shared relations are the means by which nodes are able to interact among themselves. As the name highlights, such relations are shared among multiples nodes simultaneously, so a mechanism for deciding which node is responsible for which tuple must be developed. Given a relation $S \in sdb$, an atom over S has the form $S(\texttt{->}l, u_1, \ldots, u_k)@t$, where the term $l \in \mathbf{dom} \cup \mathbf{var}$ is called *location specifier* [21] and is used to identify which node has the *exclusive access privileges* on that particular k-tuple, or, in other words, which node is responsible for that portion of data. We assume the location specifier to be always the first term of a shared relation. Moreover we assume $\mathbf{R}_{ST}$ to contain two new unary built-in relations, namely `id`, and `all`. The first stores the local node id, while the second maintains all the nodes belonging to the network, i.e., L.

3.2 Syntax

Deductive and inductive rules (respectively eq.s (2) and (3)) can be rewritten in the following form in order to incorporate shared relations:

$$H(\bar{w})@t \leftarrow B_1(\bar{u}_1)@t, \ldots, B_n(\bar{u}_n)@t, (\neg)S_1(\texttt{->}l_1, \bar{v}_1)@t, \ldots, (\neg)S_m(\texttt{->}l_m, \bar{v}_m)@t, \texttt{time}(t). \tag{10}$$

$$H(\bar{w})@t \leftarrow B_1(\bar{u}_1)@t, \ldots, B_n(\bar{u}_n)@t, (\neg)S_1(\texttt{->}l_1, \bar{v}_1)@t, \ldots, (\neg)S_m(\texttt{->}l_m, \bar{v}_m)@t, \texttt{time}(t), \texttt{succ}(t, t'). \tag{11}$$

where we allow B_i to range both over $edb \cup idb$ literals, and S_j to range over *sdb* relations, while H can space over $sdb(\mathbf{R}_{ST}, \Pi) \cup idb(\mathbf{R}_{ST}, \Pi)$, and hence if $H \in sdb(\mathbf{R}_{ST}, \Pi)$ we assume the tuple $\bar{w}$ to contain also the location specifier term. More precisely, the location specifier term will be the local node id if the rule is of type (10), while can be any identifier in L if of type (11). a deductive rule is *local* if $m = 0$, while an inductive rule is local if $m = 0$ and l is bound to the local node id. Non-local rules are labeled as *distributed*. Among all the distributed inductive rules, some rules are of special interest because they are used to explicitly state when a tuple access privileges must be transferred from one single node to another:

Definition 3. *An* explicit communication rule *is a distributed inductive rule in the following form:*

$$H(\texttt{->}l', \bar{w})@t' \leftarrow B_1(\bar{u}_1)@t, \ldots, B_n(\bar{u}_n)@t, (\neg)S_1(\texttt{->}l, \bar{v}_1)@t, \ldots, (\neg)S_m(\texttt{->}l, \bar{v}_m)@t, \texttt{time}(t), \texttt{succ}(t, t'), (\texttt{id}(l)). \tag{12}$$

where $H \in sdb$ and the location specifier term of the head l' differs from all the location specifier l (if they exist) appearing in the body's sdb relations. In case sdb relations appear in the body, the location specifier term must be bound to the local node id l.

Explicit communication rules are used to move *local* facts from one node to another. Therefore the body of such rules must contain only local atoms, i.e., atoms over *edb* or *idb* relations (thus if only these type of relations exist in the body, `id` is not necessary), or atoms over *sdb* relations where the location specifier term is bound to be the local node (in this case `id` is needed in oder to bound the location specifier terms).

Deductive, inductive and (inductive) explicit communications rules have the following syntactic-sugared form:

$$deductive : H(\bar{w}) \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n), \\ S_1(\texttt{->}l_1, \bar{v}_1), \ldots, S_m(\texttt{->}l_m, \bar{v}_m). \tag{13}$$

$$inductive : H(\bar{w})@next \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n), \\ S_1(\texttt{->}l_1, \bar{v}_1), \ldots, S_m(\texttt{->}l_m, \bar{v}_m). \tag{14}$$

$$explicit\ communication : H'(\texttt{->}l', \bar{w})@next \leftarrow B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n), \\ S_1(\texttt{->}l, \bar{v}_1), \ldots, S_m(\texttt{->}l, \bar{v}_m). \tag{15}$$

where $B_i(\bar{u}_i)$ and $S_j(\bar{v}_j)$ are literals, $S_j \in sdb$, $H \in idb \cup sdb$ and $H' \in sdb$. We name this language Datalog$^{\neg}_{ST}$. It's interesting to note that the new version of inductive rules still resemble the old version described in previous sections. But, as we will see in the next section, we can assert that also semantically are almost equivalent. In fact, if the previous versions permits to transfer tuples among time-steps, this new version permits to transfer tuples among time-steps *and* nodes, which, thanks to the synchronous communication model, we are able to affirm that takes exactly one time-step.

Continuing with the usual example, we can now actually program a distributed routing protocol. In order to describe the example of Listing 1.5 we can imagine a real network configuration where each node has the program locally installed and where each `link` relation reflects the actual state of the connection between nodes. For example, we will have the fact `link(a,b)` in node a's instance if a communication link actually exists between nodes a and b.

```
r1:   link(X,Y)@next:-link(X,Y).
r2:   path(->X,Y)@next:-path(->X,Y).

r3:   path(->X,Y):-link(X,Y).
r4:   path(->X,Z)@next:-path(->X,Y),path(->Y,Z).
```

Listing 1.5. Distributed Program

In this new version computation is performed simultaneously on multiple distributed nodes. Communication among nodes is achieved through rule `r4` which specifies that node a knows that a path from node a to a node c exists if it

knows that there is a path from a to another node b and this last node knows that a path from b to c exists. The body of rule `r4` contains relations stored at different locations, therefore implicitly assuming tuples to be exchanged among nodes. Now, if this program is run locally, for example on a shared memory multicore system, the path relation can actually be implemented using the local shared memory, therefore each path fact is also available to other nodes once derived. If instead the program is run on multiple distributed machines, rule `r4` will be never evaluated because no tuple belonging to other nodes is stored locally. This because a rule, to be satisfied, needs to have all tuples *locally stored* in order to correctly compute the joins among relations. To solve this, we can use the *rule localization rewrite* algorithm first introduced in [21], in order to make communication explicit (as specified in Definition 3), where all joins in the body are computed among locally stored relations. Rule `r4` can hence be rewritten in the two rules of Listing 1.6.

```
r4_a:   path_r(->Y,X)@next:-path(->X,Y).
r4_b:   path(->X,Z):@next-path_r(->Y,X),path(->Y,Z).
```

Listing 1.6. Rewriting of Rule `r4` of Listing 1.5

In this way, each path existing at node X is explicitly sent to the neighbor Y, and then the transitive closure can be computed at node Y because it locally has stored all the necessary tuples. This example shows the power of employing DSRs as communication means. The programmer can specify a program that is completely independent of any physical assumption and the behavior of the program can be shaped by the compiler using rewriting algorithms that are appropriate for the given physical context.

4 Semantics for Synchronous Systems

Before introducing the semantics of Datalog$^{\neg}_{ST}$, we have to introduce some concept defining what a distributed computation is. Then employing *transducer networks* [6, 7] we will show how computation is actually performed in synchronous setting with reliable communication and fully connected topology. We will then describe the model-theoretic semantics of Datalog$^{\neg}_{ST}$ programs by showing that is equivalent to the semantics of a Datalog$^{\neg}_{T}$-centralized version of the program. But first we have to introduce the notion of *relational transducer*.

4.1 Datalog$^{\neg}_{ST}$-Relational Transducer

Given a Datalog$^{\neg}_{ST}$ program Π defined over a relational schema $\mathbf{R}_{ST}$, we define *database*, *memory*, and *distributed shared relation* schemas, respectively, as $\mathbf{R}_{db} = edb(\mathbf{R}_T, \Pi)$, $\mathbf{R}_{mem} = idb(\mathbf{R}_T, \Pi)$, and $\mathbf{R}_{dsr} = sdb(\mathbf{R}_{TS}, \Pi)$. A *transducer schema* $\mathcal{R}$ is a tuple ($\mathbf{R}_{db}$, $\mathbf{R}_{mem}$, $\mathbf{R}_{dsr}$, $\mathbf{R}_{time}$, $\mathbf{R}_{sys}$) where $\mathbf{R}_{time}$ contains just the unary relation `time`, and the *system* schema $\mathbf{R}_{sys}$ contains the two unary relations `id`, and `all`. A *transducer state* for $\mathcal{R}$ is a database instance $\mathbf{T}$ over $\mathbf{R}_{db} \cup \mathbf{R}_{mem} \cup \mathbf{R}_{dsr} \cup \mathbf{R}_{time}$ and a *Datalog$^{\neg}_{ST}$-relational transducer* is a

program $\mathcal{T}$ defined over $\mathcal{R}$. Finally, a transducer *configuration*, is a tuple (L, α) where L is the set of nodes defining the relation `all`, and α is the location identifier of the node.

Initially a relational transducer $\mathcal{T}$ is assumed as loaded with the initial instance $T_{db} = \mathbf{Init}$ and `time` contains the tuple (0), In addition, we assume to have a (possibly infinite) *input tape* containing an ordered sequence of consecutive natural numbers starting from 1. This sequence will be used as input for a relational transducer in order to provide the clock driving the computation. Why we provide time-steps in this way will be more clear in Section 4.4, where we define synchronous transducer networks.

Given as input a set of input tuples T_{in} defined over $\mathbf{R}_{dsr}$, together with a configuration and the next time value taken from the input tape, the relational transducer will transit to the next state and output a set of output tuples T'_{out} defined over $\mathbf{R}_{dsr}$. More formally: similarly to what we did in Section 2.2, we define $\Upsilon = \mathcal{T} \cup \pi_{\mathbf{R}}(\mathbf{Init})$, and U_Υ, B_Υ to be respectively the Herbrand universe and Herbrand base constructed starting from the set of clauses in Υ. Then, if we denote with $\mathcal{T}_{ded}$ and $\mathcal{T}_{ind}$ respectively the set of local deductive and inductive rules in $\mathcal{T}$, we use the symbols $\mathcal{M}_{\Upsilon_{ded}}$ and $\mathcal{M}_{\Upsilon_{ind}}$ to identify the"instantaneous" models of $\mathcal{T}_{ded}$, and $\mathcal{T}_{ind}$ – i.e., subsets of B_Υ. In addition, with $\mathcal{T}_{com}$ we identify the set of rules defining the remaining distributed rules in the form of (13) and (14) – if they are adequate for the given physical context because in the opposite case they will be rewritten in explicit communication rules using the rule localization algorithm – plus the rules in form of (15). To simplify the description, however, we assume w.l.o.g. that $\mathcal{T}_{com}$ is composed only by rules in the form of (15).

Now, a *local transition* is a 5-tuple $(\mathbf{T}, t, (L, i), T_{in}, \mathbf{T}', T'_{out})$, also denoted as $\mathbf{T}$, $T_{in} \overset{t,(L,i)}{\Longrightarrow} \mathbf{T}'$, T'_{out}, where $t \in \mathbb{N}$ is taken from the input tape and T_{in}, T'_{out} are instances of $\mathbf{R}_{dsr}$. $\mathbf{T}'$, T'_{out} are transducer states such that:

- $T'_{db} = T_{db}$
- $T'_{time} = \texttt{time}(\text{t})$
- $T_{sys} := \texttt{id}(i) \cup \{\texttt{all}(j) | j \in L\}$
- $\mathcal{M}_{\Upsilon_{ded}} = deduction(\mathcal{T}_{ded})(T_{db}[t] \cup T_{mem}[t] \cup T_{in}[t] \cup T_{sys})$
- $\mathcal{M}_{\Upsilon_{ind}} = induction(\mathcal{T}_{ind})(\mathcal{M}_{\Upsilon_{ded}})$
- $\mathcal{M}_{\Upsilon_{com}} = induction(\mathcal{T}_{com})(\mathcal{M}_{\Upsilon_{ded}})$
- $T'_{dsr} = \mathcal{M}^i_{\Upsilon_{com}}$ i.e., *dsr* tuples referred to the local node i, where $\texttt{id}(i) \in T_{id}$.
- $T'_{out} = \mathcal{M}_{\Pi_{com}} / \mathcal{M}^i_{\Upsilon_{com}}$
- $T'_{mem} = \mathcal{M}_{\Upsilon_{ind}}$

where *deduction* and *induction* are the same functions described in Section 2.3. To note that transitions are deterministic, i.e., if $\mathbf{T}$, $T_{in} \overset{t,(L,i)}{\Longrightarrow} \mathbf{T}'$, T'_{out} and $\mathbf{T}$, $T_{in} \overset{t,(L,i)}{\Longrightarrow} \mathbf{T}''$, T''_{out}, then $\mathbf{T}' = \mathbf{T}''$ and $T'_{out} = T''_{out}$ [6].

4.2 Distributed Computation

At any point in time each node is in some particular *local state* incapsulating all the information of interest the node possesses. For convenience, we define the

local state s^i of a node $i \in L$ as the pair $(\mathcal{I}^i, n)$ where $\mathcal{I}^i = \pi_{\mathbf{R}_T}(\mathbf{T}^i[n])$ with $\mathbf{T}^i[n]$ a transducer state for node i at time-step n. Note that a local state is completely determined by the transducer state $\mathbf{T}^i$, however, here we use the notation $(\mathcal{I}^i, n)$ just to make explicit the fact that each state embeds internally a notion of time. We define the *global state* g of a distributed system as a tuple $(s^e, s^1, ..., s^n)$ where s^i is node i's state, while s^e is the *environment* local state. We can consider the environment as a "special" node storing all the information external to each node. In the following we will consider $\mathcal{I}^e$ to contain the sequence of facts that are in transit among nodes and not yet received. We define how global states may change over time through the notion of *run*, which binds time values to global states. Again, if we assume time values to be isomorphic to the set of natural numbers, we can define the function $\rho : \mathbb{N} \to \mathcal{G}$ where $\mathcal{G} = \{S^e \times S^1 \times ... \times S^n\}$, S^i is the set of possible local states for node $i \in L$, and S^e is the set of possible states for the environment.

If $\rho(t) = (s^e, s^1, ..., s^n)$ is the global state at time t, we define $\rho^e(t) = s^e$ and $\rho^i(t) = s^i$, for $i \in L$. We denote with *init* the initial state $\rho(0)$, with $\rho(0)^i = s^i = (\mathcal{I}^i, 0)$ and where $\mathcal{I}^i = \mathbf{Init}^i[0] \cup \texttt{time}(0)$ – i.e., the partition of **Init** containing the portion of tuples for node i at time 0. We want to note here that the time t and the notion of time-step n incapsulated in programs are two different entities. In fact, while the first one is an *external* time used to reason about the evolution of global states, the second – i.e., the one used to drive the computation of the transducer – is definitely an internal (relative) perspective that each node has about the passing of time. For instance the node i's local state at time t will have the form $\rho(t)^i = s^i$ where $s^i = (\mathcal{I}^i, n)$ and possibly $t \neq n$. In the following section, however, we will see how these two notions can be tied together in order to define what a synchronous computation is.

A system may have many possible runs, indicating all the possible ways the global state can evolve. In order to capture this, we define a *system* $\mathcal{S}$ as a set of runs. Using this definition we are able to deal with a system not as a collection of interacting nodes but, instead, directly modeling its behavior by a program specification. We think that this approach is particularly important to the aim of maintaing a high level of declarativity in our description.

4.3 Synchronous Systems

For a system to be synchronous it must satisfy the following properties:

S1 A global clock is defined and is accessible by every node
S2 The relative difference between the time-step of any two nodes is bounded
S3 Updates to remote *sdb* relations arrive at destination at most after a certain bounded physical time Δ

A *synchronous system* $\mathcal{S}^{sync}$ is therefore a set of runs fulfilling the above conditions. The first property can be expressed in our framework by linking the time-step identifier of each node with the external time. Thus, by definition, every local state $\rho^i(t) = (\mathcal{I}^i, n)$ will have now $t = n$. In this configuration, the

second property is implemented by assuming that programs proceed in *rounds*, and that each round, operationally speaking, lasts enough to permit each node computation to reach the fix-point. In the following, w.l.o.g. we use the round number as external time (and therefore as time-step identifier).

In order to express the third property, we assume Δ to be amply lower than the amount of (physical) time spent between the end of one round and the start of the consecutive one. Indeed this condition is satisfied by the communication rule syntax – a fact inferred from a communication rule is true in the subsequent time-step – and in the next section we will show how this is also satisfied by its semantics.

4.4 Synchronous Transducer Networks

We have already defined how local states evolve by defining what a relational transdcer is. Now that we have defined what a synchronous system is, it remains to specify how properties **S1** - **S3** can be enforced during the evolution of global states. To accomplish this, we employ a *synchronous transducer network* [6, 7]. A *synchronous Datalog$^{\neg}_{ST}$-transducer network* is a tuple$(L, \gamma, \mathcal{T}, \mathcal{T}^e)$ where γ is a function mapping each element $i \in L$ to a transducer state $\mathbf{T}^i$, whose configuration is (L, i). We then assume the environment to be a "special" relational transducer. In particular, we consider the transducer defining the environment $\mathcal{T}^e$ as having an empty program, and the schema composed only by *dsr* relations with $\mathbf{R}^e_{dsr} = \bigcup_{i \in L} \mathbf{R}^i_{dsr}$. As hinted before, we consider the environment as registering the *sdb* facts floating in the network and not yet received. As can be noticed, in our definition we assume each node $i \in L$ to have the same transducer $\mathcal{T}$, while the only thing that we allow to change from node to node is its instance. Then, a *state* $\mathbf{N}$ of a transducer network $(L, \gamma, \mathcal{T}, \mathcal{T}^e)$ is a tuple $(\mathbf{T}^e, \mathbf{T}^1, \ldots, \mathbf{T}^n)$ where for each $i \in L$ the i-th element is the related relational transducer state $\gamma(i) = \mathbf{T}^i$ such that $T_{db} = \mathbf{Init}^i$.

Let a transducer network initial state be a state where each `time` relation contains the value 0, and except $\mathbf{R}_{db}$, `id`, and `all`, all the remaining relations are empty. A *global transition*, is a 3-tuple $(\mathbf{N}, t, \mathbf{N}')$ also denoted as $\mathbf{N} \overset{t}{\Longrightarrow} \mathbf{N}'$ where $\mathbf{N}$ and $\mathbf{N}'$ are transducer network states, and $t \in \mathbb{N}$ is denoting the input from the time tape specifying what will be the next round. A global transition is defined such that:

- $T_{in} = \mathbf{T}^e$
- $\forall i \in L$, $\exists \mathbf{T}^i$ s. t. $\gamma(i) = \mathbf{T}^i$, $(\mathbf{T}^i, T^i_{in}, \overset{t,(L,i)}{\Longrightarrow} \mathbf{T}'^i, T'^i_{out})$ is a local transition for node i, where T^i_{in} denotes the set of facts in T_{in} for node $i \in L$
- $\mathbf{T}'^e = \bigcup_{i \in L} T'^i_{out}$

Informally, during a global transition all the transducers composing the network instantaneously make a local transition taking as input the associated *sdb* tuples output of the $t-1$ round. In addition we assume that one global transition, in order to satisfy property **S3**, can start only after that a certain amount Δ of physical time has elapsed after the end of the previous transition. Indeed, since

a global transition is composed by n local transitions all fired instantaneously and the communication is reliable, also global transitions are deterministic. If we consider a global state at round t to be defined as $\rho(t) = (s^e, s^1, \ldots, s^n)$ with $s^i = (\mathcal{I}^i, t)$ and $\mathcal{I}^i = \pi_{\mathbf{R}_T}(\mathbf{T}^i[t])$ where $\mathbf{T}^i[t]$ denotes the local Datalog$^\neg_{ST}$-transducer state for node $i \in L \cup e$ at round t, the definition of global transition specifies how global states evolve in synchronous settings because it satisfies conditions **S1 - S3**. Given a Datalog$^\neg_{ST}$ program and an initial instance **Init**, its operational semantics in synchronous settings is completely determined by the synchronous system $\mathcal{S}^{sync}$ defining its evolution.

4.5 Model-Theoretic Semantics of Synchronous Transducers Networks

Starting from a synchronous transducer network (L, γ, e), we can develop a restricted form of Datalog$^\neg_{ST}$-relational transducer, namely *Datalog$^\neg_T$-relational transducer*, which is able to simulate the behavior of such network in centralized settings. Then, after having developed the model-theoretic semantics of this restricted form of synchronous transducer networks, we will show that the declarative semantics of a transducer network is indeed the model-theoretic semantics of its centralized version.

Given a transducer network $(L, \gamma, \mathcal{T}, \mathcal{T}^e)$ with schema $\mathcal{R} = (\mathbf{R}_{db}, \mathbf{R}_{mem}, \mathbf{R}_{dsr}, \mathbf{R}_{time}, \mathbf{R}_{sys})$ and program $\mathcal{T}$, its centralized version is a Datalog$^\neg_T$-relational transducer defined as follows (where we use the apex c to denote the centralized variant):

- $\mathcal{R}^c = (\mathbf{R}_{db}, \mathbf{R}^c_{mem}, \mathbf{R}_{time})$ where $\mathbf{R}^c_{mem} = \mathbf{R}_{mem} \cup \pi_{\mathbf{R}_T}(\mathbf{R}_{dsr})$
- $\mathcal{T}^c$ is a Datalog$^\neg_T$-relational transducer, i.e., the restricted version of $\mathcal{T}$ ranging over $\mathcal{R}^c$

Informally, the centralized version of a synchronous transducer network is composed by the same program $\mathcal{T}$ defining the network – remember that the schema and the program of a transducer network are the same for each node $i \in L$ – but where each *sdb* relation is now restricted over $\mathbf{R}_T$ – i.e., all the location specifiers have been dropped – and therefore all distributed rules became normal local rules. Due to this restriction, the environment and the *system* relations are no more necessary. A transition for the Datalog$^\neg_T$-relational transducer $\mathcal{T}^c$ is simply the tuple $(\mathbf{T}, t, \mathbf{T}')$ where each component is as follows:

- $T'_{db} = T_{db}$
- $T'_{time} = \texttt{time}(t)$
- $\mathcal{M}_{\Upsilon^c_{ded}} = \mathit{deduction}(\mathcal{T}_{ded})(T_{db}[n] \cup T_{mem}[n] \cup \texttt{time}(n))$ where $n = t - 1$
- $\mathcal{M}_{\Upsilon^c} = \mathit{induction}(\mathcal{T}_{ind})(\mathcal{M}_{\Upsilon_{ded}})$
- $T'_{mem} = \mathcal{M}_{\Upsilon^c}$

where in particular $\Upsilon^c = \mathcal{T} \cup \pi_{\mathbf{R}}(\mathbf{Init})$. Also in this case transitions are indeed deterministic, and, in addition, the system describing the relational transducer $\mathcal{T}^c$ is composed by one and only one run.

Lemma 1. *Given a Datalog$_T^\neg$-relational transducer and a finite input instance, the system describing the evolution of such transducer, if exists, is composed by one and only run.*

Proof. Assuming that a system describing the Datalog$_T^\neg$-relational transducer exists, then the system is composed by at least one run by definition. This run is unique because transitions are deterministic. As a consequence, given a transducer and a finite initial instance, one single run can be used to specify how the states of the transducer evolve.

Remember that $\mathcal{T}^c$ is a Datalog$_T^\neg$ program, that we can identify with Π^c. Its canonical model $\mathcal{M}_{\Sigma^c}$ is the (possible infinite) sequence of model snapshots $\mathcal{M}_{\Sigma^c}[0], \ldots, \mathcal{M}_{\Sigma^c}[n]$ defined over $\Sigma^c = \Pi^c \cup \pi_{\mathbf{R}}(\mathbf{Init})$ and satisfying all the clauses in $\Pi^c \cup \mathbf{Init}$. Do note that $\Sigma^c = \Upsilon^c$, however we employ this notation to make a distinction between the model computed as in Section 2.3 and the model computed by the associated relational transducer. The following theorem states that the two models are indeed identical.

Theorem 1 *Given a Datalog$_T^\neg$ program Π^c and a finite input instance **Init**, its model-theoretic semantics is equivalent to the system $\mathcal{S}$ describing the evolution of the Datalog$_T^\neg$-relational transducer defining Π^c.*

Proof. We denote with $\mathcal{T}^c$ the Datalog$_T^\neg$-relational transducer describing the program Π^c. By Lemma 1 we know that given an input instance the system $\mathcal{S}$ expressing the evolution of $\mathcal{T}^c$ consists of a unique run. We use the symbol ρ^c to identify this single run. To prove the theorem we have to show that the sequence of states defined by the unique run $\{\rho^c(0), \ldots, \rho^c(n)\}$ is equivalent to the sequence of snapshot models composing $\mathcal{M}_{\Sigma^c}$, i.e., $\{\mathcal{M}_{\Sigma^c}[0], \ldots, \mathcal{M}_{\Sigma^c}[n]\}$. We are going to proceed by induction: in the base step trivially we have that $\rho^c(0) = \mathbf{Init}[0] \cup \texttt{time}(0) = \mathcal{M}_{\Sigma^c}[0]$. For the inductive step, we assume that $\rho^c(n) = \mathcal{M}_{\Sigma^c}[n]$ and we need to prove that the same happens for time $n+1$. By definition $\rho^c(n+1) = \mathbf{Init}[n+1] \cup \texttt{time}(n+1) \cup \mathcal{M}_{\Upsilon^c}[n]$, where $\mathcal{M}_{\Upsilon^c}[n] = \mathcal{M}_{\Upsilon^c_{ind}}[n]$. Now, if $\mathcal{M}_{\Upsilon^c_{ded}}[n] = \mathit{deduction}(\mathcal{T}^c_{ded})(T_{db}[n] \cup T_{mem}[n] \cup \texttt{time}(n))$, then $\rho^c(n) = T_{db}[n] \cup T_{mem} \cup \texttt{time}(n)$ and $\mathcal{M}_{\Upsilon^c}[n] = \mathit{induction}(\mathcal{T}^c_{ind})(\mathcal{M}_{\Upsilon^c_{ded}}[n])$. Analogously, from Section 2.3 we have that $\mathcal{M}_{\Sigma^c}[n+1] = \mathbf{Init}[n+1] \cup \texttt{time}(n+1) \cup \mathcal{M}_{\Sigma^c_{ind}}[n]$, where $\mathcal{M}_{\Sigma^c_{ind}}[n] = \mathit{induction}(\Pi^c_{ind})(\mathcal{M}_{\Sigma^c_{ded}}[n])$ and $\mathcal{M}_{\Sigma^c_{ded}}[n] = \mathit{deduction}(\Pi^c_{ded})(\mathcal{M}_{\Sigma^c}[n])$. Since $\mathcal{T}^c_{ded} = \Pi^c_{ded}$ by definition, $\mathcal{T}^c_{ind} = \Pi^c_{ind}$ and $\rho^c(n) = \mathcal{M}_{\Sigma^c}[n]$ applying the inductive step computation, we have that $\rho^c(n+1) = \mathcal{M}_{\Sigma^c}[n+1]$. It follows that $\forall n \in \mathbb{N}$, $\{\rho^c(0), \ldots, \rho^c(n)\} = \{\mathcal{M}_{\Sigma^c}[0], \ldots, \mathcal{M}_{\Sigma^c}[n]\}$.

We are now able to derive the model-theoretic semantics of synchronous transducer networks.

Theorem 2 *Let **Init** be a finite initial database instance, $(L, \gamma, \mathcal{T}, \mathcal{T}^e)$ a transducer network, and $\mathcal{M}_{\Sigma^c}$ the perfect model of the centralized version of $(L, \gamma, \mathcal{T}, \mathcal{T}^e)$. Then:*

$$\mathcal{M}_{\Sigma^c} = (\bigcup_{i \in L} \rho^i(0), \ldots, \bigcup_{i \in L} \rho^i(n)) \tag{16}$$

Proof. One single run exists in the system $\mathcal{S}^{sync}$ describing the network (L, γ, e) because global transitions are deterministic. Let's call this run ρ. We have to show that $(\bigcup_{i\in L} \rho^i(0), \ldots, \bigcup_{i\in L} \rho^i(n)) = (\rho^c(0), \ldots, \rho^c(n))$ with ρ^c the unique run defining the centralized version of the network, i.e., $\mathcal{T}^c$. This is enough to prove the theorem since from Theorem 1 we already know that $(\rho^c(0), \ldots, \rho^c(n)) = (\mathcal{M}_{\Sigma^c}[0], \ldots, \mathcal{M}_{\Sigma^c}[n])$. Let's proceed by induction. For the base step we have that $\bigcup_{i\in L} \rho^i(0) = \mathbf{Init} \cup \texttt{time}(0) = \rho^c(0)$. For the inductive step, instead, we assume $\bigcup_{i\in L} \rho^i(n) = \rho^c(n)$ and we have to demonstrate that $\bigcup_{i\in L} \rho^i(n+1) = \rho^c(n+1)$. From section the proof of Theorem 1 we already know that $\rho^c(n+1) = \mathbf{Init}[n+1] \cup \texttt{time}(n+1) \cup \mathcal{M}_\Upsilon[n]$, with $\mathcal{M}_{\Upsilon_{ded}}[n] = \mathit{deduction}(\mathcal{T}_{ded})(T_{db}[n] \cup T_{mem}[n] \cup \texttt{time}(n))$, $\rho^c(n) = T_{db}[n] \cup T_{mem} \cup \texttt{time}(n)$ and $\mathcal{M}_\Upsilon[n] = \mathit{induction}(\mathcal{T}_{ind})(\mathcal{M}_{\Upsilon_{ded}}[n])$. For the other side of the equivalence we have that $\bigcup_{i\in L} \rho^i(n+1) = (\bigcup_{i\in L} \mathbf{Init}^i[n+1]) \cup \texttt{time}(n+1) \cup (\bigcup_{i\in L} \mathcal{M}^i_\Upsilon[n])$. Again, $(\bigcup_{i\in L} \mathcal{M}^i_\Upsilon[n]) = \bigcup_{i\in L} \mathit{induction}\,(\mathcal{T}_{ind})(\mathcal{M}^i_{\Upsilon_{ded}}[n])$, with, as usual, $\mathcal{M}^i_{\Upsilon_{ded}}[n] = \mathit{deduction}(\mathcal{T}_{ded})(\rho^i(n))$. Then, since we have that $\bigcup_{i\in L} \mathbf{Init}^i[n+1] = \mathbf{Init}[n+1]$, it remains to prove that $\bigcup_{i\in L} \mathcal{M}^i_\Upsilon[n] = \mathcal{M}_\Upsilon[n]$. This is indeed the case since $\mathcal{M}_\Upsilon[n] = \mathit{induction}(\mathcal{T}^c_{ind})(\mathit{deduction}(\mathcal{T}^c_{ded})(\rho^c(n)))$, $(\bigcup_{i\in L} \mathcal{M}^i_\Upsilon[n]) = \mathit{induction}\,(\mathcal{T}_{ind})(\mathit{deduction}(\mathcal{T}_{ded})(\bigcup_{i\in L} \rho^i(n)))$, $\mathcal{T}^c = \mathcal{T}$ and $\bigcup_{i\in L} \rho^i(n) = \rho^c(n)$ for the inductive assumption.

5 Conclusion

Pushed by recent trends of highly distributed systems, we propose the semantics of a distributed version of Datalog$^\neg$ in synchronous settings. We introduced a new type of *Relational Transducer* [3] and *Transducer network* [6] tailored for synchronous distributed systems, and sketched the model-theoretic semantics of distributed Datalog programs for such systems.

References

1. S. Abiteboul, M. Bienvenu, A. Galland, and E. Antoine. A rule-based language for web data management. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 293–304, New York, NY, USA, 2011. ACM.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
3. S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 179–187, New York, NY, USA, 1998. ACM.
4. P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 223–236, New York, NY, USA, 2010. ACM.
5. P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In de Moor et al. [23], pages 262–281.

6. T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 283–292, New York, NY, USA, 2011. ACM.
7. T. J. Ameloot and J. Van den Bussche. Deciding eventual consistency for a simple class of relational transducer networks. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 86–98, New York, NY, USA, 2012. ACM.
8. H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. *Comput. Lang.*, 16(2):129–146, May 1991.
9. M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In *Temporal Databases*, pages 294–320. 1993.
10. C. Beeri, S. Naqvi, R. Ramakrishnan, O. Shmueli, and S. Tsur. Sets and negation in a logic data base language (ldl1). In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '87, pages 21–37, New York, NY, USA, 1987. ACM.
11. Bloom Language *http://boom.cs.berkeley.edu/index.html*.
12. Cascalog Libray *https://github.com/nathanmarz/cascalog*.
13. S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, Mar. 1989.
14. P. Cholak and H. A. Blair. The complexity of local stratification. *Fundam. Inform.*, 21(4):333–344, 1994.
15. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
16. J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.
17. D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In *ISLP*, pages 387–401, 1991.
18. P. Koutris, D. Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '11, pages 223–234, New York, NY, USA, 2011. ACM.
19. G. Lausen, B. Ludäscher, and W. May. On active deductive databases: The statelog approach. In *International Seminar on Logic Databases and the Meaning of Change, Transactions and Change in Logic Databases*, ILPS '97, pages 69–106, London, UK, UK, 1998. Springer-Verlag.
20. B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 75–90, New York, NY, USA, 2005. ACM.
21. B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 97–108, New York, NY, USA, 2006. ACM.
22. B. Ludäscher. *Integration of Active and Deductive Database Rules*, volume 45 of *DISDBIS*. Infix Verlag, St. Augustin, Germany, 1998.
23. O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, editors. *Datalog Reloaded - First International Workshop, Datalog 2010,Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702 of *Lecture Notes in Computer Science*. Springer, 2011.

24. B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, Aug. 1991.
25. J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel Distrib. Technol.*, 4(2):63–79, June 1996.
26. T. C. Przymusinski. *On the declarative semantics of deductive databases and logic programs*, pages 193–216. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.
27. R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *J. Log. Program.*, 23(2):125–149, 1995.
28. R. T. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, 1986.
29. M.-O. Stehr, M. Kim, and C. Talcott. Toward distributed declarative control of networked cyber-physical systems. In *Proceedings of the 7th international conference on Ubiquitous intelligence and computing*, UIC'10, pages 397–413, Berlin, Heidelberg, 2010. Springer-Verlag.
30. C. Zaniolo. *Advanced database systems*. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, 1997.
31. W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 615–626, New York, NY, USA, 2010. ACM.