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 of Datalog-based languages for highly distributed systems [9], in this paper we introduce a version of Datalog$^{\neg}$ specifically tailored for distributed programming [2] in synchronous settings, along with its operational and declarative semantics.

1 Introduction

Nowadays we are living the "*end of the Moore's law as we know it*": increasing performance cannot be achieved just by increasing hardware speed; instead, concurrent and parallel computation must be exploited. Thus, in the last few years new paradigms such as cloud computing, frameworks like MapReduce [7], and systems like NoSQL databases, are put at work to help programmers in developing scalable applications which exploit distributed systems. We are however aware that limitations still exist in the current literature concerning the formal models needed by the above discussed new trends. Our aim is then to develop new techniques that can be employed to solve the challenges of today's highly 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 provides a firm basis to develop more complex languages for modern parallel and distributed applications [6, 5]; (ii) its solid logical foundation provides the theoretical background that permits to formally specify and analyze complex distributed systems [9, 3].

In particular, the role of our present work is to define what a *synchronous distributed system* is and to provide the semantics for such kind of systems for a version of Datalog specifically tailored for distributed programming [2]. Albeit asynchronous systems are highly preferable today to synchronous systems, the latter are still very convenient because programs do not have to deal with much uncertainty, thus, once designed for such an ideal model, they can be refined to work in more realistic settings. Our work, in particular, proceeds of our previous work [11], and is motivated by the following two use-cases: (i) *active deductive databases* [13] have been developed in centralized settings in which updates must be issued to a unique central database: our aim is to enhance this model, developing the semantics of synchronous networks of partitioned and/or replicated databases; (ii) in the *Massive Parallel* (MP) model introduced in [10], computation proceeds by *steps* that are performed in parallel by clusters of machines on which the same program is running. This model actually is a particular instantiation of a synchronous system. In addition, the MP model is strictly related to the MapReduce model (MR)[10]. As a consequence, by developing a semantics for synchronous systems we would be able to seamlessly embed in our framework both the MP and the MR computation models.

The paper is organized as follows: in Section 2 we introduce some preliminaries on Datalog$^{\neg}_{ST}$, i.e., Datalog$^{\neg}$ augmented with a notion of time and space. In Section 3 we describe what a synchronous distributed systems is, and give an idea of the operational and declarative semantics of Datalog$^{\neg}_{ST}$ in such timing model. [1]

2 Preliminaries

In the following we define a *distributed system* to be a non-empty finite set of *fully* and *reliably connected* node identifiers $L = \{1, \ldots, n\}$. A Datalog$^{\neg}_{TS}$ program Π is composed by a set of rules, each having one of the following two forms:

$$H(\bar{w}, t) \leftarrow B_1(\bar{u}_1, t), \ldots, B_n(\bar{u}_n, t), S_1(@l_1, \bar{v}_1, t), \ldots, S_m(@l_m, \bar{v}_m, t), \quad \texttt{time}(t). \tag{1}$$

$$H(\bar{w}, t') \leftarrow B_1(\bar{u}_1, t), \ldots, B_n(\bar{u}_n, t), S_1(@l_1, \bar{v}_1, t), \ldots, S_m(@l_m, \bar{v}_m, t), \quad \texttt{time}(t), \texttt{succ}(t, t'). \tag{2}$$

where H, B_i and S_j, with $0 \leq i \leq n$, $0 \leq j \leq m$, are taken from the set **relname** of *relation* (*names*), and $(\bar{w})$, $(\bar{u}_i)$, $(\bar{v}_j)$ are tuples whose components (*terms*) can either be *variables* or *constants*. In addition, we have two "special" terms: a *temporal* term t (t') called *time-step identifier* and ranging over the set of natural numbers $\mathbb{N}_0$, and a *spatial* term l_j called *location specifier* and ranging over the set of node identifiers L. We assume that the time-step identifier term cannot appear in any of the vectors $\bar{w}$, $\bar{u}_i$, $\bar{v}_j$, while we will see that the location specifier is allowed to appear in $\bar{w}$. As usual, the left-hand part of a rule ($H(\bar{w})$) is referred to as the *head*, and the right-hand part as the *body*. The head of each rule is an *atom*, i.e. an occurrence of a predicate, while the $B_i(\bar{u}_i, t) - S_j(\bar{v}_j, t)$ are *literals*, i.e., positive or negated atoms. If $m = 0$ the rule is *local*, *distributed* otherwise. Moreover, if $n = m = 0$ the rule is expressing a *fact*. We assume each rule to be *safe*, i.e., every variable occurring in a rule-head or in negative literal, must appear in at least one positive literal of the rule body. As a consequence, we assume all facts to be *ground*, i.e., containing only constants terms.

A (*temporal-spatial*) *database schema* $\mathbf{R}_{TS}$ is defined as the set of relation names $R \in$ **relname** plus two pairs of built-in relations: `succ` (of arity two), and `time` (of arity one) ranging over time-step identifiers, and `id` and `all`, both of arity one and ranging over the set of location specifiers. The interpretation of $\texttt{succ}(t, t')$ is $t' = t + 1$, while $\texttt{time}(t)$ is used to bind the time-step of a tuple to a precise time t. As for the interpretation of `id` and `all`, an occurrence of the first is in each node and stores only that node id, while the second contains all the nodes

[1] The formal basis of this work, along with many examples, is developed in [12].

belonging to the network, i.e., L. The *intensional* part of the database schema, namely $idb(\mathbf{R}_{TS}, \Pi)$, is the subset of the database schema $\mathbf{R}_{TS}$ containing all relations that appear in at least one rule-head in Π, while with $edb(\mathbf{R}_{TS}, \Pi)$ we denote the *extensional database*. In addition, with $sdb(\mathbf{R}_{TS}, \Pi)$ – disjoined from both *edb* and *idb* – we refer to the set of *distributed shared relations* (DSR), which contains the relations represented as S_j in eq.s (1, 2) above, and are the only ones allowed to contain location specifier terms. As the name highlights, DSR's are shared among multiples nodes simultaneously, and are the means by which nodes are able to interact among themselves; therefore a mechanism for deciding which node is responsible for which tuple must be developed. The location specifier term exactly serves at this scope: it is used to identify which node is responsible for which portion of data. From the above definitions and eq.s (1, 2) follows that $B_i \in edb(\mathbf{R}_{TS}, \Pi) \cup idb(\mathbf{R}_{TS}, \Pi)$, $S_j \in sdb(\mathbf{R}_{TS}, \Pi)$ and finally $H \in idb(\mathbf{R}_{TS}, \Pi) \cup sdb(\mathbf{R}_{TS}, \Pi)$, with the meaning that H can specify an intensional relation or a distributed relation. In the latter case, we assume $\bar{w}$ to contain also the location specifier term. In particular, in case of rules of the type of (1) the location specifier term in w must be the local node id, while for rules of type (2) can be any node identifier in L, so that, with inductive rules, we are able to perform *communication* among nodes. Then, given a time-step value $t \in \mathbb{N}_0$, a (*temporal-spatial*) *database instance* is a finite set of facts $\mathbf{I}[\mathrm{t}]$ over the relations of $\mathbf{R}_{TS}$. Similarly, a (*temporal-spatial*) *relation instance* $I_R \subseteq \mathbf{I}[t]$ is a set of facts defined over R, with $R \in \mathbf{R}_{TS}$. In the following we will refer to **Init** as the *finite initial database instance* of the schema.

In Datalog$^{\neg}_{TS}$, we consider tuples to be *immutable* – once instantiated they cannot be retracted nor changed – and *ephemeral* by default, i.e., a tuple is valid only for its assigned time-step. Hence, we use rules in the form of (1) to derive new facts given the information currently available at that point in time. Adopting the same notation of [2] we call this type of rules *deductive*. Nevertheless, in Datalog$^{\neg}_{TS}$ *mutable* tuples can be *emulated* using time and 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 [2] in the form of eq. (2), we specify a new immutable version which will be valid at time-step $t + 1$. If defined by means of inductive rules, tuples from ephemeral become *persistent*: once derived, for example at time-step t, they will eventually last for every $t' \geq t$.

As example, we use the program depicted in Listing 1.1 where we employ an *edb* relation `link` containing tuples in the form `(x,y,t)` to specify the existence of a directed link between a source node `x` and a destination node `y` at time `t`. In addition, we employ a distributed shared relation `path` to define the transitive closure of the `link` relation.

```
r1:   link(x,y,t′):-link(x,y,t),time(t),succ(t,t′).
r2:   path(@x,y,t′):-path(@x,y,t′),time(t),succ(t,t′).

r3:   path(@x,y,t):-link(x,y,t),time(t).
r4:   path(@x,z,t′):-path(@x,y,t),path(@y,z,t),time(t),succ(t,t′).
```

Listing 1.1. Simple Datalog$^{\neg}_{ST}$ Program

Computation is performed simultaneously on multiple distributed nodes. Communication among nodes is achieved through rules `r4` which specifies for example 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 to c exists. Rule `r4` contains relations stored at different locations, therefore implicitly assuming tuples to be exchanged among nodes.

3 Semantics for Synchronous Systems

The operational semantics of Datalog for distributed systems is classically based on the notion of relational transducer and network of relational transducers [3]. We follow the same approach. Given a Datalog$^{\neg}_{TS}$ program Π defined over a relational schema $\mathbf{R}_{TS}$, we define the *database*, *memory*, and *dsr* schemas, respectively, as $\mathbf{R}_{db} = edb(\mathbf{R}_{TS}, \Pi)$, $\mathbf{R}_{mem} = idb(\mathbf{R}_{TS}, \Pi)$, 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 ($\mathbf{R}_{db}$, $\mathbf{R}_{mem}$, $\mathbf{R}_{dsr}$, $\mathbf{R}_{time}$), and a *Datalog$^{\neg}_{TS}$ -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 $T_{db} = \mathbf{Init}$, `time` contains the tuple (0), and all the other relations are empty. In addition, we assume to have a (possibly infinite) *input tape* containing an ordered sequence of consecutive natural numbers starting from 0. 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 3.3, where we define synchronous transducer networks. Then, 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}. Thus, a *local transition* is 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, T_{in}, T'_{out} are instances of $\mathbf{R}_{dsr}$, and (i, L) is a configuration. Now, if we denote with $\mathcal{M}_{ded}$ the *perfect model* of the deductive rules in $\mathcal{T}$ computed by employing the so called *temporal stratified semantics* [2, 12] to control negation, $\mathcal{M}_{ind}$ the model derived from $\mathcal{M}_{ded}$ by firing all the inductive rules, and analogously $\mathcal{M}_{com}$ is derived from $\mathcal{M}_{ded}$ by firing all the communication rules – i.e., inductive rules of the type of eq. (2) where the head relation is a DSR – then $\mathbf{T}'$, T'_{out} are transducer states such that:

- $T'_{db} = T_{db}$
- $T'_{time} = \texttt{time}(\mathrm{t})$
- $T'_{dsr} = \mathcal{M}^{i}_{com}$ – i.e., *sdb* tuples referred to the local node i
- $T'_{out} = \mathcal{M}_{com}/\mathcal{M}^{i}_{com}$
- $T'_{mem} = \mathcal{M}_{ind}$

3.1 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 $\mathcal{I}^i = \mathbf{T}^i[n]$, being $\mathbf{T}^i[n]$ a transducer state for node i at time-step n. 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 and s^e is the *environment* local state. We can consider the environment as a "special" node storing all the information external to all nodes. We define how global states may change over time through the notion of *run*, which binds time values to global states. If we assume the set of time values to be isomorphic to the 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 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 is definitely an internal (relative) perspective that each node has about the passing of time. 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.

3.2 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 delay Δ [8]

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, i.e., by assuming the time tape to be globally accessible by every node. 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 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.

3.3 Synchronous Transducer Networks

We have already defined how local states evolve by defining what a relational transducer is. Now that we have defined synchronous systems, 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* [3]: a *synchronous Datalog$^{\neg}_{TS}$-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 $\mathbf{T}^i$. We 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 $\mathbf{R}^e_{dsr} = \bigcup_{i \in L} \mathbf{R}^i_{dsr}$. Hence, we consider the environment as registering the *sdb* facts floating in the network and not received yet. 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 $\mathbf{R}_{db}$ all the remaining relations are empty. A *global transition* is denoted by $\mathbf{N} \overset{t}{\Longrightarrow} \mathbf{N}'$, 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. Then, a global transition is defined such that:

- $T_{in} = \mathbf{T}^e[t]$
- $\forall i \in L$, $\exists \mathbf{T}^i$ 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. 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 $\mathcal{I}^i = \mathbf{T}^i[t]$ where $\mathbf{T}^i[t]$ denotes the local Datalog$^{\neg}_{TS}$-transducer state for node $i \in L \cup e$ at time-step t, the definition of global transition specifies how global states evolve in synchronous settings because it satisfies conditions **S1** - **S3**. Given a Datalog$^{\neg}_{TS}$ 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.

3.4 Model-Theoretic Semantics

Starting from a synchronous transducer network $(L, \gamma, \mathcal{T}, \mathcal{T}^e)$, we can develop a restricted form of Datalog$^{\neg}_{TS}$-relational transducer, namely *Datalog$^{\neg}_{T}$-relational transducer*, which is able to simulate the behavior of such a network in centralized settings. Thus, given a transducer network $(L, \gamma, \mathcal{T}, \mathcal{T}^e)$ with schema $\mathcal{R}$ = ($\mathbf{R}_{db}$, $\mathbf{R}_{mem}$, $\mathbf{R}_{dsr}$, $\mathbf{R}_{time}$, $\mathbf{R}_{sys}$), 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 $\mathbf{R}^c_{mem} = \mathbf{R}_{mem} \cup \mathbf{R}^{\dagger}_{dsr}$ – where with $\mathbf{R}^{\dagger}_{dsr}$ we denote $\mathbf{R}_{dsr}$ restricted over the temporal database – and $\mathcal{T}^c$ 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 but where each *sdb* relation is now an *idb* relations. Due to this restriction, the environment and the *system* relations are no more necessary. We are now able to derive the model-theoretic semantics of synchronous transducer networks.

Theorem 1 *Let **Init** be a finite initial database instance,* $(L, \gamma, \mathcal{T}, \mathcal{T}^e)$ *a transducer network, and* $\mathcal{M}^c$ *the perfect model of the centralized version of* $(L, \gamma, \mathcal{T}, \mathcal{T}^e)$. *Then:*

$$\mathcal{M}^c = (\bigcup_{i \in L} \rho^i(0), \ldots, \bigcup_{i \in L} \rho^i(n)) \tag{3}$$

4 Conclusion

Pushed by recent trends of highly distributed systems, we tried to give an intuition of the semantics of a distributed version of Datalog$^\neg$ in synchronous settings. We introduced a new type of *Relational Transducer* [1] and *Transducer network* [3] tailored for synchronous distributed systems, and sketched the model-theoretic semantics of distributed Datalog programs for such systems.

References

1. S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *Proceedings of the seventeenth symposium on Principles of database systems*, PODS'98,179–187. 1998.
2. P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, et al. Dedalus: Datalog in time and space. In de Moor et al. Datalog Reloaded, 262–281. 2010
3. T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Proceedings of the thirtieth symposium on Principles of database systems*, PODS'11, 283–292. 2011.
4. P. Barceló and R. Pichler, editors. *Datalog in Academia and Industry - Second International Workshop, Datalog 2.0, Vienna, Austria, September 11-13, 2012. Proceedings*, volume 7494 of *Lecture Notes in Computer Science*. Springer, 2012.
5. Bloom Language *http://boom.cs.berkeley.edu/index.html.*
6. Cascalog Libray *https://github.com/nathanmarz/cascalog.*
7. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
8. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge, MA, USA. 2003.
9. J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.
10. P. Koutris, D. Suciu. Parallel evaluation of conjunctive queries. In *Proceedings of the thirtieth symposium on Principles of database systems*, PODS'11. 2011.
11. M. Interlandi. Reasoning about knowledge in distributed systems using datalog. In Barceló and Pichler [4], pages 99–110.
12. M. Interlandi, L. Tanca, S. Bergamaschi: Datalog in Time and Space. Synchoronusly. `http://www.dbgroup.unimo.it/TechnicalReport/interlandi2013.pdf`, Febbraury 2013. Technical Report.
13. C. Zaniolo. *Advanced database systems.* Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers. 1997.