Enhancing Datalog with Epistemic Operators to Reason About Knowledge in Distributed Systems*

Matteo Interlandi

University of Modena and Reggio Emilia
matteo.interlandi@unimore.it

Abstract. In the last few years, researchers started to investigate how recursive queries and deductive languages can be applied to find solutions to the new emerging trends in distributed computing. We conjecture that a missing piece in the current state-of-the-art in logic programming is the capability to express statements about the knowledge state of distributed nodes. In fact, reasoning about the state of remote nodes is fundamental in distributed contexts in order to design and analyze protocol behavior. To reach this goal, we leveraged Datalog$^\neg$ with an epistemic modal operator, allowing the programmer to directly express nodes' state of knowledge instead of low level communication details. To support the effectiveness of our proposal, we introduce, as example, the declarative implementation of a well-known protocol employed to execute distributed databases transactions: the two phase commit protocol.

1 Introduction

Pushed by the new interest that Datalog is acquiring in the database community, the goal of this paper is to open a new direction in the investigation on how Datalog could be adopted to program distributed systems. Many authors have stated how logic programming in general [6] and Datalog in particular [5] seems to particularly fit the representation of distributed programs implementation and properties. We think that a missing point is the possibility to express statements about the knowledge state of distributed nodes in Datalog. In fact, the ability to reason about the knowledge state of remote nodes has been demonstrated [4] to be a fundamental tool in multi-agent systems in order to specify global behaviors and properties of protocols. Motivated by all these facts, we leveraged Datalog$^\neg$ with an epistemic modal operator, allowing the programmer to express directly nodes' state of knowledge instead of low level communication details. To support our assertions, we describe our implementation of the two phase commit protocol. The remainder of the paper is organized as follow: Section 2 contains some preliminary notations about Datalog$^\neg$. Section 3 describes what we intend for

* This work is partially taken from [9].

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

a distributed system and introduces some concept such as *global state*, *run* and the modal operator K. Section 4 introduces Knowlog and the implementation of the two phase commit protocol. The paper finish with conclusions and future work.

2 Preliminaries

In order to define Knowlog, we first introduce some principles of Datalog$^\neg$ [1], and Datalog$^\neg$ augmented with temporal constructs [8, 3]. A Datalog$^\neg$ rule is an expression in the form:

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

where $n, m \geq 0$, H, B_i, C_j are relation names $i = 0, ..., n$ and $j = 0, ..., m$ and $\bar{u}, \bar{u}_i, \bar{v}_j$ are tuples of appropriate arities. Tuples are composed by *terms* and each term can be a constant in the domain **dom** or a variable in the set **var**. We will use interchangeably terms *predicates* and *relations*. As usual $H(\bar{u})$ is referred as the *head*, $B_i(\bar{u}_i)$, $C_j(\bar{v}_j)$ as the *body*, and in general $H(\bar{u})$, $B_i(\bar{u}_i)$ and $C_j(\bar{v}_j)$ as *atoms*. A *literal* is an atom (in this case we refer to it as *positive*) or the negation of an atom. If $m = n = 0$ and does not contains variable terms, the rule express a *fact* or equivalently a *groud atom*. In this paper we assume that each rule is *range restricted*, i.e. every variable occurring in a rule-head appears in at least one positive literal of the rule body. Then, a *Datalog$^\neg$ program* Π is a set of range restricted rules. For a database schema **R**, a *database instance* is a finite set **I** constructed by the union of the relation instances over R with $R \in \mathbf{R}$ a relation name and where each relation instance is a finite set of facts. As introductory example, we use the program depicted in Listing 1.1 where we used a relation `link`, containing tuples in the form `(S,D)`, to specify the existence of a link between a source node `S` and a destination node `D`. In addition we employ the `path` relation, which is computed starting from the `link` relation (`r1`) and recursively adding a new path when, roughly speaking, there is a link from `A` to `B` and already exists a path from `B` to `C` (`r2`).

```
r1: path(X,Y):-link(X,Y)
r2: path(X,Z):-link(X,Y),path(Y,Z)
```

Listing 1.1. Simple Recursive Datalog Program

2.1 Time in Datalog$^\neg$

With the language we are introducing, we want to model programs for distributed systems. These systems are not static, but evolving with time. Therefore it will be useful to enrich Datalog$^\neg$ with some notion of time. To reach this goal we follow the road traced by Statelog [8] and Dedalus [3]. Thus, informally, each relation is labeled with a *time-step identifier* having values in $\mathbb{N}$ and which specify at what time-step a given instance has been derived and is true. A consequence of this approach is that tuples by default are considers *ephemeral*, i.e., they are valid only for one single time-step. Obviously, tuples can became *persistent* - once derived, for example at time s, they last for every time $t \geq s$ - if

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

they are stored in *persistent* relations. Among the different temporal extensions of Datalog¬ available in the literature, we embrace the Dedalus [3] notation, thus programs' rules are divided in two sets: *inductive* and *deductive*. The former set contains all the rules employed for transfer tuples among time-steps i.e., persistency rule, while the latter encompasses the rules that are local into a single time-step. Some syntactic sugar is adopted in order to better characterize rules and relations: deductive rules appears as usual Datalog¬ rules, while a `next` suffix is introduced in head relations to characterize inductive rules. In Listing 1.2 the simple program of the previous section is rewritten to introduce the new formalism: a persistent relation rule (`r1`) and a rule for the modification of the `link` relation if a tuple is issued to the ephemeral relation `link_down`, representing an event on the link which cause the link to be disconnected.

```
r1: link(X,Y)@next:-link(X,Y),¬del_link(X,Y)
r2: del_link(X,Y):-link_down(X,Y)
r3: path(X,Y):-link(X,Y)
r4: path(X,Z):-link(X,Y),path(Y,Z)
```

Listing 1.2. Inductive and Deductive Rules

3 Distributed Logic Programming

Before starting the discussion on how we leverage the language with epistemic operators, we first introduce our model of distributed system and how communication among nodes is performed. We define a distributed message-passing system to be a non empty finite set N of share-nothing nodes joined by bidirectional communication links. Each node identifier has a value in the domain **dom** but here we consider the set $N = \{1, ..., n\}$ of node identifiers, where n is the total number of nodes in the system. We identify with *adb* a new set of *accessible* relations encompassing all the tables that are horizontally partitioned among nodes and through which nodes are able to communicate. Each relation $R \in adb$ contains a *location specifier* term [7]. This term maintains the identifier of the remote node to which every new fact inserted into the relation R must be issued. As pointed out in [5, 3], modeling communication using relations provides major advantages. Continuing with the examples introduced in the previous sections, in order to describe Listing 1.3 we can imagine a real network configuration where each node has locally installed the program, and where each `link` relation reflect the actual state of the connection between nodes. For instance, we will have the fact `link(A,B)` in node A's instance if a communication link between A and node B exists. The location specifier term is identified by the `@` prefix.

```
r1: link(X,Y)@next:-link(X,Y),¬del_link(X,Y)
r2: del_link(X,Y):-link_down(X,Y)
r3: path(@X,Y):-link(X,Y)
r4: path(@X,Z):-link(X,Y),path(@Y,Z)
```

Listing 1.3. Distributed Program

The semantics of the program in Listing 1.3 is the same as in the previous section, even though operationally it substantially differs. In fact, in this new

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

version, computation is performed simultaneously on multiple distributed nodes. Communication is achieved through rule `r4` which, informally, specify that a path from a node A to a node C exists if there is a link from A to another node B and this last knows that exist a path from B to C.

3.1 The Knowledge Model

In every point in time, each node is in some particular *local state* incapsulating all the information the node possesses. We use s_i to denote the local state of node i . We define the *global state* of a distributed system as a tuple $(s_1, ..., s_n)$ where s_i is the node i's state. We define how global states may change over time through the notion of *run*, which binds time values to global states, i.e., $r : \mathbb{N} \to \mathcal{G}$ where $\mathcal{G} = \{S_1 \times ... \times S_n\}$ and S_i is the set of possible local state for node $i \in N$. Following [4] we define a *system* 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, abstracting away many low level details. In knowledge-based systems, nodes are able to accomplish actions not only based on their local state, but also on the knowledge the node has, i.e., the information the node has about the state of the system. If we consider two runs of a system, with global states respectively $g = (s_1, ..., s_n)$ and $g' = (s'_1, ..., s'_n)$, g and g' are *indistinguishable* for process i, and we will write $g \sim_i g'$ if i has the same local state both in g and g', i.e., $s_i = s'_i$. We use the modal operator K_i and we write $K_i\psi$ to express that a node i knows sentence ψ: in every global state that i considers possible - i.e., all the global state that are indistinguishable for i - the sentence ψ is true. This definition of knowledge follows the axioms that are called *S5*. We refer the reader to [9] for a detailed discussion about the modal operator K.

4 Incorporating Knowledge: Knowlog

We employ $\Box$ to denote a (possible empty) sequence of modal operators K and we use the following statement to express it in a rule form:

$$\Box(H \leftarrow B_1, ..., B_n, \neg C_1, ..., \neg C_m) \tag{1}$$

with $n, m \geq 0$ and each positive literal is in the form $\Box R$, while negative literals are in the form $K_i\Box R$ where K_i is equal to the *model context*. From [10] we adopt the term *modal context* to refer to the sequence - with the maximum length of one - of modal operators appearing in front of a rule. We put some restriction on the sequence of operators permitted in $\Box$.

Definition 1. *Given a (possibly empty) sequence of operators $\Box$, $\Box$ is in restricted form if it does not contain K_iK_i subsequences, with i specifying a process identifier.*

Definition 2. *A Knowlog program is a set of rules in the form (1), containing only (possible empty) sequences of modal operators in the restricted form and where the subscript i of each modal operator K_i can be a constant or a variable.*

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

Informally speaking, given a Knowlog program, using modal context we are able to assign to each node the rules the node is responsible for, while atoms and facts residing in the node i are in the form $K_i \Box R$. We define communication rules as follow:

Definition 3. *A communication rule in Knowlog is a rule where no modal context is set and the body atoms have the form $K_i \Box R$ - namely they are prefixed with modal operators related to the same process - while the head atom has the form $K_j \Box R'$, with $i \neq j$ and not necessarily $R' \neq R$.*

In this way, we are able to abstract away all the low level details about how information is exchanged, leaving to the programmer just the task to specify what a process should know, and not how. For the definition of the Knowlog semantics we refer to [9].

The Two-Phase-Commit Protocol Inspired by [2], we implemented the two-phase-commit protocol (2PC) using the epistemic operator K. 2PC is used to execute distributed databases transaction and it is divided in two phases: in the first phase, called the *voting phase*, a coordinator node submit to all the transaction's participants the willingness to perform a distributed commit. Consequently, each participant sends a vote to the coordinator, expressing its intention. In the second phase - namely the *decision phase* - the coordinator collects all votes and decides if performing global *commit* or *abort*. The decision is then issued to the participants which act accordingly. In the 2PC implementation of Listing 1.4, we assume that our system is composed by three nodes: one coordinator and two partecipants. Due to the lack of space, we considerably simplify the 2PC protocol. For a more detailed discussion we refer the reader to [9].

```
\\Initialization at coordinator
   r1: Kc(part_cnt(count<N>):-nodes(N))
   r2: Kc(transaction(Tx_id,State):-log(Tx_id,State))
\\Decision Phase at coordinator
   r3: Kc(yes_cnt(Tx_id,count<part>):-vote(Vote,Tx_id,part),Vote == "yes"))
   r4: Kc(log(Tx_id,"commit")@next:-part_cnt(C),yes_cnt(Tx_id,C1),C==C1,
        State=="vote-req",transaction(Tx_id,State))
   r5: Kc(log(Tx_id,"abort"):-vote(Vote,Tx_id,part),Vote == "no",
        transaction(Tx_id,State),State =="vote-req")
\\Voting Phase at partecipants
   r6: Kp(log(Tx_id,"prepare"):-State=="vote-req",Kctransaction(Tx_id,State))
   r7: Kp(log("abort",Tx_id):-log(Tx_id,State),State=="prepare",
        db_status(Vote),Vote=="no")
\\Decision Phase at partecipants
   r8: Kp(log(Tx_id,"commit"):-log(Tx_id,State_l),State_l=="prepare",
        State_t=="commit",Kctransaction(Tx_id,State_t))
   r9: Kp(log(Tx_id,"abort"):-log(Tx_id,State_l),State_l=="prepare",
        State_t=="abort",Kctransaction(Tx_id,State_t))
\\Communication
   r10:Kxtransaction(Tx_id, State):-Kcsubs(X),
```

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

```
     Kctransaction(Tx_id,State),Kcpath(@Y,X)
r11:Kcvote(Vote,Tx_id,"sub1"):-KP1log(Tx_id,State),
     State=="prepare",KP1db_status(Vote),KP1path(@P1,C)
r12:Kcvote(Vote,Tx_id,"sub2"):-KP2log(Tx_id,State),
     State=="prepare",KP2db_status(Vote),KP2path(@P2,c)
```

Listing 1.4. Two Phase Commit Protocol

In the above example, for simplicity we wrote K_{P} as a modal context instead of K_{P1} and K_{P2}.

5 Conclusion and Future Work

In this paper we present Knowlog, a programming language based on Datalog$^{\neg}$ leveraged with a notion of time and modal operators. Through Knowlog, reasoning about state of knowledge in distributed systems can be performed, therefore lighten the programmer's burden of expressing low level communication details. What we discussed here is a first step towards the definition of a comprehensive logical framework able to define a declarative as well as operational semantics, and generic enough to be adopted in multiple contexts. We are confident that following our approach, properties such as processes coordination and replicas consistency can be exhaustively defined.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Alvaro, P., Condie, T., Conway, N., Hellerstein, J.M., Sears, R.: I do declare: consensus in a logic language. Operating Systems Review **43**(4) (2009) 25–30
3. P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, 2010, 262–28.
4. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge, MA, USA (2003)
5. Hellerstein, J.M.: The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Rec. **39** (September 2010) 5–19
6. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16** (May 1994) 872–923
7. Loo, B.T., Condie, T., Garofalakis, M., Gay, et al. : Declarative networking: language, execution and optimization. In: SIGMOD '06, New York, NY, USA, ACM (2006) 97–108
8. Ludäscher, B.: Integration of Active and Deductive Database Rules. Volume 45 of DISDBIS. Infix Verlag, St. Augustin, Germany (1998)
9. Interlandi, M.: Knowlog: A Declarative Language for Reasoning about Knowledge in Distributed Systems. ER'12 PhD Symposium Florence, Italy (2012)
10. Nguyen, L.A.: Foundations of modal deductive databases. Fundam. Inf. **79** (January 2007) 85–135

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,