Pablo Barceló
Reinhard Pichler (Eds.)

LNCS 7494

Datalog in Academia and Industry

Second International Workshop, Datalog 2.0
Vienna, Austria, September 2012
Proceedings

Springer

Pablo Barceló   Reinhard Pichler (Eds.)

Datalog in Academia and Industry

Second International Workshop, Datalog 2.0
Vienna, Austria, September 11-13, 2012
Proceedings

Springer

Volume Editors

Pablo Barceló
Universidad de Chile, Department of Computer Science
Avenida Blanco Encalada 2120, 3er Piso, 837-0459 Santiago, Chile
E-mail: pbarcelo@dcc.uchile.cl

Reinhard Pichler
Vienna University of Technology, Institute of Information Systems
Favoritenstraße 9-11, 1040 Wien, Austria
E-mail: pichler@dbai.tuwien.ac.at

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-32924-1 e-ISBN 978-3-642-32925-8
DOI 10.1007/978-3-642-32925-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012944718

CR Subject Classification (1998): H.2.4, H.4, I.2.4, C.2, H.3, H.5, D.1.6

LNCS Sublibrary: SL 3 – Information Systems and Application, incl. Internet/Web and HCI

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 2012 Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0 2012) held during September 11–13, 2012, in Vienna. Datalog 2.0 is a workshop for Datalog researchers, implementers, and users. Its main aim is to bring everyone up to date and map out directions for future research. The first edition of this workshop was held in Oxford, UK, in March 2010. It was based on invitations only. Over the past few years, Datalog has been resurrected as a lively topic with applications in many different areas of computer science as well as industry. Owing to this renewed interest and increased level of activity, we decided to open the workshop for submissions this year.

The call for papers resulted in 17 submissions. Each paper was reviewed by at least three Program Committee members. The Program Committee accepted 12 papers, based on their technical merit and potential for stimulating discussions. We also accepted one system description and one tutorial, which show the influence of Datalog in industry and practice today.

In addition, the technical program included invited talks by Thomas Eiter (Technische Universität Wien), Yuri Gurevich (Microsoft Research), Phokion Kolaitis (University of California at Santa Cruz and IBM Research - Almaden), Oege de Moor (University of Oxford), and Marie-Laure Mugnier (University of Montpellier). It also includes invited tutorials by Todd J. Green (University of California at Davis and LogicBlox) and Axel Pollers (Siemens AG Austria). The Datalog 2.0 Workshop 2012 was collocated with two further events: the 6th International Conference on Web Reasoning and Rule Systems (RR 2012) and the 4th International Conference on Computational Models of Argument (COMMA 2012). The invited talk by Robert Kowalski (Imperial College, London) was shared by all three events. Short abstracts of two invited talks are included in the front matter of this volume. Extended abstracts are given in the main body of the proceedings.

The workshop would not have been possible without the support of many people. First of all we would like to thank Georg Gottlob, the General Chair of Datalog 2.0, for his advice and help in putting together the technical program. We are very grateful to Markus Pichlmair (Technische Universität Wien) for all his hard work in the local organization. We also acknowledge EasyChair as a great tool that has significantly simplified the whole process from receiving the submissions to producing the input for the proceedings. Finally, we would like to thank all the authors who contributed to the workshop and the Program Committee members for their effort to produce timely and wise reviews.

We hope that the success of Datalog 2.0 2012 will stimulate forthcoming versions of this worskhop as well as renewed interest from the community in Datalog and its applications.

September 2012

Pablo Barceló
Reinhard Pichler

Organization

The 2012 Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0 2012) was organized by the Institute of Information Systems at the Vienna University of Technology.

Program Committee

General Chair

Georg Gottlob	University of Oxford, UK

Program Chairs

Pablo Barceló	Universidad de Chile, Chile
Reinhard Pichler	Vienna University of Technology, Austria

Members

Chitta Baral	Arizona State University, USA
Piero Bonatti	University of Naples Federico II, Italy
Loreto Bravo	Universidad de Concepción, Chile
Andrea Calì	University of London, Birkbeck College, UK
Diego Calvanese	Free University of Bozen-Bolzano, Italy
Jürgen Dix	Clausthal University of Technology, Germany
Tim Furche	University of Oxford, UK
Claudio Gutierrez	Universidad de Chile, Chile
Georg Lausen	University of Freiburg, Germany
Nicola Leone	University of Calabria, Italy
Annie Liu	State University of New York at Stony Brook, USA
Boon Thau Loo	University of Pennsylvania, USA
Raghu Ramakrishnan	Yahoo! Research, USA
Sebastian Rudolph	AIFB, Karlsruhe Institute of Technology, Germany
Tuncay Tekle	LogicBlox, USA
Miroslaw Truszczynski	University of Kentucky, USA
Stijn Vansummeren	Université Libre de Bruxelles, Belgium
Victor Vianu	UC San Diego, USA
Stefan Woltran	Vienna University of Technology, Austria
Peter Wood	Birkbeck, University of London, UK

Local Organization

Chair

Markus Pichlmair	Vienna University of Technology, Austria

Further Members

Ingo Feinerer	Vienna University of Technology, Austria
Reinhard Pichler	Vienna University of Technology, Austria

Additional Reviewers

Alviano, Mario
Arenas, Marcelo
Caroprese, Luciano
Kaminski, Roland
May, Wolfgang
Montali, Marco
Pieris, Andreas
Puehrer, Joerg
Ricca, Francesco
Terracina, Giorgio

Paraconsistent Modular Answer Set Programming (Abstract)

Thomas Eiter
Vienna University of Technology
eiter@kr.tuwien.ac.at

Paraconsistent reasoning is a well-studied approach to deal with inconsistency in logical theories in a way such that inference does not explode. It has specifically been considered in the area of knowledge representation and reasoning for a range of different formalisms, including also non-monotonic formalisms such as logic programming. In the last years, there has been increasing interest in datalog-based formalisms, including traditional Answer Set Programming, and extensions of the formalisms to encompass modularity, possibly in a distributed environment, have been conceived.

In this talk, we shall address the issue of paraconsistency for modular logic programs in a datalog setting, under the answer set semantics for logic programs. The two orthogonal aspects of modularity and paraconsistency for this semantics, which is subsumed by Equilibrium Logic, may be approached on different grounds. We shall consider developments on these aspects, including proposals for modularity and paraconsistency of answer set programs developed at TU Wien. For the latter particular emphasis is given to the issue of incoherence, i.e., non-existence of answer sets due to the lack of stability caused by cyclic dependencies of an atom from its default negation. We shall then consider possible combinations of the two aspects in a single formalism. In the course of this, we shall discuss issues and challenges regarding semantics and evaluation, both in theory and for practical concerns.

This work is a joint effort with Minh Dao-Tran, Michael Fink, Thomas Krennwallner and others, and supported by the project P20841 “Modular HEX-Programs” of the Austrian Science Fund (FWF).

A Retrospective on Datalog 1.0 (Abstract)

Phokion G. Kolaitis
University of California Santa Cruz & IBM Research - Almaden
kolaitis@cs.ucsc.edu

Datalog was introduced in the early 1980s as a database query language that embodies a recursion mechanism on relational databases and, thus, overcomes some of the inherent limitations in the expressive power of relational algebra and relational calculus. In the ensuing decade, Datalog became the subject of an in-depth investigation by researchers in database theory. This study spanned a wide spectrum of topics, including query processing and optimization of Datalog programs, the delineation of the expressive power and computational complexity of Datalog queries, and the exploration of the semantics and the expressive power of extensions of Datalog with comparison operators and negation. The investigation of these topics entailed extensive interaction of database theory with finite model theory and, in particular, with finite-variable logics and pebble games suitable for analyzing the expressive power of Datalog and its variants. From the early 1990s on, there has been a fruitful and far-reaching interaction between Datalog and constraint satisfaction; this interaction, which continues today, has contributed to the understanding of tractable cases of the constraint satisfaction problem, but has also given rise to new results about the expressive power and the computational complexity of Datalog queries.

The aim of this talk is to reflect on some of the aforementioned topics, highlight selected results, and speculate on future uses and applications of Datalog.

Table of Contents

Abstracts

Paraconsistent Modular Answer Set Programming IX
Thomas Eiter

A Retrospective on Datalog 1.0 XI
Phokion G. Kolaitis

Invited Presentations

LogicBlox, Platform and Language: A Tutorial 1
Todd J. Green, Molham Aref, and Grigoris Karvounarakis

Datalog: A Perspective and the Potential 9
Yuri Gurevich

Existential Rules: A Graph-Based View (Extended Abstract) 21
Marie-Laure Mugnier

How (Well) Do Datalog, SPARQL and RIF Interplay? 27
Axel Polleres

Regular Papers, System Descriptions and Tutorials

Magic-Sets for *Datalog* with Existential Quantifiers 31
Mario Alviano, Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri

On the CRON Conjecture 44
Tom J. Ameloot and Jan Van den Bussche

Order in Datalog with Applications to Declarative Output 56
Stefan Brass

A Broad Class of First-Order Rewritable Tuple-Generating Dependencies 68
Cristina Civili and Riccardo Rosati

Datalog Development Tools (Extended Abstract) 81
Onofrio Febbraro, Giovanni Grasso, Nicola Leone, Kristian Reale, and Francesco Ricca

Query Rewriting Using Datalog for Duplicate Resolution 86
Jaffer Gardezi and Leopoldo Bertossi

Reasoning about Knowledge in Distributed Systems Using Datalog 99
Matteo Interlandi

Declarative Datalog Debugging for Mere Mortals 111
Sven Köhler, Bertram Ludäscher, and Yannis Smaragdakis

Inconsistency-Tolerant Query Rewriting for Linear Datalog+/− 123
Thomas Lukasiewicz, Maria Vanina Martinez, and Gerardo I. Simari

Confluence Analysis for Distributed Programs: A Model-Theoretic Approach 135
William R. Marczak, Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier

Business Network Reconstruction Using Datalog 148
Daniel Ritter and Till Westmann

Data Exchange in Datalog Is Mainly a Matter of Choice 153
Domenico Saccà and Edoardo Serra

Optimizing Large-Scale Semi-Naïve Datalog Evaluation in Hadoop 165
Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu

Logical Foundations of Continuous Query Languages for Data Streams 177
Carlo Zaniolo

Author Index 191

LogicBlox, Platform and Language: A Tutorial

Todd J. Green, Molham Aref, and Grigoris Karvounarakis

LogicBlox, Inc,
1349 W Peachtree St NW,
Atlanta, GA 30309 USA
{todd.green,molham.aref,grigoris.karvounarakis}@logicblox.com

Abstract. The modern enterprise software stack—a collection of applications supporting bookkeeping, analytics, planning, and forecasting for enterprise data—is in danger of collapsing under its own weight. The task of building and maintaining enterprise software is tedious and laborious; applications are cumbersome for end-users; and adapting to new computing hardware and infrastructures is difficult. We believe that much of the complexity in today's architecture is accidental, rather than inherent. This tutorial provides an overview of the LogicBlox platform, a ambitious redesign of the enterprise software stack centered around a unified declarative programming model, based on an extended version of Datalog.

1 The Enterprise Hairball

Modern enterprise applications involve an enormously complex technology stack composed of many disparate systems programmed in a hodgepodge of programming languages. We refer to this stack, depicted in Figure 1, as "the enterprise hairball."

First, there is an *online transaction processing* (OLTP) layer that performs *bookkeeping* of the core business data for an enterprise. Such data could include the current product catalog, recent sales figures, current outstanding invoices, customer account balances, and so forth. This OLTP layer typically includes a relational DBMS—programmed in a combination of a query language (SQL), a stored procedure language (like PL/SQL or TSQL), and a batch programming language like Pro*C—an application server, such as Oracle WebLogic [35], IBM WebSphere [36], or SAP NetWeaver [26]—programmed in an object-oriented language like Java, C#, or ABAP—and a web browser front-end, programmed using HTML and Javascript.

In order to track the performance of the enterprise over time, a second *business intelligence* (BI) layer typically holds five to ten years of historical information that was originally recorded in the OLTP layer and performs read-only analyses on this information. This layer typically includes another DBMS (or, more commonly, a BI variant like Teradata [33] or IBM Netezza [25]) along with a BI application server such as Microstrategy [23], SAP BusinessObjects [5], or IBM Cognos [7], programmed using a vendor-specific declarative language. Data is

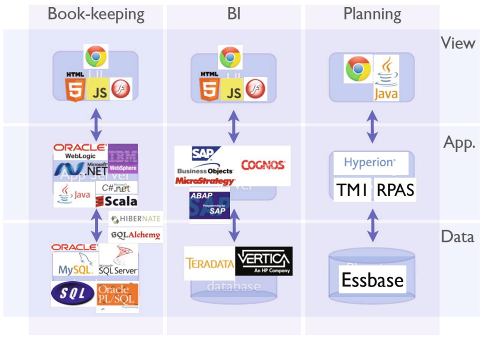


Fig. 1. Enterprise software components and technology stack example

moved from the transaction layer to the BI layer via so-called *extract-transform-load* (ETL) tools that come with their own tool-specific programming language.

Finally, in order to plan the future actions of an enterprise, there is a *planning* layer, which supports a class of read-write use cases for which the BI layer is unsuitable. This layer typically includes a planning application server, like Oracle Hyperion [13] or IBM Cognos TM1 [34], that are programmed using a vendor-specific declarative language or a standard language like MDX [21], and spreadsheets like Microsoft Excel that are programmed in a vendor-specific formula language (e.g. `A1 = B17 - D12`) and optionally a scripting language like VBA. In order to enhance or automate decisions made in the planning layer, statistical predictive models are often prototyped using modeling tools like SAS, Matlab, SPSS, or R, and then rewritten for production in C++ or Java so they can be embedded in the OLTP or OLAP layers.

In summary, enterprise software developed according to the hairball model must be programmed using a dozen or so different programming languages, running in almost as many system-scale components. Some of the languages are imperative (both object-oriented and not) and some are declarative (every possible flavor). Most of the languages used are vendor-specific and tied to the component in which they run (e.g. ABAP, Excel, R, OPL, etc.). Even the languages that are based on open standards are not easily ported from one component to another because of significant variations in performance or because of vendor specific extensions (e.g., the same SQL query on Oracle will perform very differently on DB2). Recent innovations in infrastructure technology for supporting much larger numbers of users (Web applications) and big data (predictive analytics), including NoSQL [28], NewSQL [27] and analytic databases, have introduced even more components into the technology stack described above, and have not helped reduce its overall complexity.

This complexity makes enterprise software hard to build, hard to implement, and hard to change. Simple changes—like extending a product identifier by 3 characters or an employee identifier by 1 digit—often necessitate modifications to most of the different components in the stack, requiring thousands of days of person effort and costing many millions of dollars. An extreme example of the problems caused by this accidental complexity occurred in the lead-up to the year 2000, where the simple problem of adding two digits to the year field (the Y2K problem) cost humanity over $300 billion dollars [24].

Moreover, as a result of the time required for such changes, individuals within an enterprise often resort to ad hoc, error-prone methods to perform the calculations needed in order to make timely business decisions. For example, they often roll their own extensions using spreadsheets, where they incompletely or incorrectly implement such calculations based on copies of the data that is not kept in sync with the OLTP database and thus may no longer accurately reflect the state of their enterprise.

2 LogicBlox

To address these problems, the LogicBlox platform follows a different approach, based on ruthless simplification and consolidation. Its main goal is to unify the programming model for enterprise software development that combine transactions with analytics, by using a single expressive, *declarative* language amenable to efficient evaluation schemes, automatic parallelizations, and transactional semantics. To achieve this goal, it employs *DatalogLB*, a strongly-typed, extended form of Datalog [17,1] that is expressive enough to allow coding of entire enterprise applications (including business logic, workflows, user interface, statistical modeling, and optimization tasks).

Datalog has, historically, been used as a toy language not intended for practical applications, so the choice of Datalog as a unifying language for enterprise application development may be a bit surprising. One reason for its selection for LogicBlox was that, in our experience, Datalog programs are easier to write and understand by the target users of such systems (i.e., business consultants, not Computer Science researchers), compared to languages such as Haskell [12] or Prolog [17]. Moreover, with Datalog we are able to draw on a rich literature for automatic optimizations and incremental evaluation strategies; the latter is of paramount importance for enterprise applications, where small changes to the input of a program are common, and need to be handled at interactive speeds. Changes in an Excel-like spreadsheet application, for instance, need to be reflected immediately.

Today, the LogicBlox platform has matured to the point that it is being used daily in mission-critical applications in some of the largest enterprises in the world. In the tutorial, we present an overview of the language and platform, covering standard Datalog features but emphasizing extensions to support general-purpose programming and the development of various OLTP, BI and planning components of enterprise applications. We also highlight some of the engineering challenges in implementing our platform, which must support mixed transactional and analytical workloads, high data volumes, and high numbers of concurrent users, running applications which are executed—conceptually, at least—entirely within the database system.

3 The DatalogLB Language

The bulk of the tutorial will focus on DatalogLB, the *lingua franca* of the LogicBlox platform. We sketch some highlights of the language in this section.

Rules. DatalogLB rules are specified using a `<-` notation (instead of the traditional "`:-`"), as in the example below:

```
person(x) <- father(x,y).
person(x) <- mother(x,y).
grandfather(x,z) <- father(x,y), father(y,z) ; father(x,y), mother(y,z).
mother(x) <- parent(x,y), !father(x).
```

In this example, `;` indicates disjunction while `!` is used for negation[1]. Predicate and variable names may use lower/upper case freely. The first two rules copy data from the `father` and `mother` predicates into `person`. The third rule computes the `grandfather` predicate, essentially as the union of two conjunctive queries. Finally, the fourth rule specifies that all parents that are not fathers are mothers, with negation interpreted under the stratified semantics.

Entity types and constraints. The main building-blocks of the DatalogLB type system are *entities*, i.e., specially declared unary predicates corresponding to some concrete object or abstract concept. The DatalogLB type system also includes various primitive types (e.g., numeric types, strings etc). For example, the following DatalogLB program declares (using a `->` notation) that `person` is an entity:

```
person(x) -> .
```

Entities can have various properties, expressed through predicates with the corresponding entity as the type of some argument, e.g.,:

```
ssn[x] = y -> person(x), int[32](y).
name[x] = n -> person(x), string(n).
```

The first declaration says that `ssn` is a functional predicate mapping `person` entities to integer-valued Social Security Numbers, while the second maps `person` entities to string names.

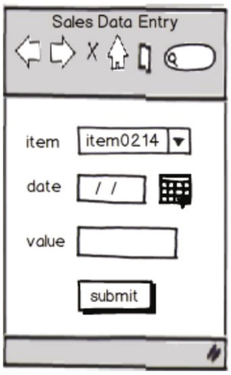
Entities can be arranged in subtyping hierarchies, e.g., the following example declares that `male` is a subtype of `person`:

```
male(x) -> person(x).
```

As expected, subtypes inherit the properties of their supertypes and can be used wherever instances of their supertypes are allowed by the type system. For example, according to the declarations above, a `male` also has an `ssn` and a `name`.

One can also use the `->` notation to specify runtime *integrity constraints*, such as that every `parent` relationship is also either a `father` or `mother` relationship, but not both:

[1] Not to be confused with the Prolog cut operator.



(a)

```
sales_entry_form(f) -> form(f).

form_title[f] = "Sales Data Entry"
   <- sales_entry_form(f).

component[f] = d, dropdown(d), label[d] = "item"
   <- sales_entry_form(f).

submit_button[f] = b, label[b] = "submit"
   <- sales_entry_form(f).
```

(b)

Fig. 2. A UIBlox form and an excerpt from its specification

```
parent(x,y) -> father(x,y), !mother(x,y) ; mother(x,y), !father(x,y).
```

As another example, the `[]` notation used earlier for the predicate `ssn` is just syntactic sugar for the following type declaration and integrity constraint:

```
ssn(x,y) -> person(x), int[32](y).
ssn(x,y), ssn(x,z) -> y = z.
```

Updates and events. The needs of interactive applications motivate procedural features in DatalogLB (inspired by previous work on Datalog with updates [2] and states [16]). For instance, LogicBlox provides a framework for user interface (UI) programming that allows the implementation of UIs over stored data through DatalogLB rules. Apart from being able to populate the UI based on results of DatalogLB programs, UI events are also handled through DatalogLB rules that are executed in response to those events.

For example, consider a simple application in which managers are allowed to use the form in Figure 2a to modify sales data for planning scenarios. This form, including the title of the page, the values in the drop-down menu and the text on the "submit" button, are generated by the DatalogLB rules shown in Figure 2b.

The selection of values for particular items from the drop-down menu, specifying a UI view, also corresponds to a database view:

```
dropdown_values(d,i)
   <- component[f] = d, sales_entry_form_user(f,u), modifiable_by(i,u).
```

UI events, such as when the submit button in Figure 2a is pushed, are represented as predicates, and one can write rules—such as the one below—that are executed when these events happen:

```
^sales[p,d,s] = v
   <- +button_clicked(f,s),
      sales_entry_form_user(f,u), dropdown_selected[f] = p,
      date_fld_value[f,_] = d, num_fld_value[f,_] = v, manager(s,u).
```

This is an example of what LogicBlox terms a *delta rule*[2], used to insert data into the edb predicate `sales`. In this body, the atom `button_clicked(f,s)` is preceded by the *insert* modifier "`+`", which indicates an insertion to the corresponding predicate. As a result, the rule will only be fired when the submit button is pushed and the corresponding fact is inserted in the `button_clicked` predicate. Similarly, the symbol "`^`" in the head is the *upsert* modifier, indicating that if the corresponding key already exists in `sales`, its value should be updated to the one produced by the rule, otherwise a new entry with this key-value pair should be inserted.

Constructors. DatalogLB allows the invention of new values during program execution through the use of *constructors* (aka *Skolem functions* [9]) in the heads of rules. DatalogLB programs using recursion through constructors are not guaranteed to terminate on all inputs. For this reason, the DatalogLB compiler implements a safety check that exploits the connection between Datalog evaluation and the chase procedure [22], and warns if termination cannot be guaranteed. (The same safety check is used for programs using recursion through arithmetic.)

Programming in the large. Enterprise applications written in DatalogLB can contain tens of thousands of predicates and rules. To support such large-scale projects, DatalogLB also supports organization of programs into *modules* and reusable *libraries*.

Second-order existential quantifiers. Combinatorial optimization problems arise in many enterprise applications. To support this class of problems, DatalogLB allows predicates to be marked as existentially quantified, subject to specified constraints, using the usual facilities of the language as a "syntax skin" for an underlying solver. For instance, the following is a fragment of a program that solves Sudoku puzzles via linear programming:

```
X[i,j,z,t,k]=v -> index(v), index(i), index(j), index(z),
   index(t), number(k), v >= 0, v <= 1.
index(i), index(j), index(z), index(t), number(k) -> X[i,j,z,t,k]= _.
lang:solver:variable(‘X).
Obj[] = v -> float[64](v).
lang:solver:minimal(‘Obj).
Obj[] += X[x,y,z,t,k] * f[x,y,k].
```

In the example, `X` is the existentially-quantified predicate predicate, and `Obj` is the objective function for the solver. (The last rule uses an aggregate syntax, `+=`, inspired by the Dyna project [8].)

Provenance. DatalogLB includes an option to record *provenance information* [10] during program evaluation and query [14] it afterwards, to facilitate debugging.

[2] Not to be confused with the delta rules transformation used in semi-naive evaluation.

BloxAnalysis. The BloxAnalysis feature of the platform allows one to import DatalogLB programs as data in appropriate predicates in a LogicBlox workspace. As a result, one can use DatalogLB programs to perform static analysis of DatalogLB programs, as well as to rewrite them for optimization purposes.

4 Academic Collaborations

We will close the tutorial by highlighting successful collaborations with academics using LogicBlox as a motivating setting and research vehicle. These collaborations have resulted in publications in diverse areas including declarative networking [18], program analysis [4,3], distributed query evaluation [37], software engineering and testing [31,20,19], data modeling [11] constraint handling rules [30,29,6], magic sets transformations [32], and debugging Datalog programs [15].

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Abiteboul, S., Vianu, V.: Datalog extensions for database queries and updates. J. Comput. Syst. Sci. 43(1) (1991)
3. Bravenboer, M., Smaragdakis, Y.: Exception analysis and points-to analysis: better together. In: ISSTA (2009)
4. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: OOPSLA (2009)
5. http://www.sap.com/solutions/analytics/business-intelligence
6. Campagna, D., Sarna-Starosta, B., Schrijvers, T.: Approximating constraint propagation in datalog. In: CICLOPS (2011)
7. http://www.ibm.com/software/analytics/cognos
8. Eisner, J., Filardo, N.: Dyna: Extending Datalog for Modern AI. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 181–220. Springer, Heidelberg (2011)
9. Enderton, H.B.: A Mathematical Introduction to Logic, 1st edn. Academic Press (1972)
10. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS (2007)
11. Halpin, T.A.: Structural aspects of data modeling languages. In: BMMDS/EMMSAD 2011 (2011)
12. Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M.M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., Peterson, J.: Report on the programming language Haskell: a non-strict, purely functional language version 1.2. SIGPLAN Notices 27(5), 1–164 (1992)
13. http://www.oracle.com/hyperion
14. Karvounarakis, G., Ives, Z.G., Tannen, V.: Querying Data Provenance. In: SIGMOD (2010)

15. Köhler, S., Ludäscher, B., Smaragdakis, Y.: Declarative Datalog Debugging for Mere Mortals. In: Barcel, P., Pichler, R. (eds.) Datalog 2.0. LNCS, vol. 7494, pp. 111–122. Springer, Heidelberg (2012)
16. Ludäscher, B.: Integration of Active and Deductive Database Rules. DISDBIS, vol. 45. Infix Verlag, St. Augustin (1998)
17. Maier, D., Warren, D.S.: Computing With Logic: Logic Programming With Prolog. Addison-Wesley (1988)
18. Marczak, W.R., Huang, S.S., Bravenboer, M., Sherr, M., Loo, B.T., Aref, M.: Secureblox: customizable secure distributed data processing. In: SIGMOD (2010)
19. McGill, M.J., Dillon, L.K., Stirewalt, R.E.K.: Scalable analysis of conceptual data models. In: ISSTA (2011)
20. McGill, M.J., Stirewalt, R.E.K., Dillon, L.K.: Automated Test Input Generation for Software That Consumes ORM Models. In: Meersman, R., Herrero, P., Dillon, T. (eds.) OTM 2009 Workshops. LNCS, vol. 5872, pp. 704–713. Springer, Heidelberg (2009)
21. `http://www.xmla.org`
22. Meier, M., Schmidt, M., Lausen, G.: On chase termination beyond stratification. PVLDB 2(1), 970–981 (2009)
23. `http://www.microstrategy.com`
24. Mitchell, R.L.: Y2K: The good, the bad and the crazy. ComputerWorld (December 2009)
25. `http://www.ibm.com/software/data/netezza`
26. `http://www.sap.com/platform/netweaver/index.epx`
27. `https://www.451research.com/report-short?entityId=66963`
28. `http://nosql-database.org/`
29. Sarna-Starosta, B., Schrijvers, T.: Transformation-based indexing techniques for constraint handling rules. In: CHR (2008)
30. Sarna-Starosta, B., Zook, D., Pasalic, E., Aref, M.: Relating Constraint Handling Rules to datalog. In: CHR (2008)
31. Stirewalt, R.E.K., Rugaber, S., Hsu, H.-Y., Zook, D.: Experience report: Using tools and domain expertise to remediate architectural violations in the logicblox software base. In: ICSE (2009)
32. Tekle, K.T., Liu, Y.A.: More efficient datalog queries: subsumptive tabling beats magic sets. In: SIGMOD (2011)
33. `http://www.teradata.com`
34. `http://www.ibm.com/software/analytics/cognos/products/tm1`
35. `http://www.oracle.com/technetwork/middleware/weblogic`
36. `http://www.ibm.com/software/webservers/appserv/was`
37. Zinn, D., Green, T.J., Ludäscher, B.: Win-move is coordination-free (sometimes). In: ICDT (2012)

Datalog: A Perspective and the Potential

Yuri Gurevich

Microsoft Research, Redmond, Washington, USA

What we see depends mainly on what we look for.
—John Lubbock

Abstract. Our main goal is to put Datalog into a proper logic perspective. It may be too early to put Datalog into a proper perspective from the point of view of applications; nevertheless we discuss why Datalog pops up so often in applications.

1 Introduction

Throughout our career, we came across Datalog many times, directly or indirectly. This exposition is, in a sense, a summary of our experience. Here we are interested in proper perspectives on Datalog from the point of view of logic and applications. The exposition contains no new hard technical results.

A short §2 on preliminaries seeks to fix the terminology and make this exposition more self-contained. In §3, we recall the notion of global relations; first-order and second-order formulas are, semantically, global relations. According to the standard model-theoretic semantics of Datalog, known also as the fixed-point semantics, Datalog queries are global relations as well. We also recall the proof-theoretic semantics of Datalog queries, and we formulate how the two semantics of Datalog are related.

The goal of the following §4–7 is to put Datalog into a proper logic perspective. The global-relation view allows us to compare the expressive power of Datalog with that of more traditional logics. Whether we speak about all structures or only finite structures, Datalog has only trivial intersection with first-order logic (§4) and constitutes only a tiny fragment of second-order logic (§5). There is, however, a more traditional logic whose expressive power is exactly that of Datalog; it is existential fixed-point logic (§7). The equiexpressivity result is rather robust.

In applications, there is a growing interest in rule-based systems, and Datalog emerges as a convenient and popular basis for such systems. One instructive example is "Dedalus: Datalog in Time and Space" [3]. In §8, we illustrate the use and limitations of Datalog for policy/trust management, and then we describe an extension of Datalog, called primal infon logic, that overcomes indicated limitations while preserving the feasibility of Datalog. To this end, Datalog is viewed as a logic calculus without axioms. Primal infon logic extends that calculus with axioms and more inference rules.

Is Datalog just a fleeting fashion or is there something objective in its coming up again and again in different applications? Following a recent article [10], our final section §9 gives an argument for the latter. It turns out that standard logic systems (and even many non-logic systems) reduce to Datalog. While many of these reductions are infeasible, some of them are rather practical and allow one to exploit well-optimized Datalog tools.

2 Preliminaries

By default, first-order formulas are without equality or function symbols of positive arity, and a term is a variable or constant.

A (pure) Datalog program is built from atomic formulas of first-order logic. The relation symbols of the program split into *extensional* and *intensional*; accordingly the atomic formulas are extensional or intensional. The program itself is a finite set of facts and rules. Facts are extensional atomic formulas, and rules have the form

$$\beta_0 \; :- \; \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_\ell \tag{1}$$

where all formulas α_i are atomic and extensional, formulas β_j are atomic and intensional, and k, ℓ may be zero. The *implication form* of (1) is a formula

$$(\alpha_1 \wedge \cdots \wedge \alpha_k \wedge \beta_1 \wedge \cdots \wedge \beta_\ell) \to \beta_0, \tag{2}$$

and the *closed form* of (1) is a sentence

$$\forall \bar{x}\big((\alpha_1 \wedge \cdots \wedge \alpha_k \wedge \beta_1 \wedge \cdots \wedge \beta_\ell) \to \beta_0\big) \tag{3}$$

where $\bar{x}$ comprises the individual variables of (1). Similarly, the *closed form* of a fact $\alpha(\bar{x})$ is the sentence $\forall \bar{x}\, \alpha(\bar{x})$.

A Datalog query Q is a pair (Π, γ) where Π is a Datalog program and γ an intensional atomic formula. The extensional vocabulary of Π (resp. Q) comprises the constants and extensional relation symbols in Π (resp. Q). The number of distinct variables in γ is the arity of Q. Nullary queries are also known as ground queries.

Example 1. Let Π_1 be the Datalog program

$$\begin{aligned} &E(x,1)\\ &E(2,3)\\ &T(3,x) :-\\ T(x,y) \;&:-\; E(x,y)\\ T(x,y) \;&:-\; T(x,z), T(z,y) \end{aligned}$$

with two facts and three rules. Relation E is extensional, and relation T is intensional. The extensional vocabulary of query $(\Pi_1, T(1,4))$ consists of the constants $1, 2, 3, 4$ and the relation symbol E. □

Let Π be a Datalog program, Υ the extensional vocabulary of Π and Υ' the result of adding to Υ the intensional symbols of Π. The program Π gives rise to an operator O that, given any Υ'-structure B, does the following. For every rule (1) of Π and every instantiation

$$\xi\beta_0 \ :-\ \xi\alpha_1, \ldots, \xi\alpha_k, \xi\beta_1, \ldots, \xi\beta_\ell$$

of the variables of the rule with elements of B such that the ground atomic formulas $\xi\alpha_1, \ldots, \xi\alpha_k, \xi\beta_1, \ldots, \xi\beta_\ell$ hold in B, the operator O sets $\xi\beta_0$ true (unless it was true already in B). The result is an Υ'-structure $O(B)$ which is like B except that the intensional relations might have grown.

Given any Υ-structure A modeling (the closed form of) the facts of Π, let A_0 be the Υ'-expansion of A where all the intensional relations are empty, $A_{n+1} = O(A_n)$, and A^* be the limit of structures A_n so that, for every intensional relation symbol R, the interpretation of R in A^* is the union of its interpretations in structures A_n. The intensional relations of A^* are the smallest intensional relations closed under the rules of Π over A. In other words, they are the smallest intensional relations that make the closed forms of the rules true.

Definition 2. A query $(\Pi, \gamma(\bar{x}))$ is *bounded* if there exists a number n such that, for every structure A, we have

$$\{\bar{a} : \ A_n \models \gamma(\bar{a})\} = \{\bar{a} : \ A^* \models \gamma(\bar{a})\}. \tag{4}$$

The query is bounded on a class $\mathcal{C}$ of structures if there exists n such that (4) holds on all structures in $\mathcal{C}$. □

3 Proof-Theoretic and Model-Theoretic Semantics of Datalog Programs and Queries

There are two different semantics of Datalog queries in the literature. It may be useful to clarify what they are and how they are related.

3.1 Proof-Theoretic Semantics

View a given Datalog program Π as a deductive system. The axioms of Π are its facts. Each rule (1) of Π gives rise to an inference rule

$$\frac{\xi\alpha_1, \ \ldots, \ \xi\alpha_k, \ \xi\beta_1, \ \ldots, \ \xi\beta_\ell}{\xi\beta_0} \tag{5}$$

where ξ is an arbitrary substitution (of variables with terms). And there is one additional inference rule, the substitution rule

$$\frac{\varphi}{\xi\varphi} \tag{6}$$

where ξ is again an arbitrary substitution.

A nullary query (Π, γ) is an assertion that γ is Π-deducible. It is easy to see that, in Example 1, $T(1,4)$ is not Π_1-deducible.

Lemma 3. *Let $(\Pi, \gamma(\bar{x}))$ be a Datalog query, H a set of atomic formulas, and $\gamma(\bar{c})$ the result of replacing the variables $\bar{x}$ with fresh constants $\bar{c}$. Then*

$$H \vdash_\Pi \gamma(\bar{x}) \Longleftrightarrow H \vdash_\Pi \gamma(\bar{c}).$$

Proof. $\Longrightarrow$. Use the substitution rule.
$\Longleftarrow$. Given a derivation of $\gamma(\bar{c})$, replace the constants $\bar{c}$ with fresh variables $\bar{y}$ and then use the substitution rule. □

3.2 Model-Theoretical Semantics

Definition 4 ([14]). An r-ary *(abstract) global relation* $\mathcal{R}$ of vocabulary Υ associates with any given Υ-structure A an r-ary relation $\mathcal{R}A$ on (the base set of) A in such a way that

$$\mathcal{R}\eta A = \eta\mathcal{R}A$$

for every isomorphism η from A to another Υ-structure. □

Any first-order or second-order formula (without free 2nd order variables) φ is, semantically, a global relation $A \models \varphi$. Model-theoretically, a Datalog query $Q = (\Pi, \gamma(\bar{x}))$ is also a global relation $\mathcal{R}$; the vocabulary of $\mathcal{R}$ is the extensional vocabulary of Q, and the arity of $\mathcal{R}$ is the number of distinct individual variables in γ. A relation $\mathcal{R}A(\bar{x})$ asserts that $A \models \gamma(\bar{x})$ if A models the (closed form of the) facts of Π and if the intensional relations over A are as in the definition of A^* in §2.

We illustrate that global relation $\mathcal{R}$ on the example of a binary query $(\Pi_1, T(x,y))$ where Π_1 is the the program of Example 1. Let G be an arbitrary directed graph (V, E) with distinguished (and not necessarily distinct) elements $1, 2, 3$. If $E(2,3)$ or $\forall x\, E(x,1)$ fails in G, then relation $\mathcal{R}G(x,y)$ is universally true. If $E(2,3)$ and $\forall x\, E(x,1)$ hold in G, consider the closed forms

$$\begin{aligned}
\rho_1 &= \forall x\, T(3,x)\\
\rho_2 &= \forall x,y\, (E(x,y) \to T(x,y))\\
\rho_3 &= \forall x,y,z\, ((T(x,z) \wedge T(z,y)) \to T(x,y))
\end{aligned}$$

of the rules of Π_1. Interpret T as the least relation on V such that the expansion $G^* = (V, E, T)$ satisfies ρ_1, ρ_2 and ρ_3. The relation $\mathcal{R}G(x,y)$ is $G^* \models T(x,y)$.

3.3 Relating the Two Semantics

Theorem 5. *Let $\mathcal{R}$ be the global relation of a Datalog query $(\Pi, \gamma(\bar{x}))$, F the collection of the facts of Π and H a set of atomic formulas with relation symbols different from the intensional symbols of Π. The following claims are equivalent.*

1. $H \vdash_{\Pi} \gamma(\bar{x})$.
2. $\mathcal{R}A(\bar{x})$ *is universally true in every structure* A *satisfying* $F \cup H$.

Proof. Without loss of generality, we may assume that $\gamma(\bar{x})$ is ground. Indeed, instantiate the variables $\bar{x}$ with fresh constants $\bar{c}$. Now use Lemma 3 and the obvious fact that $\mathcal{R}A(\bar{x})$ is universally true in a structure A if and only if $\mathcal{R}A(\bar{c})$ holds in every expansion of A with constants $\bar{c}$ (and the same base set).

$1 \to 2$ is obvious.
$2 \to 1$. We suppose that claim 1 fails and prove that claim 2 fails as well. Let Υ be the extensional vocabulary of the query extended with that of H. Without loss of generality, Υ contains at least one constant. Consider the Υ-structure A on the Υ-constants where an extensional ground atomic formula α holds if and only if it is an instantiation of a hypothesis or fact. It suffices to prove that $\mathcal{R}A$ is false. Obviously the facts and hypotheses are universally true in A.

Let A' be the expansion of A with the intensional relations of Π where an intensional ground atomic formula β holds if and only if it is Π-deducible from H. This instantiates intensional variables to the least values satisfying the closed forms of the rules of Π. Taking into account that γ is not Π-deducible from H, it follows that $\mathcal{R}A$ fails. $\square$

One case of interest is $H = \emptyset$. Another one is where H is the positive diagram $\Delta^+(A)$ of a structure A such that A models F and every element of A is distinguished (a constant). Here $\Delta^+(A)$ is the set of all ground atomic formulas in the vocabulary of A that are true in A.

4 Datalog and First-Order Logic

There is a bit of confusion in the literature about the relation of Datalog and first-order logic. "Query evaluation with Datalog is based on first order logic" [23]. "Datalog is declarative and is a subset of first-order logic" [19]. In fact, Datalog is quite different from first-order logic. Datalog is all about recursion, and first-order logic does not have any recursion (though recursion is available in some first-order theories, e.g. arithmetic). We say that a Datalog query and a first-order formula are equivalent if their global relations coincide. More generally, the query and formula are equivalent on a class $\mathcal{C}$ of structures if their global relations coincide on $\mathcal{C}$.

Theorem 6. *If a Datalog query is equivalent to a first-order formula then the query is bounded and equivalent to a positive existential first-order formula.*

Theorem 6 is a straightforward consequence of the compactness theorem [2, Theorem 5]. Unfortunately the compactness argument involves infinite structures of little relevance to Datalog applications. The finite version of Theorem 6 was more challenging.

Theorem 7 (Ajtai-Gurevich). *If a Datalog query is equivalent to a first-order formula on finite structures then, on finite structures, the query is bounded and equivalent to a positive existential first-order formula.*

Theorem 7 is fragile [2, §10] as far as extensions of Datalog are concerned. It was generalized in [4] to some classes of finite structures. Later Benjamin Rossman proved the powerful Homomorphism Preservation Theorem [21] that implies Theorem 7.

5 Datalog and Second-Order Logic

Theorem 8. *Every Datalog query* (Π, γ) *is equivalent to a second-order formula* Φ *of the form* $\forall \bar{X} \exists \bar{y} \varphi$ *where* $\bar{X}$ *is a sequence of relation variables,* $\bar{y}$ *is a sequence of individual variables and* φ *is quantifier-free.*

Again, the equivalence of a query and formula means that their global relations coincide. The form $\forall \bar{X} \exists \bar{y} \varphi$ is known as the strict $\forall^1_1$ form where "strict" refers to the fact that the first-order part is existential. Strict $\forall^1_1$ formulas were studied by logicians long before Datalog was introduced [20].

For illustration consider Datalog query $(\Pi_1, T(a,b))$ where Π_1 is the program of Example 1 and a, b are fresh constants. Let F be the formula

$$(\forall x E(1,x)) \wedge E(2,3)$$

reflecting the facts of Π_1 and let ρ_1, ρ_2 and ρ_3 be the closed forms of the rules of Π_1, as in §3. Then the desired Φ is (the strict $\forall^1_1$ formula equivalent to) the formula

$$\neg F \vee \forall T \big((\rho_1 \wedge \rho_2 \wedge \rho_3) \rightarrow T(a,b) \big)$$

The converse of Theorem 8 is not true: non-3-colorability is expressible by a strict $\forall^1_1$ formula while it cannot be expressed by any Datalog query unless P = NP [6]. The converse can be obtained by severely restricting the form of the quantifier-free formula φ.

6 Liberal Datalog

The version of Datalog considered above is known as pure Datalog. It has been generalized in numerous ways. In particular, Constraint Datalog is popular; see for example [19] and references there. One very different generalization started with article [14] where we defined and studied inflationary fixed points. Abiteboul and Vianu used inflationary-fixed-point semantics to define an elegant version of Datalog with negations [1] that was popularized by Ullman [22] and used e.g. in [3]. One simple and most natural liberalization of pure Datalog is this:

(a) Make extensional atomic formulas negatable, so that the facts of a program are extensional atomic formulas or their negations, and rules have the form (1) where $\alpha_1, \ldots, \alpha_k$ are extensional atomic formulas or their negations and formulas β_j are atomic and intensional.

While the positivity of intensional formulas is essential for the least-fixed-point construction, the requirement that extensional formulas be positive has not been essential above. The whole §3 remains valid under liberalization (a). Furthermore Theorem 5 and its proof remain valid if "H is a set of atomic formulas" is replaced with "H is a set of atomic formulas or their negations." A new special case of interest is where H is the diagram $\Delta(A)$ of a structure A on constants that models F. Here $\Delta(A)$ is the set of all ground atomic formulas and their negations in the vocabulary of A that are true in A.

A further liberalization of Datalog was introduced in [16], by the name Liberal Datalog, and was studied in [8]. In addition to (a), there are two additional liberalizations in Liberal Datalog.

(b) The extensional vocabulary of a program may contain function symbols of any arity. Intensional formulas may contain extensional function symbols.
(c) Equality has its usual meaning and may occur in programs as an extensional relation symbol.

Model-theoretically, liberal Datalog queries are global relations. Theorem 8 remains valid for Liberal Datalog queries [6, Theorem 5].

7 Datalog, Liberal Datalog, and Existential Fixed-Point Logic

We have seen that Datalog has a trivial intersection with first-order logic and constitutes only a sliver of second-order logic. Existential fixed-point logic (EFPL) was introduced as the right logic to formulate preconditions and postconditions of Hoare logic [6]. The same authors continued to investigate EFPL in [7,8,9].

Theorem 9 ([8]). *Every global relation expressible by an EFPL formula is expressible by a Liberal Datalog query, and the other way round.*

> `Q:` If you want a logic with the expressivity of Liberal Datalog, why not declare Liberal Datalog a logic in its own right?
>
> `A:` In traditional logics, like first-order logic or second-order logic, logic operators are explicit and can be nested. EFPL is traditional from that point of view. While in Datalog, pure or liberal, the fixed-point operation is implicit and can't be nested, in EFPL it is explicit and can be nested.

For the purpose of the following theorem, equality is considered part of logic and therefore is not counted in the definition of the vocabulary of an EFPL formula.

Theorem 10 (Blass-Gurevich). *Every global relation expressible by an EFPL formula without negations or function symbols of positive arity is expressible by a pure Datalog query, and the other way round.*

We explain how to prove Theorem 10 given the proof of Theorem 9. In the proof of Theorem 9, given an EFPL formula φ, we construct a Datalog query Q with the same global relation, and the other way round. The vocabulary of φ coincides with the extensional vocabulary of Q; if one of them has no function symbols of positive arity, neither does the other. If the given φ has no negation then the constructed Q has no negation, and the other way round. However, the construction of φ from Q makes use of equality. Equality is legal in EFPL but pure Datalog does not have it. The problem arises how to deal with the equality of φ in the construction of Q from φ. Pure Datalog does not have equality as an extensional relation. But, since we need only positive occurrences of equality, we can compute the equality as an intensional relation:

$$E(x, x) \ :-$$

The intensional relation E represents the equality of φ in the construction of Q.

8 Datalog, Policies and Primal Logic

The advent of cloud computing forces us to be more careful with policies and trust. Numerous policies, that might have been implicit and vague in the world of brick and mortar, need be explicit, precise and automated in the cloud. Many policy rules are expressible in Datalog.

```
X can read File 13  :-  Alice owns File 13,
                        Alice and X are friends.
```

But there are common policy rules that are not expressible in Datalog, primarily because they involve quotations and trust implications.

```
X can read File 13 :-   Alice owns File 13,
                        Alice said Friends(A,X),
                        Alice is trusted on Friends(A,X).
```

where `Friends(A,X)` is a more formal version of `Alice and X are friends` and `Alice is trusted on Friends(A,X)` abbreviates the implication

$$(\texttt{Alice said Friends}(\texttt{A}, \texttt{X})) \to \texttt{Friends}(\texttt{A}, \texttt{X}).$$

Primal infon logic, introduced in [17], is a proof-theoretic extension of Datalog that allows one to use quotations and nested implications. (Infons are statements treated as pieces of information.)

In the rest of this section, formulas are by default quantifier-free first-order formulas without equality or function symbols of positive arity. Datalog rules (1) will be viewed as implications (2) so that a Datalog program is a finite set of formulas.

8.1 Proof-Theoretic Semantics of Datalog

As we have seen in §3, Datalog programs can be viewed as logic calculi, but there is a broader proof-theoretic view of Datalog according to which Datalog itself is a logic calculus. The logic calculus DL of pure Datalog has no axioms and just three inference rules. One of them is the substitution rule (6). The other two are the the following conjunction introduction rule and implication elimination rule:

$$\frac{\varphi \quad \psi}{\varphi \land \psi} \qquad\qquad \frac{\varphi \quad \varphi \to \psi}{\psi}.$$

We write $H \vdash_{\mathrm{DL}} \varphi$ to indicate that hypotheses H entail formula φ in DL.

Theorem 11. *For any Datalog query* (Π, γ)*, we have* $\vdash_{\Pi} \gamma \Longleftrightarrow \Pi \vdash_{\mathrm{DL}} \gamma$.

Proof. $\Longrightarrow$. To simulate a rule of Π, use conjunction introduction to derive the body of the rule, and then use implication elimination to derive the head.
$\Longleftarrow$. Check by induction on the derivation length that Π entails only Π-derivable atomic formulas and their conjunctions.

The entailment problem for the ground fragment of DL, obtained from DL by removing the substitution rule, is solvable in linear time [13].

8.2 Primal Infon Logic

The quote-free fragment of primal infon logic is obtained from DL by adding an axiom $\top$ and the following conjunction elimination rules and implication introduction rule:

$$\frac{\varphi \land \psi}{\varphi} \qquad \frac{\varphi \land \psi}{\psi} \qquad\qquad \frac{\varphi}{\psi \to \varphi}.$$

To form quotations, primal infon logic has countably infinite lists of variables and constants of type Principal. The logic uses unary connectives p `said`[1] where p is a term of type Principal. A quotation prefix is a string of the form

$$p_1 \ \texttt{said}\ p_2\ \texttt{said}\ \ldots\ p_k\ \texttt{said}.$$

Primal infon logic is given by the following logic calculus where `pref` ranges over quotation prefixes.

1 Originally primal infon logic had two kinds of quotations p `said` φ and p `implied` φ but later the second kind was removed.

Axioms $\texttt{pref}\ \top$

Inference rules

$$\frac{\texttt{pref}\,(x \wedge y)}{\texttt{pref}\,x} \qquad \frac{\texttt{pref}\,(x \wedge y)}{\texttt{pref}\,y}$$

$$\frac{\texttt{pref}\,x \qquad \texttt{pref}\,y}{\texttt{pref}\,(x \wedge y)}$$

$$\frac{\texttt{pref}\,x \qquad \texttt{pref}\,(x \to y)}{\texttt{pref}\,y}$$

$$\frac{\texttt{pref}\,y}{\texttt{pref}\,(x \to y)}$$

The entailment problem for the ground fragment of primal logic, obtained from primal logic by removing the substitution rule, is solvable in linear time [15,12]. This is important because, while policy rules typically have variables, deduction often deals with fully instantiated cases. An article [12] is being written to become a standard initial reference for primal infon logic.

9 It All Reduces to Datalog

In a way, pure Datalog reflects the essence of deductive systems. By default, this section follows [10].

Definition 12. A *Hilbertian system* (or an *abstract Hilbertian deductive system*) is given by a set F of so-called *formulas*, a subset $\mathrm{Ax} \subseteq F$ of so-called *axioms*, and a set Ru of so-called *rules of inference* $\langle P, \alpha \rangle$ where P is a finite subset of F and $\gamma \in F$.

`Q:` Why these "so-called"?

`A:` There are Hilbertian systems that do not look at all like logics. For example, let F be the set of edges of a fixed digraph, $\mathrm{Ax} = \emptyset$, and Ru comprise the pairs $\langle \{(a,b),(b,c)\},(a,c)\rangle$. Yet we'll call the elements of F, Ax and Ru formulas, axioms and rules respectively.

Theorem 13. *For every Hilbertian system S there is a (possibly infinite) Datalog program Π such that S-formulas are propositional symbols of Π and*

$$H \vdash_S \gamma \Longleftrightarrow H \vdash_\Pi \gamma.$$

Definition 14. A Hilbertian system is *substitutional* if:

1. Formulas are certain finite strings in a specified alphabet.
2. The alphabet includes a countably infinite set of so-called *variables*, and some (possibly none) of the non-variable symbols are so-called *constants*. The variables and constants are *terms*.

3. If α is a formula then the result $\xi\alpha$ of replacing in α distinct variables x by terms $\xi(x)$ respectively is also a formula, and $\langle\{\alpha\}, \xi\alpha\rangle$ is a rule of inference. Such rules are *substitution rules*.
4. If $\langle\{\alpha_1, \ldots, \alpha_n\}, \beta\rangle$ is a rule of inference that is not a substitution rule, then $\langle\{\xi\alpha_1, \ldots, \xi\alpha_n\}, \xi\beta\rangle$ is also a rule of inference, for any substitution ξ. □

The requirement 4 is often superfluous. For example, the inference rules of primal infon logic are closed under substitutions. Here is a simple example when the requirement is essential. Consider a deductive system with axiom $P(1)$, the substitution rule and a rule $\dfrac{P(x)}{Q(x)}$. One may expect to derive $Q(1)$, but it is not derivable in the system.

Theorem 15. *For every substitutional Hilbertian system S there is a (possibly infinite) Datalog program Π such that Π treats any S-formula α with k distinct variables as a relation symbol of arity k, and*

$$H \vdash_S \gamma \Longleftrightarrow H \vdash_\Pi \gamma.$$

Theorem 16. *There is an algorithm that converts any instance of the derivability problem for primal infon logic into an instance of the derivability problem for pure Datalog, with the same answer.*

A more practical algorithm for the same purpose is constructed in [5].

Acknowledgment. Many thanks to Andreas Blass, Carlos Cotrini and Phokion Kolaitis for useful discussions.

References

1. Abiteboul, S., Vianu, V.: Datalog Extensions for Database Queries and Updates. J. of Computer and System Sciences 43, 62–124 (1991)
2. Ajtai, M., Gurevich, Y.: Datalog vs First-Order Logic. J. of Computer and System Sciences 49(3), 562–588 (1994)
3. Alvaro, P., Marczak, W., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.C.: Dedalus: Datalog in Time and Space. EECS Dept, University of California, Berkeley Technical Report No. UCB/EECS-2009-173, December 16 (2009)
4. Atserias, A., Dawar, A., Kolaitis, P.: On Preservation under Homomorphisms and Unions of Conjunctive Queries. JACM 53(2), 208–237 (2006)
5. Bjørner, N., de Caso, G., Gurevich, Y.: From Primal Infon Logic with Individual Variables to Datalog. In: Erdem, E., Lee, J., Lierler, Y., Pearce, D. (eds.) Correct Reasoning. LNCS, vol. 7265, pp. 72–86. Springer, Heidelberg (2012)
6. Blass, A., Gurevich, Y.: Existential Fixed-Point Logic. In: Börger, E. (ed.) Computation Theory and Logic. LNCS, vol. 270, pp. 20–36. Springer, Heidelberg (1987)
7. Blass, A., Gurevich, Y.: The Underlying Logic of Hoare Logic. Bull. EATCS 70, 82–110 (2000); also in Current Trends in Theoretical Computer Science, pp. 409–436. World Scientific (2001)

8. Blass, A., Gurevich, Y.: Two Forms of One Useful Logic: Existential Fixed Point Logic and Liberal Datalog. Bull. EATCS 95, 164–182 (2008)
9. Blass, A., Gurevich, Y.: One Useful Logic That Defines Its Own Truth. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 1–15. Springer, Heidelberg (2008)
10. Blass, A., Gurevich, Y.: Hilbertian Deductive Systems, Infon Logic, and Datalog. Microsoft Research Technical Report MSR-TR-2011-81 (June 2011); to appear in Postproceeding from FCT 2011 in Information and Computation
11. Cook, S.A.: Soundness and Completeness of an Axiom System for Program Verification. SIAM J. Computing 7, 70–90 (1978)
12. Cotrini, C., Gurevich, Y.: Revisiting Primal Infon Logic (in preparation)
13. Dantzin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Sureveys 33(3), 374–425 (2001)
14. Gurevich, Y.: Toward Logic Tailored for Computational Complexity. Springer Lecture Notes in Math, vol. 1104, pp. 175–216 (1984)
15. Gurevich, Y.: Two Notes on Propositional Primal Logic. Microsoft Research Tech. Report MSR-TR-2011-70 (May 2011)
16. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. In: 21st IEEE Computer Security Foundations Symposium (CSF 2008), pp. 149–162 (2008)
17. Gurevich, Y., Neeman, I.: DKAL 2 — A Simplified and Improved Authorization Language, Microsoft Research Tech. Report MSR-TR-2009-11
18. Gurevich, Y., Shelah, S.: Fixed-Point Extensions of First-Order Logic. Annals of Pure and Applied Logic 32, 265–280 (1986)
19. Li, N., Mitchell, J.C.: DATALOG with Constraints: A Foundation for Trust Management Languages. In: Dahl, V., Wadler, P. (eds.) PADL 2003. LNCS, vol. 2562, pp. 58–73. Springer, Heidelberg (2002)
20. Maltsev, A.I.: Model Correspondences. Izv. Akad. Nauk SSSR. Ser. Mat. 23, 313–336 (1959) (Russian)
21. Rossman, B.: Homomorphism Preservation Theorems. JACM 55(3), Article No. 15 (2008)
22. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. The New Technologies, vol. II. W. H. Freeman & Co., New York (1990)
23. Wikipedia, Datalog, `http://en.wikipedia.org/wiki/Datalog` (viewed on June 14, 2012)

Existential Rules: A Graph-Based View
(Extended Abstract)

Marie-Laure Mugnier

University of Montpellier, France

1 Introduction

We consider rules that allow to assert the existence of new individuals, an ability called *value invention* in databases [AHV95]. These rules are of the form *body* $\rightarrow$ *head*, where the body and the head are function-free conjunctions of atoms, and variables that occur only in the head are *existentially* quantified, hence their name $\forall\exists$-rules in [BLMS09, BLM10] or existential rules in [BMRT11, KR11]. Existential rules have long been studied in databases as high-level constraints called *tuple generating dependencies* (TGDs) [BV84]. Recently, there has been renewed interest for these rules in the context of ontology-based data access (OBDA), a new paradigm that seeks to exploit the semantics encoded in ontologies while querying data. The deductive database language Datalog could be seen as a natural candidate for expressing ontological knowledge in this context, however its limitation is that it does not allow for value invention, since all variables in a rule head necessarily occur in the rule body. Value invention has been recognized as a necessary prerequisite in an open-world perspective, where all individuals are not known *a priori*. It is in particular a feature of description logics (DLs), well-known languages dedicated to ontological representation and reasoning. This prerequisite motivated the recent extension of Datalog to existential rules, which gave rise to the *Datalog +/-* formalism [CGK08, CGL09].

Existential rules have indeed some particularly interesting features in the context of OBDA. On the one hand, they generalize lightweight DLs dedicated to query answering (DL-Lite [CGL$^+$07] and $\mathcal{EL}$ [BBL05] families, and more generally Horn DLs) while being more powerful and flexible [CGL09, BLM10, BMRT11]. In particular, they have unrestricted predicate arity (while DLs consider unary and binary predicates only). This allows for a natural coupling with database schemas, in which relations may have any arity; moreover, adding pieces of information, for instance to take contextual knowledge into account, is made easier, since these pieces can be added as new predicate arguments. On the other hand, existential rules cover plain Datalog, while allowing for incompleteness in the data.

Historically, we studied existential rules as part of another research line that seeks to develop a knowledge representation and reasoning formalism based on (hyper)graphs. This formalism is *graph-based* not only in the sense that all objects are defined as graphs, while being equipped with a logical semantics, but also in the sense that reasoning relies on graph mechanisms, which are sound and complete with respect to the logical semantics. This framework, presented thoroughly in [CM09], is rooted in conceptual graphs [Sow84]. The logical translation of the graph rules yield exactly existential rules (and other Datalog+/- constructs, like constraints, have their equivalent in this framework).

In this talk, we present a graph view of the existential rule framework and some related results. Generally speaking, seeing formulas as graphs or hypergraphs allows to focus on their structure: paths, cycles or decompositions are then fundamental notions. Two examples of results exploiting the graph structure will be detailed: the decidable class of *(greedy) bounded-treewidth sets* of rules, which is based on the tree decomposition of a graph, and a backward chaining mechanism based on subgraphs called *pieces*.

2 The Logical Framework

An *existential rule* is a first-order formula $R = \forall \mathbf{x} \forall \mathbf{y}(B[\mathbf{x}, \mathbf{y}] \rightarrow (\exists \mathbf{z} H[\mathbf{y}, \mathbf{z}]))$ where B and H are conjunctions of atoms (without function symbol except constants). A *fact* is the existential closure of a conjunction of atoms. Note that we extend the classical notion of a fact as a ground atom in order to take existential variables produced by rules into account. Moreover, this allows to cover naturally languages such as RDF/S, in which a blank node is logically translated into an existentially quantified variable, or basic conceptual graphs. In this talk, a knowledge base is composed of a set of facts, seen as a single fact, and of existential rules (other components could be added, see e.g. [CGL09] [BMRT11]). Query answering consists of computing the set of answers to a query in the knowledge base. We consider conjunctive queries (CQs), which are the standard basic queries. Boolean CQs have the same form as facts. The fundamental decision problem associated with query answering can be expressed in several equivalent ways, in particular as a Boolean CQ entailment problem: is a Boolean CQ logically entailed by a knowledge base ? In the following this problem is refered as the "entailment" problem.

A fundamental tool for query answering is homomorphism: given two facts/Boolean queries F and Q seen as sets of atoms, a *homomorphism* h from Q to F is a substitution of the variables in Q by terms in F such that $h(Q) \subseteq F$. It is well-known that F logically entails Q iff there is a homomorphism from Q to F. Sound and complete mechanisms for entailment with rules are obtained with classical paradigms, namely *forward chaining* (also called bottom-up approach, and chase when applied to TGDs) and *backward chaining* (also called top-down approach). Forward chaining enriches the initial fact by applying rules —with rule application being based on homomorphism— and checks if a fact can be derived to which the query maps by homomorphism. Backward chaining uses the rules to rewrite the query in different ways —with rewriting being based on unification— with the aim of producing a query that maps to the initial fact by homomorphism. Note that, due to the existential variables in the rule heads, unification cannot operate atom by atom as it is classically done for Horn clauses, a more complex operation is required.

3 The Graph-Based Framework

A set of atoms $\mathcal{A}$ can be seen as an ordered labeled hypergraph $\mathcal{H}_{\mathcal{A}}$, whose nodes and ordered hyperedges respectively encode the terms and the atoms from $\mathcal{A}$. One may also encode $\mathcal{A}$ as an undirected bipartite graph which is exactly the *incidence graph* of $\mathcal{H}_{\mathcal{A}}$ (it is a multigraph actually since there may be several edges between two nodes), where one class of nodes encodes the terms and the other the atoms (see Figure 1): for each atom $p(t_1, \ldots, t_k)$ in $\mathcal{A}$, instead of a hyperedge, there is an atom node labeled by p and this node is incident to k edges linking it to the nodes assigned to $t_1, \ldots, t_k$. Each edge is labeled by the position of the corresponding term in the atom. Therefore, all objects of the preceding logical framework can be defined as graphical objects. In particular, a fact or a query is encoded as a (hyper)graph and an existential rule can be seen as a pair of (hyper)graphs or equivalently as a bicolored (hyper)graph. Entailment between facts/queries is computed by a (hyper)graph homomorphism, which corresponds to the homomorphism notion defined on formulas; entailment using rules relies on homomorphism in the same way as in the logical framework.

Example 1. Figure 1 pictures a fact $F = siblingOf(a, b)$, a rule $R = siblingOf(X, Y) \rightarrow parentOf(Z, X) \wedge parentOf(Z, Y)$ (quantifiers are omitted) with its body in white and its head in gray, as well as the fact $F' = sibblingOf(a, b) \wedge parentOf(Z_0, a) \wedge parentOf(Z_0, b)$ obtained by applying R to F, where Z_0 is the newly created existential variable. Note that it is not necessary to label nodes representing variables, the graph structure being sufficient to encode co-occurrences of variables.

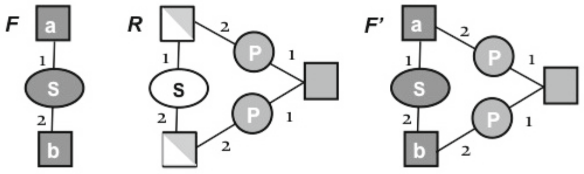


Fig. 1. Graph Representation of facts and rules

As mentioned in the introduction, this graph-based framework can be seen as a specific member of the conceptual graph fragments we have defined and developed. Conceptual graphs are defined with respect to a vocabulary, which can be seen as a very basic ontology. This vocabulary contains two finite (pre)ordered sets of concepts and of relations with any arity —and it can be further enriched by relation signatures, concept disjointness assertions, etc. The orders are interpreted as a specialization relation. Concepts and relations are logically translated into predicates and the specialization orders into formulas of the form $\forall x_1 \ldots x_k p_2(x_1 \ldots x_k) \rightarrow p_1(x_1 \ldots x_k)$ for $p_2 \leq p_1$. A basic conceptual graph is a bipartite multigraph where so-called concept nodes represent instances of concepts (i.e., terms) and relation nodes represent relations between concept instances (i.e., atoms). A concept node is labeled by a set of concepts (interpreted as a conjunction) and a marker (which can be the generic marker $\star$, referring to an unknown individual, or a constant). A relation node is labeled by a relation. Concept labels are partially ordered in a lattice obtained from the order on concepts and the order on markers ($\star$ is greater than all constants, which are pairwise incomparable). Homomorphism takes the orders on labels into account: for all concept or relation node x, one must have $label(x) \geq label(h(x))$. This allows to take the ontology into account in a very efficient way as the label comparisons can be compiled then performed in constant time. Note that the semantic web language RDFS can be encoded in the basic conceptual graph fragment. Conceptual graph rules are defined as pairs of basic conceptual graphs.

The existential rule framework can thus be seen as a conceptual graph fragment in which the vocabulary is restricted to a singleton concept set and a flat relation set. Both have the same expressivity, since the orders on concepts and relations can be encoded into the graphs —as if the rules translating these orders were applied in forward chaining.

4 Procedures for Entailement with Existential Rules

The ability to generate existential variables, associated with arbitrarily complex conjunctions of atoms, makes entailment undecidable in general. Since the birth of TGDs various conditions of decidability have been exhibited. We focus here on two abstract properties, which come with finite procedures based on forward and backward chaining respectively, and for which the graph view is particularly relevant. These properties are said to be *abstract* in the sense that they are not recognizable, i.e., deciding if a given set of rules has the property is undecidable [BLM10]. However, they provide generic algorithmic schemes that can be further customized for specific recognizable classes of rules.

A set of rules $\mathcal{R}$ is said to have the *bounded treewidth set (bts)* property if for any initial fact F, there is an integer b such that the treewidth of any fact derived from F with $\mathcal{R}$ is bounded by b (property essentially introduced in [CGK08]). The treewidth is defined with respect to a graph (the "primal graph") associated with the hypergraph encoding a fact. The decidability proof of entailment with *bts* rules does not provide a halting algorithm (at least not directly). A subclass of *bts* has been defined recently, namely *greedy bts (gbts)*, which is equipped with a forward-chaining-like halting algorithm [BMRT11, TBMR12]. For this class of rules a bounded width tree decomposition of any derived fact can be built in a greedy way. The set of all possibly derived facts can be encoded in such tree, however this tree may be infinite. An appropriate equivalence relation on the nodes of this tree allows to build only a finite part of it. The *gbts* class is very expressive, as it includes plain Datalog, (weakly) guarded rules [CGK08], *frontier-one (fr1)* rules [BLMS09], and their generalizations (weakly / jointly) frontier-guarded rules [BLM10, KR11]. The algorithm provided in [TBMR12] can be customized to run in the "good" complexity class for these subclasses, which is important since some of them have polynomial data complexity.

A set of rules is said to have the *finite unification set (fus)* property if the set of rewritten queries restricted to its most general elements is finite. This class includes for instance rules with atomic body (also called linear Datalog+/-), domain-restricted rules and sticky(-join) rules [BLMS09, CGL09, CGP10a, CGP10b]. We propose a (sound and complete) backward chaining mechanism which computes such minimal set of rewritings when the rules are *fus*. Its originality lies in the rewriting step which is based on a graph notion, that of a *piece*. Briefly, a piece is a subgraph (i.e., subset of atoms) of the query that must be erased as a whole during a rewriting step (see [SM96] for the first piece-based backward chaining on conceptual graph rules, [BLMS11] for a revised version dedicated to existential rules and [KLMT12] for an effective implementation). We point out that the unification operation can take a preorder on predicates into account, similarly to conceptual graph operations, which can save an exponential number of rewritings.

5 Conclusion

The existential framework in the context of OBDA is rather young and challenging issues need to be solved before systems effectively able to deal with large amounts data can be built. We believe that having a double view —logical and graphical— of this framework is likely be fruitful. The logical view allows to connect to Datalog, description logics and other logic-based knowledge representation and reasoning languages. The graph view allows to connect to studies in graph theory, or in other areas in which (hyper)graphs structures have been particularly well-studied from an algorithmic viewpoint, such as constraint programming. It also allows to relate directly to graph representations of data, such as RDF/S triplestores, and other emerging graph-based paradigms for data management.

References

[AHV95] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)

[BBL05] Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: IJCAI 2005, pp. 364–369 (2005)

[BLM10] Baget, J.-F., Leclère, M., Mugnier, M.-L.: Walking the decidability line for rules with existential variables. In: KR 2010, pp. 466–476. AAAI Press (2010)

[BLMS09] Baget, J.-F., Leclère, M., Mugnier, M.-L., Salvat, E.: Extending decidable cases for rules with existential variables. In: IJCAI 2009, pp. 677–682 (2009)

[BLMS11] Baget, J.-F., Leclère, M., Mugnier, M.-L., Salvat, E.: On rules with existential variables: Walking the decidability line. Artificial Intelligence 175(9-10), 1620–1654 (2011)

[BMRT11] Baget, J.-F., Mugnier, M.-L., Rudolph, S., Thomazo, M.: Walking the complexity lines for generalized guarded existential rules. In: IJCAI 2011, pp. 712–717 (2011)

[BV84] Beeri, C., Vardi, M.Y.: A proof procedure for data dependencies. Journal of the ACM 31(4), 718–741 (1984)

[CGK08] Calì, A., Gottlob, G., Kifer, M.: Taming the infinite chase: Query answering under expressive relational constraints. In: KR 2008, pp. 70–80 (2008)

[CGL$^+$07] Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. J. Autom. Reasoning 39(3), 385–429 (2007)

[CGL09] Calì, A., Gottlob, G., Lukasiewicz, T.: A general datalog-based framework for tractable query answering over ontologies. In: PODS 2009, pp. 77–86 (2009)

[CGP10a] Calì, A., Gottlob, G., Pieris, A.: Advanced processing for ontological queries. PVLDB 3(1), 554–565 (2010)

[CGP10b] Calì, A., Gottlob, G., Pieris, A.: Query answering under non-guarded rules in datalog+/-. In: Hitzler, P., Lukasiewicz, T. (eds.) RR 2010. LNCS, vol. 6333, pp. 1–17. Springer, Heidelberg (2010)

[CM09] Chein, M., Mugnier, M.-L.: Graph-based Knowledge Representation and Reasoning—Computational Foundations of Conceptual Graphs. Advanced Information and Knowledge Processing. Springer (2009)

[KLMT12] König, M., Leclère, M., Mugnier, M.-L., Thomazo, M.: A sound and complete backward chaining algorithm for existential rules. In: RR 2012 (to appear, 2012)

[KR11] Krötzsch, M., Rudolph, S.: Extending decidable existential rules by joining acyclicity and guardedness. In: IJCAI 2011, pp. 963–968 (2011)

[SM96] Salvat, E., Mugnier, M.-L.: Sound and Complete Forward and Backward Chainings of Graph Rules. In: Eklund, P., Mann, G.A., Ellis, G. (eds.) ICCS 1996. LNCS (LNAI), vol. 1115, pp. 248–262. Springer, Heidelberg (1996)

[Sow84] Sowa, J.F.: Conceptual Structures: Information Processing in Mind and Machine. Addison-Wesley (1984)

[TBMR12] Thomazo, M., Baget, J.-F., Mugnier, M.-L., Rudolph, S.: A generic querying algorithm for greedy sets of existential rules. In: KR 2012 (2012)

How (Well) Do Datalog, SPARQL and RIF Interplay?

Axel Polleres

Siemens AG Österreich, Siemensstraße 90, 1210 Vienna, Austria
axel.polleres@siemens.com

Abstract. In this tutorial we will give an overview of the W3C standard query language for RDF – SPARQL – and its relation to Datalog as well as on the interplay with another W3C standard closely related to Datalog, the Rule Interchange Format (RIF). As we will learn – while these three interplay nicely on the surface and in academic research papers – some details within the W3C specs impose challenges on seamlessly integrating Datalog rules and SPARQL.

1 SPARQL Official Semantics vs. Academia

The formal semantics of SPARQL in its original recommendation in 2008 [19] has been very much inspired by academic results, such as by the seminal papers of Pérez et al. [13,14]. Angles and Gutierrez [1] later showed that SPARQL – as defined in those papers – has exactly the expressive power of non-recursive safe Datalog with negation. Another translation from SPARQL to Datalog has been presented in [15]. In the tutorial we will present the semantics of SPARQL, starting with the semantics as per [14] as well as its translation to Datalog; we will then discuss adaptions needed to be considered with regards to the official W3C specification, particularly:

1. SPARQL's multi-set semantics and solution modifiers
2. the treatment of complex expressions and errors in FILTERs
3. the treatment of FILTER expressions in OPTIONAL

Let us sketch briefly some reasons how these features affect the translation.

SPARQL's multi-set semantics. While Datalog is set-based, SPARQL queries – just like SQL – allow for duplicates in solutions. Since duplicates can stem from only certain patterns, the translation to Datalog can be "fixed" to cater for these; however, the translation becomes less elegant [18]. Likewise, solution modifiers such as ORDER BY and LIMIT/OFFSET in SPARQL have no straightforward equivalent within Datalog. Notably, multi-set semantics has been considered in some earlier works about Datalog [12].

Complex expressions and errors in FILTERs. Unlike the semantics given in [13], FILTERS in SPARQL have a 3-valued semantics for connectives such as "&&" and "||", to cater for errors. We will give some examples where these make sense and discuss how the translation to Datalog can be adapted.

FILTERs in OPTIONALs. Another specialty of the SPARQL semantics, as noted by [1] outer joins in SPARQL – denoted by the OPTIONAL keyword – are not compositional due to the fact that certain "non-safe" FILTERs are possible, i.e., it is allowed that FILTERs refer to variables bound outside the OPTIONAL pattern. While a rewriting of SPARQL queries with safe FILTERS only is possible [1], a translation from SPARQL to Datalog could also cater for this semantics directly.

2 SPARQL in Combination with Rules

In the second part of the tutorial, we will have a closer look at using SPARQL itself as a rules language in the spirit of Datalog, from academic approaches [20,17] over practically motivated & implemented ones – such as SPIN [9] and R2R [4] – to combinations within the W3C standards themselves, namely SPARQL in combination with RIF. As for the latter, we will discuss both (i) whether or why not the translation from SPARQL to Datalog as per [15] works with RIF [5,16] and (ii) what a SPARQL query means in combination with a RIF rule set [8].

3 New Features in SPARQL1.1

Finally, we will discuss and sketch how translations to or a combinations with Datalog style rules could be extended to new features of the upcoming SPARQL 1.1 recommendation [21], namely:

Aggregate functions. Aggregate functions will allow operations on the query engine side such as counting, numerical min/max/average and so on, by operating over columns of results. This feature is commonly known from other query languages such as SQL, but also well investigated in terms of extensions of Datalog, cf. for instance [7]. A proposal to extend SPARQL with aggregates following these ideas for Datalog has been made in [17], whereas the SPARQL1.1 working group rather follows the SQL design.

Subqueries. This feature will allow nesting the results of a SPARQL query within another query. The SPARQL1.1 specification will only allow very limited subqueries, whereas a discussion of further options for subqueries within SPARQL has been presented by Angles and Gutierrez [2]; again we will discuss where Datalog fits in the picture.

Project expressions. This feature will allow one to compute values from expressions within queries, rather than just returning terms appearing in the queried RDF graph; built-ins within Datalog provide similar functionality.

Property paths. Many classes of queries over RDF graphs require traversing hierarchical data structures and involve arbitrary-length paths. While such queries over graph-based structures can be naturally expressed in languages like Datalog, is was not possible to express such queries using the original

SPARQL recommendation. The ability to formulate certain path queries has now been added in SPARQL1.1 and again the design choices have been influenced by discussions in academia [3,10].

Inferred results under different Entailment Regimes. The [8] specifies various entailment regimes for SPARQL, particularly for RDF Schema, OWL, and RIF; apart from the above-mentioned combination of SPARQL with RIF, we will particularly discuss those entailment regimes that are most closely related to Datalog, i.e. those based on the OWL fragments OWL 2 RL (which essentially includes RDF Schema [6]) and OWL 2 QL [11].

Acknowledgements. The presenter – who has been actively contributing to both the RIF and SPARQL specifications – would like to thank all the members of the RIF[1] and SPARQL[2] W3C working groups.

References

1. Angles, R., Gutierrez, C.: The Expressive Power of SPARQL. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 114–129. Springer, Heidelberg (2008)
2. Angles, R., Gutierrez, C.: Subqueries in SPARQL. In: Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW2011), Santiago, Chile. CEUR Workshop Proceedings, vol. 749. CEUR-WS.org (May 2011)
3. Arenas, M., Conca, S., Pérez, J.: Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In: Proceedings of the 21st World Wide Web Conference (WWW 2012), Lyon, France, pp. 629–638. ACM (April 2012)
4. Bizer, C., Schultz, A.: The R2R framework: Publishing and discovering mappings on the web. In: 1st International Workshop on Consuming Linked Data (COLD 2010), Shanghai, China (November 2010)
5. Boley, H., Hallmark, G., Kifer, M., Paschke, A., Polleres, A., Reynolds, D.: RIF Core Dialect. W3C recommendation, W3C (June 2010), `http://www.w3.org/TR/rif-core/`
6. Brickley, D., Guha, R., McBride, B. (eds.): RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, W3C (February 2004), W3C Recommendation.
7. Faber, W., Leone, N., Pfeifer, G.: Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 200–212. Springer, Heidelberg (2004)
8. Glimm, B., Ogbuji, C., Hawke, S., Herman, I., Parsia, B., Polleres, A., Seaborne, A.: SPARQL 1.1 Entailment Regimes. W3C working draft, W3C (January 2012), `http://www.w3.org/TR/sparql11-entailment/`
9. Knublauch, H., Hendler, J.A., Idehen, K.: SPIN - Overview and Motivation (February 2011); W3C member submission

[1] `http://www.w3.org/2005/rules/wiki/RIF_Working_Group`
[2] `http://www.w3.org/2009/sparql/wiki/Main_Page`

10. Losemann, K., Martens, W.: The complexity of evaluating path expressions in sparql. In: Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2012), Scottsdale, AZ, USA, pp. 101–112. ACM (May 2012)
11. Motik, B., Grau, B.C., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C., Calvanese, D., Carroll, J., De Giacomo, G., Hendler, J., Herman, I., Parsia, B., Patel-Schneider, P.F., Ruttenberg, A., Sattler, U., Schneider, M.: OWL 2 Web Ontology Language Profiles. W3C recommendation, W3C (October 2009), http://www.w3.org/TR/owl2-profiles/
12. Mumick, I.S., Shmueli, O.: Finiteness properties of database queries. In: 4th Australian Database Conference (1993)
13. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and Complexity of SPARQL. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 30–43. Springer, Heidelberg (2006)
14. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. ACM Transactions on Database Systems, 34(3), Article 16, 45 pages (2009)
15. Polleres, A.: From SPARQL to rules (and back). In: Proceedings of the 16th World Wide Web Conference (WWW 2007), Banff, Canada, pp. 787–796. ACM Press (May 2007)
16. Polleres, A., Boley, H., Kifer, M.: RIF Datatypes and Built-Ins 1.0. W3C recommendation, W3C (May 2010), http://www.w3.org/TR/2010/rif-dtb/
17. Polleres, A., Scharffe, F., Schindlauer, R.: SPARQL++ for Mapping Between RDF Vocabularies. In: Meersman, R., Tari, Z. (eds.) OTM 2007, Part I. LNCS, vol. 4803, pp. 878–896. Springer, Heidelberg (2007)
18. Polleres, A., Schindlauer, R.: dlvhex-sparql: A SPARQL-compliant query engine based on dlvhex. In: 2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS 2007), Porto, Portugal. CEUR Workshop Proceedings, vol. 287, pp. 3–12. CEUR-WS.org. (September 2007)
19. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3c recommendation, W3C (January 2008), http://www.w3.org/TR/rdf-sparql-query/
20. Schenk, S., Staab, S.: Networked graphs: A declarative mechanism for sparql rules, sparql views and rdf data integration on the web. In: Proceedings WWW 2008, Beijing, China, pp. 585–594. ACM Press (2008)
21. Seaborne, A., Harris, S.: SPARQL 1.1 Query Language. W3C working draft, W3C (January 2012), http://www.w3.org/TR/sparql11-query/

Magic-Sets for *Datalog* with Existential Quantifiers

Mario Alviano, Nicola Leone, Marco Manna[⋆],
Giorgio Terracina, and Pierfrancesco Veltri

Department of Mathematics, University of Calabria, Italy
{alviano,leone,manna,terracina,veltri}@mat.unical.it

Abstract. *Datalog*$^{\exists}$ is the extension of *Datalog* allowing existentially quantified variables in rule heads. This language is highly expressive and enables easy and powerful knowledge-modelling, but the presence of existentially quantified variables makes reasoning over *Datalog*$^{\exists}$ undecidable in the general case. Restricted classes of *Datalog*$^{\exists}$, such as *Shy*, have been proposed in the literature with the aim of enabling powerful, yet decidable query answering on top of *Datalog*$^{\exists}$ programs. However, in order to make such languages attractive it is necessary to guarantee good performance for query answering tasks. This paper works in this direction: improving the performance of query answering on *Datalog*$^{\exists}$. To this end, we design a rewriting method extending the well-known Magic-Sets technique to any *Datalog*$^{\exists}$ program. We demonstrate that our rewriting method preserves query equivalence on *Datalog*$^{\exists}$, and can be safely applied to *Shy* programs. We therefore incorporate the Magic-Sets method in DLV$^{\exists}$, a system supporting *Shy*. Finally, we carry out an experiment assessing the positive impact of Magic-Sets on DLV$^{\exists}$, and the effectiveness of the enhanced DLV$^{\exists}$ system compared to a number of state-of-the-art systems for ontology-based query answering.

1 Introduction

Datalog is one of the best-known rule-based languages, and extensions of it are used in a wide context of applications. *Datalog* is especially useful in various Artificial Intelligence applications as it allows for effective encodings of incomplete knowledge. However, in recent years an important shortcoming of *Datalog*-based languages became evident, especially in the context of Semantic Web applications: The language does not permit the generation and the reasoning about unnamed individuals in an obvious way. In particular, it is weak in supporting many cases of existential quantification needed in the field of ontology-based query answering (QA), which is becoming more and more a challenging task [11,13,9,17] attracting also interest of commercial companies.

As an example, big companies such as Oracle are adding ontological reasoning modules on top of their existing software. In this context, queries are not merely evaluated on an extensional relational database D, but over a logical theory combining D with an *ontological theory* Σ. More specifically, Σ describes rules and constraints for inferring intensional knowledge from the extensional data stored in D. Thus, for a conjunctive query (CQ) q, it is not only checked whether D entails q, but rather whether $D \cup \Sigma$ does.

⋆ Supported by the European Commission through the European Social Fund and by Calabria Region.

A key issue in ontology-based QA is the design of the language that is provided for specifying the ontological theory Σ. In this regard, Datalog$^{\pm}$ [9], the novel family of *Datalog*-based languages proposed for tractable QA over ontologies, is arousing increasing interest. This family, generalizing well-known ontology specification languages, is mainly based on *Datalog*$^{\exists}$, the natural extension of *Datalog* [1] that allows $\exists$-quantified variables in rule heads. Unfortunately, a major challenge for *Datalog*$^{\exists}$ is decidability. In fact, without any restriction, QA over *Datalog*$^{\exists}$ is not decidable; thus, the identification of subclasses for which QA is decidable is desirable.

Different *Datalog*$^{\exists}$ fragments have been proposed in the literature, which essentially rely on four main syntactic paradigms called *guardedness* [8], *weak-acyclicity* [15], *stickiness* [10] and *shyness* [18]. The complexity of QA on these fragments, which offer different levels of expressivity, ranges from $\mathbf{AC}_0$ to **EXP**. Hence, optimization techniques are crucial to make QA effectively usable in real world scenarios, especially for those fragments providing high degrees of expressiveness.

The contribution of this paper goes exactly in this direction. We first focus on optimization strategies for improving QA tasks over decidable *Datalog*$^{\exists}$ fragments, and in particular on the well-known Magic-Sets optimization. We then focus on *Shy*, a *Datalog*$^{\exists}$ class based on shyness, enabling tractable QA, offering a good balance between expressivity and complexity, and suitable for an efficient implementation.

The original Magic-Sets technique was introduced for *Datalog* [5]. Many authors have addressed the issue of extending Magic-Sets to broader languages, including non-monotonic negation [14], disjunctive heads [16,2], and uninterpreted function symbols [12,3]. In order to bring Magic-Sets to the more general framework of *Datalog*$^{\exists}$, two main difficulties must be faced: the first is, obviously, the presence of existentially quantified variables; the second regards the correctness proof of a Magic-Sets rewriting. In fact, while a *Datalog* program can be associated with a universal model that comprises finitely many atoms, the universal model of a *Datalog*$^{\exists}$ program comprises in general infinitely many atoms. These difficulties are faced and solved in this paper, whose main contributions are as follows:

- We design a Magic-Sets rewriting algorithm handling existential quantifiers, and thus suitable for *Datalog*$^{\exists}$ programs in general.
- We demonstrate that our Magic-Sets algorithm preserves query equivalence for any *Datalog*$^{\exists}$ program.
- We show how Magic-Sets can be safely applied to *Shy* programs.
- We implement the Magic-Sets strategy in DLV$^{\exists}$, a bottom-up evaluator of CQs over *Shy* programs.
- We experiment on QA over a well-known benchmark ontology, named LUBM. The results evidence the optimization potential provided by Magic-Sets and confirm the effectiveness of DLV$^{\exists}$, which outperforms all compared systems in the benchmark.

2 *Datalog*$^{\exists}$

In this section we introduce *Datalog*$^{\exists}$ programs and CQs, and we equip such structures with a formal semantics.

2.1 Preliminaries

The following notation will be used throughout the paper. We always denote by Δ_C, Δ_N, $\Delta_\forall$ and $\Delta_\exists$, countably-infinite pairwise-disjoint domains of *terms* called *constants*, *nulls*, *universal variables* and *existential variables*, respectively; by Δ, the union of these four domains; by t, a generic *term*; by $\mathtt{c}$, $\mathtt{d}$ and $\mathtt{e}$, constants; by φ, a null; by $\mathtt{X}$ and $\mathtt{Y}$, variables; by $\mathbf{X}$ and $\mathbf{Y}$, sets of variables; by Π an alphabet of *predicate symbols* each of which, say $\mathtt{p}$, has a fixed nonnegative arity, denoted by $\mathsf{arity}(\mathtt{p})$; by $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, *atoms* being expressions of the form $\mathtt{p}(t_1,\ldots,t_k)$, where $\mathtt{p}$ is a predicate symbol and $t_1,\ldots,t_k$ is a *tuple* of terms (also denoted by $\bar{t}$). Moreover, if the tuple of an atom consists of only constants and nulls, then this atom is called *ground*; if $T \subseteq \Delta_C \cup \Delta_N$, then $\mathsf{base}(T)$ denotes the set of all ground atoms that can be formed with predicate symbols in Π and terms from T; if $\mathbf{a}$ is an atom, then $\mathsf{pred}(\mathbf{a})$ denotes the predicate symbol of $\mathbf{a}$; if ς is any formal structure containing atoms, then $\mathsf{dom}(\varsigma)$ denotes all the terms from $\Delta_C \cup \Delta_N$ occurring in the atoms of ς.

A *mapping* is a function $\mu : \Delta \to \Delta$ s.t. $c \in \Delta_C$ implies $\mu(c) = c$, and $\varphi \in \Delta_N$ implies $\mu(\varphi) \in \Delta_C \cup \Delta_N$. Let T be a subset of Δ. The application of μ to T, denoted by $\mu(T)$, is the set $\{\mu(t) \mid t \in T\}$. The restriction of μ to T, denoted by $\mu|_T$, is the mapping μ' s.t. $\mu'(t) = \mu(t)$ for each $t \in T$, and $\mu'(t) = t$ for each $t \notin T$. In this case, we also say that μ is an *extension* of μ', denoted by $\mu \supseteq \mu'$. For an atom $\mathbf{a} = \mathtt{p}(t_1,\ldots,t_k)$, we denote by $\mu(\mathbf{a})$ the atom $\mathtt{p}(\mu(t_1),\ldots,\mu(t_k))$. For a formal structure ς containing atoms, we denote by $\mu(\varsigma)$ the structure obtained by replacing each atom $\mathbf{a}$ of ς with $\mu(\mathbf{a})$. The *composition* of a mapping μ_1 with a mapping μ_2, denoted by $\mu_2 \circ \mu_1$, is the mapping associating each $t \in \Delta$ to $\mu_2(\mu_1(t))$. Let ς_1 and ς_2 be two formal structures containing atoms. A *homomorphism* from ς_1 to ς_2 is a mapping h s.t. $h(\varsigma_1)$ is a substructure of ς_2 (for example, if ς_1 and ς_2 are sets of atoms, $h(\varsigma_1) \subseteq \varsigma_2$). A *substitution* is a mapping σ s.t. $t \in \Delta_N$ implies $\sigma(t) = t$, and $t \in \Delta_V$ implies $\sigma(t) \in \Delta_C \cup \Delta_N \cup \{t\}$.

2.2 Programs and Queries

A *Datalog*$^\exists$ *rule* r is a finite expression of the following form:

$$\forall \mathbf{X} \exists \mathbf{Y}\ \ \mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]} \leftarrow \mathbf{conj}_{[\mathbf{X}]} \tag{1}$$

where (i) $\mathbf{X} \subseteq \Delta_\forall$ and $\mathbf{Y} \subseteq \Delta_\exists$ (next called $\forall$-variables and $\exists$-variables, respectively); (ii) $\mathbf{X}' \subseteq \mathbf{X}$; (iii) $\mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]}$ stands for an atom containing only and all the variables in $\mathbf{X}' \cup \mathbf{Y}$; and (iv) $\mathbf{conj}_{[\mathbf{X}]}$ stands for a *conjunction* of zero or more atoms containing only and all the variables in $\mathbf{X}$. Constants are also allowed in r. In the following, $\mathsf{head}(r)$ denotes $\mathbf{atom}_{[\mathbf{X}' \cup \mathbf{Y}]}$, and $\mathsf{body}(r)$ the set of atoms in $\mathbf{conj}_{[\mathbf{X}]}$. Universal quantifiers are usually omitted to lighten the syntax, while existential quantifiers are omitted only if $\mathbf{Y}$ is empty. In the second case, r coincides with a standard *Datalog* rule. If $\mathsf{body}(r) = \emptyset$, then r is usually referred to as a *fact*. In particular, r is called *existential* or *ground* fact according to whether r contains some $\exists$-variable or not, respectively. A *Datalog*$^\exists$ program P is a finite set of *Datalog*$^\exists$ rules. We denote by $\mathsf{preds}(P) \subseteq \Pi$ the predicate symbols occurring in P, by $\mathsf{data}(P)$ all the atoms constituting the ground facts of P, and by $\mathsf{rules}(P)$ all the rules of P being not ground facts. A predicate $\mathtt{p} \in \Pi$ is called *intentional* if there is a rule $r \in \mathsf{rules}(P)$ s.t. $\mathtt{p} = \mathsf{pred}(\mathsf{head}(r))$; otherwise, p is called *extensional*. We denote by $\mathsf{idb}(P)$ and $\mathsf{edb}(P)$ the sets of the intentional and extensional predicates occurring in P, respectively.

Example 1. The next rules belong to a *Datalog*$^\exists$ program hereafter called *P-Jungle*:

```
r1 : ∃Z pursues(Z,X)  ←  escapes(X)
r2 : hungry(Y)  ←  pursues(Y,X), fast(X)
r3 : pursues(X,Y)  ←  pursues(X,W), prey(Y)
r4 : afraid(X)  ←  pursues(Y,X), hungry(Y), strongerThan(Y,X).
```

This program describes a funny scenario where an escaping, yet fast animal X may induce many other animals to be afraid. Data for *P-Jungle* could be escapes(gazelle), fast(gazelle), prey(antelope), strongerThan(lion,antelope), and possibly pursues(lion,gazelle). We will use *P-Jungle* as a running example. □

Given a *Datalog*$^\exists$ program P, a *conjunctive query* (CQ) q over P is a first-order expression of the form $\exists \mathbf{Y}\ \mathbf{conj}_{[\mathbf{X}\cup\mathbf{Y}]}$, where $\mathbf{X} \subseteq \Delta_\forall$ are its free variables, $\mathbf{Y} \subseteq \Delta_\exists$, and $\mathbf{conj}_{[\mathbf{X}\cup\mathbf{Y}]}$ is a conjunction containing only and all the variables in $\mathbf{X}\cup\mathbf{Y}$ and possibly some constants. To highlight the free variables, we write $q(\mathbf{X})$ instead of q. Query q is called *Boolean CQ* (BCQ) if $\mathbf{X} = \emptyset$. Moreover, q is called *atomic* if $\mathbf{conj}$ is an atom. Finally, $\mathsf{atoms}(q)$ denotes the set of atoms in $\mathbf{conj}$.

Example 2. Animals pursed by a *lion* and stronger than some other animal can be retrieved by means of a CQ ∃Y pursues(lion,X), strongerThan(X,Y). □

2.3 Semantics and Query Answering

Given a set S of atoms and an atom $\mathbf{a}$, we say S entails $\mathbf{a}$ ($S \models \mathbf{a}$ for short) if there is a substitution σ s.t. $\sigma(\mathbf{a}) \in S$. Let $P \in$ *Datalog*$^\exists$. A set $M \subseteq \mathsf{base}(\Delta_C \cup \Delta_N)$ is a *model* for P ($M \models P$) if $M \models \sigma|_\mathbf{X}(\mathsf{head}(r))$ for each $r \in P$ of the form (1) and substitution σ s.t. $\sigma(\mathsf{body}(r)) \subseteq M$. Let $\mathsf{mods}(P)$ denote the set of models of P. Let $M \in \mathsf{mods}(P)$. A BCQ q is *true* w.r.t. M ($M \models q$) if there is a substitution σ s.t. $\sigma(\mathsf{atoms}(q)) \subseteq M$. Analogously, the answer of a CQ $q(\mathbf{X})$ w.r.t. M is the set $\mathsf{ans}(q, M) = \{\sigma|_\mathbf{X} : \sigma \text{ is a substitution } \wedge\ M \models \sigma|_\mathbf{X}(q)\}$. The answer of a CQ $q(\mathbf{X})$ w.r.t. a program P is the set $\mathsf{ans}_P(q) = \{\sigma : \sigma \in \mathsf{ans}(q, M)\ \forall M \in \mathsf{mods}(P)\}$. Note that for a BCQ q either $\mathsf{ans}_P(q) = \{\sigma|_\emptyset\}$ or $\mathsf{ans}_P(q) = \emptyset$; in the first case we say that q is *cautiously true* w.r.t. P, denoted by $P \models q$.

Query answering (QA) is the problem of computing $\mathsf{ans}_P(q)$, where P is a *Datalog*$^\exists$ program and q a CQ. It is well-known that QA can be carried out by using a *universal model* of P [15], that is, a model U of P s.t. for each $M \in \mathsf{mods}(P)$ there is a homomorphism h satisfying $h(U) \subseteq M$. In this regard, given a universal model U of P, for each CQ $q(\mathbf{X})$ and for each substitution σ s.t. $\sigma(\mathbf{X}) \subseteq \Delta_C$, it has been shown that $\sigma \in \mathsf{ans}_P(q)$ iff $\sigma \in \mathsf{ans}(q, U)$ [15]. However, although each *Datalog*$^\exists$ program admits a universal model, deciding whether a substitution belongs to $\mathsf{ans}_P(q)$ is undecidable in the general case [15]. Finally, we mention the CHASE as a well-known procedure for constructing a universal model for a *Datalog*$^\exists$ program. (See the extended version [4] of this paper for details.)

3 Magic-Sets for *Datalog*$^\exists$

The original Magic-Sets technique was introduced for *Datalog* [5]. In order to bring it to the more general framework of *Datalog*$^\exists$, we have to face two main difficulties. The first is that originally the technique was defined to handle $\forall$-variables only. How does the technique have to be extended to programs containing $\exists$-variables? The second difficulty, which is eventually due to the first one, concerns how to establish the correctness of an extension of Magic-Sets to *Datalog*$^\exists$. In fact, any *Datalog* program is characterized by a unique universal model of finite size. In this case, the correctness of Magic-Sets can be established by proving that the universal model of the rewritten program (modulo auxiliary predicates) is a subset of the universal model of the original program and contains all the answers for the input query. On the other hand, a *Datalog*$^\exists$ program may have in general many universal models of infinite size. Due to this difference, it is more difficult to prove the correctness of a Magic-Sets technique.

The difficulty associated with the presence of $\exists$-variables is circumvented by means of the following observation: A hypothetical top-down evaluation of a query over a *Datalog*$^\exists$ program would only consider the rules whose head atoms unify with the (sub)queries. Therefore, the Magic-Sets algorithm has to skip those rules whose head atoms have some $\exists$-variables in arguments that are bound from the (sub)queries. Concerning the second difficulty, we prove the correctness of the new Magic-Sets technique by considering all models of original and rewritten programs, showing that the same set of substitution answers is determined for the input query.

3.1 Magic-Sets Algorithm

Magic-Sets stem from SLD-resolution, which roughly acts as follows: Each rule r s.t. $\sigma(\mathsf{head}(r)) = \sigma'(q)$, where σ and σ' are two substitutions, is considered in a first step. Then, the atoms in $\sigma(\mathsf{body}(r))$ are taken as subqueries, and the procedure is iterated. During this process, if a (sub)query has some arguments bound to constant values, this information is used to limit the range of the corresponding variables in the processed rules, thus obtaining more targeted subqueries when processing rule bodies. Moreover, bodies are processed in a certain sequence, and processing a body atom may bind some of its arguments for subsequently considered body atoms. The specific propagation strategy adopted in a top-down evaluation scheme is called *sideways information passing strategy* (SIPS). Roughly, a SIPS is a strict partial order over the atoms of each rule which also specifies how the bindings originate and propagate [6].

In order to properly formalize our Magic-Sets algorithm, we first introduce adornments, a convenient way for representing binding information for intentional predicates.

Definition 1 (Adornments). *Let* $\mathtt{p}$ *be a predicate of arity* k. *An adornment for* $\mathtt{p}$ *is a string* $\alpha = \alpha_1 \cdots \alpha_k$ *defined over the alphabet* $\{b, f\}$. *The* i*-th argument of* $\mathtt{p}$ *is considered* bound *if* $\alpha_i = b$, *or* free *if* $\alpha_i = f$ *(*$i \in [1..k]$*).*

Binding information can be propagated in rule bodies according to a SIPS.

Algorithm 1. MS(q,P)

```
Input  : an atomic query $q = \mathtt{g}(u_1, \ldots, u_k)$ and a Datalog$^\exists$ program $P$
Output: an optimized Datalog$^\exists$ program
1  begin
2      $\alpha := \alpha_1 \cdots \alpha_k$, where $\alpha_i = b$ if $u_i \in \Delta_C$, and $\alpha_i = f$ otherwise ($i \in [1..k]$);
3      $S := \{\langle \mathtt{g}, \alpha \rangle\}$;   $D := \emptyset$;   $R^{mgc} := \{\mathsf{mgc}(q, \alpha) \leftarrow \}$;   $R^{mod} := \emptyset$;
4      while $S \neq \emptyset$ do
5          $\langle \mathtt{p}, \alpha \rangle$ := any element in $S$;   $S := S \setminus \{\langle \mathtt{p}, \alpha \rangle\}$;   $D := D \cup \{\langle \mathtt{p}, \alpha \rangle\}$;
6          foreach $r \in \mathsf{rules}(P)$ s.t. $\mathsf{head}(r) = p(t_1, \ldots, t_n)$ and
                   $t_i \in \Delta_\exists$ implies $\alpha_i = f$ ($i \in [1..k]$) do
               // $\mathbf{a} := \mathtt{p}(t_1, \ldots, t_n)$
7              $R^{mod} := R^{mod} \cup \{\mathsf{head}(r) \leftarrow \mathsf{mgc}(\mathbf{a}, \alpha) \wedge \mathsf{body}(r)\}$;
8              foreach $q(s_1, \ldots, s_m) \in \mathsf{body}(r)$ s.t. $q \in \mathsf{idb}(P)$ do
                   // $\mathbf{b} := \mathtt{q}(s_1, \ldots, s_m)$
9                  $B := \{\mathbf{c} \in \mathsf{body}(r) \mid \mathbf{c} \prec_r^\alpha \mathbf{b}\}$;
10                 $\beta := \beta_1 \cdots \beta_m$, where $\beta_i = b$ if $s_i \in \Delta_C \cup f_r^\alpha(B)$, and
                       $\beta_i = f$ otherwise ($i \in [1..k]$);
11                 $R^{mgc} := R^{mgc} \cup \{\mathsf{mgc}(\mathbf{b}, \beta) \leftarrow \mathsf{mgc}(\mathbf{a}, \alpha) \wedge B\}$;
12                 if $\langle q, \beta \rangle \notin D$ then  $S := S \cup \{\langle \mathtt{q}, \beta \rangle\}$;
13     return $R^{mgc} \cup R^{mod} \cup \{\mathbf{a} \leftarrow \ \mid \mathbf{a} \in \mathsf{data}(P)\}$;
```

Definition 2 (SIPS). *Let r be a Datalog$^\exists$ rule and α an adornment for* $\mathsf{pred}(\mathsf{head}(r))$. *A SIPS for r w.r.t. α is a pair $(\prec_r^\alpha, f_r^\alpha)$, where: $\prec_r^\alpha$ is a strict partial order over* $\mathsf{atoms}(r)$ *s.t.* $\mathbf{a} \in \mathsf{body}(r)$ *implies* $\mathsf{head}(r) \prec_r^\alpha \mathbf{a}$*; f_r^α is a function assigning to each atom* $\mathbf{a} \in \mathsf{atoms}(r)$ *the subset of the variables in* $\mathbf{a}$ *that are made bound after processing* $\mathbf{a}$*; f_r^α must guarantee that $f_r^\alpha(\mathsf{head}(r))$ contains only and all the variables of* $\mathsf{head}(r)$ *corresponding to bound arguments according to α.*

The auxiliary atoms introduced by the algorithm are obtained as described below.

Definition 3 (Magic Atoms). *Let* $\mathbf{a} = p(t_1, \ldots, t_k)$ *be an atom and α be an adornment for* $\mathtt{p}$*. We denote by* $\mathsf{mgc}(\mathbf{a}, \alpha)$ *the magic atom* $\mathtt{mgc\_p}^\alpha(\bar{t})$*, where: $\bar{t}$ contains all terms in $t_1, \ldots, t_k$ corresponding to bound arguments according to α; and* $\mathtt{mgc\_p}^\alpha$ *is a new predicate symbol (we assume that no standard predicate in P has the prefix "*$\mathtt{mgc\_}$*").*

We are now ready to describe the MS algorithm (Algorithm 1), associating each atomic query q over a *Datalog*$^\exists$ program P with a rewritten and optimized program MS(q,P). (More complex queries can be encoded by means of auxiliary rules.) The algorithm uses two sets, S and D, to store pairs of predicates and adornments to be propagated and already processed, respectively. Magic and modified rules are stored in the sets R^{mgc} and R^{mod}, respectively. The algorithm starts by producing the adornment associated with the query (line 1), which is paired with the query predicate and put into S (line 2). Moreover, the algorithm stores a ground fact named *query seed* into R^{mgc} (line 2). Sets D and R^{mod} are initially empty (line 2).

After that, the main loop of the algorithm is repeated until S is empty (lines 3–11). More specifically, a pair $\langle \mathtt{p}, \alpha \rangle$ is moved from S to D (line 4), and each rule r s.t.

$\mathsf{head}(r) = \mathbf{a}$ and $\mathsf{pred}(\mathbf{a}) = \mathtt{p}$ is considered (lines 5–11). Considered rules are constrained to comply with the binding information from α, that is, no existential variables have to receive a binding during this process (line 5). The algorithm adds to R^{mod} a rule named *modified rule* which is obtained from r by adding $\mathsf{mgc}(\mathbf{a}, \alpha)$ to its body.

Binding information from α are then passed to body atoms according to a specific SIPS (lines 7–11). Specifically, for each body atom $\mathbf{b} = \mathtt{q}(\bar{s})$, the algorithm determines the set B of predecessor atoms in the SIPS (line 8), from which an adornment string β for $\mathtt{q}$ is built (line 9). B and β are then used to generate a *magic rule* whose head atom is $\mathsf{mgc}(\mathbf{b}, \beta)$, and whose body comprises $\mathsf{mgc}(\mathbf{a}, \alpha)$ and atoms in B (line 10). Moreover, the pair $\langle \mathtt{q}, \beta \rangle$ is added to S unless it was already processed in a previous iteration (that is, unless $\langle \mathtt{q}, \beta \rangle \in D$; line 11). Finally, the algorithm terminates returning the program obtained by the union of R^{mgc}, R^{mod} and $\{\mathbf{a} \leftarrow \ \mid \mathbf{a} \in \mathsf{data}(P)\}$ (line 12).

Example 3. Resuming program *P-Jungle* of Example 1, we now give an example of the application of Algorithm 1. In particular, we consider SIPS s.t. atoms are totally ordered from left-to-right and binding information is propagated whenever possible. In this setting, Algorithm 1 run on query `afraid(antelope)` and *P-Jungle* yields the following rewritten program:

```
mgc_afraid^b(antelope)  ←
mgc_pursues^fb(X)  ←  mgc_afraid^b(X)
mgc_pursues^ff  ←  mgc_pursues^fb(Y)
mgc_pursues^bf(Y)  ←  mgc_hungry^b(Y)
mgc_hungry^b(Y)  ←  mgc_afraid^b(X), pursues(Y,X)

∃Z pursues(Z,X)  ←  mgc_pursues^fb(X), escapes(X)
∃Z pursues(Z,X)  ←  mgc_pursues^ff, escapes(X)
hungry(Y)  ←  mgc_hungry^b(Y), pursues(Y,X), fast(X)
pursues(X,Y)  ←  mgc_pursues^fb(Y), pursues(X,W), prey(Y)
pursues(X,Y)  ←  mgc_pursues^ff, pursues(X,W), prey(Y)
pursues(X,Y)  ←  mgc_pursues^bf(X), pursues(X,W), prey(Y)
afraid(X)  ←  mgc_afraid^b(X), pursues(Y,X), hungry(Y),
               strongerThan(Y,X)
```

A detailed description is reported in the extended version [4] of this paper. □

3.2 Query Equivalence Result

We start by establishing a relationship between the model of P and those of MS(q,P). The relationship is given by means of the next definition.

Definition 4 (Magic Variant). *Let* $I \subseteq \mathsf{base}(\Delta_C \cup \Delta_N)$, *and* $\{\mathsf{var}_i(I)\}_{i \in \mathbb{N}}$ *be the following sequence:* $\mathsf{var}_0(I) = I$*; for each* $i \geq 0$, $\mathsf{var}_{i+1}(I) = \mathsf{var}_i(I) \cup \{\mathbf{a} \in I \mid \exists \alpha$ *s.t.* $\mathsf{mgc}(\mathbf{a}, \alpha) \in \mathsf{var}_i(I)\} \cup \{\mathsf{mgc}(\mathbf{a}, \alpha) \mid \exists r, \sigma$ *s.t.* $r \in R^{mgc} \wedge \sigma(\mathsf{head}(r)) = \mathsf{mgc}(\mathbf{a}, \alpha) \wedge \sigma(\mathsf{body}(r)) \subseteq \mathsf{var}_i(I)\}$. *The fixpoint of this sequence is denoted by* $\mathsf{var}(I)$.

We point out that the magic variant of a set of atoms I comprises magic atoms and a subset of I. Intuitively, these atoms are enough to achieve a model of MS(q,P) if I is a model of P. This intuition is formalized below and proven in the extended version [4] of this paper.

Lemma 1. *If* $M \models P$*, then* $\mathsf{var}(M) \models \mathrm{MS}(q, P)$*.*

The soundness of Algorithm 1 w.r.t. QA can be now established.

Theorem 1 (Soundness). *If* $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$*, then* $\sigma \in \mathsf{ans}_P(q)$*.*

Proof. Assume $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$. Let $M \models P$. By Lemma 1, $\mathsf{var}(M) \models \mathrm{MS}(q, P)$. Since $\sigma \in \mathsf{ans}(q, \mathrm{MS}(q, P))$ by assumption, $\sigma(q) \in \mathsf{var}(M)$. Thus, $\sigma(q) \in M$ because $\mathsf{var}(M)$ comprises magic atoms and a subset of M by construction. □

To prove the completeness of Algorithm 1 w.r.t. QA we identify a set of atoms that are not entailed by the rewritten program but not due to the presence of magic atoms.

Definition 5 (Killed Atoms). *Let* $M \models \mathrm{MS}(q, P)$*. The set* $\mathsf{killed}(M)$ *is defined as follows:* $\{\mathbf{a} \in \mathsf{base}(\Delta) \setminus M \mid$ *either* $\mathsf{pred}(\mathbf{a}) \in \mathsf{edb}(P)$*, or* $\exists\alpha$ *s.t.* $\mathsf{mgc}(\mathbf{a}, \alpha) \in M\}$*.*

Since the falsity of killed atoms is not due to the Magic-Sets rewriting, one expects that their falsity can also be assumed in the original program. This intuition is formalized below and proven in the extended version [4] of this paper.

Lemma 2. *If* $M \models MS(q, P)$*,* $M' \models P$ *and* $M' \supseteq M$*, then* $M' \setminus \mathsf{killed}(M) \models P$*.*

We can finally prove the completeness of Algorithm 1 w.r.t. QA, which then establishes the correctness of Magic-Sets for queries over *Datalog*$^{\exists}$ programs.

Theorem 2 (Completeness). *If* $\sigma \in \mathsf{ans}_P(q)$*, then* $\sigma \in \mathsf{ans}(q, MS(q, P))$*.*

Proof. Assume $\sigma \in \mathsf{ans}_P(q)$. Let $M \models \mathrm{MS}(q, P)$. Let $M' \models P$ and be s.t. $M' \supseteq M$. By Lemma 2, $M' \setminus \mathsf{killed}(M) \models P$. Since $\sigma \in \mathsf{ans}_P(q)$ by assumption, $\sigma(q) \in M' \setminus \mathsf{killed}(M)$. Note that all instances of the query which are not in M are contained in $\mathsf{killed}(M)$ because the query seed belongs to M. Thus, $\sigma(q) \in M$ holds. □

4 Magic-Sets for *Shy* Programs

Among various *Datalog*$^{\exists}$ subclasses making QA computable, we are going to focus on *Shy* [18], an attractive *Datalog*$^{\exists}$ fragment which guarantees both easy recognizability and efficient answering even to CQs. After recalling basic definitions and computational results about *Shy*, we show how to guarantee shyness in the rewritten of a *Shy* program.

4.1 *Shy* Programs

Intuitively, the key idea behind *Shy* programs relays on the following *shyness* property: *During a chase execution on a* Shy *program* P*, nulls (propagated body-to-head in ground rules) do not meet each other to join.*

We now introduce the notion of *null-set* of a position in an atom. More precisely, φ^r_{X} denotes the "representative" null that can be introduced by the $\exists$-variable X occurring in rule r. (If $\langle r, \mathrm{X}\rangle \neq \langle r', \mathrm{X}'\rangle$, then $\varphi^r_{\mathrm{X}} \neq \varphi^{r'}_{\mathrm{X}'}$.) Let P be a *Datalog*$^{\exists}$ program, $\mathbf{a}$ be an atom, and X a variable occurring in $\mathbf{a}$ at position i. The *null-set* of position i in

$\mathbf{a}$ w.r.t. P, denoted by $\mathsf{nullset}(i, \mathbf{a})$, is inductively defined as follows: In case $\mathbf{a}$ is the head of some rule $r \in P$, $\mathsf{nullset}(i, \mathbf{a})$ is the singleton $\{\varphi^r_{\mathtt{X}}\}$ if $\mathtt{X} \in \Delta_\exists$; otherwise ($\mathtt{X} \in \Delta_\forall$), $\mathsf{nullset}(i, \mathbf{a})$ is the intersection of every $\mathsf{nullset}(j, \mathbf{b})$ s.t. $\mathbf{b} \in \mathsf{body}(r)$ and $\mathtt{X}$ occurs at position j in $\mathbf{b}$. In case $\mathbf{a}$ is not a head atom, $\mathsf{nullset}(i, \mathbf{a})$ is the union of $\mathsf{nullset}(i, \mathsf{head}(r))$ for each $r \in P$ s.t. $\mathsf{pred}(\mathsf{head}(r)) = \mathsf{pred}(\mathbf{a})$.

A representative null φ *invades* a variable $\mathtt{X}$ that occurs at position i in an atom $\mathbf{a}$ if φ is contained in $\mathsf{nullset}(i, \mathbf{a})$. A variable $\mathtt{X}$ occurring in a conjunction $\mathbf{conj}$ is *attacked* in $\mathbf{conj}$ by a null φ if each occurrence of $\mathtt{X}$ in $\mathbf{conj}$ is invaded by φ. A variable $\mathtt{X}$ is *protected* in $\mathbf{conj}$ if it is attacked by no null.

Definition 6. *Let Shy be the class of all Datalog*$^\exists$ *programs containing only shy rules, where a rule r is called* shy *w.r.t. a program P if the following conditions are satisfied:*

- *If a variable $\mathtt{X}$ occurs in more than one body atom, then $\mathtt{X}$ is protected in* $\mathsf{body}(r)$.
- *If two distinct $\forall$-variables are not protected in* $\mathsf{body}(r)$ *but occur both in* $\mathsf{head}(r)$ *and in two different body atoms, then they are not attacked by the same null.* □

According to Definition 6, program *P-Jungle* of Example 1 is *Shy*. Let $\mathbf{a}_1, \ldots, \mathbf{a}_{12}$ be the atoms of rules r_1–r_4 in left-to-right/top-to-bottom order, and $\mathsf{nullset}(1, \mathbf{a}_1)$ be $\{\varphi^{r_1}_{\mathtt{Z}}\}$. To show the shyness of *P-Jungle*, we first propagate $\varphi^{r_1}_{\mathtt{Z}}$ (head-to-body) to $\mathsf{nullset}(1, \mathbf{a}_4)$, $\mathsf{nullset}(1, \mathbf{a}_7)$, and $\mathsf{nullset}(1, \mathbf{a}_{10})$. Next, this singleton is propagated (body-to-head) from $\mathbf{a}_4$, $\mathbf{a}_7$ and $\mathbf{a}_3$ to $\mathsf{nullset}(1, \mathbf{a}_3)$, $\mathsf{nullset}(1, \mathbf{a}_6)$ and $\mathsf{nullset}(1, \mathbf{a}_{11})$, respectively. Finally, we observe that rules r_1–r_3 are trivially shy, and that r_4 also is because variable $\mathtt{Y}$ is not invaded in $\mathbf{a}_{12}$ even if $\varphi^{r_1}_{\mathtt{Z}}$ invades $\mathtt{Y}$ both in $\mathbf{a}_{10}$ and $\mathbf{a}_{11}$.

Shy enjoys the following notable computational properties:

- Checking whether a program is *Shy* is doable in polynomial-time.
- Query answering over *Shy* is polynomial-time computable in data complexity.[1]

4.2 Preserving Shyness in the Magic-Sets Rewriting

In Section 3, the correctness of MS has been established for *Datalog*$^\exists$ programs in general. Our goal now is to preserve the desirable shyness property in the rewritten of a *Shy* program. In fact, shyness is not preserved by MS per sé. Resuming Example 3, MS run on query `afraid(antelope)` and program *P-Jungle* may produce from r_4 a rule $\mathtt{mgc\_hungry}^b\mathtt{(Y)} \leftarrow \mathtt{mgc\_afraid}^b\mathtt{(X)},\ \mathtt{pursues(Y,X)}$, which assumes $\mathtt{hungry(}\varphi\mathtt{)}$ relevant whenever some $\mathtt{pursues(}\varphi\mathtt{,X)}$ is derived, for any $\varphi \in \Delta_N$. However, shyness guarantees that any extension of this substitution for r_4 is actually annihilated by `strongerThan(Y,X)`, which thus enforces protection on `Y`. Unfortunately, SIPS cannot represent this kind of information in general, and thus MS may yield a non-shy program. Actually, the rewritten program in Example 3 is not shy because it contains rule $\mathtt{hungry(Y)} \leftarrow \mathtt{mgc\_hungry}^b\mathtt{(Y)},\ \mathtt{pursues(Y,X)},\ \mathtt{fast(X)}$.

The problem described above originates by the inability to represent in SIPS that no join on nulls is required to evaluate *Shy* programs. We thus explicitly encode this information in rules by means of the following transformation strategy: Let r be a rule

[1] In this setting, $\mathsf{data}(P)$ are the only input while q and $\mathsf{rules}(P)$ are considered fixed.

of the form (1) in a program P, and #dom be an auxiliary predicate not occurring in P. We denote by $r^\star$ the rule obtained from r by adding a body atom #dom(X) for each protected variable X in body(r). Moreover, we denote by $P^\star$ the program comprising each rule $r^\star$ s.t. $r \in P$, and each fact #dom(c) $\leftarrow$ s.t. c $\in$ dom(P). (Note that the introduction of these facts is not really required because #dom can be treated as a built-in predicate, thus introducing no computational overhead.)

Proposition 1. *If P is Shy, then $P^\star$ is shy as well and* mods(P) = mods($P^\star$).

Now, for an atomic query q over a *Shy* program P, in order to preserve shyness, we apply Algorithm 1 to $P^\star$ and force SIPS to comply with the following restriction: Let $r \in P^\star$ and α be an adornment. For each $\mathbf{a}, \mathbf{b} \in$ body(r) s.t. $\mathbf{a} \prec_r^\alpha \mathbf{b}$, and for each variable X occurring in both $\mathbf{a}$ and $\mathbf{b}$, SIPS $(\prec_r^\alpha, f_r^\alpha)$ is s.t. $\mathbf{a} \prec_r^\alpha$ #dom(X) $\prec_r^\alpha \mathbf{b}$. (See the extended version [4] of this paper for an example.)

Theorem 3. *Let q be an atomic query. If P is Shy, then* $\mathrm{MS}(q, P^\star)$ *is Shy.*

Proof. All arguments of magic predicates have empty null-sets. Indeed, each variable in the head of a magic rule r either occurs in the unique magic atom of body(r), or appears as the argument of a #dom atom. Consequently, all rules in R^{mgc} are shy. Moreover, each rule in R^{mod} is obtained from a rule of $P^\star$ by adding a magic atom to its body. No attack can be introduced in this way because arguments of magic atoms have empty null-sets. Thus, since the original rule is shy, the modified rule is also shy. □

In order to handle CQs of the form $\exists \mathbf{Y}\ \mathbf{conj}_{[\mathbf{X} \cup \mathbf{Y}]}$, we first introduce a rule r_q of the form $q(\mathbf{X}) \leftarrow \mathbf{conj}$. We then compute $P' = \mathrm{MS}(q(\mathbf{X}), (P \cup \{r_q\})^\star)$ further restricting the SIPS for r_q to not propagate bindings via attacked variables, that is, to be s.t. Z $\in f_{r_q}^\alpha(\mathbf{conj})$ implies that Z is protected in $\mathbf{conj}$ (where α is the adornment for q). After that, we remove from P' the rule associated with the query, thus obtaining a *Shy* program P''. Finally, we evaluate the original query $\exists \mathbf{Y}\ \mathbf{conj}_{[\mathbf{X} \cup \mathbf{Y}]}$ on program P''.

Table 1. Query evaluation time (seconds) of DLV$^\exists$ and improvements (IMP) of Magic-Sets

	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}
lubm-10														
DLV$^\exists$	3.40	3.21	0.93	1.37	5.73	2.29	5.12	3.97	4.83	3.53	0.33	0.86	5.26	1.88
DLV$^\exists$+MS	1.83	1.95	0.63	0.39	1.20	0.48	2.95	1.08	3.45	2.54	0.08	0.85	0.76	1.88
IMP	46%	39%	32%	72%	79%	79%	42%	73%	29%	28%	76%	1%	86%	0%
lubm-30														
DLV$^\exists$	11.90	11.49	2.09	4.40	18.42	8.07	18.02	13.53	15.87	12.42	1.13	2.93	18.95	6.41
DLV$^\exists$+MS	6.20	6.28	1.44	1.28	3.91	1.67	9.85	3.11	11.82	7.95	0.24	2.85	2.42	6.23
IMP	48%	45%	31%	71%	79%	79%	45%	77%	26%	36%	79%	3%	87%	3%
lubm-50														
DLV$^\exists$	21.15	19.05	3.72	7.71	31.80	14.46	31.47	23.63	28.96	21.80	1.99	5.48	32.50	11.52
DLV$^\exists$+MS	10.86	11.39	2.42	2.23	6.36	3.03	16.32	5.23	20.30	14.10	0.39	5.32	4.13	11.49
IMP	49%	40%	35%	71%	80%	79%	48%	78%	30%	35%	80%	3%	87%	0%

5 Experimental Results and Discussion

We incorporated Magic-Sets in DLV$^\exists$ [18], a system supporting QA over *Shy*. Empirical evidence of the effectiveness of the implemented system is provided by means of an experiment on the well-known benchmark suite LUBM (see http://swat.cse.lehigh.edu/projects/lubm/). It refers to a university domain and includes a synthetic data generator, which we used to generate three increasing data sets, namely `lubm-10`, `lubm-30` and `lubm-50`. LUBM incorporates a set of 14 queries referred to as q_1–q_{14}, where q_2, q_6, q_9 and q_{14} contain no constants. Tests have been carried out on an Intel Xeon X3430, 2.4 GHz, with 4 Gb Ram, running Linux Operating System. For each query, we allowed 7200 seconds (two hours) or running time and 2 Gb of memory.

We first evaluated the impact of Magic-Sets on DLV$^\exists$. Specifically, we measured the time taken by DLV$^\exists$ to answer the 14 LUBM queries with and without the application of Magic-Sets. Results are reported in Table 1, where times do not include data parsing and loading as they are not affected by Magic-Sets. On the considered queries, Magic-Sets reduce running time of 50% in average, with a peak of 87% on q_{13}. If only queries with no constants are considered, the average improvement of Magic-Sets is 37%, while the average improvement rises up to 55% for queries with at least one constant. We also point out that the average improvement provided by Magic-Sets is always greater than 25% if q_{12} and q_{14} are not considered. Regarding these two queries, Magic-Sets do not provide any improvement because the whole data sets are relevant for their evaluation.

Next, we compared DLV$^\exists$ enhanced by Magic-Sets with three state-of-the-art reasoners, namely Pellet [19], OWLIM-SE [7] and OWLIM-Lite [7]. Results are reported in Table 2, where times include the *total time* required for query answering. We measured the total time, including data parsing and loading, because ontology reasoning

Table 2. Systems comparison: running time (sec.), solved queries ($\#_s$) and average time (G.Avg)

	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}	$\#_s$	G.Avg
lubm-10																
DLV$^\exists$	5	4	2	4	6	1	6	4	8	5	<1	1	6	2	14	2.87
Pellet	82	84	84	82	80	88	81	89	95	82	82	89	82	84	14	84.48
OWLIM-Lite	33	–	33	33	33	33	4909	70	–	33	33	33	33	33	12	53.31
OWLIM-SE	105	105	105	105	105	105	105	106	106	105	105	105	105	105	14	105.14
lubm-30																
DLV$^\exists$	16	13	7	14	21	3	21	12	25	18	<1	5	23	8	14	9.70
Pellet	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	–
OWLIM-Lite	107	–	107	106	107	106	–	528	–	107	106	106	107	106	11	123.18
OWLIM-SE	323	328	323	323	323	323	323	323	326	323	323	323	323	323	14	323.57
lubm-50																
DLV$^\exists$	27	23	12	23	35	6	34	22	42	31	<1	9	33	14	14	16.67
Pellet	–	–	–	–	–	–	–	–	–	–	–	–	–	–	0	–
OWLIM-Lite	188	–	190	187	189	188	–	1272	–	189	187	187	189	187	11	223.79
OWLIM-SE	536	547	536	536	536	537	536	536	542	536	536	536	536	537	14	537.35

is usually performed in contexts where data and knowledge rapidly vary, even within hours. DLV$^{\exists}$ significantly outperforms all other systems in all tested queries and data sets. Comparing the other systems, OWLIM-Lite is in general faster than Pellet and OWLIM-SE. Pellet is faster than OWLIM-SE on `lubm-10`, but it answered no tested queries in the allotted time on `lubm-30` and `lubm-50`.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases: The Logical Level. Addison-Wesley Longman Publishing Co., Inc. (1995)
2. Alviano, M., Faber, W., Greco, G., Leone, N.: Magic sets for disjunctive datalog programs. Artificial Intelligence 187–188, 156–192 (2012)
3. Alviano, M., Faber, W., Leone, N.: Disjunctive ASP with functions: Decidable queries and effective computation. Theory and Practice of Logic Programming 10(4–6), 497–512 (2010)
4. Alviano, M., Leone, N., Manna, M., Terracina, G., Veltri, P.: Magic-Sets for Datalog with Existential Quantifiers (Extended Version). Technical report, Department of Mathematics, University of Calabria, Italy (June 2012), http://www.mat.unical.it/datalog-exists/pub/12dl2.pdf
5. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proc. Int. Symposium on Principles of Database Systems, pp. 1–16 (1986)
6. Beeri, C., Ramakrishnan, R.: On the power of magic 10(1-4), 255–259 (1991)
7. Bishop, B., Kiryakov, A., Ognyanoff, D., Peikov, I., Tashev, Z., Velkov, R.: OWLIM: A family of scalable semantic repositories. Semant. Web 2, 33–42 (2011)
8. Calì, A., Gottlob, G., Kifer, M.: Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. In: Proc. of the 11th KR Int. Conf., pp. 70–80 (2008), http://dbai.tuwien.ac.at/staff/gottlob/CGK.pdf
9. Calì, A., Gottlob, G., Lukasiewicz, T.: A general datalog-based framework for tractable query answering over ontologies. In: Proc. of the 28th PODS Symp., pp. 77–86 (2009)
10. Calì, A., Gottlob, G., Pieris, A.: Advanced Processing for Ontological Queries. PVLDB 3(1), 554–565 (2010)
11. Calì, A., Gottlob, G., Pieris, A.: New Expressive Languages for Ontological Query Answering. In: Proc. of the 25th AAAI Conf. on AI, pp. 1541–1546 (2011)
12. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Magic Sets for the Bottom-Up Evaluation of Finitely Recursive Programs. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 71–86. Springer, Heidelberg (2009)
13. Calvanese, D., Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. J. Autom. Reason. 39, 385–429 (2007)
14. Faber, W., Greco, G., Leone, N.: Magic Sets and their Application to Data Integration. Journal of Computer and System Sciences 73(4), 584–609 (2007)
15. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: semantics and query answering. TCS 336(1), 89–124 (2005)
16. Greco, S.: Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries 15(2), 368–385 (2003)

17. Kollia, I., Glimm, B., Horrocks, I.: SPARQL Query Answering over OWL Ontologies. In: Antoniou, G., Grobelnik, M., Simperl, E., Parsia, B., Plexousakis, D., De Leenheer, P., Pan, J. (eds.) ESWC 2011, Part I. LNCS, vol. 6643, pp. 382–396. Springer, Heidelberg (2011)
18. Leone, N., Manna, M., Terracina, G., Veltri, P.: Efficiently Computable Datalog$^{\exists}$ Programs. In: Proc. of the 13th KR Int. Conf. (page forthcoming, 2012)
19. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semant. 5(2), 51–53 (2007)

On the CRON Conjecture

Tom J. Ameloot * and Jan Van den Bussche

Hasselt University & Transnational University of Limburg, Hasselt, Belgium

Abstract. Declarative networking is a recent approach to programming distributed applications with languages inspired by Datalog. A recent conjecture posits that the delivery of messages should respect causality if and only if they are used in non-monotone derivations. We present our results about this conjecture in the context of Dedalus, a Datalog-variant for distributed programming. We show that both directions of the conjecture fail under a strong semantical interpretation. But on a more syntactical level, we can show that positive Dedalus programs can tolerate non-causal messages, in the sense that they compute the correct answer when messages can be sent into the past.

1 Introduction

In declarative networking, distributed computations and networking protocols are modeled and programmed using formalisms based on Datalog [17]. Hellerstein has made a number of intriguing conjectures concerning the expressiveness of declarative networking [14, 15]. In the present paper, we are focusing on the CRON conjecture (Causality Required Only for Non-monotonicity).

Causality stands for the physical constraint that an effect can only happen after its cause. Applied to message delivery, this intuitively means that a sent message can only be delivered in the future, not in the past. Now, the conjecture relates the causal delivery of messages to the nature of the computations that those messages participate in, like monotone versus non-monotone, and asks us to think about the cases where causality is really needed.

There seem to be interesting real-world applications of the CRON conjecture, one of which is crash recovery. During crash recovery, a program can read an old checkpointed state and a log of received messages, which is disjoint from that state. These messages could appear to come from the "future" when put side-by-side with the old state because according to the old state, those messages have yet to be sent. Then, it is not always clear how the program should combine the old state and the message log, certainly if negation and more generally non-monotone operations are involved. One can understand the CRON conjecture as saying that during recovery, for non-monotone operations, messages from the log should be read in causal order, like the order in which they are received, and they should not be exposed all at once. From the other direction, if you know that only monotone operations are involved, the recovery could perhaps become more

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

efficient by reading the messages all at once. Distributed computations happen often in large clusters of compute nodes, where failure of nodes is not uncommon [21], and indeed distributed computing software should be robust against failures [9]. We want to avoid restarting entire computations when only a few nodes fail, and therefore it seems natural to use some lightweight crash recovery facility for individual nodes that can still make the computation succeed, although perhaps some partial results might have to be recomputed. The CRON conjecture could help us better understand how such recovery facilities can be designed.

In this paper we formally investigate the CRON conjecture in the setting of the language Dedalus, which is a Datalog-variant for distributed programming [4, 5, 15]. It turns out that stable models [12] provide a way to reason about non-causality, and we use this to formalize the CRON conjecture. A strong interpretation of the conjecture posits that causality is not needed if and only if the query computed by a Dedalus program is monotone. Neither the "if" nor the "only if" direction holds, however, which is perhaps not entirely surprising as we can do special tricks with negation. Therefore we have turned attention to a more syntactic version of the conjecture, and there we indeed find that causal message ordering is not needed for *positive* Dedalus programs in order to compute meaningful results, if these programs already behave correctly in a causal operational semantics.

This paper is organized as follows. Preliminaries on databases and Dedalus are given in Sect. 2. In Sect. 3 we give an intuitive operational semantics for Dedalus. The formalization of non-causality, the CRON conjecture, and the related results are all in Sect. 4. We conclude in Sect. 5.

2 Preliminaries

2.1 Databases and Network

A *database schema* $\mathcal{D}$ is a nonempty finite set of pairs (R, k) where R is a relation name and $k \in \mathbb{N}$ its associated arity. A relation name occurs at most once in a database schema. We also write (R, k) as $R^{(k)}$.

We assume a countably infinite universe **dom** of atomic data values that includes the set $\mathbb{N}$ of natural numbers. A fact $\boldsymbol{f}$ with *predicate* R is of the form $R(a_1, \ldots, a_k)$ with $a_i \in \mathbf{dom}$ for each $i = 1, \ldots, k$. We say that a fact $R(a_1, \ldots, a_k)$ is over a database schema $\mathcal{D}$ if $R^{(k)} \in \mathcal{D}$. A *database instance* I over $\mathcal{D}$ is a set of facts over $\mathcal{D}$.

A *network* $\mathcal{N}$ is a nonempty finite subset of **dom**. Intuitively, $\mathcal{N}$ represents a set of identifiers of compute *nodes* involved in a distributed system. This model is general enough to represent distributed computing on any network topology, because we can restrict attention to programs where nodes only send messages to nodes to which they are explicitly linked, as expressed by input relations. Now, a *distributed database instance H over $\mathcal{N}$ and a database schema $\mathcal{D}$* is a total function mapping every node of $\mathcal{N}$ to a finite (normal) database instance over $\mathcal{D}$.

2.2 Dedalus Programs

We now recall the language Dedalus, that can be used to describe distributed computations [4, 5, 15]. Essentially, Dedalus is an extension of Datalog$^\neg$ to represent updateable memory for the nodes of a network and to provide a mechanism for communication between these nodes. Here, we present Dedalus as Datalog$^\neg$ extended with annotations, which simplify the presentation.[1]

Let $\mathcal{D}$ be a database schema. Below, we write $\mathbf{B}\{\bar{\mathtt{w}}\}$, where $\bar{\mathtt{w}}$ is a tuple of variables, to denote any sequence β of atoms and negated atoms over database schema $\mathcal{D}$, such that the variables in β are precisely those in the tuple $\bar{\mathtt{w}}$. Also, let R be a relation name in $\mathcal{D}$. There are three types of Dedalus *rules over* $\mathcal{D}$:

– A *deductive* rule is a normal Datalog$^\neg$ rule over $\mathcal{D}$.
– An *inductive* rule is of the form

$$R(\bar{\mathtt{u}})\bullet \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}.$$

– An *asynchronous* rule is of the form

$$R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}.$$

So, inductive and asynchronous rules are basically normal Datalog$^\neg$ rules with respectively head-annotations "$\bullet$" and "$\mid$ y", where y is a variable. For asynchronous rules, the notation "$\mid$ y" means that the derived head facts are transferred ("piped") to the node represented by y. Intuitively, deductive, inductive and asynchronous rules express respectively local computation, updateable memory, and message sending (cf. Sect. 3). We will only consider *safe* rules: all variables of these rules occur in at least one positive body atom. Moreover, because constants can always be represented by unary input relations, we will assume that no values of **dom** occur in the rules. For technical simplicity, we also assume that rule-bodies contain at least one positive atom.

To illustrate, if $\mathcal{D} = \{R^{(2)}, S^{(1)}, T^{(2)}\}$, then the following three rules are examples of, respectively, deductive, inductive and asynchronous rules over $\mathcal{D}$:

$$\begin{aligned}
&T(\mathtt{u}, \mathtt{v}) \leftarrow R(\mathtt{u}, \mathtt{v}),\ \neg S(\mathtt{v}).\\
&T(\mathtt{u}, \mathtt{v})\bullet \leftarrow R(\mathtt{u}, \mathtt{v}).\\
&T(\mathtt{u}, \mathtt{v}) \mid \mathtt{y} \leftarrow R(\mathtt{u}, \mathtt{v}),\ S(\mathtt{y}).
\end{aligned}$$

Definition 1. *A Dedalus* program over a schema $\mathcal{D}$ *is a set of deductive, inductive and asynchronous Dedalus rules over* $\mathcal{D}$*, such that the set of deductive rules is syntactically stratifiable.*

Let $\mathcal{P}$ be a Dedalus program. We write $sch(\mathcal{P})$ to denote the schema that $\mathcal{P}$ is over. We define $idb(\mathcal{P}) \subseteq sch(\mathcal{P})$ to be the relations that occur in rule-heads of $\mathcal{P}$. We abbreviate $edb(\mathcal{P}) = sch(\mathcal{P}) \setminus idb(\mathcal{P})$. An *input* for $\mathcal{P}$ is a distributed database instance H over some network $\mathcal{N}$ and the schema $edb(\mathcal{P})$.

[1] These annotations correspond to syntactic sugar in the previous presentations of Dedalus.

3 Operational Semantics

Let $\mathcal{P}$ be a Dedalus program. Let H be an input distributed database instance for $\mathcal{P}$, over a network $\mathcal{N}$. We give an operational semantics for Dedalus, which respects causality. This operational semantics is in line with earlier formal work on declarative networking [10, 19, 13, 6, 1]. In Sect. 4, we will contrast the operational semantics with a non-causal semantics, to formalize the CRON conjecture.

In this section we will sketch the most important concepts of the operational semantics. The interested reader can consult the formal details in the appendix. To represent a possible execution of $\mathcal{P}$ on input H, we use a *run*. A run consists of *configurations* and *transitions*. A configuration describes for each node of $\mathcal{N}$ the facts that it has stored locally (state), and also what messages are in flight on the network. At the beginning, the *start configuration* assigns to each node only its local input fragment in H, and there are no messages. Now, a transition transforms one configuration into another: it selects one *active node* $x \in \mathcal{N}$ to receive some messages addressed to x and to do a local computation. Specifically, the active node x reads its old state together with the received messages. The node then executes the deductive rules to "complete" these facts, using the stratified semantics. We consider the resulting set D of deductive facts as being "all" facts that x locally has during the transition. Next, the inductive rules are given input D, and the derived facts are stored in the next state of x, always together with the local input fragment of x in H (which is preserved). Similarly, the asynchronous rules are also given input D, and the derived facts are considered messages that are sent around the network. The first component in these facts represents the addressee. The resulting configuration reflects all these actions taken by x. Then, a run $\mathcal{R}$ is an infinite sequence of such transitions, initially departing from the start configuration. Natural fairness conditions are imposed: we consider only runs in which each node is made active an infinite number of times and every sent message is eventually delivered. The operational semantics closely corresponds to that of the language Webdamlog [1].

This operational semantics is highly nondeterministic because in each transition we can choose which node is made active and also what messages it receives (from those that are in flight).

Assume a subset $out(\mathcal{P}) \subseteq idb(\mathcal{P})$, called the *output schema*, is selected: the relation names in this schema designate the intended output of the program. Following Marczak et al. [18], we define this output based on *ultimate* facts. In a run $\mathcal{R}$, we say that a fact $\boldsymbol{f}$ over schema $out(\mathcal{P})$ is *ultimate* at some node x if there is some transition after which $\boldsymbol{f}$ is present at x during every subsequent transition of x once the deductive rules are executed. Thus, this is a fact that will eventually always be present at x. The output of $\mathcal{R}$, denoted $output(\mathcal{R})$, is the union of all ultimate facts over all nodes. In this definition we ignore what node is responsible for what piece of the output, which follows the intuition of cloud computing. Since the operational semantics is nondeterministic, there can be different runs producing a different output. Program $\mathcal{P}$ is called *consistent* if individually for every input distributed database instance H, every run produces the *same* output, which we denote as $outInst(\mathcal{P}, H)$. This is an instance over *out*($\mathcal{P}$). Guaranteeing or deciding consistency in special cases is an important research topic [1, 18, 7].

As some additional terminology, in a run, for each transition t, we define the *timestamp* of the active node x during t to be the number of transitions of x that come strictly before t. This can be thought of as the local (zero-based) clock of x during t. For example, suppose we have the following sequence of active nodes: x, y, y, x, x, etc. If we would write the timestamps next to the nodes, we get this sequence: $(x, 0)$, $(y, 0)$, $(y, 1)$, $(x, 1)$, $(x, 2)$, etc.

4 CRON Conjecture

Conjecture 1. Causality Required Only for Non-monotonicity (CRON) [15]: Program semantics require causal message ordering if and only if the messages participate in non-monotonic derivations.

The CRON conjecture talks about an intuitive notion of "causality" on messages. As mentioned in the introduction, causality here stands for the physical constraint that an effect can only happen after its cause. Our operational semantics respects causality because a message can only be delivered after it was sent. When the delivery of one message causes another one to be sent, then the second one is delivered in a later transition. For this reason, we want a new formalism to reason about non-causality, which entails sending messages into the "past". We introduce such a formalism in Sect. 4.1, and in Sect. 4.2 we look at our formalizations of the CRON conjecture and the associated results.

4.1 Modeling Non-causality

In a previous work [3], we have shown that the operational semantics of Dedalus is equivalent to a declarative semantics based on stable models [12]. There, we described a *causality transform* that converts a Dedalus program to a pure Datalog$^\neg$ program containing extra rules, called the *causality rules.* When the stable model semantics is applied to this pure Datalog$^\neg$ program, these rules enforce causality on message sending. For the current work, we will remove the causality rules, and now stable models can represent non-causal message sending.

Let $\mathcal{P}$ be a Dedalus program. Below, we present the *SZ-transformation* that transforms $\mathcal{P}$ into $pure_{\mathrm{SZ}}(\mathcal{P})$, which is a pure Datalog$^\neg$ program that models the distributed computation in a holistic fashion: the distributed data of a network across all nodes and their local timestamps is modeled as facts of the form $R(x, s, \bar{a})$, representing that the fact $R(\bar{a})$ is present at node x at its timestamp s. In $pure_{\mathrm{SZ}}(\mathcal{P})$, for asynchronous rules, we also use a rewriting technique inspired by the work of Saccà and Zaniolo, who show how to express dynamic choice under the stable model semantics [20].

In $pure_{\mathrm{SZ}}(\mathcal{P})$, we will use relations of the following database schema:

$$\mathcal{D}_{\mathrm{time}} = \{\mathtt{time}^{(1)}, \mathtt{tsucc}^{(2)}, \neq^{(2)}\} \ .$$

We may assume that these relations are not in $sch(\mathcal{P})$, which can be solved with namespaces if needed. The relation '$\neq$' will be written in infix notation. We consider only the following instance over $\mathcal{D}_{\mathrm{time}}$:

$$I_{\mathtt{time}} = \{\mathtt{time}(s),\, \mathtt{tsucc}(s, s+1) \mid s \in \mathbb{N}\} \cup \{(s \neq t) \mid s, t \in \mathbb{N} :\ s \neq t\}\ .$$

This instance provides timestamps, together with a non-equality relation. Next, we will specify $pure_{\mathrm{SZ}}(\mathcal{P})$ incrementally. Let $\mathtt{x}$, $\mathtt{s}$, $\mathtt{t}$ and $\mathtt{t}'$ be variables not yet used in $\mathcal{P}$. For any sequence L of atoms and negated atoms, let $L^{\Uparrow \mathtt{x},\mathtt{s}}$ denote the sequence obtained by adding $\mathtt{x}$ and $\mathtt{s}$ as first and second components to each atom in L (negated atoms stay negated).

For each *deductive* rule '$R(\bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}$' in $\mathcal{P}$, we add to $pure_{\mathrm{SZ}}(\mathcal{P})$ the following rule:

$$R(\mathtt{x}, \mathtt{s}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}^{\Uparrow \mathtt{x},\mathtt{s}}. \tag{1}$$

This expresses that deductively derived facts are directly visible within the same step (of the same node) in which they were derived.

For each *inductive* rule '$R(\bar{\mathtt{u}})\bullet \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}$' in $\mathcal{P}$, we add to $pure_{\mathrm{SZ}}(\mathcal{P})$ the following rule:

$$R(\mathtt{x}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}^{\Uparrow \mathtt{x},\mathtt{s}},\ \mathtt{tsucc}(\mathtt{s}, \mathtt{t}). \tag{2}$$

This expresses that inductively derived facts appear in the *next* step of the *same* node.

We will also assume that the following relation names are not in $sch(\mathcal{P})$: name $\mathtt{All}$, and the names $\mathtt{cand}_R$, $\mathtt{chosen}_R$ and $\mathtt{other}_R$ for each name R in $idb(\mathcal{P})$. Now, for each asynchronous rule '$R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}$' in $\mathcal{P}$, we add to $pure_{\mathrm{SZ}}(\mathcal{P})$ the following rules, for which the intuition is given below:

$$\mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}^{\Uparrow \mathtt{x},\mathtt{s}},\ \mathtt{All}(\mathtt{y}),\ \mathtt{time}(\mathtt{t}). \tag{3}$$

$$\mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}),\ \neg\mathtt{other}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}). \tag{4}$$

$$\mathtt{other}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}),\ \mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}', \bar{\mathtt{u}}),\ \mathtt{t} \neq \mathtt{t}'. \tag{5}$$

$$R(\mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}). \tag{6}$$

A fact of the form $\mathtt{All}(x)$ means that x is a node of the network. Rule (3) represents message sending: it derives messages by evaluating the original asynchronous rule, verifies that the addressee of each message is in the network, and it considers for each message all possible candidate arrival timestamps at the addressee. In the derived facts, we include the sender's location and send-timestamp, the addressee's location and arrival-timestamp, and the actual transmitted data. Next, rules (4) and (5) together enforce under the stable model semantics that precisely one arrival timestamp will be chosen for every sent message, using the technique of [20]. Rule (6) models the actual arrival of messages, where the sender-information is projected away, and the data-tuple in the message becomes part of the addressee's state for relation R. We repeat the above transformation for all asynchronous rules in $\mathcal{P}$, and $pure_{\mathrm{SZ}}(\mathcal{P})$ is now completed. Remark: multiple asynchronous rules in $\mathcal{P}$ can have the same head predicate R, and after the above transformation, there can be multiple rules with head predicates $\texttt{cand}_R$, $\texttt{chosen}_R$, $\texttt{other}_R$ and R.

Let H be an input for $\mathcal{P}$, over a network $\mathcal{N}$. We define

$$\mathit{input}_{\mathrm{SZ}}(H) = \bigcup_{x\in\mathcal{N}} \bigcup_{s\in\mathbb{N}} \{R(x,s,\bar{a}) \mid R(\bar{a}) \in H(x)\} \cup \{\texttt{All}(x) \mid x \in \mathcal{N}\} \cup I_{\texttt{time}} \ .$$

Intuitively, in $\mathit{input}_{\mathrm{SZ}}(H)$, for each node its input facts are available at each of its local timestamps; relation All represents the network; and all timestamps are provided, together with a non-equality relation. Now, we call any stable model M of $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ on input $\mathit{input}_{\mathrm{SZ}}(H)$ an *SZ-model* of $\mathcal{P}$ on input H. Program $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ does not enforce causality on the messages in M since the arrival timestamps can be chosen arbitrarily, even into the past.

Similar to [3], we only consider "fair" models, defined as follows. We say that an SZ-model M is *fair* if for each pair $(y,t) \in \mathcal{N} \times \mathbb{N}$ there are only a finite number of facts in M of the form $\texttt{chosen}_R(x,s,y,t,\bar{a})$. This expresses that every node receives only a finite number of messages at any given timestamp. We focus on fair models because in reality a node always processes a finite number of messages at each computation step.

We define the *output* of an SZ-model M, denoted $\mathit{output}(M)$, as

$$\bigcup_{R^{(k)}\in \mathit{out}(\mathcal{P})} \{R(\bar{a}) \mid \exists x \in \mathcal{N},\, \exists s \in \mathbb{N},\, \forall t \in \mathbb{N} : \, t \geq s \Rightarrow R(x,t,\bar{a}) \in M\} \ .$$

Thus, we use the intuition of ultimate facts, as was used in the operational semantics (cf. Sect. 3). Now, a consistent program $\mathcal{P}$ is called *SZ-consistent* if individually for every input distributed database instance H, every SZ-model M yields the output $\mathit{outInst}(\mathcal{P},H)$. Intuitively, if a consistent program is SZ-consistent, then it also computes the same result when messages can be sent into the past.

4.2 Results

We have first formalized the CRON conjecture purely on the semantical level, by relating causality to the monotonicity of the queries computed by Dedalus programs. A query $\mathcal{Q}$ is a function from database instances over an input schema $\mathcal{D}_1$ to database instances over an output schema $\mathcal{D}_2$. A Dedalus program $\mathcal{P}$ can compute a query as follows: we say that $\mathcal{P}$ *(distributedly) computes* a query $\mathcal{Q}$ if $\mathcal{P}$ is consistent and for every input instance I for $\mathcal{Q}$, for every network $\mathcal{N}$, for every partition H of I over $\mathcal{N}$, we have $\mathit{outInst}(\mathcal{P},H) = \mathcal{Q}(I)$. To compute non-monotone queries, every node needs its own identifier and the identifiers of the other nodes, or equivalent information [6]. Therefore, we restrict attention to Dedalus programs $\mathcal{P}$ for which $\{\texttt{Id}^{(1)}, \texttt{Node}^{(1)}\} \subseteq \mathit{edb}(\mathcal{P})$, where relation Id is initialized to contain on every node the identifier of that node, and relation Node is initialized to contain the identifiers of all nodes (including the local node). These node identifiers are not considered part of the query input. In this context, we have looked at the following formalization of the CRON conjecture:

A Dedalus program computes a monotone query if and only if it is SZ-consistent.

Both directions of this conjecture can be refuted by counterexamples. First, for the if-direction, we give a Dedalus program that computes the *non-monotone* emptiness query on a nullary relation S, that is, output "true" (encoded by a nullary relation T) if and only if S is empty (at all nodes):

$$\begin{array}{ll} \texttt{empty}(x) \mid y \leftarrow \neg S(\,),\ \texttt{Id}(x),\ \texttt{Node}(y). & \texttt{notDone}(\,) \leftarrow \texttt{Node}(y),\ \neg\texttt{empty}(y). \\ \texttt{empty}(y)\bullet \leftarrow \texttt{empty}(y). & T(\,) \leftarrow \texttt{Id}(x),\ \neg\texttt{notDone}(\,). \end{array}$$

Here, the asynchronous rule lets each node broadcast its own identifier if its relation S is empty. The inductive rule lets a node remember all received node identifiers. The rules on the right let a node output $T(\,)$ starting at the moment that it has all identifiers (including its own). This program is consistent. There are no causal message dependencies, so it does not really matter at what time a node receives some identifier: in every SZ-model, after a while this node will still have received and stored the identifier. Thus every SZ-model yields the output $T(\,)$ iff all nodes have an empty relation S. The program is SZ-consistent.

Second, for the only-if direction, we give a (contrived) Dedalus program that computes the *monotone* non-emptiness query on a nullary relation S, that is, output "true" if and only if S is not empty (on at least one node):

$$\begin{array}{ll} A(\,) \mid x \leftarrow S(\,),\ \texttt{Id}(x). & B(\,) \mid x \leftarrow A(\,),\ \neg\texttt{sent}_B(\,),\ \texttt{Id}(x). \\ A(\,)\bullet \leftarrow A(\,). & T(\,) \leftarrow A(\,),\ B(\,). \\ \texttt{sent}_B(\,)\bullet \leftarrow A(\,). & T(\,)\bullet \leftarrow T(\,). \end{array}$$

Here, when a node has a nonempty relation S, it sends $A(\,)$ to itself continuously. On receipt of $A(\,)$, it stores this fact, and it sends $B(\,)$ to itself if it has not previously done so. Thus, if a node sends $A(\,)$ then it sends $B(\,)$ precisely once. When the $B(\,)$ is later received, it is paired with the stored $A(\,)$, producing the fact $T(\,)$ that is stored indefinitely. The program is consistent, but is however not SZ-consistent, which we now explain. Let H be the input over singleton network $\{z\}$ with $H(z) = \{S(\,)\}$. On input H, we can exhibit an SZ-model M in which $A(\,)$-facts arrive at node z starting at timestamp 1, which implies that $\texttt{sent}_B(\,)$ will exist starting at timestamp 2. This implies that $B(\,)$ is sent precisely once in M, namely, at timestamp 1. Now, the trick is to violate the causal dependency between relations A and B, and to let $B(\,)$ arrive in the past, at timestamp 0 of z, which is before any $A(\,)$ is received. Then the arriving $B(\,)$ cannot pair with any stored or arriving $A(\,)$. Since $B(\,)$ itself is not stored, we have thus erased the single chance of producing $T(\,)$. Hence $\mathit{output}(M) = \emptyset$, and the program is not SZ-consistent.

So, contrary to the CALM conjecture [15, 6, 22], a formalization of the CRON conjecture that is situated purely on the semantical level does not seem to give any results. A Dedalus program without negation is called *positive*. Our main result now is that the following does hold:

Theorem 1. *Every positive, consistent Dedalus program is SZ-consistent.*

Note that the converse direction of Theorem 1, to the effect that every SZ-consistent Dedalus program is equivalent to a positive program, cannot hold by our above counterexample for the if-direction. We sketch the proof of Theorem 1. Let $\mathcal{P}$ be a positive, *consistent* Dedalus program, and let H be an input for $\mathcal{P}$. Let M be an SZ-model of $\mathcal{P}$ on H. To show $\mathit{output}(M) \subseteq \mathit{outInst}(\mathcal{P}, H)$, we show $\mathit{output}(M) \subseteq \mathit{output}(\mathcal{R})$ where $\mathcal{R}$ is the run of $\mathcal{P}$ on H that operates in rounds: in every round all nodes empty their entire buffer, and this run saturates towards the derivation of all "possible" deductive facts per node. To show $\mathit{outInst}(\mathcal{P}, H) \subseteq \mathit{output}(M)$, we convert M to a run $\mathcal{R}$ in which we create no more opportunities for messages to "join" in comparison to M. Concretely, we make sure that in $\mathcal{R}$ we only send messages that are sent an infinite number of times in M, which, by the "fairness" assumption on M, allows us to pick an arrival time that is *still* represented by M. Hence, $\mathit{output}(\mathcal{R}) \subseteq \mathit{output}(M)$.

5 Discussion

In future work, we may want to understand better the spectrum of causality. We have seen that for positive programs no causality at all is required, and perhaps richer classes of programs can tolerate some relaxations of causality as well. We would also like to investigate how the CRON conjecture can be concretely linked to crash recovery applications, and the design of recovery mechanisms. It might also be interesting to look at other local operational semantics for Dedalus, besides the stratified semantics used here.

Acknowledgment. We thank Joseph M. Hellerstein for his thoughtful comments on an earlier draft of this paper.

References

[1] Abiteboul, S., Bienvenu, M., Galland, A., et al.: A rule-based language for Web data management. In: Proceedings 30th ACM Symposium on Principles of Database Systems, pp. 293–304. ACM Press (2011)

[2] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)

[3] Alvaro, P., Ameloot, T.J., Hellerstein, J.M., Marczak, W., Van den Bussche, J.: A declarative semantics for dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley (November 2011)

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

[5] Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in Time and Space. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 262–281. Springer, Heidelberg (2011)

[6] Ameloot, T.J., Neven, F., Van den Bussche, J.: Relational transducers for declarative networking. In: Proceedings 30th ACM Symposium on Principles of Database Systems, pp. 283–292. ACM Press (2011)
[7] Ameloot, T.J., Van den Bussche, J.: Deciding eventual consistency for a simple class of relational transducers. In: Proceedings of the 15th International Conference on Database Theory, pp. 86–98. ACM Press (2012)
[8] Apt, K.R., Francez, N., Katz, S.: Appraising fairness in languages for distributed programming. Distributed Computing 2, 226–241 (1988)
[9] Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics. Wiley (2004)
[10] Deutsch, A., Sui, L., Vianu, V., Zhou, D.: Verification of communicating data-driven Web services. In: Proceedings 25th ACM Symposium on Principles of Database Systems, pp. 90–99. ACM Press (2006)
[11] Francez, N.: Fairness. Springer-Verlag New York, Inc., New York (1986)
[12] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the Fifth International Conference on Logic Programming, pp. 1070–1080. MIT Press (1988)
[13] Grumbach, S., Wang, F.: Netlog, a Rule-Based Language for Distributed Programming. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 88–103. Springer, Heidelberg (2010)
[14] Hellerstein, J.M.: Datalog redux: experience and conjecture. Video available (under the title The Declarative Imperative) (2010), PODS 2010 keynote `http://db.cs.berkeley.edu/jmh/`
[15] Hellerstein, J.M.: The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Record 39(1), 5–19 (2010)
[16] Lamport, L.: Fairness and hyperfairness. Distributed Computing 13, 239–245 (2000)
[17] Loo, B.T.: et al. Declarative networking. Communications of the ACM 52(11), 87–95 (2009)
[18] Marczak, W., Alvaro, P., Conway, N., Hellerstein, J.M., Maier, D.: Confluence analysis for distributed programs: A model-theoretic approach. Technical Report UCB/EECS-2011-154, EECS Department, University of California, Berkeley (December 2011)
[19] Navarro, J.A., Rybalchenko, A.: Operational Semantics for Declarative Networking. In: Gill, A., Swift, T. (eds.) PADL 2009. LNCS, vol. 5418, pp. 76–90. Springer, Heidelberg (2008)
[20] Saccà, D., Zaniolo, C.: Stable models and non-determinism in logic programs with negation. In: Proceedings of the Ninth ACM Symposium on Principles of Database Systems, pp. 205–217. ACM Press (1990)
[21] Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications 1, 7–18 (2010)
[22] Zinn, D., Green, T.J., Ludaescher, B.: Win-move is coordination-free. In: Proceedings of the 15th International Conference on Database Theory, pp. 99–113. ACM Press (2012)

Appendix

A Operational Semantics

Let $\mathcal{P}$ be a Dedalus program. Let H be an input distributed database instance for $\mathcal{P}$, over a network $\mathcal{N}$. We define formally an operational semantics for Dedalus.

A.1 Subprograms

We split the program $\mathcal{P}$ into three subprograms, that contain respectively the deductive, inductive and asynchronous rules. First, we define $\mathit{deduc}_{\mathcal{P}}$ to be the Datalog$^{\neg}$ program consisting of precisely all deductive rules of $\mathcal{P}$. Secondly, we define $\mathit{induc}_{\mathcal{P}}$ to be the Datalog$^{\neg}$ program consisting of all inductive rules of $\mathcal{P}$ after the annotation "$\bullet$" in their head is removed. Thirdly, we define $\mathit{async}_{\mathcal{P}}$ to be the Datalog$^{\neg}$ program consisting of precisely all rules '$T(\mathtt{y}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \mathtt{y}\}$' where '$T(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \mathtt{y}\}$' is an asynchronous rule of $\mathcal{P}$. The first component in the rules of $\mathit{async}_{\mathcal{P}}$ will represent the addressee of messages. The programs $\mathit{deduc}_{\mathcal{P}}$, $\mathit{induc}_{\mathcal{P}}$ and $\mathit{async}_{\mathcal{P}}$ are just Datalog$^{\neg}$ programs over the schema $\mathit{sch}(\mathcal{P})$, or a sub-schema thereof. Moreover, $\mathit{deduc}_{\mathcal{P}}$ is syntactically stratifiable because the deductive rules in $\mathcal{P}$ must be syntactically stratifiable. The semantics of these subprograms is given below.

Let I be an instance over $\mathit{sch}(\mathcal{P})$. We define the *output of $\mathit{deduc}_{\mathcal{P}}$ on input I*, denoted as $\mathit{deduc}_{\mathcal{P}}(I)$, to be given by the stratified semantics [2]. This implies $I \subseteq \mathit{deduc}_{\mathcal{P}}(I)$. We define the *output of $\mathit{induc}_{\mathcal{P}}$ on input I* to be the set of facts derived by the rules of $\mathit{induc}_{\mathcal{P}}$ for all possible satisfying valuations in I, in just one derivation step. This output is denoted as $\mathit{induc}_{\mathcal{P}}\langle I \rangle$. The *output of $\mathit{async}_{\mathcal{P}}$ on input I* is defined in the same way as for $\mathit{induc}_{\mathcal{P}}$, except that we now use the rules of $\mathit{async}_{\mathcal{P}}$ instead of $\mathit{induc}_{\mathcal{P}}$. This output is denoted as $\mathit{async}_{\mathcal{P}}\langle I \rangle$.

A.2 Configurations

A *configuration* ρ of $\mathcal{P}$ on input H is a pair $(\mathrm{st}^{\rho}, \mathrm{bf}^{\rho})$ where st^{ρ} is a function that maps each node of $\mathcal{N}$ to a set of facts over $\mathit{sch}(\mathcal{P})$, and bf^{ρ} is a function that maps each node of $\mathcal{N}$ to a set of pairs of the form $\langle i, \boldsymbol{f} \rangle$, where $i \in \mathbb{N}$ and $\boldsymbol{f}$ is a fact over $\mathit{idb}(\mathcal{P})$. The set st^{ρ} represents the *state* of each node. The set bf^{ρ}, called *(message) buffer*, represents for each node all messages addressed to that node but that are not yet received. The reason for having numbers i, called *send-tags*, attached to facts in the image of bf^{ρ} is to differentiate between multiple instances of the same message being sent at different moments (to the same addressee), and these tags are not visible to the Dedalus program. The *start configuration* of $\mathcal{P}$ on input H, denoted $\mathit{start}(\mathcal{P}, H)$, is the configuration ρ defined for each $x \in \mathcal{N}$ as $\mathrm{st}^{\rho}(x) = H(x)$ and $\mathrm{bf}^{\rho}(x) = \emptyset$.

A.3 Transitions and Runs

To transform one configuration ρ_a into another configuration ρ_b, we describe *transitions* in each of which one active node does a local computation and possibly sends messages around the network. Such transitions can be chained to form a *run* that describes a full execution of the Dedalus program on the given input. As a small notational aid, for a set m of pairs of the form $\langle i, \boldsymbol{f}\rangle$, we define $\mathit{untag}(m) = \{\boldsymbol{f} \mid \exists i \in \mathbb{N} : \langle i, \boldsymbol{f}\rangle \in m\}$. Now, a *transition with send-tag* $i \in \mathbb{N}$ is a five-tuple $(\rho_a, x, m, i, \rho_b)$ such that ρ_a and ρ_b are configurations of $\mathcal{P}$ on input H, $x \in \mathcal{N}$, $m \subseteq \mathrm{bf}^{\rho_a}(x)$, and, letting

$$\begin{aligned} I = \quad & \mathrm{st}^{\rho_a}(x) \cup \mathit{untag}(m), \qquad D = \mathit{deduc}_{\mathcal{P}}(I),\\ \delta^{i\to y} = \; & \{\langle i,\, R(\bar{a})\rangle \mid R(y, \bar{a}) \in \mathit{async}_{\mathcal{P}}\langle D\rangle\} \text{ for each } y \in \mathcal{N}, \end{aligned}$$

for x and each $y \in \mathcal{N} \setminus \{x\}$ we have

$$\begin{aligned} \mathrm{st}^{\rho_b}(x) &= H(x) \cup \mathit{induc}_{\mathcal{P}}\langle D\rangle, & \mathrm{st}^{\rho_b}(y) &= \mathrm{st}^{\rho_a}(y),\\ \mathrm{bf}^{\rho_b}(x) &= (\mathrm{bf}^{\rho_a}(x) \setminus m) \cup \delta^{i\to x}, & \mathrm{bf}^{\rho_b}(y) &= \mathrm{bf}^{\rho_a}(y) \cup \delta^{i\to y} \;. \end{aligned}$$

We say that this transition is *of* the *active* node x. The transition models that the active node x reads its old state $\mathrm{st}^{\rho_a}(x)$ together with the received facts in $\mathit{untag}(m)$ (thus without the tags), and then completes this information with subprogram $\mathit{deduc}_{\mathcal{P}}$. Next, the state of x is changed to $\mathrm{st}^{\rho_b}(x)$, which always contains the input facts of x, over schema $\mathit{edb}(\mathcal{P})$, and it also includes all facts derived by subprogram $\mathit{induc}_{\mathcal{P}}$, which is applied to the deductive fixpoint. This represents that input facts are never lost, and that the facts over $\mathit{idb}(\mathcal{P})$ that are explicitly derived by $\mathit{induc}_{\mathcal{P}}$ are remembered. Only the state of x changes. The facts generated by $\mathit{async}_{\mathcal{P}}$ are called *messages*. By the syntax of $\mathit{async}_{\mathcal{P}}$, these facts have an additional first component to indicate the addressee. For each $y \in \mathcal{N}$, the set $\delta^{i\to y}$ contains all messages addressed to y: we drop the addressee-component because it is now redundant, and we attach the send-tag i. The set $\delta^{i\to y}$ is added to the buffer of y. We ignore messages with an addressee outside $\mathcal{N}$.

A *run* $\mathcal{R}$ of $\mathcal{P}$ on input H is an infinite sequence of transitions, such that *(i)* the very first configuration is $\mathit{start}(\mathcal{P}, H)$, *(ii)* the output configuration of each transition is the input configuration for the next transition, and *(iii)* the transition at ordinal i of the sequence uses send-tag i. The transition system is highly non-deterministic because in each transition we can choose the active node and also what messages to deliver. Note that messages with a valid addressee are never lost.

It is natural to require certain “fairness” conditions on the execution of a system [11, 8, 16]. A run $\mathcal{R}$ of $\mathcal{P}$ on H is called *fair* if *(i)* every node does an infinite number of transitions, and *(ii)* every sent message is eventually delivered.

Order in Datalog with Applications to Declarative Output

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
`brass@informatik.uni-halle.de`

Abstract. We propose an extension of Datalog that has "ordered predicates" (lists/arrays of tuples instead of sets of tuples). We previously suggested to specify output of Datalog programs declaratively by defining text pieces with their position. The proposal in the current paper reaches significantly farther by making order a first class citizen in the language. For database application programs, the output is an important part of the program, and should be fully integrated into the declarative language. However, order has many more applications besides specifying output. For instance, SQL has recently been extended by ranking functions, and aggregates over windows looping over sorted data — all this is needed in Datalog, too.

1 Introduction

Currently, database application programs are usually developed in a combination of two or more languages, e.g. PHP for programming and SQL for the database queries and updates. While SQL is declarative, most languages used for the programming part are not. However, SQL cannot be used for specifying complex output (e.g., generating a web page), so the programming part is necessary.

The goal of deductive databases is that a single, declarative language is used for programming *and* database tasks. The advantages of declarativity have been shown in SQL: The productivity is higher (because the programs are shorter and there is no need to think about efficient evaluation), and new technology (parallel hardware, new data structures/algorithms) can be used for existing application programs without changing them, because only the DBMS needs to be updated.

Although generating output is important in practice, it seems that there is no really good solution for Datalog yet. The standard solution in Prolog with a `write`-predicate is clearly non-declarative because it depends on the specific evaluation order used in Prolog. Non-declarative output might be acceptable for programs that do a complex calculation (specified declaratively), and contain only a small part that prints the result in the end. However, for database application programs, output is usually a significant part. Therefore, being able to specify the output declaratively is important in this case.

A well-known solution to declarative output in logic programming is to use a state argument as an accumulator pair (`IOStateIn`, `IOStateOut` in every predicate with output). While this is a good solution for programmers who think "top-down" (if coupled with syntactic abbreviations and a determinism analysis as in Mercury [1]), it contradicts a basic requirement in deductive databases:

- In deductive databases, one usually thinks bottom-up. Thus, it should be possible to understand any kind of extended Datalog programs by applying (some variant of) the usual T_P (fixpoint) operator. I.e. a naive execution of the rules in the direction of the arrow "$\leftarrow$" should be possible. Given this, using the state argument is simply no option: There could be infinitely many possible states for the first body literal of a rule. Furthermore, it seems natural that actions like output should be done in the head, not in the body.
- If Prolog is used for database applications, several solutions to a query are usually generated via backtracking (e.g. a relation is specified as a set of facts). But with the IO state solution, backtracking must be avoided (one cannot backtrack over an already performed output). Thus predicates like `findall` must be used, plus recursion over the produced list, even for really simple queries. This does not seem adequate and would deter users new to logic programming. Note that this is a consequence of the set-oriented evaluation which is a classical characteristic of deductive databases.

In the same way, using monads as in functional languages [7] is not a solution that fits to the classical Datalog bottom-up programming paradigm, and is therefore no alternative.

The basic idea of our approach is that predicates can be declared as ordered. This can be semantically understood as introducing an additional, usually hidden argument that defines the order of the facts for the predicates (of course, there are often more efficient implementations, which avoid actually storing such an argument).

If the order is not explicitly specified, the default for an ordered predicate takes into account the order of the facts or rules and the possible order defined for body literals (see Subsection 2.6). Thus, a sequence of facts becomes actually a list or array. Experience shows that relational tables not seldom need extra "position" columns for defining an order. Making this automatic and hidden could simplify working with such data.

As in [2], output is done by defining a predicate `output` which contains text pieces to be printed. With the ordered predicates of the current proposal, the position information becomes implicit, and a simple example is:

```
ordered output/1.
output('Hello, ').
output(Name) ← name(Name).
output('.\n').
name('Nina').
```

Of course, with special syntactic abbreviations, output can be made still more natural and easy (see the pattern syntax in Section 3).

But order has many more applications than just output, for instance

- ranking, top-n queries and window functions as recently added to SQL,
- list and array processing,

- aggregation functions, and
- specification of algorithms that pass through a sequence of computation states (i.e., more or less imperative algorithms).

The last two points appear already in LDL (XY-stratification) [8], but otherwise our approach is quite different and in some aspects more general.

Whereas in [2], all ordering requirements for facts of predicates were only implicitly derived from the requested ordering from the top `output` predicate, now explicit sorting information can be specified for any predicate. This facilitates the design of reusable components, and it also corresponds to a recent development in SQL. In SQL, top-n and window queries have recently become important — every major DBMS has special support for them, although the syntax is different in different systems. For instance, in Oracle it is possible to retrieve the three employees with highest salary as follows:

```
SELECT ENAME, SAL
FROM   (SELECT ENAME, SAL
        FROM   EMP
        ORDER  BY SAL DESC)
WHERE  ROWNUM <= 3
```

This example is interesting, because it shows that a subquery, corresponding to a view or a derived predicate, might need a defined order. In older SQL standards, it was not possible to use `ORDER BY` in subqueries or view definitions. In our proposal, the subquery corresponds to the following ordered predicate:

```
ordered emp_by_sal/2.
emp_by_sal<^Sal>(EName, Sal) ← emp(EName, Sal, Job).
```

In this case, the special ordering argument must be explicitly defined — this is done in "`<...>`". The "`^`" means "descending order", i.e. the highest salary first (the intuition is to think of "$\uparrow$" instead of the default order "$\downarrow$").

Now a powerful function of the system is that it can condense any ordering argument (possibly list-valued for several ordering criteria of different priority) to a single integer (in an order-preserving manner). There are several methods for doing this, but the default gives a simple array index:

```
answer(EName, Sal) ← emp_by_sal[N](EName, Sal) ∧ N ≤ 3.
```

This syntax corresponds to the understanding that `emp_by_sal` is now an array, and the array entries are records/facts.

The possibility to explicitly refer e.g. to the first, previous, next, and last fact with respect to some order significantly increases the expressiveness of the language. Of course, the construct is nonmonotonic, and already a check for the first tuple gives the possibility to simulate negation.

2 Datalog with Ordering: Syntax and Semantics

We start with Datalog with stratified negation. Terms are constants or variables (no function symbols). Rules must be range-restricted, i.e. variables that appear in the head literal or in a negative body literal must also appear in a positive body literal. Of course, all this could be generalized, but those are questions orthogonal to the issue of the current paper.

2.1 Declaration of Ordered Predicates

We assume that a subset of the predicates are declared as "ordered predicates":

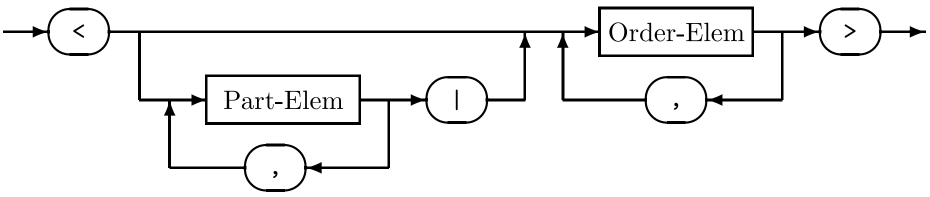
```
ordered p/n.
```

This makes predicate `p` of arity `n` an ordered predicate. When rules for `p` contain explicit ordering specifications, a special declaration is not necessary. However, there is also a default sort order for ordered predicates, and then it must be made clear that this is not a normal predicate.

2.2 Order Specifications in the Head Literal

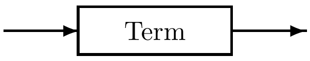
If the predicate in the head literal is an ordered predicate, an order specification is expected after the predicate name and before the argument list. The order specification is written in angle brackets `<...>`. It consists of an optional partitioning part, and an ordering part. Partitioning is necessary if one wants to rank the data values in several groups, e.g. the top 3 salaries for each job. This means that the order on the facts is not a linear order, but there can be incomparable facts. One can also view the predicate as a two-dimensional array, where the first dimension is the partitioning value (e.g. the job), and the second is the position in the defined order. With partitioning, it is possible to represent more than one list in a predicate.

Order-Specification:



The partitioning part is a comma-separated list of partitioning elements, which are simply terms. Two facts are comparable if they agree in the values for all partitioning elements. Therefore, the sequence of the partitioning elements is not important.

Part-Elem:



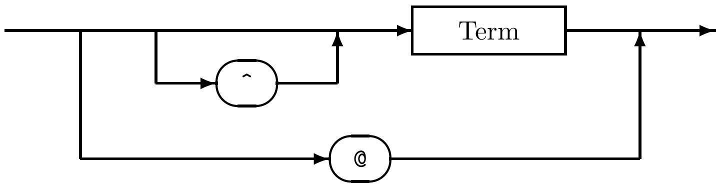
The ordering part is a comma-separated list of order elements, which are

- terms, possibly marked with the "descending" operator ^ (inversing the sort direction), or

– the special marker "`@`", which is replaced by the number of the current rule (this permits to order facts by in the sequence in which they are written in the program without having to use explicit numeric "labels" as in Basic).

The value of the first ordering element has highest priority in determining the sort order of two facts, and only if it is equal, the value of the second ordering element is considered, and so on (as in the `ORDER BY` clause of SQL).

Order-Elem:



2.3 Accessing the Sort Index in Body Literals

For body literals with an ordered predicate, one can optionally access the position in the ordered list (the "array index"). However, if several facts have the same ordering value (e.g. several employees with equal salary), this position (the `ROWNUM` or `ROW_NUMBER` in SQL), is arbitrary (defined by the implementation). Therefore, SQL introduced two more ranking functions:

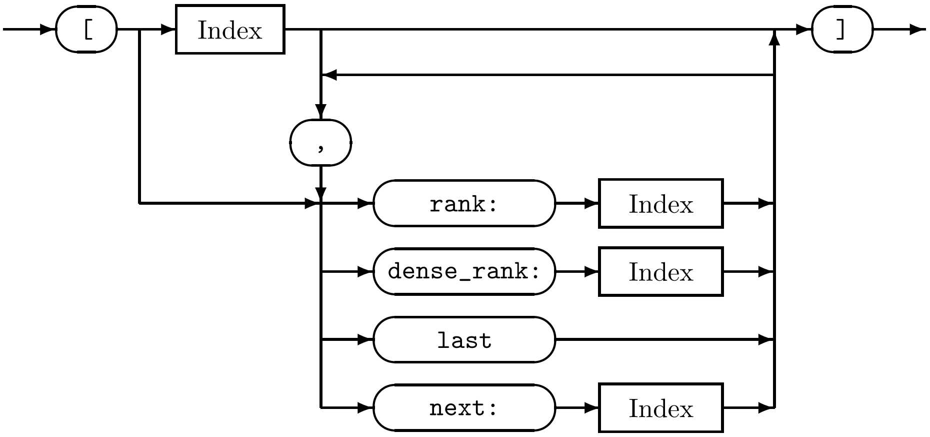
– `RANK` counts the number of tuples/facts with an ordering value less than the value in the current fact/tuple (and then adds 1, so that the first position is 1 and not 0).
– `DENSE_RANK` counts the number of distinct ordering values less than the value in the current fact/tuple (again, 1 is added).

We also permit to check for the last tuple and to get the index of the next tuple (with respect to the row number, i.e. the standard array index):

`EName`	`Sal`	`row_number`	`rank`	`dense_rank`	`last`	`next`
`Andrew`	4000	1	1	1	false	2
`Betty`	3000	2	2	2	false	3
`Chris`	3000	3	2	2	false	4
`Doris`	2000	4	4	3	false	5
`Eddy`	1000	5	5	4	false	6
`Fred`	1000	6	5	4	true	nil

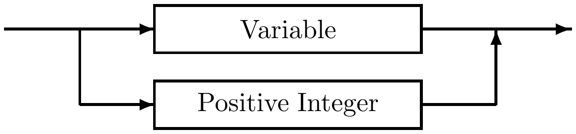
All these functions can be accessed in a single pair of brackets, since these functions can be determined in a single scan over the sorted list. The functions are distinguished by a prefix in front of the index, only the row number has no prefix, because it is most similar to an array index.

Order-Position:



An index is a variable or a positive integer constant:

Index:



As an example, consider the following SQL query:

```
SELECT ENAME, SAL, RANK() OVER (ORDER BY SAL DESC)
FROM   EMP
ORDER  BY ENAME
```

This shows that more than one ordering might be necessary during the evaluation of a query: First, the employee tuples must be sorted by salary in order to compute the rank (position in that order), and then the tuples must be sorted by employee name for output. This looks as follows in our Datalog extension:

```
emp_by_sal<^Sal>(EName, Sal) ← emp(EName, Sal, Job).
answer<EName>(EName, Sal, N) ← emp_by_sal[rank:N](EName, Sal).
```

Later, we will discuss a possibility to use an order specification directly in the body literal, so that the auxillary predicate `emp_by_sal` can be avoided.

2.4 Stratification

It is not surprising that with recursion and the possibility to determine the first literal, one can get contradictory/inconsistent situations:

```
p<10>(a) ← p[1](b).
p<20>(b).
```

If `p(b)` is the first element in the sorted list `p`, then `p(a)` is true, which then would come first. But then `p(b)` is no longer the first element. This program has no reasonable semantics and must be excluded. The solution is the same as in the case of negation: We require that there is a level mapping l, which assigns positive integers to the predicates, such that for rules containing `p[...](...)` in the body, and `q` in the head, $l(\mathtt{q}) > l(\mathtt{p})$. In any case, if `p` occurs in the body, and `q` in the head, $l(\mathtt{q}) \geq l(\mathtt{p})$.

The stratification ensures that it is possible to compute all facts about an ordered predicate before the position of a fact can be determined. It is possible to reduce the stratification requirements at least in the following cases:

- When the index position in the body is only used to determine an ordering value in the head, the exact value is not important if this is the only rule about the predicate, or the rule number is a sort criterion of higher priority. Then any order-preserving values can be used, and recursion is possible.
- When it can be ensured that a recursive rule yields only facts that will get a higher index than facts used in the body, also no contradiction can occur (this is similar to XY-stratification in LDL).

However, in order to keep the explanations simple, we will assume the stratification requirement in the following.

2.5 Evaluation

We propose a simple, "naive" evaluation method here in order to define clearly the semantics of programs in our extended Datalog dialect. Of course, for real query evaluation, many optimizations are possible (and needed in order to reach acceptable performance). But space is not sufficient to treat this topic here.

Our approach consists of rewriting the rules, evaluating them in the order of the stratification levels bottom-up, and having a special sorting and ranking step between the fixpoint computations for each level. For every ordered predicate `p` of arity n, two new predicates are introduced:

- $\mathsf{p\_head}$ of arity $n + 2$, where the additional arguments are used for the partitioning and ordering values (stored as lists),
- $\mathsf{p\_body}$ of arity $n + 4$, where the additional values are used for `row_number` (normal index), `rank`, `dense_rank`, and `next`.

The rules are rewritten in the obvious way: The head literal

$$\mathsf{p}\langle \mathsf{a}_1, \ldots, \mathsf{a}_k | \mathsf{b}_1, \ldots, \mathsf{b}_m \rangle (\mathsf{t}_1, \ldots, \mathsf{t}_n)$$

is translated to

$$\mathsf{p\_head}([\mathsf{a}_1, \ldots, \mathsf{a}_k],\ [\bar{\mathsf{b}}_1, \ldots, \bar{\mathsf{b}}_m],\ \mathsf{t}_1, \ldots, \mathsf{t}_n),$$

where $\bar{\mathsf{b}}_i$ is

- $\mathsf{desc}(\mathsf{b}'_i)$ if b_i has the form $\hat{}\,\mathsf{b}'_i$,
- the number of the current rule, if b_i is "`@`", and
- b_i otherwise.

The body literal

$$\mathsf{p}[\mathsf{a}_1,\ \mathsf{rank}\text{:}\ \mathsf{a}_2,\ \mathsf{dense\_rank}\text{:}\ \mathsf{a}_3,\ \mathsf{next}\text{:}\ \mathsf{a}_4](\mathsf{t}_1, \ldots, \mathsf{t}_n)$$

is translated to

$$\mathsf{p\_body}(\mathsf{a}_1,\ \mathsf{a}_2,\ \mathsf{a}_3,\ \mathsf{a}_4,\ \mathsf{t}_1, \ldots, \mathsf{t}_n),$$

where for each part that is not used, an anonymous variable a_i is inserted. If `last` appears in the index, $\mathsf{a}_4 = \mathsf{nil}$ is added to the body of the rule (or nil is directly inserted in the fourth argument position if next is not used).

Now the following algorithm can be used for query evaluation:

```
(1)  Let the input program $\mathsf{P}$ be split into strata $\mathsf{P}_1, \ldots, \mathsf{P}_m$;
(2)  $\mathsf{F} := \emptyset$; /* set of derived facts to be computed */
(3)  **for** $i = 1, \ldots, m$ {
(4)       Let $\mathsf{P}'_i$ be the rewritten version of $\mathsf{P}_i$;
(5)       Compute the fixpoint $\mathsf{F}_i$ of $\mathsf{T}_{\mathsf{P}'_i \cup \mathsf{F}}$;
(6)       $\mathsf{F} := \mathsf{F} \cup \{\mathsf{p}(\mathsf{c}_1, \ldots, \mathsf{c}_n) \in \mathsf{F}_i \mid \mathsf{p}$ is standard predicate of level $i\}$;
(7)       **foreach** ordered predicate $\mathsf{p}$ of stratification level $i$ {
(8)            Sort the facts about $\mathsf{p\_head}$ in $\mathsf{F}_i$ by first arg., second arg.
(9)            $\mathsf{last\_a} := \mathsf{nil}$; $\mathsf{last\_b} := \mathsf{nil}$; $\mathsf{last\_fact} := \mathsf{nil}$;
(10)           **foreach** fact $\mathsf{p\_head}(\mathsf{a}, \mathsf{b}, \mathsf{c}_1, \ldots, \mathsf{c}_n)$ in sorted sequence {
(11)                **if** ($\mathsf{a} \neq \mathsf{last\_a}$) { /* new partition */
(12)                     $\mathsf{rownum} := 1$; $\mathsf{rank} := 1$; $\mathsf{dense\_rank} := 1$;
(13)                     **if** ($\mathsf{last\_fact} \neq \mathsf{nil}$) insert $\mathsf{last\_fact}$ into $\mathsf{F}$;
(14)                } **else** { /* Not first row in partition */
(15)                     $\mathsf{rownum}$++;
(16)                     **if** ($\mathsf{b} \neq \mathsf{last\_b}$) {
(17)                          $\mathsf{rank} = \mathsf{rownum}$;
(18)                          $\mathsf{dense\_rank}$++;
(19)                     }
(20)                     insert $\mathsf{last\_fact}$ into $\mathsf{F}$ with 4th arg ($\mathsf{nil}$)
(21)                                         replaced by $\mathsf{rownum}$;
(22)                }
(23)                $\mathsf{last\_a} := \mathsf{a}$; $\mathsf{last\_b} := \mathsf{b}$;
(24)                $\mathsf{last\_fact} := \mathsf{p\_body}(\mathsf{rownum}, \mathsf{rank}, \mathsf{dense\_rank}, \mathsf{nil}$,
(25)                                         $\mathsf{c}_1, \ldots, \mathsf{c}_n)$;
(26)           }
(27)           **if** ($\mathsf{last\_fact} \neq \mathsf{nil}$) insert $\mathsf{last\_fact}$ into $\mathsf{F}$;
(28)      }
(29) }
(30) Print tuples in $\mathsf{F}$ of main predicate (e.g. $\mathsf{output\_body}$ or $\mathsf{answer\_body}$)
(31)      without the first four added arguments, sorted by first arg.
```

The sorting requires some explanation. For the partitioning argument, no particular sort sequence is needed. It is only important that facts with the same value in the partitioning argument appear next to each other. If the partitioning argument is equal, the order argument determines the sort sequence. The order argument is a list, and the first position, where the two lists differ, determines the order of the two facts. If one list is a prefix of the other, the shorter list comes first. Otherwise, the list elements are ordered as follows:

- numbers come first (in the usual order),
- then strings (in alphabetic order),
- then terms `desc(S)`, where `S` is a string (in inverse alphabetic order),
- then terms `desc(N)`, where `N` is a number (in inverse numeric order).

If different types are compared, this is likely an error, and one could print a warning. It is possible that distinct facts have identical ordering values: For the functions `rank` and `dense_rank` this is important, for the function `row_number` (and thus for output) the implementation can decide which fact to put first.

2.6 An Abbreviation: Default Order Specification

If a predicate is declared as ordered, but the head of a rule contains no order specification, a default order specification is constructed as follows: First the number of the rule, and then the index value of every body literal with an ordered predicate in the body (in the order of occurrence in the body). E.g. consider

$$\texttt{p(...)} \leftarrow \texttt{q(...)} \wedge \texttt{r(...)} \wedge \texttt{s(...)}.$$

If this is the i-th rule about `p`, and `p`, `q`, `s` are ordered predicates (while `r` is a standard predicate), this rule is an abbreviation for

$$\texttt{p<i,X,Y>(...)} \leftarrow \texttt{q[X](...)} \wedge \texttt{r(...)} \wedge \texttt{s[Y](...)}.$$

Here `X` and `Y` are the index positions of the facts used for the body literals with ordered predicates — this order is reflected in the derived facts.

Note that this order corresponds to the order in which Prolog would compute the `p`-facts (for non-recursive rules). Note also that rules become somewhat similar to comprehension syntax [3]: One constructs a list in the body with ordered predicates and can then add other conditions to filter the list elements.

If an ordered predicate is defined by a set of facts, without explicit order specification, the default order is the order in which the facts are written down.

This abbreviation also fits to the understanding of a query (goal) as a rule body: It suffices to translate

$$\leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_n \wedge \mathsf{L}_{n+1} \wedge \cdots \wedge \mathsf{L}_m.$$

to

$$\texttt{ordered answer}/k.$$
$$\texttt{answer}(X_1, \ldots, X_k) \leftarrow \mathsf{L}_1 \wedge \cdots \wedge \mathsf{L}_n \wedge \mathsf{L}_{n+1} \wedge \cdots \wedge \mathsf{L}_m.$$

where $X_1, \ldots, X_k$ are the variables appearing in the query ("answer variables"). Then the default order specification is used to automatically propagate orders declared for the query literals to the result of the query.

3 Applications to Output

As mentioned in the introduction, output is done by defining an ordered predicate "`output`" which contains text pieces to be printed in the defined order. Of course, the predicate "`output`" is only the "main" predicate, which composes the final document out of text fragments defined in many other ordered predicates.

Suppose we want to generate an HTML-table with name and salary of the employees, ordered first by salary (descending) and for equal salary by name. Using standard rules with ordered predicates this is possible, but for longer texts, it looks somewhat clumsy. Instead, we propose an "output pattern" syntax, in which one can write any text (between the special symbols "`(#`" and "`#)`"), and mark places in the text where argument values "`<$>`" or texts defined by other predicates "`<#>`" should be inserted. Each pattern corresponds to a predicate.

```
sal_table(#
    <table>
    <tr><th>Employee</th><th>Salary</th></tr>
    <#sal_table_row>
    </table>
#).

sal_table_row<^Sal,EName>(#
        <tr><td><$EName></td><td><$Sal></td></tr>
#) ← emp(EName, Sal, _).
```

This is automatically translated to standard rules with ordered predicates by splitting the text of the pattern into pieces where something must be inserted. Note that the piece numbers do not have to be written by the user, they are automatically assigned by the system. Even if one writes the rules directly, piece numbers can usually be avoided or replaced by "`@`" (current rule number).

```
sal_table<1>('<table>\n').
sal_table<2>('<tr><th>Employee</th><th>Salary</th></tr>\n').
sal_table<3,Pos>(Text) ← sal_table_row[Pos](Text).
sal_table<4>('\n</table>\n').

sal_table_row<^Sal, EName, 1>('<tr><td>')   ← emp(Ename, Sal, _).
sal_table_row<^Sal, EName, 2>(EName)        ← emp(Ename, Sal, _).
sal_table_row<^Sal, EName, 3>('</td><td>')  ← emp(Ename, Sal, _).
sal_table_row<^Sal, EName, 4>(Sal)          ← emp(Ename, Sal, _).
sal_table_row<^Sal, EName, 5>('</td></tr>') ← emp(Ename, Sal, _).
```

4 More Applications

The possibility to number facts and to compute the next number permits to write loops over sets of facts. This can be used to write aggregation functions. E.g., the following program computes the sum of all salaries.

```
emp_list<EName>(EName, Sal) ← emp(EName, Sal, Job).
sal_sum(1, 0).
sal_sum(N1, S1) ←
    sal_sum(N, S),
    emp_list[N,next:N1](EName, Sal),
    S1 is S + Sal.
answer(S) ← sal_sum(nil, S).
```

5 Discussion of Alternatives

Quite often, the values used for ordering and partitioning are arguments of the head literal, so they could be directly marked there: `emp_list(EName, <^Sal>)`. It would also be possible to declare sorting and partitioning in the "`ordered`"-declaration for the predicate. However, this works only sometimes: E.g. for output applications, there often is no explicit ordering argument, and each rule has a different ordering specification. The solution proposed here is more general.

Instead of a single declaration "`ordered`", one could consider many different types of predicates. For instance, an "`ordered_set`" might eliminate duplicate facts which differ only in the value of the hidden ordering argument (one could use always the smallest/first ordering value) — this would be helpful for termination if one allows recursion. The converse case is when the hidden argument is used only for duplicates, and no ordering is required.

User-defined orders could be permitted. Besides several chains of linear orders, as in the current proposal, arbitrary partial orders could be used.

If one wants to determine the sequence number of a row, it looks nice if the ordering specification can be done in the body literal, i.e. the top-earning employees could be determined without an auxillary predicate:

```
answer(EName, Sal) ← emp<^Sal>[N](EName, Sal, Job) ∧ N ≤ 3.
```

However, consider this case:

```
answer(EName, Sal) ← emp<^Sal>[N](EName, Sal, Job) ∧
                     N ≤ 3 ∧ programmer(Job).
programmer('Programmer').
```

The question is whether only programmers are considered when assigning row numbers, or row numbers are assigned first, and then programmers are selected (in which case the result may be empty when there is no programmer among the top earning employees). The problem is that it is no longer sufficient to consider a single assignment of values to variables when the rule is applied. Basically an aggregation is done here in the body (one counts the number of higher earning employees). The predicate `findall` in Prolog has a similar problem ($\wedge$ is not commutative). A solution is to mark arguments that should not be used for the purpose of determining row numbers, i.e. that are temporarily replaced by an anonymous variable, and after the numbers are assigned, the original argument is used (which can perform a selection or join). This can also be described by introducing a temporary predicate for the call.

6 Related Work

Datalog with arrays was studied in [5,4], but there the arrays are terms and the emphasis is on using the indexed memory access for efficiency. Datalog with a multiset semantics for predicates (i.e. allowing duplicates) was investigated in [6]. They use "colored sets" which is similar to an additional hidden argument. A database query language for more general collection types based on comprehension syntax was discussed in [3].

7 Conclusions

We proposed an extension of Datalog that permits to specify the order of facts for predicates. If the vision of a deductive database as a declarative, integrated system for developing database applications should come true, such an extension is needed for two reasons: (1) The system must support more or less all features of SQL, and SQL has not only `ORDER BY`, but also functions like `RANK`, which permit to use order in conditions. (2) Output is an essential part of database applications, and output is necessarily a sequence of text pieces. Actually, every query result that consists of more than just a few rows will be easier to read and understand if it is ordered in some reasonable way.

The proposed extension is also interesting for the following reasons: (3) It permits to work on lists in a very direct and simple way, without structured terms, and often without recursion. This might help the non-sophisticated user. (4) For the power user, also loops over facts can be written, e.g. for computing aggregation functions. In this way, the expressiveness of the language is extended.

A small prototype is being implemented that allows to experiment with the language, see

`http://www.informatik.uni-halle.de/~brass/order/`

The prototype is also interesting because the input program is internally represented as a set of Datalog facts. Our long-term goal is to develop a Datalog-to-C++ compiler written in Datalog. The constructs introduced in this paper, especially for output, are necessary for this purpose.

References

1. Becket, R.: Mercury tutorial. Tech. rep., University of Melbourne, Dept. of Computer Science (2010)
2. Brass, S.: Declarative output by ordering text pieces. In: Gallagher, J., Gelfond, M. (eds.) Technical Communications of the 27th International Conference on Logic Programming (ICLP 2011). Leibniz International Proceedings in Informatics (LIPIcs), vol. 11, pp. 151–161. Schloss Dagstuhl (2011)
3. Buneman, P., Libkin, L., Suciu, D., Tannen, V., Wong, L.: Comprehension syntax. SIGMOD Record 23(1), 87–96 (1994)
4. Greco, S., Palopoli, L., Spadafora, E.: Querying datalog with arrays: Design and implementation issues. Journal of Systems Integration 6, 299–327 (1996)
5. Greco, S., Palopoli, L., Spadafora, E.: Extending datalog with arrays. Data & Knowledge Engineering 17(1), 31–57 (1995)
6. Mumick, I.S., Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates. In: McLeod, D., Sacks-Davis, R., Schek, H.J. (eds.) Proc. of the 16th International Conf. on Very Large Data Bases (VLDB 1990), pp. 264–277. Morgan Kaufmann (1990)
7. Wadler, P.: How to declare an imperative. ACM Computing Surveys 29(3), 240–263 (1997), `http://homepages.inf.ed.ac.uk/wadler/topics/monads.html`
8. Zaniolo, C., Arni, N., Ong, K.: Negation and Aggregates in Recursive Rules: The LDL++ Approach. In: Ceri, S., Tanaka, K., Tsur, S. (eds.) DOOD 1993. LNCS, vol. 760, pp. 204–221. Springer, Heidelberg (1993)

A Broad Class of First-Order Rewritable Tuple-Generating Dependencies

Cristina Civili and Riccardo Rosati

Dipartimento di Ingegneria Informatica, Automatica e Gestionale Antonio Ruberti,
Sapienza Università di Roma, Italy
lastname@dis.uniroma1.it

Abstract. We study reasoning, and in particular query answering, over databases with tuple-generating dependencies (TGDs). Our focus is on classes of TGDs for which conjunctive query answering is *first-order rewritable*, i.e., can be reduced to the standard evaluation of a first-order query over the database. In this paper, we define the class of *weakly recursive* TGDs, and prove that this class comprises and generalizes every previously known FOL-rewritable class of TGDs, under fairly general assumptions on the form of the TGDs.

1 Introduction

A lot of interesting recent works on Datalog extensions [4,1,2,14,8,12,15] are based on the idea of extending datalog rules with existential variables in rule heads. Rules with existential variables correspond to *tuple-generating dependencies*, a well-known form of database dependencies in database theory (see, e.g., [3]). In fact, the problem of reasoning over datalog programs with existential variables in rule heads corresponds to the problem of reasoning over a database with tuple-generating dependencies under an *open-world assumption*: specifically, the semantics of a database D with a set P of TGDs is given by the set of databases B corresponding to all the supersets of D that satisfy the TGDs P. This implies that, differently from the standard closed-world assumption of databases, the TGDs are not supposed to be necessarily satisfied by the database D. Almost all the recent approaches to this problem focus on *conjunctive query answering* under TGDs, i.e., the problem of answering a conjunctive query over a database with TGDs. We recall that many other problems involving TGDs can be reduced in a straightforward way to this problem, e.g., implication of TGDs, conjunctive query containment under TGDs, etc.

In this paper we are interested in *first-order rewritable* TGDs, i.e., classes of TGDs for which conjunctive query answering can be reduced to the evaluation of a first-order query over the database. We believe that identifying expressive, yet first-order rewritable classes of TGDs is very important not only from the theoretical viewpoint but also from a practical one: indeed, restricting to such classes of TGDs allows in principle for building efficient query answering systems that delegate data management to standard relational database technology, in a way analogous to recent successful systems for ontology-based data access (e.g., [11]).

Several first-order rewritable classes of TGDs have been identified in the last years, in particular: linear TGDs [9,4], multi-linear TGDs [5], sticky TGDs [6], sticky-join

TGDs [7]. Multi-linear TGDs generalize linear TGDs, and sticky-join TGDs generalize sticky TGDs. However, multi-linear TGDs and sticky-join TGDs are incomparable classes of TGDs. Moreover, *acyclic TGDs*, which are trivially first-order rewritable, are not encompassed by the two above classes.

Our aim is to identify a broader class of TGDs that comprises all known FOL-rewritable classes of TGDs, and in particular acyclic TGDs, multi-linear TGDs, and sticky-join TGDs. In this paper, we reach the above goal under the restriction that TGDs do not allow for occurrences of constants and repeated occurrences of a variable in the same atom: we call *simple TGDs* the TGDs satisfying the above restriction.

More precisely, the contributions of the present paper are the following:

1. We define the class of *weakly recursive* TGDs (Section 3).
2. We prove that weakly recursive TGDs are first-order rewritable, by defining an algorithm that is able to compute the first-order rewriting of conjunctive queries over weakly recursive TGDs (Section 4) and proving termination of this algorithm over weakly recursive TGDs (Section 5).
3. We prove that, under the restriction to simple TGDs, the class of weakly recursive TGDs comprises and generalizes every previously known FOL-rewritable class of TGDs, in particular, acyclic TGDs, linear TGDs, multi-linear TGDs, sticky TGDs, sticky-join TGDs (Section 6).

2 Preliminaries: TGDs and Queries

Syntax We start from three pairwise disjoint alphabets: (i) a relational schema $\mathcal{R}$, where, as usual, every relation symbol r is associated with an arity, denoted by $\mathit{Arity}(r)$, which is a positive integer; (ii) a countably infinite domain of constants; (iii) a countably infinite domain of variables. An atom is an expression of the form $r(t_1, \ldots, t_k)$ where $k = \mathit{Arity}(r)$ and where every t_i is either a constant symbol or a variable symbol. A database is a (possibly infinite) set of ground atoms. Given an atom γ, we denote by $\mathit{Rel}(\gamma)$ the relation symbol occurring in γ.

A *tuple-generating dependency (TGD)* R is an expression of the form $\beta_1, \ldots, \beta_n \rightarrow \alpha_1, \ldots, \alpha_m$, where $\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_n$ are atoms and $m \geq 1$, $n \geq 1$ (we omit the existential quantification in the head of the rule). We call the expression $\alpha_1, \ldots, \alpha_m$ the *head* of R and call the expression $\beta_1, \ldots, \beta_n$ the *body* of R. Given a set P of TGDs, the *signature of* P is the set of relation symbols occurring in P.

We call *distinguished variables of* R the variables occurring both in the head and in the body of R. We call *existential body variables* of R the variables that occur only in the body of R (we call such variables *join variables* of R if they occur in at least two atoms of the body of R), and call *existential head variables* of R the variables that occur only in the head of R.

In the following, we call *FOL query* any domain-independent first-order query. A *conjunctive query (CQ)* is an existentially quantified conjunction of positive atoms (possibly with free variables): in this paper, we use a datalog notation for CQs, i.e., a CQ q is an expression of the form $q(\mathbf{x})$:- $\alpha_1, \ldots, \alpha_n$, where $\alpha_1, \ldots, \alpha_n$ is a sequence of atoms, called the *body* of q, the variables $\mathbf{x}$ are the *distinguished variables* of q and every variable of $\mathbf{x}$ occurs at least once in the body of q; the non-distinguished variables occurring in the body of q are called *existential variables* of q. The number of variables of $\mathbf{x}$ is called the *arity of* q. A *union of conjunctive queries (UCQ)* is a set of CQs of the same arity.

Semantics. We give the semantics of TGDs through first-order logic. We assume the reader is familiar with the notion of first-order interpretation and evaluation of a first-order formula over an interpretation. We adopt the Unique Name Assumption, i.e., different constant symbols are interpreted as different domain elements in every interpretation.

Given a TGD R of the form $\beta_1, \ldots, \beta_n \rightarrow \alpha_1, \ldots, \alpha_m$ and a database B, we say that B *satisfies* R if the first-order interpretation $\mathcal{I}^B$ (i.e., the first-order interpretation isomorphic to B) satisfies the first-order sentence $\forall \boldsymbol{x}.\beta_1 \wedge \ldots \wedge \beta_n \rightarrow \exists \boldsymbol{y}.\alpha_1 \wedge \ldots \wedge \alpha_m$, where $\boldsymbol{x}$ denotes all the variables occurring in the body of R and $\boldsymbol{y}$ denotes the existential head variables of R. Given a set P of TGDs and a database D over the signature of P, we say that a database B over the signature of P *satisfies* (P, D) if $B \supseteq D$ and B satisfies every TGD in P. Moreover, we denote by $\mathit{sem}(P, D)$ the set of all databases B over the signature of P such that B satisfies (P, D).

Let q be a FOL query and let B be a database. We denote by $\mathit{ans}(q, B)$ the set of tuples of constants $\mathbf{c}$ such that $\mathcal{I}^B$ satisfies $q(\mathbf{c})$, where $q(\mathbf{c})$ is the first-order sentence obtained from q by replacing its free variables with the constants $\mathbf{c}$.

Let P be a set of TGDs, let q be a UCQ and let D be a database. We define the *certain answers* to q over P and D, denoted by $\mathit{cert}(q, P, D)$, as the set of tuples of costants $\mathbf{c}$ such that $\mathbf{c} \in \bigcap_{B \in \mathit{sem}(P,D)} \mathit{ans}(q, B)$.

Finally, we introduce the notion of first-order (FOL) rewritable set of TGDs. Let P be a set of TGDs. We say that P is *FOL-rewritable* if, for every UCQ q, there exists a FOL query q' such that, for every database D, $\mathit{cert}(q, P, D) = \mathit{ans}(q', D)$.

In this paper, we restrict our attention to *simple TGDs*, i.e., TGDs of the above form in which every atom does not contain occurrences of constants and does not contain repeated occurrences of variables.

3 Weakly Recursive Simple TGDs

In this section we define weakly recursive (WR) simple TGDs.

A *position* σ is either an expression of the form $r[i]$ or an expression of the form $r[\,]$, where r is a relation symbol (and is denoted by $\mathit{Rel}(\sigma)$), and i is an integer such that $1 \leq i \leq k$, where k is the arity of r.

Let R be a TGD of the form $\beta_1, \ldots, \beta_n \rightarrow \alpha_1, \ldots, \alpha_m$, and let α be an atom of $\mathit{head}(R)$. Then: (i) given a position $r[\,]$, we say that α *is R-compatible with* $r[\,]$ if $\mathit{Rel}(\alpha) = r$; (ii) given a position $r[i]$, we say that α *is R-compatible with* $r[i]$ if $\mathit{Rel}(\alpha) = r$ and $\alpha[i]$ is a distinguished variable of R. Then, given an atom β of the body and a variable x occurring in β in position i, we denote by $\mathit{Pos}(x, \beta)$ the position $r[i]$.

Based on the above notions, we are now ready to define the position graph.

Definition 1. ***(Position Graph)*** *Given a set P of TGDs, the* position graph of P, *denoted by $PG(P)$, is a triple $\langle V, E, L \rangle$ where V (the set of nodes of $PG(P)$) is a set* *of positions, E is a set of edges (pairs of nodes), and L is an edge labeling function $L : E \to 2^{\{m,s\}}$. $PG(P)$ is inductively defined as follows:*

- *for every TGD $R \in P$ of the form $\beta_1, \ldots, \beta_n \to \alpha_1, \ldots, \alpha_m$ and for every i such that $1 \le i \le m$, $r[\,] \in V$ where $r = Rel(\alpha_i)$;*
- *if $\sigma \in V$, then for every TGD $R \in P$ and for every atom α of $head(R)$ that is R-compatible with σ:*
  1. *for every atom $\beta \in body(R)$:*
     - *(a) $\langle \sigma, s[\,] \rangle \in E$, where $s = Rel(\beta)$;*
     - *(b) for each existential body variable z of R occurring in β, $\langle \sigma, \sigma' \rangle \in E$, where $\sigma' = Pos(z, \beta)$;*
     - *(c) if σ is of the form $r[i]$, then $\langle \sigma, \sigma'' \rangle \in E$, where $\sigma'' = Pos(y, \beta)$ and y is the variable occurring in α at position i;*
     - *(d) if there exists a distinguished variable of R that does not occur in β, then, for every edge e added to E at points (a), (b), (c), $m \in L(e)$;*
  2. *if there exists an existential body variable x of R occurring in at least two atoms of $body(R)$, then for every edge e added to E at point 1, $s \in L(e)$;*
  3. *if σ is of the form $r[i]$, y is the variable occurring in α at position i, and y occurs in at least two atoms of $body(R)$, then, for every edge e added to E at point 1, $s \in L(e)$.*

We call m-*edge* is an edge e such that $m \in L(e)$, and call s-*edge* an edge e such that $s \in L(e)$. We are now ready to define weakly recursive sets of TGDs.

Definition 2. ***(Weakly Recursive TGDs)*** *A set of simple TGDs P is* weakly recursive (WR) *if in $PG(P)$ there exists no cycle that contains both an m-edge and an s-edge.*

Example 1. Let P be the following set of TGDs:

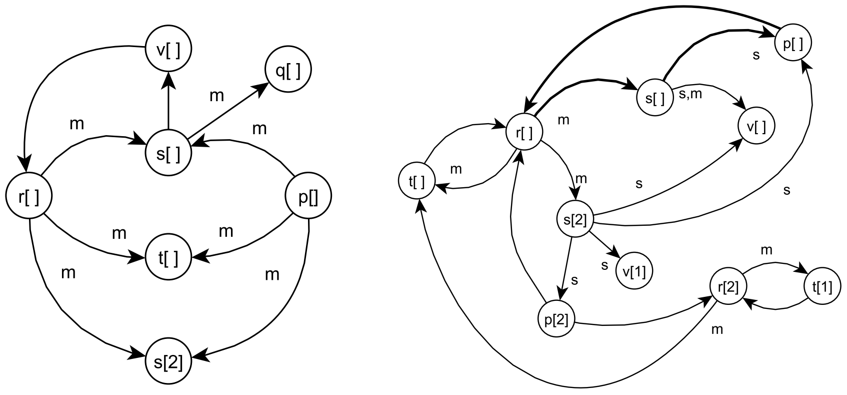
$$\begin{aligned} R_1 &= s(x,w,y), t(z) \to r(x,y), p(y,z) \\ R_2 &= v(x,y), q(y) \to s(x,z,y) \\ R_3 &= r(x,y) \to v(x,y) \end{aligned}$$

The position graph $PG(P)$ for P is displayed in Figure 1(a). Since there are no s-edges in $PG(P)$, it immediately follows that P is a weakly recursive set of TGDs. □

Example 2. Let P be the following set of TGDs:

$$\begin{aligned} R_1 &= s(x,w), t(y,z) \to r(x,y,z) \\ R_2 &= p(x,y), v(y) \to s(x,y) \\ R_3 &= r(x,y,z) \to p(x,y), t(y,z) \end{aligned}$$

The position graph $PG(P)$ for P is displayed in Figure 1(b). This graph contains a cycle (highlighted by bold edges in the figure) that contains both an m-edge and an s-edge. Consequently, P is not a weakly recursive set of TGDs. □



(a) The position graph of Example 1. (b) The position graph of Example 2.

Fig. 1. Postion graphs of Example 1 and Example 2

Essentially, the position graph of a set of TGDs P reflects the dependencies between relations induced by the TGDs: there is an edge from position $r[]$ to position $s[]$ if r occurs in the head of a TGD R and s occurs in the body of R. In addition, the position graph marks its edges with the symbols s and e. An s-edge represents the introduction of an existential join variable: e.g., if P contains the TGD $s(x, y), t(y) \rightarrow r(x)$, there is an s-edge from position $r[]$ to position $s[]$ (and to position $t[]$ too). Besides the above "direct" (i.e., due to a single TGD) introduction of existential joins, the position graph also considers an indirect introduction of existential join variables, through the propagation of (non-join) existential variables through the nodes labeled by positions of the form $r[i]$. For instance, in the above Example 2, variable w in R_1 is a (non-join) existential body variable: for this reason, the node with position $s[2]$ is added to the graph, and since in R_2 the distinguished variable y (which occurs at position $s[2]$) occurs in two atoms in the body of R_2, the outcoming edges of $s[2]$ are marked as s-edges. That is, position $s[2]$ is *split* over two atoms by the TGD R_2 (this is why we call such edges s-edges). Finally, m-edges represent those situations in which a body atom of the TGD *misses* distinguished variables of the TGD: e.g., in the TGD R_1 of Example 1, the body atom $s(x, w, y)$ does not contain any occurence of the distinguished variable z: hence, in the position graph of that example, the edge from $p[]$ to $s[]$ is marked as an m-edge.

Then, in the definition of weakly recursive TGDs, we identify "dangerous" cycles (for the FOL-rewritability of queries) as those cycles of the position graph which contain both an m-edge and an s-edge: as we will show later on, in the absence of one of the two kinds of edges, a cycle in the position graph is not really able to affect the FOL-rewritability property.

Finally, it can be easily verified that the problem of establishing whether a set of simple TGDs is weakly recursive is in PTIME.

4 Query Rewriting for Weakly Recursive TGDs

In this section we present an algorithm that, given a UCQ Q and a set P of TGDs, computes a FOL perfect rewriting of Q under P.

The only purpose of this algorithm is to show FOL-rewritability of weakly recursive TGDs. So, the algorithm is not optimized for minimizing the size of the query generated. The structure of the algorithm is very similar to known query rewriting techniques for TGDs (in particular, [13,12]): the only novel and distinguishing feature of the algorithm is the fact that it guarantees a fundamental property (Lemma 2) that in turn implies termination of the execution of the algorithm over weakly recursive TGDs, as we shall see in Section 5. Such a property is guaranteed by a restricted application of the reduce step (described below).

The algorithm assumes that TGDs are *single-head TGDs*, i.e., TGDs of the form $\beta_1, \ldots, \beta_n \rightarrow \alpha$, in which only one atom occurs in the head of every TGD. This is not a restriction, as shown in [3]: in fact, it is always possible to translate a set of TGDs into a set of single-head TGDs by a polynomial transformation Φ of the TGDs, such that the following property holds.

Lemma 1. *For every set P of TGDs, for every FOL query q over the signature of P and for every database D over the signature of P, $\mathit{cert}(q, P, D) = \mathit{cert}(q, \Phi(P), D)$. Moreover, P is weakly recursive iff $\Phi(P)$ is weakly recursive.*

Thus, in the rest of this section we assume that TGDs are simple TGDs of the form $\beta_1, \ldots, \beta_n \rightarrow \alpha$. We are now ready to present the algorithm *WRQueryRewrite*.

```
Algorithm WRQueryRewrite
Input: set of simple, single-head TGDs $P$, UCQ $Q$
Output: UCQ $Q'$
  $P' := P$;
  $Q' := \mathit{Normalize}(Q)$;
  $Q_{new} := Q'$;
  repeat
    $Q_{aux} := Q'$;
    $Q_P := Q_{new}$;
    $Q_{new} := \emptyset$;
    for each $q \in Q_P$ do
      for each $g1, g2 \in \mathit{body}(q)$ do
        if $g1$ and $g2$ unify and belong to the same join-connected component of $q$
          then $Q_{new} := Q_{new} \cup \{\mathit{reduce}(q, g1, g2)\}$;
      for each $g \in \mathit{body}(q)$ do
        for each $R \in P'$ do
          if $R$ is applicable to $g$
            then $Q_{new} := Q_{new} \cup \{\mathit{atom\text{-}rewrite}(q, g, R)\}$;
    $Q_{new} := \mathit{Normalize}(Q_{new})$;
    for each $q \in Q_{new}$ do
      if $q$ is not equal to any query of $Q'$ up to variable renaming
        then $Q' := Q' \cup \{q\}$;
  until $Q' = Q_{aux}$;
  return $Q'$
```

Due to space limitations, here we provide a brief description of the algorithm, and refer the reader to [9,13,12] for more details. Given a UCQ Q and a set P of TGDs, the algorithm tries to expand Q, generating new conjunctive queries. This is done using three different auxiliary sets of queries: Q_{new}, Q_{aux} and Q_P, where Q_{new} is the set of new CQs obtained in each iteration, Q_{aux} is used to check whether the set of CQs has changed during the iteration, and Q_P is the set of CQs that must be processed in each iteration. The three main steps of the algorithm are the following:

Step 1: **Query Normalization** - For each query $q \in Q_{new}$, the algorithm scans all the body atoms of q and checks if they are redundant, i.e., for each pair of atoms $a_1, a_2 \in body(q)$, we say that a_2 is redundant in q if the following condition hold: (i) $Rel(a_1) = Rel(a_2)$; (ii) there exists a renaming μ of the existential variables of q occurring only in a_2 such that every argument of $\mu(a_2)$ that is not a non-join existential variable is equal to the corresponding argument of a_1. In this case, the atom a_2 can be removed from the query q without changing its semantics. This step is done at the beginning and is repeated on the new CQs generated at every iteration. Given a UCQ Q, we denote by *Normalize*(Q) the UCQ obtained by applying the above normalization step to Q.

Step 2: **Reduction** - For each normalized query $q' \in Q_P$, the algorithm scans all the body atoms of q' and, for each pair of atoms $a_1, a_2 \in body(q')$, if they are unifiable, applies their MGU to q. The resulting query, q'', is added to the set of new queries Q_{new}. Differently from previous techniques, the algorithm does not apply the above reduction to every pair of unifiable atoms, but only considers pairs of atoms that belong to the same join-connected component of the CQ (it can be shown that such a restriction preserves completeness of the algorithm). A *join-connected component* of a CQ q is a set of atoms of the body of q connected by existential join variables. E.g., the CQ $q(x, y) \mathrel{:-} s(x, z, w), t(w, y, z'), u(y, v), r(v, v', a)$ has two join-connected components: $s(x, z, w), t(w, y, z')$ and $u(y, v), r(v, v', a)$.

Step 3: **Atom Rewrite** - Let R be a TGD and let q be a CQ. Let g be an atom appearing in $body(q)$. We say that R *is applicable to* g if and only if: (i) there exists a *most general unifier (MGU)* τ for $head(R)$ and g (i.e., there exists a substitution of the variables that makes $head(R)$ and g equal); (ii) each existential head variable of R matches one non-join existential variable of q. Let R be a TGD and let q be a CQ. Let g be a goal atom appearing in $body(q)$. If R is applicable to g, then we denote by *atom-rewrite*(q, g, R) the query q' obtained from q by first replacing g with $body(R)$, and then applying to the query the MGU τ for $head(R)$ and g. Now, for each normalized query $q' \in Q_P$, the algorithm scans all the body atoms $g \in body(q')$ and, for each TGD $R \in P$, if R is applicable to g, adds the query *atom-rewrite*(q, g, R) to the set of new queries Q_{new}. (This is the step that may be responsible for the non-termination of the algorithm.)

At the end of each iteration, the algorithm adds to Q' all the queries in Q_{new} that are not already in Q' (up to variable renaming) and checks whether Q' has changed, i.e., whether it is equal to Q_{aux}. If so, the algorithm goes on, starting a new iteration, replacing the set Q_P with Q_{new}, emptying Q_{new} and repeating the two steps; otherwise it stops because a fixpoint has been reached.

The following theorem follows from correctness of the algorithm presented in [12] and from the fact that the modifications of the present algorithm preserve completeness.

Theorem 1. *Let P be a set of single-head TGDs and let Q be a UCQ such that WRQueryRewrite(Q, P) terminates. Let Q' be the UCQ returned by WRQueryRewrite(Q, P). Then, Q' is a perfect rewriting of Q with respect to P.*

5 FOL-Rewritability of Weakly Recursive TGDs

In this section we prove that the class of weakly recursive TGDs is FOL-rewritable. In particular, we prove that the algorithm *WRQueryRewrite*(q, P) always terminates if P is a weakly recursive set of single-head TGDs.

We start by stating a fundamental property of the algorithm *WRQueryRewrite*. From now on, we call *ebj-variable* an existential join variable of q that occurs in at least two atoms of the body of q.

Lemma 2. *Let P be a set of simple TGDs and let Q be a UCQ. If WRQueryRewrite(Q, P) does not terminate, then for every integer n there exists a CQ q returned by WRQueryRewrite(Q, P) such that q has at least one join-connected component with more than n ebj-variables.*

In the rest of this section, we prove that, if P is a weakly recursive set of TGDs, then for every UCQ Q the number of ebj-variables in every join-connected component of a CQ returned by *WRQueryRewrite*(Q, P) is actually bounded, and therefore, by the above lemma, the algorithm *WRQueryRewrite*(Q, P) terminates. This in turn implies, by Theorem 1, that weakly recursive TGDs are a FOL-rewritable class of TGDs.

To show that, we need to define the preliminary notions of CQ-tree (a tree whose nodes are CQs and that represents an execution of the algorithm *WRQueryRewrite*), CQ-path (a path in the CQ-tree), atom-path (a path in the CQ-tree that connects single query atoms), and extended path (an extended notion of path in the graph $PG(P)$).

A *rewriting action* ξ is: either (i) the execution of an atom-rewrite step, and in this case ξ has the form *atom-rewrite*(q, α, R), where q is a CQ, α is an atom of the body of q, and R is a TGD of P; or (ii) the execution of a reduce step, and in this case ξ has the form *reduce*(q, α_1, α_2) where q is a CQ and α_1, α_2 are atoms of the body of q.

Given a CQ q, an atom $\alpha \in body(q)$, and a CQ q' obtained from q by an atom-rewrite action ξ that applies the TGD $\beta_1, \ldots, \beta_k \rightarrow \gamma$ and produces the MGU μ, we define $succ(\alpha, \xi, q)$ as follows: (i) if α is not the atom to which the atom-rewrite action ξ is applied, then $succ(\alpha, \xi, q) = \{\mu(\alpha)\}$; (ii) if α is the atom to which the atom-rewrite action ξ is applied, then $succ(\alpha, \xi, q) = \{\mu(\beta_1), \ldots, \mu(\beta_k)\}$.

Given a CQ q, an atom $\alpha \in body(q)$, and a CQ q' obtained from q by a reduce action ξ that unifies the atoms α_1, α_2 of q producing the atom β and the MGU μ, we define $succ(\alpha, \xi, q)$ as follows: (i) if $\alpha \neq \alpha_i$ for every i such that $1 \leq i \leq 2$, then $succ(\alpha, \xi, q) = \{\mu(\alpha)\}$; (ii) if $\alpha = \alpha_i$ for some i such that $1 \leq i \leq 2$, then $succ(\alpha, \xi, q) = \{\beta\}$. Intuitively, the relation $succ(\alpha, \xi, q)$ associates every body atom α of a CQ q with the transformation of such an atom obtained through the rewriting action ξ.

Definition 3. *Let P be a set of TGDs and let q be a CQ. We define the* CQ-tree of q and P*, denoted by CQ-tree(q, P), as the tree T, whose nodes are labeled with sets of atoms and whose edges are labeled with action-labels and atom-labels, such that:*

- *$body(q)$ is the label of the root of T;*
- *there is an edge e from a node n_1 (with label q_1) to a node n_2 (with label q_2) in T if and only if the CQ q_2 appears in the UCQ Q' returned by the algorithm*

WRQueryRewrite(q, P)*, and* q_2 *has been added by the algorithm to* Q' *through a rewriting action* ξ *applied to the CQ* q_1*;*
- *the action-label of the above edge* e *is* ξ*;*
- *the atom-label of the above edge* e *is the set of pairs* $\bigcup_{\alpha \in q_1} \bigcup_{\beta \in succ(\alpha,\xi)} \{\langle \alpha, \beta \rangle\}$.

Given a CQ-tree T, a *CQ-path of* T is a sequence $q_1, \xi_1, q_2, \xi_2, \dots, q_{m-1}, \xi_{m-1}, q_m$ such that: (1) every q_i is a CQ and every ξ_i is a rewriting action of the algorithm; (2) there exists a sequence of nodes $n_1, \dots, n_m$ such that: (i) for every $i \in \{1, \dots, m\}$, q_i is the label of n_i in T; (ii) for every $i \in \{1, \dots, m-1\}$, there is an edge from n_i to n_{i+1} in T with action-label ξ_i.

Given a CQ-tree T, an *atom-path of* T is a sequence $\alpha_1, \psi_1, \alpha_2, \dots, \alpha_{n-1}, \psi_{n-1}, \alpha_n$ such that: (1) every α_i is an atom and every ψ_i is either a rewriting action of the algorithm or the symbol *ident*; (2) there exists a CQ-path $q_1, \xi_1, q_2, \dots, q_{n-1}, \xi_{n-1}, q_n$ in T such that: (i) for every i such that $1 \leq n$, $\alpha_i \in body(q_i)$; (ii) for every i such that $1 \leq n-1$, $\alpha_{i+1} \in succ(\alpha_i, \xi_i, q_i)$; (iii) for every i such that $1 \leq n-1$, $\psi_i = \xi_i$ if either ξ has the form *atom-rewrite*(q', α_i, R) or ξ_i has the form *reduce*(q', β_1, β_2) and $\alpha_i \in \{\beta_1, \beta_2\}$, otherwise $\psi_i =$ *ident*.

We call *extended path* of $PG(P)$ a sequence of the form $\pi' = \sigma_1, \sigma_2, \dots, \sigma_n$ such that every σ_i is a position, σ_1 is a node of $PG(P)$ and, for every i such that $2 \leq i \leq n$, either $\langle \sigma_{i-1}, \sigma_i \rangle \in E$ or $\sigma_i = \sigma_{i-1}$.

Let $\alpha_1, \psi_1, \alpha_2, \dots, \alpha_{n-1}, \psi_{n-1}, \alpha_n$ be an atom-path in *CQ-tree*(q, T), and let $\pi' = \sigma_1, \sigma_2, \dots, \sigma_k$ be an extended path in $PG(P)$. We say that π' *maps to* π if, for every i such that $1 \leq i \leq k$, *Rel*$(\sigma_i) =$ *Rel*(α_i) and, for every i such that $1 \leq i \leq k-1$, if either $\psi_i =$ *ident* or ψ_i is a reduce action then $\sigma_{i+1} = \sigma_i$.

Let ξ be an atom-rewrite action of the form *atom-rewrite*(q, α, R) such that ξ returns a CQ that contains an ebj-variable y that is not an ebj-variable in q. Then, we call ξ an *ebj-generating* action. Observe that such atom-rewrite actions are the only ones that introduce new ebj-variables in the CQ (since reduce actions can never introduce new ebj-variables).

Lemma 3. *Let* P *be a set of TGDs and let* q *be a CQ. If* $\pi = \alpha_1, \psi_1, \alpha_2, \psi_2, \dots, \alpha_n$ *is an atom-path in CQ-tree*(q, P)*, then there exists an extended path* $\pi' = \sigma_1, \sigma_2, \dots, \sigma_n$ *in* $PG(P)$ *such that: (i)* π' *maps to* π*; (ii) for every* i *such that* $1 \leq i \leq n-1$ *if* ψ_i *is an ebj-generating action, then* $\langle \sigma_i, \sigma_{i+1} \rangle$ *is an s-edge of* $PG(P)$*.*

Theorem 2. *Let* P *be a set of TGDs and let* Q *be a UCQ. If the execution of WRQueryRewrite*(Q, P) *does not terminate, then there exists a cycle in* $PG(P)$ *containing both an* m*-edge and an* s*-edge.*

Proof (sketch). Suppose that the execution of *WRQueryRewrite*(Q, P) does not terminate. From Lemma 2 it follows that, for every integer n, there exists a CQ-path π^n_{cq}, ending with a CQ q^n, that contains at least n atom-rewrite actions that introduce at least a new ebj-variable in the same join-connected component of q^n.

Then, it is easy to see that the existence of the above CQ-paths π^n_{cq} implies that, for every integer n, there exists at least one atom-path $\pi = q_1, \xi_1, q_2, \xi_2, \dots, q_m$ of *CQ-tree*(q, P) containing n atom-rewrite actions introducing at least one new ebj-variable in the CQ q_m. Let us then assume that such an n is sufficiently large, in particular, n is greater than the number of edges in $PG(P)$. But then, from Lemma 3 it follows that there exists an extended path $\pi' = \sigma_1, \ldots, \sigma_m$ in $PG(P)$ such that π' maps to π and, for every i such that $1 \leq i \leq m-1$, if ξ_i is an atom-rewrite action introducing a new ebj-variable in the CQ, then $\langle \sigma_i, \sigma_{i+1} \rangle$ is an s-edge of $PG(P)$. Now, consider the path π'' of $PG(P)$ obtained from π' by deleting all the *ident* edges: then, π'' is a path of $PG(P)$ such that π'' has at least n s-edges. But since n is greater than the number of edges in $PG(P)$, it follows that there exists a cycle containing an s-edge in $PG(P)$. Moreover, since n is arbitrarily large, we can assume w.l.o.g. that the above cycle is repeated more than k times in π'', where k is the maximum arity k of the relations occurring in P (indeed, if in *CQ-tree*(q, P) there were no atom-path π from which such a path π'' can be derived, we could immediately conclude that every join-connected component in every CQ generated by the algorithm has a bounded number of ebj-variables, thus contradicting Lemma 2).

Moreover, it can be shown that, if the above cycle in $PG(P)$ does not contain m-edges, then the following property (*) holds: if y_i is a new ebj-variable introduced (at the i-th iteration of the above cycle) by the atom-rewrite action corresponding to the i-th application of a TGD R (through the atom-rewrite step) such that y_i occurs in a join-connected component of q_m (the final query of the atom-path), then *every* atom in the above join-connected component contains at least one new ebj-variable introduced in q at the i-th application of R.

As a consequence of Property (*), the number of applications of the above TGD R is bounded by the maximum arity k of the relations occurring in P, thus contradicting the above hypothesis that the above cycle of $PG(P)$ is repeated more than k times in π''. Therefore, the above cycle must also contain at least one m-edge, which implies the thesis. □

As a corollary of Theorem 2, we get that the execution of *WRQueryRewrite*(Q, P) always terminates over weakly recursive TGDs. Therefore, from the above property, Theorem 1, and Lemma 1, we obtain the following theorem.

Theorem 3. *Every weakly recursive set of TGDs is FOL-rewritable.*

6 Comparison

In this section we recall the definitions of some important classes of TGDs and we show that the weakly recursive class comprises and generalizes every previously known FOL-rewritable class of TGDs.

A TGD is *linear* [4] iff it contains only a singleton body atom (i.e., the TGD is of the form $\beta \rightarrow \alpha_1, \ldots, \alpha_n$). As already noted in [4], the class of linear TGDs generalizes all the DLs belonging to the DL-Lite family [10]. Moreover, linear TGDs are a particular instance of the more general class of multi-linear TGDs. A TGD R is *multi-linear* [5] if every variable that occurs in $body(R)$ occurs in every atom of $body(R)$.

Theorem 4. *The class of weakly recursive simple TGDs strictly contains the class of multi-linear simple TGDs.*

Proof. From the definition of multi-linear TGDs, it immediately follows that, if P is a set of multi-linear TGDs, then $PG(P)$ does not contain any m-edge. Therefore, P is also weakly recursive. To prove that the containment is strict, we refer to Example 3. □

Another relevant FOL-rewritable class of TGDs, which is incomparable with the class of multi-linear TGDs, is the class of *sticky* TGDs, presented in [6]. Moreover, *sticky-join* TGDs [7] are a further generalization of the class of sticky TGDs, which encompasses the class of linear TGDs but is incomparable with the class of multi-linear TGDs. For the definition of sticky-join TGDs we refer to [7].

Theorem 5. *The class of weakly recursive simple TGDs strictly contains the class of sticky-join simple TGDs.*

Proof (sketch). It can be easily verified that, if a set of simple TGDs P is sticky-join, then the graph $PG(P)$ does not contain any s-edge, thus P is weakly recursive as well. To prove that the containment is strict, we refer to Example 3. □

Example 3. The following example presents a set P of TGDs that is weakly recursive but neither sticky-join, nor multi-linear. Let P be the following set of TGDs:

$$R_1 = s(x, y, z, \underline{v}) \to r(x, y, z)$$
$$R_2 = t(x, \underline{w}), r(x, \underline{w}, y) \to s(x, y, z, w)$$

As shown in Figure 2, the position graph $PG(P)$ does not contain any cycle with both m-edges and s-edges, therefore P is weakly recursive.

On the other hand, it is easy to verify that P is not multi-linear, since in R_2 the atom $t(x, w)$ does not contain all the distinguished variables of the head (in particular, x is missing). Moreover, according to the sticky-join marking procedure on the expanded set of TGDs $P^\star$ (in this case $P = P^\star$) described in [7], the variable v is first marked in R_1, thus the variable w is also marked in R_2 and, since w occurs in two distinct atoms, P is not sticky-join. □

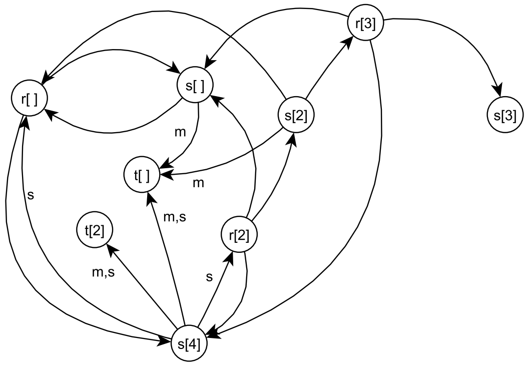


Fig. 2. The position graph of Example 3

Finally, we focus on *acyclic TGDs*. A set P of TGDs is acyclic if its position graph $PG(P)$ is acyclic. Of course, acyclic TGDs are trivially FOL-rewritable, since every known query rewriting algorithm for TGDs terminates over such TGDs. However, neither the class of multi-linear TGDs nor the class of sticky-join TGDs encompasses the class of acyclic TGDs. On the other hand, every set of acyclic TGDs is obviously also weakly recursive.

7 Conclusions

This work is a further step towards the definition of a maximal class of TGDs supporting efficient query answering, but there are many challenging issues that still need to be tackled. First, we are working towards extending the results presented in this paper to non-simple TGDs, i.e., TGDs with repeated variables within the same atom and with constants. Actually, we have already identified a generalization of the present notion of weakly recursive TGDs to this more general case. Then, we would like to generalize the class of weakly recursive TGDs to other decidable, and possibly tractable, classes. In particular, we believe that some of our results can pave the way towards the identification of new classes of TGDs for which query answering is in PTIME with respect to data complexity. Finally, the *WRQueryRewrite* algorithm presented in this paper had the only purpose of showing that weakly recursive TGDs are FOL-rewritable. An important issue towards the usage of weakly recursive TGDs in real applications is to define optimizations of such a query rewriting algorithm.

Acknowledgments. This research has been partially supported by the ICT Collaborative Project ACSI (Artifact-Centric Service Interoperation), funded by the EU under FP7 ICT Call 5, 2009.1.2, grant agreement n. FP7-257593.

References

1. Baget, J.-F., Leclère, M., Mugnier, M.-L., Salvat, E.: On rules with existential variables: Walking the decidability line. Artificial Intelligence 175(9-10), 1620–1654 (2011)
2. Baget, J.-F., Mugnier, M.-L., Rudolph, S., Thomazo, M.: Walking the complexity lines for generalized guarded existential rules. In: Proc. of IJCAI 2011, pp. 712–717 (2011)
3. Calì, A., Gottlob, G., Kifer, M.: Taming the infinite chase: Query answering under expressive relational constraints. In: Proc. of KR 2008 (2008)
4. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. In: Proc. of PODS 2009, pp. 77–86 (2009)
5. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. Technical Report CL-RR-10-21, Oxford University Computing Laboratory (2010)
6. Calì, A., Gottlob, G., Pieris, A.: Advanced processing for ontological queries. PVLDB 3(1), 554–565 (2010)
7. Calì, A., Gottlob, G., Pieris, A.: Query Answering under Non-guarded Rules in Datalog+/- . In: Hitzler, P., Lukasiewicz, T. (eds.) RR 2010. LNCS, vol. 6333, pp. 1–17. Springer, Heidelberg (2010)
8. Calì, A., Gottlob, G., Pieris, A.: New expressive languages for ontological query answering. In: Proc. of AAAI 2011 (2011)
9. Calì, A., Lembo, D., Rosati, R.: Query rewriting and answering under constraints in data integration systems. In: Proc. of IJCAI 2003, pp. 16–21 (2003)
10. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. J. of Automated Reasoning 39(3), 385–429 (2007)
11. Calvanese, D., Giacomo, G.D., Lembo, D., Lenzerini, M., Poggi, A., Rodriguez-Muro, M., Rosati, R., Ruzzi, M., Savo, D.F.: The mastro system for ontology-based data access. Semantic Web 2(1), 43–53 (2011)

12. Gottlob, G., Orsi, G., Pieris, A.: Ontological queries: Rewriting and optimization. In: Proc. of ICDE 2011, pp. 2–13 (2011)
13. Gottlob, G., Orsi, G., Pieris, A.: Ontological Query Answering via Rewriting. In: Eder, J., Bielikova, M., Tjoa, A.M. (eds.) ADBIS 2011. LNCS, vol. 6909, pp. 1–18. Springer, Heidelberg (2011)
14. Krötzsch, M., Rudolph, S.: Extending decidable existential rules by joining acyclicity and guardedness. In: Proc. of IJCAI 2011, pp. 963–968 (2011)
15. Leone, N., Manna, M., Terracina, G., Veltri, P.: Efficiently computable Datalog$^{\exists}$ programs. In: Proc. of KR 2012 (to appear, 2012)

Datalog Development Tools
(Extended Abstract)

Onofrio Febbraro[1], Giovanni Grasso[2], Nicola Leone[3],
Kristian Reale[3], and Francesco Ricca[3]

[1] DLVSystem s.r.l. - P.zza Vermicelli, Polo Tecnologico, 87036 Rende, Italy
febbraro@dlvsystem.com
[2] Oxford University, Department of Computer Science - , Oxford, UK
giovanni.grasso@cs.ox.ac.uk
[3] Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{leone,reale,ricca}@mat.unical.it

1 Introduction

The recent successful application of Datalog in a number of advanced projects, has renewed the interest in Datalog-based systems for developing real-world applications. Indeed, in the last few years Datalog has been successifully applied in many different areas of computer science, including: Artificial Intelligence, Data Extraction, Information Integration and Knowledge Management. Interestingly, besides the scientific applications, Datalog-based systems were also applied for developing some industrial systems. Nonetheless, in order to boost the adoption of Datalog-based technologies in the scientific community and especially in industry, it is important to provide effective programming tools, which support the activities of researchers and implementors and simplify users' interactions with Datalog systems. Indeed, development frameworks and tools provide indispensable means for assisting and simplifying application development. For this reason, the most popular programming languages and also commercial off-of-the-shelf software products (e.g., DBMSs) are always complemented by Integrated Development Environments (IDE) and Software Development Kits (SDK).

We have dealt with this issue, and we designed development tools for the Datalog-based system DLV [1]. The language of DLV is an extension of Datalog allowing disjunction in rule heads, aggregate atoms, and both weak and strong constraints in rule bodies [1]. DLV is widely considered a state-of-the-art implementation of Disjunctive Datalog under the stable models semantics [2,3] (also called Answer Set Programming (ASP) [4]), and it is undergoing an industrialization process [5] conducted by a spin-off company named DLVSYSTEM s.r.l..

The development tools described in this paper are born from our experience in developing Datalog real-world applications (see. e.g., [6,7]) with DLV. In particular, while implementing Datalog-based applications, we recognized two basic needs: (i) the availability of an IDE supporting Datalog program development (as it is customary for languages like C++ or Java); and (ii) the strong need of integrating Datalog programs and solvers in the well-assessed software-development processes and platforms, which are tailored for imperative/object-oriented programming languages. Indeed, complex business-logic features can be developed with Datalog-based technologies at a lower (implementation) price than in traditional imperative languages, and there are several additional advantages from a Software Engineering viewpoint, in flexibility, readability, extensibility, ease of maintenance, etc. However, since Datalog is not a full general-purpose language, logic programs *must be embedded*, at some point, in systems components that are usually built by employing imperative/object-oriented programming languages, e.g., for developing visual user-interfaces. To respond to the above mentioned needs, we have developed two tools:

1. *ASPIDE* [8]: a complete IDE for disjunctive Datalog programs, which integrates a cutting-edge *editing tool* (featuring dynamic syntax highlighting, on-line syntax correction, autocompletion, code-templates, quick-fixes, refactoring, etc.) with a collection of user-friendly *graphical tools* for program composition, debugging, profiling, database access, solver execution configuration and output-handling; and
2. *JDLV* [9]: a plug-in for the Eclipse platform that implements $\mathcal{JASP}$, a hybrid language that transparently supports a bilateral interaction between disjunctive Datalog and Java. The Datalog program can access Java variables, and the results of the evaluation are automatically stored in Java objects, possibly populating Java collections, transparently. A key ingredient of $\mathcal{JASP}$ is the mapping between (collections of) Java objects and Datalog facts, which can be customized by following widely-adopted standards for Object-Relational Mapping (ORM).

These tools speed-up and empower the development of Datalog-based solutions and their integration in the well-assessed development processes and platforms, which are tailored for imperative/object-oriented programming languages.

2 *ASPIDE*

ASPIDE [8] supports the entire life-cycle of the development of Datalog-based applications, from (assisted) programs editing to application deployment. In the following we overview the main features that make *ASPIDE* one of the most comprehensive development environment for logic programming.[1]

Workspace organization. The system allows for organizing logic programs in projects à la Eclipse, which are collected in a special directory (called workspace). *ASPIDE* allows to manage logic programs in DLV syntax [1] and `ASPCore` [10]; other file types can be added by providing input plugins (see below).

Advanced text editor. *ASPIDE* features an editor tailored for logic programs that offers, besides the basic functionalities also *text coloring*, *automatic completion*, and program *refactorings*.

Outline navigation. *ASPIDE* creates an outline view which graphically represents program elements. Each item in the outline can be used to quickly access the corresponding line of code (a very useful feature when dealing with long files).

Visual editor. The users can *draw* logic programs by exploiting a full graphical environment that offers a QBE-like tool for building logic rules. The user can switch from the text editor to the visual one (and vice-versa) thanks to a reverse-rengineering mechanism from text to graphical format.

[1] For an exhaustive comparison among the available tools for logic programming see [8].

Dependency graph. The system provides a graphical representation of several variants of the (non-ground) dependency graphs associated with the project.

Dynamic code checking and errors highlighting. Programs are parsed while writing, and both errors or possible warnings are immediately outlined.

Quick fixes and Code templates. The system suggests quick fixes to reported errors or warnings, and provides support for assisted writing of rule patterns (guessing patterns, etc.) by means of code templates that can be instantiated while writing.

Debugger and Profiler. *ASPIDE* embeds the debugging tool *spock* and the DLV Profiler [11].

Unit Testing. The testing feature consists on a unit testing framework for logic programs in the style of JUnit. For an exhaustive description the testing language and the graphical testing tool of *ASPIDE* we refer the reader to [12].

Annotation Management. *ASPIDE* supports annotations for indicating meta information of programs like rule names, predicate name, arity, etc. Meta-information given through annotations is exploited for auto-completion, test case composition, etc.

Schema Management and Interaction with Databases. *ASPIDE* simplifies access to external databases by a graphical tool connecting to DBMSs via JDBC. The database management of *ASPIDE* supports both DLV with ODBC interface and DLV^{DB} [13].

Configuration of the execution and Presentation of the Results. The RunConfiguration Dialog allows one to setup a DLV invocation; whereas the results are presented to the user in a comfortable tabular representation and they can be also saved in text files for subsequent analysis.

User-defined Plugins. *ASPIDE* can be extended with user defined plugins for handling: (i) new input formats, (ii) program rewritings, and even (iii) customizing the visualization/format of results. An input plugin can take care of input files that appear in *ASPIDE* as a logic program, and an output plugin can handle the external conversion of the computed results. A rewriting plugin may encode a procedure that can be applied to rules in the editor.

Availability. *ASPIDE* is available for all the major operating systems, including Linux, Mac OS and Windows, and can be downloaded from the system website `http://www.mat.unical.it/ricca/aspide`.

3 JDLV

We overview in the following a new programming framework blending logic programming with Java and its implementation as plug-in for the Eclipse platform [14], called *JDLV*. *JDLV* is based on $\mathcal{JASP}$ [9], a hybrid language that transparently supports a bilateral interaction between (disjunctive) Datalog and Java. A key ingredient of $\mathcal{JASP}$ is the mapping between (collections of) Java objects and ASP facts. $\mathcal{JASP}$ shares with Object-Relational Mapping (ORM) frameworks, such as Hibernate and TopLink, the structural issues of the *impedance mismatch* [15,16] problem. In $\mathcal{JASP}$, Java Objects are mapped to logic facts (and vice versa) by adopting a structural mapping strategy similar to the one employed by ORM tools for retrieving/saving persistent objects from/to relational databases. $\mathcal{JASP}$ supports both a default mapping strategy, which fits the most common programmers' requirements, and custom ORM specifications, which comply with the Java Persistence API (JPA) [17], to perfectly suit enterprise application development standards. In this section, we present $\mathcal{JASP}$ by exploiting an example. We refer the reader to [9] for a complete description of the language and its semantics.

Integrating Disjunctive Datalog with Java. The $\mathcal{JASP}$ code is very natural and intuitive for a programmer skilled in both Disjunctive Datalog and Java; we introduce it by exploiting an example program, in which a monolithic block of plain Datalog code (called *module*) is embedded in a Java class, which is executed "in-place", i.e., the solving process is triggered at the end of the module specification. Consider the following $\mathcal{JASP}$ code:

```
1 class GraphUtil {
   public static Set<Colored> compute3Coloring(Set<Arc> arcs,
3                                                Set<String> nodes ){
     Set<Colored> res = new HashSet<Colored>();
5    <# in=arcs::arc, nodes::node out=res::col
        col(X,red) v col(X,green) v col(X,blue) :- node(X).
7       :- col(X,C), col(Y,C), arc(X,Y).   #>
     if_no_answerset { res = null; }
9    return res; }}
```

GraphUtil defines the method compute3Coloring(), that encompass a $\mathcal{JASP}$-module to computes a 3-coloring of the given graph. The parameters arcs and nodes are mapped to corresponding predicates (Line 5) `arc` and `node`, respectively, whereas the local variable res is mapped as output variable to the predicate `col` (Line 5). Intuitively, when compute3Coloring() is invoked, Java objects are transformed into logic facts, by applying a default ORM strategy. In this example, each string x in nodes is transformed in unary facts `node(x)`; similarly, each instance of Arc in the variable arcs produces a binary fact, e.g., `arc(from,to)`. These facts are input of the logic program, which is evaluated "in-place". If no 3-coloring exists, the variable res is set to **null** (Line 8); else, when the first result (called *answer set* [4]) is computed, for each fact `col` contained in the solution a new object of the class Colored is created and added to res, which, in turn, is returned by the method. Syntactically, $\mathcal{JASP}$ directly extends the syntax of Java by few new keywords (e.g.,« <# »,« #> »), in such a way that $\mathcal{JASP}$ module statements are allowed in Java *block statements*. Concerning the syntax allowed within modules, $\mathcal{JASP}$ is compliant with the language of DLV. The $\mathcal{JASP}$'s default ORM strategy uses compound keys, i.e., keys made of all basic attributes, and *embedded values*, for one to one associations. This choice naturally fits the usual way of representing information in Datalog, e.g., in the example, one fact models one node. Such mapping can be inverted to obtain Java objects from logic facts. Although, this strategy poses (a few) restrictions to Java specifications (e.g., such as non-recursive type definition, bean-like structure), based on our experience, it is sufficient to handle common use cases.

In the example above we have employed the basic syntax of $\mathcal{JASP}$. The language supports also other advanced features conceived for easing the development of complex applications, including: syntactic enhancements (e.g., named non-positional notation), incremental programs (to enable building programs throughout the application), access to Java variables (for accessign the Java environment), database table mappings (to directly access data stored in a DBMS), and complex mappings with JPA [17] annotations (for complex ORM). Indeed, $\mathcal{JASP}$ spouses the work done in the field ORM and supports the standard JPA Java annotations for defining how Java classes map to relations (logic predicates).

Acknowledgments. This work has been partially supported by the Calabrian Region under PIA (Pacchetti Integrati di Agevolazione industria, artigianato e servizi) project DLVSYSTEM approved in BURC n. 20 parte III del 15/05/2009 - DR n. 7373 del 06/05/2009. Giovanni Grasso has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement DIADEM, no. 246858.

References

1. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL 7(3), 499–562 (2006)
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS 22(3), 364–418 (1997)
3. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC 9, 365–385 (1991)
4. Lifschitz, V.: Answer Set Planning. In: ICLP 1999, pp. 23–37 (1999)
5. Grasso, G., Leone, N., Manna, M., Ricca, F.: ASP at Work: Spin-off and Applications of the DLV System. In: Balduccini, M., Son, T.C. (eds.) Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning. LNCS, vol. 6565, pp. 432–451. Springer, Heidelberg (2011)
6. Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S., Leone, N.: Team-building with answer set programming in the gioia-tauro seaport. TPLP 12(3), 361–381 (2012)
7. Ricca, F., Dimasi, A., Grasso, G., Ielpa, S.M., Iiritano, S., Manna, M., Leone, N.: A Logic-Based System for e-Tourism. FI 105(1–2), 35–55 (2010)
8. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated Development Environment for Answer Set Programming. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 317–330. Springer, Heidelberg (2011)
9. Febbraro, O., Grasso, G., Leone, N., Ricca, F.: JASP: a framework for integrating Answer Set Programming with Java. In: Proc. of KR 2012. AAAI Press (2012)
10. Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M.C., Sirianni, M., Terracina, G., Veltri, P.: The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 388–403. Springer, Heidelberg (2011)
11. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A Visual Tracer for DLV. In: Proc. of SEA 2009, Potsdam, Germany (2009)
12. Febbraro, O., Leone, N., Reale, K., Ricca, F.: Unit testing in aspide. CoRR abs/1108.5434 (2011)
13. Terracina, G., Leone, N., Lio, V., Panetta, C.: Experimenting with recursive queries in database and logic programming systems. TPLP 8, 129–165 (2008)
14. Eclipse: Eclipse (2001), `http://www.eclipse.org/`
15. Maier, D.: Representing database programs as objects. In: Advances in Database Programming Languages, pp. 377–386. ACM Press (1990)
16. Keller, A.M., Jensen, R., Agrawal, S.: Persistence software: Bridging object-oriented programming and relational databases. In: Proc. of ACM SIGMOD 1993, pp. 523–528. ACM Press (1993)
17. Oracle: JSR 317: JavaTM Persistence 2.0 (2009), `http://jcp.org/en/jsr/detail?id=317`

Query Rewriting Using Datalog for Duplicate Resolution*

Jaffer Gardezi[1] and Leopoldo Bertossi[2]

[1] University of Ottawa, SITE,
Ottawa, Canada
[2] Carleton University, SCS,
Ottawa, Canada

Abstract. Matching Dependencies (MDs) are a recent proposal for declarative entity resolution. They are rules that specify, given the similarities satisfied by values in a database, what values should be considered duplicates, and have to be matched. On the basis of a chase-like procedure for MD enforcement, we can obtain clean (duplicate-free) instances; actually possibly several of them. The clean answers to queries (which we call the resolved answers) are invariant under the resulting class of instances. In this paper, we investigate a query rewriting approach to obtaining the resolved answers (for certain classes of queries and MDs). The rewritten queries are specified in stratified Datalog^{not,s} with aggregation. In addition to the rewriting algorithm, we discuss the semantics of the rewritten queries, and how they could be implemented by means of a DBMS.

1 Introduction

For various reasons, databases may contain different coexisting representations of the same external, real world entity. This can occur, for example, because of errors or because the data comes from different sources using different formats. Those "duplicates" can be entire tuples or values within them. To obtain accurate information, in particular, query answers from the data, those tuples or values should be merged into a single representation.

Identifying and merging duplicates is a process called *entity (or duplicate) resolution* (ER) [15, 9]. Matching dependencies (MDs) are a recent proposal for declarative duplicate resolution [20, 21]. An MD expresses, in the form of a rule, that if the values of certain attributes in a pair of tuples are similar, then the values of other attributes in those tuples should be matched (or merged) into a common value.

For example, the MD $R_1[X_1] \approx R_2[X_2] \rightarrow R_1[Y_1] \doteq R_2[Y_2]$ is a symbolic expression saying that, if an R_1-tuple and R_2-tuple have similar values for their attributes X_1, X_2, then their values for attributes Y_1, Y_2 should be made equal. This is a *dynamic* dependency, in the sense that its satisfaction is checked against a pair of instances: the first one where the antecedent holds, and the second one where the identification of values takes place. This semantics of MDs was sketched in [21].

In this paper we use a refinement of that original semantics that was put forth in [25] (cf. also [26]). It improves wrt the latter in that it disallows changes that are irrelevant to

* Research supported by the NSERC Strategic Network on Business Intelligence (BIN ADC05) and NSERC/IBM CRDPJ/371084-2008.

the duplicate resolution process. Actually, [25] goes on to define the clean versions of the original database instance D_0 that contains duplicates. They are called the *resolved instances* (RIs) of D_0 wrt the given set M of matching dependencies. A resolved instance is obtained as the fixed point of a chase-like procedure that starts from D_0 and iteratively applies or enforces the MDs from M. Each step of this chase generates a new instance by making equal values that are identified as duplicates by the MDs.

In [25] it was shown that resolved instances always exist, and that they have certain desirable properties. For example, the set of allowed changes is just restrictive enough to prevent irrelevant changes, while still guaranteeing existence of resolved instances. The resolved instances that minimize the *overall number* of attribute value changes (associated to a same tuple identifier) wrt the original instance are called *minimally resolved instances* (MRIs). On this basis, given a query Q posed to a database instance D_0 that may contain duplicates, we define the *resolved answers* wrt Σ as the query answers that are true of all the minimally resolved instances [25].

The concept of resolved query answer has similarities to that of *consistent query answer* (CQA) in a database that fails to satisfy a set of integrity constraints [4, 11]. The consistent answers are invariant under the *repairs* of the original instance. However, data cleaning and CQA are different problems. For the former, we want to compute a clean instance, determined by MDs; for the latter, the goal is obtaining semantically correct query answers. MDs are not (static) ICs. In principle, we could see clean instances as repairs, treating MDs similarly to static FDs. However, the existing repair semantics do not capture the matchings as dictated by MDs (cf. [25, 26] for a more detailed discussion).

In this paper, we investigate the *problem of computing the resolved answers*, simply called *resolved answer problem* (RAP). The motivation for addressing this problem is that even in a database instance containing duplicates, much or most of the data may be duplicate-free. One can therefore obtain useful information from the instance without having to perform data cleaning on the instance. This would be convenient if the user does not want, or cannot afford, to go through a data cleaning process. In other situations the user may not have write access to the data being queried, or any access to the data sources, as in virtual data integration systems.

In [27] we identified classes of MDs and conjunctive queries for which RAP can be solved in polynomial time in data complexity. Furthermore, a recursively-defined predicate was introduced for identifying the sets of duplicate values within a database instance. This predicate can be combined with a query, opening the ground for a query rewriting approach to RAP.

In this paper we present a *query rewriting methodology for the RAP problem* (for the identified classes of MDs and queries). It can be used to rewrite the original query Q into a new query Q', in such a way that the latter, posed as usual to the original instance D_0, returns the resolved answers to the original query.

More precisely, we show that queries Q (in a restricted but broad class of conjunctive queries) that expect to obtain resolved answers from a given dirty database D can be rewritten into a (non-disjunctive) recursive Datalog^{not,s} query Q' with stratified negation and aggregation. Q' posed to D returns the answers to Q. As expected, such a query can be computed in polynomial time in the size of the initial database. The recursion arises from the fact that identifying duplicate values requires computing the transitive closure of binary similarity operators. Transitivity is not assumed for similarity operators, and in fact, common similarity relations used in practice, such as those based on the edit distance and related string similarity metrics, are not transitive. Aggregation is needed to enforce the minimality constraint, since this involves finding the frequency of occurrence of values within a set of duplicates.

To the best of our knowledge, this is the first result on query rewriting in the context of MD-based entity resolution. Furthermore, our rewritings into Datalog are non-trivial, in the sense that they are not the result of translations into Datalog of first-order rewritings. Our rewriting uses in an essential manner the elements of the resulting Datalog queries, namely recursion and aggregation. It is worth mentioning that the polynomial-time rewritings for conjunctive queries proposed for consistent query answering have been all been first-order (FO) [4, 18, 24, 31].

On the other hand, the general answer set programs that have been proposed as *repair programs* [5, 7, 29, 19, 17], that specify database repairs and can be used for highly expressive query rewritings, have a higher expressive power and complexity than Datalog programs with stratified negation and aggregation.[1] The attempts in [10] to obtain lower complexity programs for CQA from repair programs for some tractable classes of queries and constraints led back to FO rewritings. Thus classical, i.e. non-disjunctive and stratified, Datalog was missed as an "intermediate" language for CQA.

This paper is organized as follows. In Section 2 we introduce basic concepts and notation on MDs. In Section 3, we define the important concepts used in this paper, in particular, (minimally) resolved instances and resolved answers to queries. Section 4 contains the main results of this paper, that includes a query rewriting algorithm for a special case of the resolved query answer problem. Section 5 concludes the paper and discusses related and future work. Proofs of results can be found in [27].

2 Preliminaries

We consider a relational schema $\mathcal{S}$ that includes an enumerable, possibly infinite domain U, and a finite set $\mathcal{R}$ of database predicates. Elements of U are represented by lower case letters near the beginning of the alphabet. $\mathcal{S}$ determines a first-order (FO) language $L(\mathcal{S})$. An instance D for $\mathcal{S}$ is a finite set of ground atoms of the form $R(\bar{a})$, with $R \in \mathcal{R}$, say of arity n, and $\bar{a} \in U^n$. $R(D)$ denotes the extension of R in D. Every predicate $R \in \mathcal{S}$ has a set of attribute, denoted $attr(R)$. As usual, we sometimes refer to attribute A of R by $R[A]$. We assume that all the attributes of a predicate are different, and that we can identify attributes with *positions* in predicates, e.g. $R[i]$, with $1 \leq i \leq n$. If the ith attribute of predicate R is A, for a tuple $t = (c_1, \ldots, c_n) \in R(D)$, $t_R^D[A]$ (usually, simply $t_R[A]$ or $t[A]$ if the instance is understood) denotes the value c_i. For a sequence $\bar{A}$ of attributes in $attr(R)$, $t[\bar{A}]$ denotes the tuple whose entries are the values of the attributes in $\bar{A}$. Attributes have and may share subdomains of U.

In the rest of this section, we summarize some of the assumptions, definitions, notation, and results from two previous papers, [25] and [27], that we will need.

[1] Under the common assumption that the polynomial hierarchy does not collapse.

We will assume that every relation in an instance has an auxiliary attribute, a surrogate key, holding values that act as *tuple identifiers*. Tuple identifiers are never created, destroyed or changed during the duplicate resolution process. They do not appear in MDs, and are used to identify different versions of the same original tuple that result from the matching process. We usually leave them implicit; and "tuple identifier attributes" are commonly left out when specifying a database schema. However, when explicitly represented, they will be the "first" attribute of the relation. For example, if $R \in \mathcal{R}$ is n-ary, $R(t, c_1, \ldots, c_n)$ is a tuple with id t, and is usually written as $R(t, \bar{c})$. We usually use the same symbol for a tuple's identifier as for the tuple itself. Tuple identifiers are unique over the entire instance. Two instances over the same schema that share the same tuple identifiers are said to be *correlated*, and they can be unambiguously compared tuple by tuple.

As expected, some of the attribute domains, say A, have a built-in binary similarity relation $\approx_A \subseteq Dom(A) \times Dom(A)$ that is reflexive and symmetric. Such a relation can be extended to finite lists of attributes (or domains therefor), componentwise. For single attributes or lists thereof, the similarity relation is is generically denoted with $\approx$.

A *matching dependency* (MD) [20] is an expression of the form

$$m: \ R[\bar{A}] \approx S[\bar{B}] \ \rightarrow \ R[\bar{C}] \doteq S[\bar{E}], \tag{1}$$

with $\bar{A} = (A_1, ..., A_k)$, $\bar{C} = (C_1, ..., C_{k'})$ lists of (different) attributes from $attr(R)$; and $\bar{B} = (B_1, ..., B_k)$, $\bar{E} = (E_1, ..., E_{k'})$ lists of attributes from $attr(S)$.[2]

The set of attributes on the left-hand-side (LHS) of the arrow in m is denoted with $LHS(m)$. Similarly for the right-hand-side (RHS). In relation to (1), the attributes in a *corresponding pair* (A_i, B_i) or (C_i, E_i) are assumed to share a common domain; and in particular, a *similarity relation* $\approx_i$. In consequence, the condition on the LHS of (1) means that, for a pair of tuples t_1 in R and t_2 in S, $t_1[A_i] \approx_i t_2[B_i]$, $1 \leq i \leq k$. Similarly, the expression on the RHS means $t_1[A_i] \doteq t_2[B_i]$, $1 \leq i \leq k'$. Here, $\doteq$ means that the values should be updated to the same value.

Accordingly, the intended semantics of the MD in (1) is that, for an instance D, if any pair of tuples, $t_1 \in R(D)$ and $t_2 \in S(D)$, satisfy the similarity conditions on the LHS, then for the same tuples (or tuple ids), the attributes on the RHS have to take the same values [21], possibly through updates that may lead to a new version of D.

We assume that all sets M of MDs are in *standard form*, i.e. for no two different MDs $m_1, m_2 \in M$, $LHS(m_1) = LHS(m_2)$. All sets of MDs can be put in this form. MDs in a set M can interact in the sense that a matching enforced by one of them may create new similarities that lead to the enforcement of another MD in M. This intuition is captured through the *MD-graph*.

Definition 1. [26] Let M be a set of MDs in standard form. The *MD-graph* of M, denoted $MDG(M)$, is a directed graph with a vertex m for each $m \in M$, and an edge from m to m' iff $RHS(m) \cap LHS(m') \neq \emptyset$.[3] If $MDG(M)$ contains edges, M is called *interacting*. Otherwise, it is called *non-interacting* (NI). □

[2] We assume that the MDs are defined in terms of the same schema $\mathcal{S}$.

[3] That is, they share at least one corresponding pair of attributes.

3 Matching Dependencies and Resolved Answers

Updates as prescribed by an MD m are not arbitrary. The updates based on m have to be justified by m, as captured through the notion of *modifiable value* in an instance.

Definition 2. Let D be an instance, M a set of MDs, and $\mathcal{P}$ be a set of pairs (t, G), where t is a tuple of D and G is an attribute of t. (a) For a tuple $t_R \in R(D)$ and C an attribute of R, the value $t_R^D[C]$ is *modifiable wrt* $\mathcal{P}$ if there exist $S \in \mathcal{R}$, $t_S \in S(D)$, an $m \in M$ of the form $R[\bar{A}] \approx S[\bar{B}] \rightarrow R[\bar{C}] \doteq S[\bar{E}]$, and a corresponding pair (C, E) of $(\bar{C}, \bar{E})$ in m, such that $(t_S, E) \in \mathcal{P}$ and one of the following holds:

1. $t_R[\bar{A}] \approx t_S[\bar{B}]$, but $t_R[C] \neq t_S[E]$.
2. $t_R[\bar{A}] \approx t_S[\bar{B}]$ and $t_S[E]$ is modifiable wrt $\mathcal{P} \smallsetminus \{(t_S, E)\}$.

(b) The value $t_R^D[C]$ is *modifiable* if it is modifiable wrt $\mathcal{V} \smallsetminus \{(t_R, C)\}$, where $\mathcal{V}$ is the set of all pairs (t, G) with t a tuple of D and G an attribute of t. □

Definition 2 is recursive. The base case occurs when either case 1 applies (with any $\mathcal{P}$) or when there is no tuple/attribute pair in $\mathcal{P}$ that can satisfy part (a). Notice that recursion must terminate eventually, since the latter condition must be satisfied when $\mathcal{P}$ is empty, and each recursive call reduces the size of $\mathcal{P}$.

Example 1. Consider $m: R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$ on schema $R[A, B]$, and the following instance. Assume that the only non-trivial similarities are $a_1 \approx a_2 \approx a_3$ and $b_1 \approx b_2$. Since $a_2 \approx a_3$ and $c_1 \neq c_3$, $t_2[B]$ and $t_3[B]$ are modifiable (base case). With case 2 of Definition 2, since $a_1 \approx a_2$, and $t_2[B]$ is modifiable, we obtain that $t_1[B]$ is modifiable.

$R(D)$	A	B
t_1	a_1	c_1
t_2	a_2	c_1
t_3	a_3	c_3
t_4	b_1	c_3
t_5	b_2	c_3

For $t_5[B]$ to be modifiable, it must be modifiable wrt $\{(t_i, B) \mid 1 \leq i \leq 4\}$, and via t_4. According to case 2 of Definition 2, this requires $t_4[B]$ to be modifiable wrt $\{(t_i, B) \mid 1 \leq i \leq 3\}$. However, this is not the case since there is no t_i, $1 \leq i \leq 3$, such that $t_4[A] \approx t_i[A]$. Therefore $t_5[B]$ is not modifiable. A symmetric argument shows that $t_4[B]$ is not modifiable. □

Definition 3. [25] Let D, D' be correlated instances, and M a set of MDs. (D, D') *satisfies* M, denoted $(D, D') \vDash M$, iff: 1. For any pair of tuples $t_R \in R(D)$, $t_S \in S(D)$, if there exists an $m \in M$ of the form $R[\bar{A}] \approx S[\bar{B}] \rightarrow R[\bar{C}] \doteq S[\bar{E}]$ and $t_R[\bar{A}] \approx t_S[\bar{B}]$, then for the corresponding tuples (i.e. with same ids) $t'_R \in R(D')$ and $t'_S \in S(D')$, it holds $t'_R[\bar{C}] = t'_S[\bar{E}]$. 2. For any tuple $t_R \in R(D)$ and any attribute G of R, if $t_R[G]$ is *non-modifiable*, then $t'_R[G] = t_R[G]$. □

Intuitively, D' in Definition 3 is a new version of D that is produced after a single update. Since the update involves matching values (i.e. making them equal), it may produce "duplicate" tuples, i.e. that only differ in their tuple ids. They would possibly be merged into a single tuple in the a data cleaning process. However, we keep the two versions. In particular, D and D' have the same number of tuples. Keeping or eliminating duplicates will not make any important difference in the sense that, given that tuple ids are never updated, two duplicates will evolve in exactly the same way as subsequent updates are performed. Duplicate tuples will never be subsequently "unmerged".

This definition of MD satisfaction departs from [21], which requires that updates preserve similarities. Similarity preservation may force undesirable changes [25]. The existence of the updated instance D' for D is guaranteed [25]. Furthermore, wrt [21], our definition does not allow unnecessary changes from D to D'. Definitions 2 and 3 imply that only values of attributes that appear to the right of the arrow in some MD are subject to updates. Hence, they are called *changeable attributes*.[4]

Definition 3 allows us to define a *clean instance* wrt M as the result of a chase-like procedure, each step being satisfaction preserving.

Definition 4. [25] (a) A *resolved instance* (RI) for D wrt M is an instance D', such that there are instances $D_1, D_2, ...D_n$ with: $(D, D_1) \vDash M$, $(D_1, D_2) \vDash M$,..., $(D_{n-1}, D_n) \vDash M$, $(D_n, D') \vDash M$, and $(D', D') \vDash M$. We say D' is *stable*. (b) D' is a *minimally resolved instance* (MRI) for D wrt M if it is a resolved instance that minimizes the overall number of attribute value changes wrt D and in relation with the same tuple ids. (c) $\mathit{MRI}(D, M)$ denotes the class of MRIs of D wrt M. □

Example 2. Consider the MD $R[A] \approx R[A] \rightarrow R[B] \doteq R[B]$ on predicate R, and the instance D. It has several resolved instances, among them, four that

$R(D)$	A	B
t_1	a_1	c_1
t_2	a_1	c_2
t_3	b_1	c_3
t_4	b_1	c_4

$R(D_1)$	A	B
t_1	a_1	c_1
t_2	a_1	c_1
t_3	b_1	c_3
t_4	b_1	c_3

$R(D_2)$	A	B
t_1	a_1	c_1
t_2	a_1	c_1
t_3	b_1	c_1
t_4	b_1	c_1

minimize the number of changes. One of them is D_1. A resolved instance that is not minimal in this sense is D_2. □

In this work, as in [25, 26], we are investigating what we could call "the pure case" of MD-based entity resolution. It adheres to the original semantics outlined in [21], which does not specify how the matchings are to be done, but only which values must be made equal. That is, the MDs have implicit existential quantifiers (for the values in common). The semantics we just introduced formally captures this pure case. We find situations like this in other areas of data management, e.g. with referential integrity constraints, tuple-generating dependencies in general [1], schema mappings in data exchange [8], etc. It can be shown that an RI exists for any instance [25]. It follows immediately that every instance has an MRI. A non-pure case that uses matching functions to realize the matchings as prescribed by MDs is investigated in [13, 14, 6].

Definition 5. [25] Let $\mathcal{Q}(\bar{x})$ be a query expressed in the FO language $L(\mathcal{S})$. A tuple of constants $\bar{a}$ from U is a *resolved answer* to $\mathcal{Q}(\bar{x})$ wrt the set M of MDs, denoted $D \models_M \mathcal{Q}[\bar{a}]$, iff $D' \models \mathcal{Q}[\bar{a}]$, for every $D' \in \mathit{MRI}(D, M)$. $\mathit{ResAn}(D, \mathcal{Q}, M)$ is the set of resolved answers to $\mathcal{Q}$ from D wrt M. □

[4] Not to be confused with "modifiability", that applies to tuples.

Example 3. (example 1 cont.) Since the only MRI for the original instance D is $R(D')$ $= \{\langle t_1, a_1, c_1\rangle, \langle t_2, a_2, c_1\rangle, \langle t_3, a_3, c_1\rangle, \langle t_4, b_1, c_3\rangle, \langle t_5, b_2, c_3\rangle\}$, the resolved answers to the query $\mathcal{Q}(x,y)\colon\ R(x,y)$ are $\{\langle a_1, c_1\rangle, \langle a_2, c_1\rangle, \langle a_3, c_1\rangle, \langle b_1, c_3\rangle, \langle b_2, c_3\rangle\}$. $\square$

4 Query Rewriting for Resolved Answers

In this section, we present a query rewriting method for retrieving the resolved answers for certain classes of queries and sets of MDs. We provide an intuitive and informal presentation of the rewritten queries. For details and a proof of correctness, see [27].

It has been shown in previous work that the problem of deciding resolved answers (the *resolved answer decision problem*) is generally intractable in data [25, 26, 27]. However, there are tractable cases of this decision problem that are practically relevant [27]. Two of those cases are considered here: that of *non-interacting* (NI) sets of MDs (cf. Definition 1), and that of *hit-set-cyclic* (HSC) sets of MDs, that we now define.

Definition 6. A set M of MDs is *hit-simple-cyclic* iff the following hold: (a) In all MDs in M and in all their corresponding pairs, the two attributes (and predicates) are the same. (b) In all MDs $m \in M$, at most one attribute in $\mathit{LHS}(m)$ is changeable. (c) Each vertex v_1 in $\mathit{MDG}(M)$ is on at least one cycle, or there is a vertex v_2 on a cycle with at least two vertices such that there is an edge from v_1 to v_2. $\square$

Example 4. For schema $R[A, C, F, G]$, consider the following set M of MDs:

$$m_1\colon\ R[A] \approx R[A]\ \rightarrow\ R[C, F, G] \doteq R[C, F, G],$$
$$m_2\colon\ R[C] \approx R[C]\ \rightarrow\ R[A, F, G] \doteq R[A, F, G].$$

Set M obviously satisfies (a) and (b) of Definition 6. Also, $\mathit{MDG}(M)$ consists of a single cycle through the two vertices, so M satisfies (c). M is then HSC.

Predicate R subject to the given M has two "keys", $R[A]$ and $R[C]$. Such relations are common in practice. For example, R may be used in a database about people: $R[A]$ could be used for the person's name, $R[C]$ the address, and $R[F]$ and $R[G]$ for non-distinguishing information, e.g. gender and age. $\square$

HSC sets have properties similar to those of NI sets wrt the resolved answer problem [27]. For both classes, the value positions identified as duplicates are the same for all MRIs, and they are characterized via equivalence classes of the *tuple-attribute closure*.

Definition 7. [27] Let $M = \{m_i \mid i = 1, \ldots, n\}$ be a set of MDs, with $m_i\colon\ R_i[\bar{A}_i] \approx_i S_i[\bar{B}_i]\ \rightarrow\ R_i[\bar{C}_i] \doteq S_i[\bar{E}_i]$. (a) The *previous set* of m_i, denoted $PS(m_i)$, is the set of all MDs $m_j \in M$ with a path in $\mathit{MDG}(M)$ from m_j to m_i. (b) For an instance D, and tuples ids t_1, t_2 for R, S, resp. (i.e. ids for tuples $t_1 \in R(D), t_2 \in S(D)$): $(t_1, C_i) \approx' (t_2, E_i)\ :\Longleftrightarrow\ t_1[\bar{A}_j] \approx_j t_2[\bar{B}_j]$, where (C_i, E_i) is a corresponding pair of $(\bar{C}_i, \bar{E}_i)$ in m_i and $m_j \in PS(m_i)$. (c) The *tuple-attribute closure* (TA closure) of M wrt D, denoted $\mathit{TA}^{M,D}$, is the reflexive, symmetric, and transitive closure of $\approx'$. $\square$

Example 5. (example 4 continued) In this case, $PS(m_1) = PS(m_2) = \{m_1, m_2\}$. Consider the instance D, where the only similarities are: $a_i \approx a_j$, $b_i \approx b_j$, $d_i \approx d_j$,

$R(D)$	A	B
t_1	a_1	d_1
t_2	a_2	e_2
t_3	b_1	e_1
t_4	b_2	d_2

$e_i \approx e_j$, with $i, j \in \{1, 2\}$. The relations $(t_{imod4+1}, A) \approx' (t_{(i+1)mod4+1}, A)$ and $(t_{imod4+1}, B) \approx' (t_{(i+1)mod4+1}, B)$, $0 \leq i \leq 3$, hold. The TA closure is given by $\{TA(t_i, x, t_j, x) \mid 1 \leq i, j \leq 4,\ x \in \{A, B\}\}$. Notice that this relation involves just tuple ids and attributes. However, it depends on D through the similarity conditions in (b) in the definition. □

For the set of MDs as in Definition 7, the TA closure can be specified by Datalog rules. The database predicates in them have a first argument (attribute) to explicitly represent the tuple id. More precisely, for $1 \leq i \leq n$, for each corresponding pair (C, E) of $(\bar{C}_i, \bar{E}_i)$, and for each $m_j \in PS(m_i)$, we have the rule:[5]

$$(t_1, R_i[C]) \approx' (t_2, S_i[E]) \ \leftarrow\ R_i(t_1, \bar{x}), S_i(t_2, \bar{y}), t_1[\bar{A}_j] \approx_j t_2[\bar{B}_j].$$

Additionally, for all attributes A of R_i and ids t of tuples in R_i, we have

$$TA(t, A, t, A) \leftarrow R_i(t, \bar{x}); \tag{2}$$

similarly for S_i. For arbitrary tuple ids t_1, t_2, and t_3, and attributes A, B, and C,

$$TA(t_1, A, t_2, B) \ \leftarrow\ TA(t_2, B, t_1, A), \tag{3}$$
$$TA(t_1, A, t_2, B) \ \leftarrow\ (t_1, A) \approx' (t_2, B), \tag{4}$$
$$TA(t_1, A, t_3, C) \ \leftarrow\ TA(t_1, A, t_2, B), (t_2, B) \approx' (t_3, C). \tag{5}$$

Rules (4) and (5) express that TA is the transitive closure of relation $\approx'$. Rules (2) and (3), that TA is reflexive and symmetric. A related concept is the *attribute closure*.

Definition 8. [26] Let M be a set of MDs on schema $\mathcal{S}$. (a) The symmetric binary relation $\doteq_r$ relates attributes $R[A]$, $S[B]$ of $\mathcal{S}$ whenever there is an MD m in M where $R[A] \doteq S[B]$ appears in $RHS(m)$. (b) The *attribute closure* of M is the reflexive, symmetric, transitive closure of $\doteq_r$. (c) $E_{R[A]}$ denotes the equivalence class of attribute $R[A]$ in the attribute closure of M. □

Example 6. Let M be the set of MDs: $R[A] \approx_1 S[B] \rightarrow R[C] \doteq S[D]$, $S[E] \approx_2 T[F] \wedge S[G] \approx T[H] \rightarrow S[D, K] \doteq T[J, L]$, $T[F] \approx_3 T[H] \rightarrow T[L, N] \doteq T[M, P]$. The equivalence classes of T_{at} are $E_{R[C]} = \{R[C], S[D], T[J]\}$, $E_{S[K]} = \{S[K], T[L], T[M]\}$, and $E_{T[N]} = \{T[N], T[P]\}$. □

It is easy to show that if (u_1, A), (u_2, B) are in the same equivalence class of tuple-attribute closure, then A and B are in the same equivalence class of attribute closure.

Definition 9. Let M be a set of MDs and D and instance and a a data value. For an equivalence class E of $TA^{M,D}$, the *frequency of a in E* is the quantity $freq^D(a, E) := |\{(t, A) \mid (t, A) \in E,\ t[A] = a \text{ in } D\}|$. □

Proposition 1. [27] For M an NI or HSC set of MDs, and D an instance, each MRI for D wrt M is obtained by setting, for each equivalence class E of $TA^{M,D}$, all the values for $t[A]$, with $(t, A) \in E$, to a value a that maximizes $freq^D(a, E)$. □

[5] Remember that the first argument in R_i, S_i stands for the tuple id.

Example 7. (example 5 continued) The two equivalence classes of TA closure are $E_1 = \{(t_i, A) \mid 1 \leq i \leq 4\}$ and $E_2 = \{(t_i, B) \mid 1 \leq i \leq 4\}$. All values in the A (B) column of the table have frequency 1 in E_1 (E_2). Thus, there are 16 MRIs, obtained by setting all values in each column to a common value chosen from those in the column. $\square$

Proposition 1 tells us that the minimally resolved instances for an instance D can be obtained by identifying most frequently occurring values. Thus, resolved query answers from D can be computed by imposing this requirement on the original query. As a consequence, the rewritten queries will become aggregate queries.

In Datalog notation, aggregate queries take the form $P(\bar{a}, \bar{x}, Agg(\bar{u})) \leftarrow B(\bar{y})$, where P is answer collecting predicate, the body $B(\bar{y})$ represents a conjunction of literals all of whose variables are among those in $\bar{y}$, $\bar{a}$ is a list of constants, $\bar{x} \cup \bar{u} \subseteq \bar{y}$, and Agg is an aggregate operator such as $Count$ or Sum. The variables $\bar{x}$ are the "group-by" variables. That is, for each fixed value $\bar{b}$ for $\bar{x}$, aggregation is performed over all tuples that make $B\frac{\bar{x}}{\bar{b}}$, the instantiation of B on $\bar{b}$ for $\bar{x}$, true. $Count(\bar{u})$ counts the number of distinct values of $\bar{u}$, while $Sum(\bar{u})$ sums over all $\bar{u}$, whether distinct or not.

Our query rewriting methodology for computing resolved answers will be applicable to a certain class of conjunctive queries, the called UJCQ queries defined below. In [27] a counterexample for the general applicability to all conjunctive queries is given.

Definition 10. [27] For a set M of MDs, a conjunctive query $\mathcal{Q}$ without built-ins is an *unchangeable join conjunctive query* (UJCQ query) if there are no existentially quantified variables in a join in $\mathcal{Q}$ in the position of a changeable attribute. For fixed, M, $UJCQ$ denotes this class of queries. $\square$

In the rest of this paper we assume that the we have a fixed and finite set M of MDs that satisfies the hypothesis of Proposition 1. The queries posed to the initial, possibly non-resolved instance belong to $UJCQ$.

The rewritten queries will be in Datalog^{not,s} [1], i.e. Datalog queries with stratified negation and aggregation, and the built-ins $\neq$ and $\leq$. For simplicity, the rewriting makes use of tuple identifiers only. In the absence of such a surrogate key, whole tuples could be used instead of identifiers.

Given a $UJCQ$ query $\mathcal{Q}$, with answer predicate Q:

$$Q(\bar{x}) \leftarrow R_1(\bar{v}_1), R_2(\bar{v}_2), ..., R_n(\bar{v}_n), \tag{6}$$

the rewritten query $\mathcal{Q}'$ is the conjunction of the rewritings Q_i of each of the R_i, to be given in (8) below, i.e.

$$Q'(\bar{x}) \leftarrow Q_1(\bar{v}_1), Q_2(\bar{v}_2), ..., Q_n(\bar{v}_n). \tag{7}$$

Now, for a fixed atom $R_i(\bar{v}_i)$ in (6), let C be the set of changeable attributes corresponding to a free variable in $\bar{v}_i$, i.e. also appearing in $Q(\bar{x})$. We denote the list of such variables by $\bar{v}_C$.

If C is empty, then its rewriting becomes $Q_i(\bar{v}_i) \leftarrow R_i(\bar{v}_i)$. Intuitively, this is because, by Definition 10, only attributes corresponding to free variables can participate in joins, so changes to values of attributes corresponding to bound variables cannot affect satisfaction of the body in (6).

Suppose C is non-empty, and consider $R_i[A] \in C$. From Proposition 1, deciding whether or not all MRIs have the same value v for $R_i[A]$ for a given tuple id t will involve finding the frequency of v in E for the equivalence class E of the TA closure to which $(t, R_i[A])$ belongs. We introduce aggregation operators to express this count, of values for attributes in $E_{R_i[A]}$ (cf. the remark following Example 6).

We introduce a predicate $C^{R_i[A]}$, with an attribute at the start of its attribute list whose value is the attribute in $E_{R_i[A]}$ over whose values aggregation is performed. For an attribute A and list of variables $\bar{v}$, we denote with v_A the variable holding the value for A. For each $S[B] \in E_{R_i[A]}$, we have the rule

$$C^{R_i[A]}(S[B], t_1, v_{S[B]}, \mathit{Count}(t_2)) \leftarrow TA(t_1, R_i[A], t_2, S[B]), R_i(t_1, \bar{u}), S(t_2, \bar{v}),$$

in which all predicate arguments are variables except for the attribute labels $S[B]$ and $R_i[A]$, that are constants.

In each tuple in the head predicate of the above expression, the value of the *Count* expression is $|\{t \mid (t, S[B]) \in E,\ t[S[B]] = v_{S[B]}\}|$, where E is the equivalence class of the TA closure to which $(t_1, R_i[A])$ belongs.

To find the frequency of the value of $v_{S[B]}$ in E, this count must be extended to all attributes in $E_{R_i[A]}$. We introduce the predicate $\mathit{Total}^{R_i[A]}$ for this purpose:

$$\mathit{Total}^{R[A_i]}(t, v, \mathit{Sum}(z)) \leftarrow C^{R_i[A]}(x, t, v, z).$$

Tuples in $\mathit{Total}^{R_i[A]}$ specify in their last argument the frequency of v in the equivalence class of the TA-closure to which $(u, R[A_i])$ belongs.

To compare these aggregate quantities for different values of v, we use the *Compare* predicate:

$$\mathit{Compare}^{R[A_i]}(t, v) \leftarrow \mathit{Total}^{R_i[A]}(t, v, z_1), \mathit{Total}^{R_i[A]}(t, v', z_2), z_1 \leq z_2, v' \neq v.$$

Tuples in $\mathit{Compare}^{R_i[A]}$ consist of a tuple identifier t in R_i and a value v for attribute $R_i[A]$. For such a pair (t, v) there is another value v' whose frequency in the equivalence class of the TA closure to which $(t, R_i[A])$ belongs is at least as large as that of v.

In order for a value to be a "certain" for a given attribute $R_i[A]$ in a given tuple, the tuple and value must not occur as a tuple in $\mathit{Compare}^{R_i[A]}$. Let $\bar{v}_i'$ be $\bar{v}_i$ with all variables in $\bar{v}_C$ replaced with new variables. Then,

$$Q_i(\bar{v}_i) \leftarrow R_i(t, \bar{v}_i'), \bigwedge_{R_i[A] \in C} \mathit{not}\ \mathit{Compare}^{R_i[A]}(t, v_{R_i[A]}), \mathit{Total}^{R_i[A]}(t, v_{R_i[A]}, z). \quad (8)$$

Example 8. Consider the schema $R[ABC], S[EFG], U[HI]$ with non-interacting MDs: $R[A] \approx S[E] \rightarrow R[B] \doteq S[F],\ S[E] \approx U[H] \rightarrow S[F] \doteq U[I]$, and the UJCQ query: $Q(x, y, z) \leftarrow R(x, y, z), S(u, v, z), U(p, q)$. Since the S and U atoms have no free variables holding the values of changeable attributes, they remain unchanged. Therefore, the rewritten query Q' has the form

$$Q' \leftarrow R'(x, y, z), S(u, v, z), U(p, q), \quad (9)$$

where R' is the rewritten form of R. The only free variable holding the value of a changeable attribute is y. This variable corresponds to attribute $R[B]$, which belongs to the equivalence class $E_{R[B]} = \{R[B], S[F], U[I]\}$. Therefore, we have the rules:

$$C^{R[B]}(R[B], t_1, y, \mathit{Count}(t_2)) \leftarrow TA(t_1, R[B], t_2, R[B]), R(t_1, x', y', z'), R(t_2, x, y, z).$$
$$C^{R[B]}(S[F], t_1, y, \mathit{Count}(t_2)) \leftarrow TA(t_1, R[B], t_2, S[F]), R(t_1, x', y', z'), S(t_2, x, y, z).$$
$$C^{R[B]}(U[I], t_1, y, \mathit{Count}(t_2)) \leftarrow TA(t_1, R[B], t_2, U[I]), R(t_1, x', y', z'), U(t_2, x, y).$$
$$\mathit{Total}^{R[B]}(t, y, \mathit{Sum}(u)) \leftarrow C^{R[B]}(x, t, y, u).$$
$$\mathit{Compare}^{R[B]}(t, y) \leftarrow \mathit{Total}^{R[B]}(t, y, z_1),\ \mathit{Total}^{R[B]}(t, y'', z_2),\ z_1 \leq z_2,\ y'' \neq y.$$

The rewriting of R becomes

$$R'(x, y, z) \leftarrow R(t, x, y', z), \mathit{not}\ \mathit{Compare}^{R[B]}(t, y),\ \mathit{Total}^{R[B]}(t, y, w). \qquad (10)$$

Thus, the rewriting of the original query is the stratified Datalog program [1] with aggregation consisting of rules (9), (10), plus the five rules preceding (10). □

In order to obtain the resolved answers to a query on a possibly non-resolved instance D, the resulting Datalog program can be run on D in polynomial time in the size of D. Remarks on implementation and an example are included in an extended version [28].

5 Conclusions

This paper considered a novel approach based on query rewriting to the duplicate resolution problem within the framework of matching dependencies. The transformed queries return the resolved answers to the original query, which are the answers that are true in all minimally resolved instances.

We used minimal resolved instances (MRIs) as our model of a clean database. Another possibility is to use arbitrary, not necessarily minimal, resolved instances (RIs). While MRIs have the advantage of being "closer" to the original instance than RIs, they have the downside of being overly restrictive.

In practice, update values are typically chosen by applying a merging function to the sets of duplicates [9, 13, 14], rather than by imposing a minimal change constraint. RIs are more flexible in that they take into account all ways of choosing the update values that lead to a clean database. We are currently investigating query answering over RIs, identifying tractable cases of the problem that are not tractable for MRIs.

Matching dependencies first appeared in [20], and their semantics is given in [21]. The original semantics was refined in [13, 14], including the use of *matching functions* (MFs) for matching two attribute values. The approach in [13, 14] uses a chase to define clean instances. The MDs are applied one at a time to pairs of tuples, rather than all at once to all tuples as in the present paper. Another important difference is that here we do not use MFs to do a mathcing, but implicit existential quantifiers for the values in common. When the update values are determined by the matching functions there is no uncertainty arising from different possible choices for update values. Rather, the different clean instances are produced by applying the MDs in different orders. Clean answers are obtained by taking a glb (or lub) over the clean instances wrt a partial ordering that is based on semantic domination of one value by another.

The alternative refinement of the semantics used in this paper was first introduced in [25, 26]. A thorough complexity analysis, as well as the derivation of a query rewriting algorithm for the resolved answer problem was done in [27].

Our work in some ways resembles work on query answering over ontologies [16]. As in our duplicate resolution setting, a chase is applied repeatedly to an initial instance, terminating in a "repaired" instance which is a fixed point of the chase rules. The set of chase rules can include tuple generating dependencies (TGDs) and equality generating dependencies (EGDs). Despite these similarities, our chase differs from those based on EGDs and TGDs in that it does not generate new tuples, but modifies values in existing tuples. Also, despite the fact that MDs are similar to EGDs, issues arise as a result of the non-transitivity of similarity operators that do not occur in the case of EGDs.

In [3], Datalog is used for identifying groups of tuples that could be merged. However, they do not do the merging (a main contribution in our approach) or base their approach on MDs. Actually, that work could be considered as complimentary to ours, in the sense that, in essence, the authors address the problem of identifying similarities. This is the starting point for the actual matchings that we address in this paper.

References

[1] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases.Addison-Wesley (1995)
[2] Afrati, F., Kolaitis, P.: Repair checking in inconsistent databases: Algorithms and complexity. In: Proc. ICDT (2009)
[3] Arasu, A., Ré, C., Suciu, D.: Large-scale deduplication with constraints using dedupalog. In: Proc. ICDE (2009)
[4] Arenas, M., Bertossi, L., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proc. PODS (1999)
[5] Arenas, M., Bertossi, L., Chomicki, J.: Answer sets for consistent query answering in inconsistent databases. Theory and Practice of Logic Programming 3(4-5), 393–424 (2003)
[6] Bahmani, Z., Bertossi, L., Kolahi, S., Lakshmanan, L.: Declarative Entity Resolution via Matching Dependencies and Answer Set Programs. In: Proc. KR 2012 (2012)
[7] Barceló, P., Bertossi, L., Bravo, L.: Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets. In: Bertossi, L., Katona, G.O.H., Schewe, K.-D., Thalheim, B. (eds.) Semantics in Databases. LNCS, vol. 2582, pp. 7–33. Springer, Heidelberg (2003)
[8] Barcelo, P.: Logical foundations of relational data exchange. SIGMOD Record 38(1), 49–58 (2009)
[9] Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., Euijong Whang, S., Widom, J.: Swoosh: A generic approach to entity resolution. VLDB Journal 18(1), 255–276 (2009)
[10] Bertossi, L.: From database repair programs to consistent query answering in classical logic. In: Proc. AMW. CEUR-WS, vol. 450 (2009)
[11] Bertossi, L.: Database Repairing and Consistent Query Answering. Synthesis Lectures on Data Management. Morgan & Claypool (2011)
[12] Bertossi, L., Bravo, L., Franconi, E., Lopatenko, A.: The complexity and approximation of fixing numerical attributes in databases under integrity constraints. Information Systems 33(4), 407–434 (2008)
[13] Bertossi, L., Kolahi, S., Lakshmanan, L.: Data cleaning and query answering with matching dependencies and matching functions. In: Proc. ICDT (2011)
[14] Bertossi, L., Kolahi, S., Lakshmanan, L.: Data cleaning and query answering with matching dependencies and matching functions. Theory of Computing Systems (2012), doi:10.1007/s00224-012-9402-7
[15] Bleiholder, J., Naumann, F.: Data fusion. ACM Computing Surveys 41(1), 1–41 (2008)

[16] Cali, A., Gottlob, G., Lukasiewicz, T., Pieris, A.: A logical toolbox for ontological reasoning. ACM Sigmod Record 40(3), 5–14 (2011)

[17] Caniupan, M., Bertossi, L.: The consistency extractor system: answer set programs for consistent query answering in databases. Data & Know. Eng. 69(6), 545–572 (2010)

[18] Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Information and Computation 197(1/2), 90–121 (2005)

[19] Eiter, T., Fink, M., Greco, G., Lembo, D.: Repair localization for query answering from inconsistent databases. ACM Trans. Database Syst. 33(2) (2008)

[20] Fan, W.: Dependencies revisited for improving data quality. In: Proc. PODS (2008)

[21] Fan, W., Jia, X., Li, J., Ma, S.: Reasoning about record matching rules. In: Proc. VLDB (2009)

[22] Flesca, S., Furfaro, F., Parisi, F.: Querying and repairing inconsistent numerical databases. ACM Trans. Database Syst. 35(2) (2010)

[23] Franconi, E., Laureti Palma, A., Leone, N., Perri, S., Scarcello, F.: Census Data Repair: A Challenging Application of Disjunctive Logic Programming. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 561–578. Springer, Heidelberg (2001)

[24] Fuxman, A., Miller, R.: First-order query rewriting for inconsistent databases. J. Computer and System Sciences 73(4), 610–635 (2007)

[25] Gardezi, J., Bertossi, L., Kiringa, I.: Matching dependencies with arbitrary attribute values: semantics, query answering and integrity constraints. In: Proc. LID (2011)

[26] Gardezi, J., Bertossi, L., Kiringa, I.: Matching dependencies: semantics, query answering and integrity constraints. Frontiers of Computer Science 6(3), 278–292 (2012)

[27] Gardezi, J., Bertossi, L.: Query answering under matching dependencies for data cleaning: Complexity and algorithms, arXiv:1112.5908v1

[28] Gardezi, J., Bertossi, L.: Query Rewriting using Datalog for Duplicate Resolution (extended version), http://people.scs.carleton.ca/~bertossi/papers/datalog22Long.pdf

[29] Greco, G., Greco, S., Zumpano, E.: A logical framework for querying and repairing inconsistent databases. IEEE Trans. Knowledge and Data Eng. 15(6), 1389–1408 (2003)

[30] Wijsen, J.: Database repairing using updates. ACM Trans. Database Systems 30(3), 722–768 (2005)

[31] Wijsen, J.: On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In: Proc. PODS (2010)

Reasoning about Knowledge in Distributed Systems Using Datalog

Matteo Interlandi

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

Abstract. Logic programming has been considered a viable solution for distributed computing since the Fifth Generation Computer Systems project [8]. Nowadays, this line of thought is gaining new verve, pushed by the need for new programming paradigms for addressing new emerging issues in distributed computing. We argue that a missing piece in the current state-of-the-art is the capability to express statements about the knowledge state of distributed nodes. In fact, reasoning about the knowledge state of (group of) nodes has been demonstrated to be fundamental in order to design and analyze distributed protocols [7]. To reach this goal, we designed Knowlog: Datalog$^\neg$ augmented with a set of epistemic modal operators, allowing the programmer to directly express what a node "*knows*" instead of low level communication details.

1 Introduction

Since the Fifth Generation Computer Systems project [8], many authors have stated how logic programming [10,9] could be used to express distributed programs' specifications. New emerging trends that are arising nowadays provide new chances for proving how logic programming can be used to tackle distributed computing concerns. For instance, the tight coupling among distributed nodes and the related overhead introduced by traditional mechanisms implementing ACID properties starts to be considered unacceptable by cloud computing operators [5]. To address these issues, monotonic logic programming has been employed to formally specify eventually consistent distributed programs [3].

Motivated by all these facts, our goal is to open a new direction in the investigation on how Datalog could be adopted to implement distributed systems. We conjecture, in fact, that a missing piece in the litterature exists: this is the possibility to express in Datalog statements about the knowledge state of distributed nodes. The ability to reason about the knowledge state of (group of) nodes has been demonstrated to be a fundamental tool in multi-agent systems in order to specify global behaviors and properties of protocols [7]. Therefore, inspired by previous works in distributed logic programming [9,3] and knowledge-base programs for multi-agent systems [7], we develop Knowlog: Datalog$^\neg$ leveraged with a set of epistemic modal operators. In this way programmers are able to directly express nodes' state of knowledge instead of low level communication details. The advantage of this formalism is that it abstracts away all the mechanisms by which the knowledge is exchanged (message passing, shared memory, etc) and permits to explicitly reason about the nodes' state of knowledge. To support our assertions, we describe an implementation of the two phase commit protocol.

The remainder of the paper is organized as follows: Section 2 contains some preliminary notations about Datalog$^\neg$ and Datalog$^\neg$ augmented with a notion of time. Section 3 describes what we intend for a distributed system and introduces some concepts such as *global state* and *run* that will be used in Section 4 to specify the modal operators K, E and D. In addition, Section 4 contains the two phase commit protocol implementation in Knowlog. The paper finishes with Section 5 which specifies Knowlog semantics, and conclusions.

2 Preliminaries

As usual, 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 $H(\bar{u})$, $B_i(\bar{u}_i)$ and $C_j(\bar{v}_j)$ are *atoms*, H, B_i, C_j are relation names in **relname** 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**, with both **dom** and **var** disjoined from **relname**. In the followings we will interchangeably use the words *predicate*, *relation* and *table*. A *literal* is an atom (in this case we refer to it as *positive*) or the negation of an atom. We will usually refer to a ground atom as a *fact*. We allow *built-in* predicates to appear in the body of rules. Thus, we allow relation names such as $=,\neq,\leqslant, <, \geqslant$, and $>$. We also allow *aggregate operations* in rule heads in the form $R(\bar{u}, \Lambda < \bar{\mu} >)$ with Λ one of the usual aggregate functions and $< \bar{\mu} >$ defining the grouping of arguments $\bar{\mu}$. In this paper we assume each rule to be *safe*, 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 safe rules. As usual, we refer to $idb(\Pi)$ as the *itensional* part of the database schema, while we refer to the *extensional* schema as $edb(\Pi)$. Given a database schema $\mathbf{R}$, a *database instance* is a finite set $\mathbf{I}$ of facts.

As introductory example, we use the program depicted in Listing 1.1 where we employed an *edb* relation `link` to specify the existence of a link between two nodes. In addition, we employ an intensional relation `path` which is computed starting from the `link` relation (`r1`) and recursively adding a new path when a link exists from `A` to `B` and a path already exists 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$^\neg$

Distributed systems are not static, but evolving with time. Therefore it will be useful to enrich Datalog$^\neg$ with a notion of time. For this purpose we follow the road traced by Dedalus$_0$ [6]. Thus, starting with considering time isomorphic to the set of natural numbers $\mathbb{N}_0$, a new schema $\mathbf{R}^T$ is defined incrementing the arity of each relation $R \in \mathbf{R}$ by one. By convention, the new extra term, called *time suffix*, appears as the last attribute in every relation and has values in $\mathbb{N}_0$. We will sometimes adopt the term *timestamp* to refer to the time suffix value. In fact, each tuple can be viewed as timestamped with the evaluation step in which it is valid. For conciseness we will employ the term *time-step* to denote an evaluation step. By incorporating the time suffix term in the schema definition, we now have multiple instances for each relation, one for each timestamp. In this situation, with $\mathbf{I}[0]$ it is named the *initial database instance* comprising at least all ground atoms existing at the initial time 0, while with $\mathbf{I}[n]$ the instance at time-step n. In accordance with this approach, tuples by default are considered *ephemeral*, i.e., they are valid only for one single time-step. In order to make tuples *persistent* - i.e., once derived, for example at time-step s, they will eventually last for every time-step $t \geq s$ - a new built-in relation `succ` with arity two and ranging over the set of natural numbers is introduced. $\texttt{succ}(x, y)$ is interpreted true if $y = x + 1$. Program rules are then divided in two sets: *inductive* and *deductive*. The former set contains all the rules employed for transferring tuples through time-steps, while the latter encompasses rules that are instantaneous, i.e, local into a single time-step. Some syntactic sugar is adopted to better characterize rules: all time suffixes are omitted together with the `succ` relation, and a `next` suffix is introduced in head relations to characterize inductive rules. For a complete discussion on how to incorporate time in Datalog$^{\neg}$, we refer the reader to [6].

In Listing 1.2 the simple program of the previous section is rewritten in order to introduce the new formalism. Rule `r1` is a *persistency* rule which moves towards time-steps tuples that are not been explicitly deleted. To note that relation *del_P* is a not mandatory *idb* relation which contains all the facts of P that must be deleted (will not appear in P at state $t = s + 1$).

```
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 distributed system model and how communication among nodes is performed. We define a distributed message-passing system to be a non empty finite set $\mathcal{N} = \{id_1, id_2, ..., id_n\}$ of share-nothing nodes joined by bidirectional communication links. Each node identifier has a value in the domain **dom** but, for simplicity, we assume that a node id_i is identified by its subscript i. Thus, in the followings we consider the set $N = \{1, ..., n\}$ of node identifiers, where n is the total number of nodes in the system.

With *adb* we denote a new set of *accessible* relations encompassing all the tables in which either facts are created remotely or they need to be delivered to another node. These relations can be viewed as 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 [12]. This term maintains the identifier of the node to which every new fact inserted into the relation R belongs. Hence, the nature of *adb* relations can be considered twofold: for one perspective they act as normal relations, but from another perspective they are local buffers associated to relations stored in remote nodes. As pointed out in [9,6], modeling communication using relations provides major advantages. For instance, the disordered nature of sets appears particularly appropriate to represent the basic communication channel behavior by which messages are delivered out of order.

Continuing with the same example of previous sections, we can now employ it to actually program a distributed routing protocol. In order to describe the example of Listing 1.3 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 instance, we will have the fact `link(A,B)` in node A instance if a communication link between A and node B actually 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. Inductive and Deductive Rules

The semantics of the program of Listing 1.3 is the same of the previous sections' ones, even though operationally it substantially differs. In fact, in this new version, computation is performed simultaneously on multiple distributed nodes. Communication is achieved through rule `r4` which, informally, specifies that a path from a generic node A to node C exists if there is a link from A to another node B and this last knows that a path exists from B to C.

3.1 Local State, Global State and Runs

In every point in time, each node is in some particular *local state* encapsulating all the information the node is in possess. The local state s_i of a node $i \in N$ can then be defined as a tuple $(\Pi_i, \mathcal{I}_i)$ where Π_i is the finite set of rules composing node i's program, and $\mathcal{I}_i \subseteq \mathbf{I}[n]_i$ is a set of facts belonging to 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 (real) time values to global states. If we assume time values to be isomorphic to the set of natural numbers, we can define the function $r : \mathbb{N} \rightarrow \mathcal{G}$ where $\mathcal{G} = \{S_1 \times ... \times S_n\}$ with S_i be the set of possible local states for node $i \in N$. We refer to the a tuple (r, t) consisting of a run r and a time t as a *point*. If $r(t) = (s_1, ..., s_n)$ is the global state at point (r, t), we define $r_i(t) = s_i$

[7]. A system may have many possible runs, indicating all the possible ways the global state of the system can evolve. 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. We think that this approach is particularly important in our scope of maintaing in our language an high level of declarativity.

4 Reasoning about Knowledge in Distributed Systems

In the model we have developed so far, all computations that a node can accomplish are consequences of its local state. 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 node i, and we will write $g \sim_i g'$ if i has the same local state both in g and g', i.e. $s_i = s'_i$. It has been shown in [7] that a system $\mathcal{S}$ can be viewed as a *Kripke frame.* A Kripke frame is a tuple $\mathcal{F} = (W, \mathcal{K}_1, ..., \mathcal{K}_n)$ where W is a non empty set of *possible worlds* (in our case a set of possible global states) and $\mathcal{K}_i$ with $i \in N$ is a binary relation in $W \times W$ which is intended to capture the *accessibility relation* according to node i: this is, $(w, u) \in \mathcal{K}_i$ if node i consider world u possible given its information in world w. Or, in other words, we want $\mathcal{K}$ to be equivalent to the $\sim$ relation, therefore maintaining the intuition that a node i considers u possible in global state w if they are indistinguishable, i.e., in both global states, node i has the same local state. In order to model this situation, $\mathcal{K}$ must be an *equivalence relation* on $W \times W$.

To map each rule and fact to the global states in which they are true, we define an *interpreted system* Γ as the tuple $(\mathcal{S}, \pi)$ with $\mathcal{S}$ a system over a set of global states $\mathcal{G}$ and π an interpretation function which maps first-order clauses to global states [7]. More formally, we build a structure over the Kripke frame $\mathcal{F}$ in order to map each program Π_i and each ground atom in $\mathcal{I}_i$ to the possible worlds in which they are true. To reach this goal, we define a *Kripke structure* $\mathcal{M} = (\mathcal{F}, U, \pi)$ where $\mathcal{F}$ is a Kripke frame, U is the *Herbrand Universe*, π is a function which maps every possible world to a *Herbrant interpretation* over first-order clauses $\Sigma_{\Pi,\mathbf{I}}$ associated with the rules of the program Π and the input instance $\mathbf{I}$, and $\Pi = \bigcup_{i=1}^{n} \Pi_i$, $\mathbf{I} = \bigcup_{i=1}^{n} \mathbf{I}_i[0]$. To be precise, $\Sigma_{\Pi,\mathbf{I}}$ can be constructed starting from the program Π and translating each rule $\rho \in \Pi$ in its first-order *Horn clause* form. This process creates the set of sentences Σ_{Π}. To get the logical theory $\Sigma_{\Pi,\mathbf{I}}$, starting from Σ_{Π} we add one sentence $R(\bar{u})$ for each fact $R(\bar{u})$ in the instance [2,11]. A valuation v on $\mathcal{M}$ is now a function that assign to each variable a value in U. In our settings both the interpretation and the variables valuation are fixed. This means that $v(x)$ is independent of the state, and a constant c has the same meaning in every state in which exists. Thus, constants and relation symbols in our settings are *rigid designators* [7,14]. Given a Kripke structure $\mathcal{M}$, a world $w \in W$ and a valuation v on $\mathcal{M}$, the *satisfaction relation* $(\mathcal{M}, w, v) \models \psi$ for a formula $\psi \in \Sigma_{\Pi,\mathbf{I}}$ is:

- $(M, w, v) \models R(t_1, ..., t_n)$ with $n = arity(R)$, iff $(v(t_1), ..., v(t_n)) \in \pi(w)(R)$
- $(M, w, v) \models \neg\psi$ iif $(M, w, v) \models \psi$

- $(M, w, v) \models \psi \wedge \phi$ iff $(M, w, v) \models \psi$ and $(M, w, v) \models \phi$
- $(M, w, v) \models \forall \psi$ iif $(M, w, v[x/a]) \models \psi$ for every $a \in U$ with $v[x/a]$ be a substitution of x with a constant a

We use $(M, w) \models \psi$ to denote that $(M, w, v) \models \psi$ for every valuation v. It could be also interesting to know not only whether certain formula ψ is true in a certain world, but also the formulas that are true in all the worlds of W. In particular, a formula ψ is *valid* in a structure M, and we write $M \models \psi$, if $(M, w) \models \psi$ for every world w in W. We say that ψ is valid, and write $\models \psi$, if ψ is valid in all structures. We now introduce the modal operator K_i in order to express what a node i "*knows*", namely which of the sentences in $\Sigma_{\Pi,\mathbf{I}}$ are known by the node i. Given $\psi \in \Sigma_{\Pi,\mathbf{I}}$, a world w in the Kripke structure $\mathcal{M}$, the node i knows ψ - we will write $K_i\psi$ - in world w if ψ is true in all the worlds that i considers possible in w [7]. Formally:

$$(\mathcal{M}, w, v) \models K_i\psi \text{ iff for all } w \text{ such that } (w, u) \in \mathcal{K}_i$$

This definition of knowledge has the following valid properties that are called *S5*:

1. *Distributed Axiom*: $\models (K_i\psi \wedge K_i(\psi \rightarrow \phi)) \rightarrow K_i\phi$
2. *Knowledge Generalization Rule*: For all structures M, if $M \models \psi$ then $M \models K_i\psi$
3. *Truth Axiom*: $\models K_i\psi \rightarrow \psi$
4. *Positive Introspection Axiom*: $\models K_i\psi \rightarrow K_iK_i\psi$
5. *Negative Introspection Axiom*: $\models \neg K_i\psi \rightarrow K_i\neg K_i\psi$

Informally, the first axiom allows us to distribute the epistemic operator K_i over implication; the knowledge generalization rule instead says that if ψ is valid, then so is $K_i\psi$. This rule differ from the formula $\psi \rightarrow K_i\psi$, in the sense that the latter tells that if ψ is true, then node i knows it, but a node does not necessarily know all things that are true. Even though a process may not know all facts that are true, axiom 3 says that if it knows a fact, then it is true. The last two properties say that nodes can do introspection regarding their knowledge: they know what they know and what they do not know [7].

4.1 Incorporating Knowledge: Knowlog^{K}

In the previous section we described how knowledge assumptions can be expressed using first-order Horn clauses representing our program. We can now move back to the rule form. We use symbols $\square$ and $\boxdot$ to denote a (possibly empty) sequence of modal operators K_i, with i specifying a node identifier. Given a sentence in the modal Horn clause form, we use the following statement to express it in a rule form:

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

with $n, m \geq 0$ and each positive literal in the form $\boxdot R$, while negative literals are in the form $\square \boxdot R$.

Definition 1. *The modal context $\Box$ is the sequence - with the maximum length of one - of modal operators K appearing in front of a rule.*

We put some restriction on the sequence of operators permitted in $\boxdot$.

Definition 2. *Given a (possibly empty) sequence of operators $\boxdot$, we say that $\boxdot$ is in restricted form if it does not contain K_iK_i subsequences.*

Definition 3. *A Knowlog^K program is a set of rules in the form (1), containing only (possibly 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.*

Informally speaking, given a Knowlog^K program, with the 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 $K_i \boxdot R$. In order to specify how communication is achieved we define communication rules as follows:

Definition 4. *A communication rule in Knowlog^K is a rule where no modal context is set and body atoms have the form $K_i \boxdot R$ - they are all prefixed with a modal operators pointing to the same node - while the head atom has the form $K_j \boxdot R'$, with $i \neq j$.*

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 node should know, and not *how*.

The Two-Phase-Commit Protocol. Inspired by [4], we implemented the two-phase-commit protocol (2PC) using the epistemic operator K. 2PC is used to execute distributed transaction and it is divided in two phases: in the first phase, called the *voting phase*, a coordinator node submits 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 (a *yes vote* in case it is ready, a *no vote* otherwise). 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 participants. We considerably simplify the 2PC protocol by disregarding failures and timeouts actions, since our goal is not an exhaustive exposition of the 2PC. In addition, we employ some syntactic sugar to have a more clean code: we omit the modal context in each rule, and instead we group rules in *programs* identified by a name and the identifier of the node where the program should be installed. If the program must be installed on multiple nodes, we permit to specify, as location, a relation ranging over node identifiers.

```
    \\Initialization at coordinator
    #Program Initialization @C
r1:   transaction(Tx_id,State)@next:-transaction(Tx_id,State),
        ¬Kcdel_transaction(Tx_id,State).
```

```
r2:   log(Tx_id,State)@next:-log(Tx_id,State).
r3:   part_cnt(count<N>):-participants(N).
r4:   transaction(Tx_id,State):-log(Tx_id,State).
r5:   participants(P1).
r6:   participants(P2).

    \\Initialization at participants
    #Program Initialization @participants
r7:   transaction(Tx_id,State)@next:-transaction(Tx_id,State),
        ¬K_cdel_transaction(Tx_id,State).
r8:   log(Tx_id,State)@next:-log(Tx_id,State).

    \\Decision Phase at coordinator
    #Program DecisionPhase @C
r9:   yes_cnt(Tx_id,count<Part>):-vote(Vote,Tx_id,Part),Vote == "yes").
r10:  log(Tx_id,"commit")@next:-part_cnt(C),yes_cnt(Tx_id,C1),C==C1,
         State=="vote-req",transaction(Tx_id,State).
r11:  log(Tx_id,"abort"):-vote(Vote,Tx_id,Part),Vote == "no",
         transaction(Tx_id,State),State =="vote-req".

    \\Voting Phase at participants
    #Program VotingPhase @participants
r12:  log(Tx_id,"prepare"):-State=="vote-req",K_ctransaction(Tx_id,State).
r13:  log("abort",Tx_id):-log(Tx_id,State),State=="prepare",db_status(Vote),
         Vote=="no".

    \\Decision Phase at participants
    #Program DecisionPhase @participants
r14:  log(Tx_id,"commit"):-log(Tx_id,State_l),State_l=="prepare",
         State_t=="commit",K_ctransaction(Tx_id,State_t).
r15:  log(Tx_id,"abort"):-log(Tx_id,State_l),State_l=="prepare",
         State_t=="abort",K_ctransaction(Tx_id,State_t).

    \\Communication
r16:K_Xtransaction(Tx_id, State):-K_Cparticipants(@X),K_Ctransaction(Tx_id,State).
r17: K_Cvote(Vote,Tx_id,"sub1"):-K_P1log(Tx_id,State),State=="prepare",
       K_P1db_status(Vote).
r18: K_Cvote(Vote,Tx_id,"sub2"):-K_P2log(Tx_id,State),State=="prepare",
       K_P2db_status(Vote).
```

Listing 1.4. Two Phase Commit Protocol

4.2 Incorporating Higher Levels of Knowledge: Knowlog

Rules `r10`, `r11` of Listing 1.4 indicates that each participant, once written `"prepare"` in the log, sends to the coordinator its status together with its identifier. Then, the votes are aggregated at coordinator side and the final decision is issued. This process can also be seen in another way: the coordinator node will deliver `"commit"` for transaction `Tx_id` if it knows that every participant knows the fact `vote("yes",Tx_id)`; `"abort"` otherwise. Consequently, the decision phase at the coordinator will become as in Listing 1.5.

```
r10_a: log(Tx_id,"commit")@next:-Ks1vote("yes",Tx_id),Ks2vote("yes",Tx_id),
       transaction(Tx_id,State),State=="vote-req".
r11_a: log(Tx_id,"abort")@next:-Kxvote(Vote,Tx_id),Vote=="no",
       participants(X),transaction(Tx_id,State),State=="vote-req".
```

Listing 1.5. 2PC coordinator's program revisited

We chose this new revisited form described in Listing 1.5 for a purpose, in fact we want to show that other types of knowledge could be appealing to be incorporated in our language. For example, a node could be interested not only in knowing some fact, but also in knowing if *every* node in the system know something (rule `r1`) or a new information is derived by *combining* the knowledge belonging to different nodes (rule `r2`). The discussion about higher levels of knowledge can be started pointing out that both rules in Listing 1.5 have a common denominator: i.e., the notion of knowledge inside a *group of nodes*. In fact, both the above mentioned rules are used to declare which is the state of knowledge inside the group of participant nodes. Thus, given a non empty set of nodes G, we can hence augment Knowlog^K with modal operators E_G and D_G, which respectively are informally stating that "every node in the group G knows" and "it is distributed knowledge among the nodes in G". From a more operational point of view, $E_G\psi$ states that the sentence ψ is replicated in all the nodes belonging to G, while $D_G\psi$ states that ψ is fragmented among the nodes in G. We can easily extend the definition of satisfiability to handle the two new types of knowledge just introduced. $E_G\psi$ is true exactly if everyone in the group G know ψ:

$$(M, w, v) \models E_G\psi \text{ iff } (M, w, v) \models K_i\psi \text{ for all } i \in G$$

On the other side, a group G has distributed knowledge of ψ if the coalesced knowledge of the members of G implies ψ. This is accomplished by eliminating all worlds that some agent in G considers impossible:

$$(M, w, v) \models D_G\psi \text{ iff } (M, u, v) \models \psi \text{ for all } t \text{ that are } (w, u) \in \bigcap_{i \in G} R_i$$

Not surprisingly, for both E_G and D_G axioms analogous to the Knowledge Axiom, Distribution Axiom, Positive Introspection Axiom, and Negative Introspection Axiom all hold. In addition, distributed knowledge of a group of size one is the same as knowledge, so if G contains only one node i [7]:

$$\models D_{\{i\}}\psi \leftrightarrow K_i\psi$$

and the larger the subgroup, the greater the distributed knowledge of that subgroup is :

$$\models D_G\psi \rightarrow D_{G'}\psi \text{ if } G \subseteq G'$$

Before reformulating the definition of Knowlog and rewriting Listing 1.5 using the new operators, we first update the definition 2 in order to incorporate the new operators D_G and E_G.

Definition 5. *Given a (possibly empty) sequence of operators $\boxdot$, we say that $\boxdot$ is in restricted form if it does not contain either K_iK_i, D_GD_G or $E_{G'}E_{G'}$ subsequences, with i specifying a node identifier, G a group of nodes $G \subseteq N$ and G' is singleton.*

Definition 6. *A Knowlog program is a Knowlog^K program augmented with operators E_G and D_G, with $G \subseteq N$ and where the sequence of operators $\boxdot$ is in the restrict form of definition 5.*

Listing 1.6 shows Knowlog version of Listing 1.5.

```
r10_b: log(Tx_id,"commit")@next:-Exvote("yes",Tx_id),participants(X),
        transaction(Tx_id,State),State=="vote-req".
r11_b: log(Tx_id,"abort")@next:-Dxvote(Vote,Tx_id),Vote=="no",
        participants(X),transaction(Tx_id,State),State=="vote-req".
```

Listing 1.6. Knowlog 2PC coordinator's program

Operationally, E_G is used when we want that a fact, to be considered true, is correctly replicated in every node $i \in G$. On the other side, D_G is employed when facts that are fragmented inside multiple relations distributed in the node enclosed in G must be assembled in one place for computation. Employing the E_G operator in the head of communication rules we are able to express the sending of a message to multiple destinations, therefore emulating the multicast primitive behavior.

Definition 7. *A communication rule in Knowlog is a Knowlog^K communication rule where the body may contains atoms both in the form $D_G \boxdot R$ and $E_G \boxdot R$, while head atoms may also have the form $E_G \boxdot R$, with R a relation and $G \subseteq N$.*

As a future work we will investigate how the operator D can be used in front of communication rule to implement data dissemination [15].

5 Knowlog Semantics

The first step towards the definition of the Knowlog semantics will be the specification of the *reified* version of Knowlog. For this purpose, we augment **dom** with a new set of constants $\triangle$ which will encompass the modal operators symbols. We also assume a new set of variables O that will range over the just defined set of modal operator elements. We then construct $\mathbf{R}^{TK}$ adding to each relation $R \in \mathbf{R}^T$ a new term called *knowledge accumulator* and a new set of build-in relations K, D, E and $\oplus$. A tuple over the $\mathbf{R}^{TK}$ schema will have the form $(k, t_1, ..., t_n, s)$ where $k \in O \cup \triangle$ identify the knowledge accumulator term, $s \in S \cup \mathbb{N}$ and $t_1, ..., t_n \in \mathbf{var} \cup \mathbf{dom}$. Conversely, a tuple over *adb* relations, i.e., relations in the head of at least one communication rule, will have the form $(k, l, t_1, ..., t_n, s)$, with l the location specifier term. If the knowledge operator used in front of a non-*adb* relation is a constant, i.e. K_SK_R`input("value")`, the reified version will be `input(Y,"value",n),Y =` $K_S \oplus K_R$ for example at time-step n. The operator $\oplus$ is hence employed to concatenate epistemic operators.

Instead, in case the operator employes a variable to identify a particular node (as in rule `r10_a` of Listing 1.5) or a set of nodes (as shown in rule `r10_b` of Listing 1.6), we need to introduce in $\mathbf{R}^{TK}$ relations `K(X,Y)`, `E(<X>,Y)`, and `D(<X>,Y)` in order to help us in the effort of building the knowledge accumulator term. The first term of the `K` relation is a node identifier $i \in N$ (respectively a set of node identifiers for relations `E`, and `D`) and the `Y` term is a value in $\triangle$ determined by the relation name and the node identifier(s). So for example, the reified version of E_X`vote("yes",Tx_id),participants(X)` will be `E(<X>,Y),vote(Y,"yes",Tx_id,n),participants(X)`.

For what concern communication rules, the process is the same as above, but this time we have to fill also the location specifier field of the head-relation. To accomplish this, if the head relation $R \in adb$ is in the form $K_i\Box R(t_1,...,t_n)$, the reified version will be $R(K_i\Box, i, t_1, ..., t_n, s)$. On the other side, if the head relation is in the form $E_G\Box R(t_1,...,t_n)$ the rule that includes it, is rewritten in m rules, one for each node identifier in G, and each of them having the head relation in the form $K_i\Box R(t_1,...,t_n)$ with $i \in G$. The reified version is then computed as described above. Using this semantics, nodes are able to communicate using the mechanism described in Section 3.

5.1 Operational Semantics

Given as input a Knowlog program Π in the reified version, first Π is separated in two subset: Π^l containing local rules (informally these are the rules that the local nodes i knows) and Π^r containing rules that must be installed in remote nodes. These last are rules having as a modal context K_j with $j, i \in N$, i the identifier of the local node and $i \neq j$. Following the delegation approach illustrated in [1] given a program Π_i local to node $i \in N$, we denote with Π^r_{ij} the remote rules in i related to node j and with $\Pi_j \leftarrow \Pi^r_{ij}$ the action of installing Π^r_{ij} in j's program. For what concern the evaluation of local rules, we partition the local program Π^l in inductive and deductive rule sets, respectively Π^i and Π^d. Then a pre-processing step orders the deductive rules following the dependency graph stratification. After this pre-processing step, the model $\mathcal{M}_\Pi$ is computed. In order to evaluate the stratified program Π^d we use the semi-naive algorithm depicted in [16]. To correctly evaluate rules and facts with modal operators, the *saturation (Sat)* and the *normalization* (*Norm*) operators are used to assist the immediate consequence operators T_{Π^d} [14]. This because Knowlog facts and rules are labeled by modal operators and therefore the immediate consequence operator must be enhanced in order to be employed in our context. More precisely, given a Knowlog instance $\mathbf{I}$ as input, $Sat(\mathbf{I})$ saturate facts in the instances following operators' properties. Lastly, the *Norm* operator converts to restricted form the modal operators in $T_{\Pi^d}(Sat(\mathbf{I}))$.

6 Conclusion and Future Work

We have presented Knowlog, a programming language for distributed systems based on Datalog$^{\neg}$ leveraged with a notion of time and modal operators. We described the communication and knowledge model behind Knowlog and we introduce as example, an implementation of the two phase commit protocol. As a future work, we will incorporate in Knowlog the *common knowledge* operator that has been proven to be linked to concepts such as coordination, agreement and consistency [7]. The successive step will be the definition in Knowlog of weaken forms of common knowledge such as *eventual* common knowledge.

References

1. Abiteboul, S., Bienvenu, M., Galland, A., Antoine, E.: A rule-based language for web data management. In: PODS 2011, pp. 293–304. ACM, New York (2011)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
3. Alvaro, P., Conway, N., Hellerstein, J., Marczak, W.R.: Consistency analysis in bloom: a calm and collected approach. In: CIDR, pp. 249–260 (2011)
4. Alvaro, P., Condie, T., Conway, N., Hellerstein, J.M., Sears, R.: I do declare: consensus in a logic language. Operating Systems Review 43(4), 25–30 (2009)
5. Birman, K., Chockler, G., van Renesse, R.: Toward a cloud computing research agenda. SIGACT News 40(2), 68–80 (2009)
6. Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in Time and Space. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 262–281. Springer, Heidelberg (2011)
7. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (2003)
8. Furukawa, K.: Logic programming as the integrator of the fifth generation computer systems project. Commun. ACM 35(3), 82–92 (1992)
9. Hellerstein, J.M.: The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Rec. 39, 5–19 (2010)
10. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16, 872–923 (1994)
11. Lloyd, J.: Foundations of logic programming. Symbolic Computation: Artificial Intelligence. Springer (1987)
12. Loo, B.T., Condie, T., Garofalakis, M., Gay, et al.: Declarative networking: language, execution and optimization. In: SIGMOD 2006, pp. 97–108. ACM, New York (2006)
13. Ludäscher, B.: Integration of Active and Deductive Database Rules. DISDBIS, vol. 45. Infix Verlag, St. Augustin (1998)
14. Nguyen, L.A.: Foundations of modal deductive databases. Fundam. Inf. 79, 85–135 (2007)
15. Suri, N., Benincasa, G., Choy, S., Formaggi, S., Gilioli, M., Interlandi, M., Rota, S.: Disservice: a peer-to-peer disruption tolerant dissemination service. In: Proceedings of the 28th IEEE Conference on Military Communications, MILCOM 2009, pp. 2514–2521. IEEE Press, Piscataway (2009)
16. Zaniolo, C.: Advanced database systems. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers (1997)

Declarative Datalog Debugging for Mere Mortals

Sven Köhler[1], Bertram Ludäscher[1], and Yannis Smaragdakis[2]

[1] Department of Computer Science, University of California, Davis
{svkoehler,ludaesch}@ucdavis.edu
[2] LogicBlox, Inc., Atlanta, GA and Univ. of Athens, Greece
yannis.smaragdakis@logicblox.com

Abstract. Tracing why a "faulty" fact A is in the model $M = P(I)$ of program P on input I quickly gets tedious, even for small examples. We propose a simple method for debugging and "logically profiling" P by generating a provenance-enriched rewriting $\hat{P}$, which records rule firings according to the logical semantics. The resulting provenance graph can be easily queried and analyzed using a set of predefined and ad-hoc queries. We have prototypically implemented our approach for two different Datalog engines (DLV and LogicBlox), demonstrating the simplicity, effectiveness, and system-independent nature of our method.

1 Introduction

Developing declarative, rule-based programs can be surprisingly difficult in practice, despite (or because of) their declarative semantics. Possible reasons include what Kunen long-ago called *the PhD effect* [10], i.e., that a PhD in logic seems necessary to understand the meaning of certain logic programs (with negation). Similarly, *An Amateur's Introduction to Recursive Query Processing* [2] from the early days of deductive databases, rather seems to be for experts only. So what is the situation now, decades later, for a brave, aspiring Datalog 2.0 programmer who wants to develop complex programs?

The meaning and termination behavior of a *Prolog* program P depends on, among other things, the order of rules in P, the order of subgoals within rules, and even (apparently minor) updates to base facts. Consider, e.g., the program for computing the transitive closure of a directed graph, i.e., $P_{\mathsf{tc}} =$

$$\begin{aligned} r_1\colon\quad & \mathsf{tc}(X,Y) \;\text{:-}\; \mathsf{e}(X,Y). \\ r_2\colon\quad & \mathsf{tc}(X,Z) \;\text{:-}\; \mathsf{e}(X,Y), \mathsf{tc}(Y,Z). \end{aligned}$$

Seasoned logic programmers know that P_{tc} is *not* a correct way to compute the transitive closure in Prolog.[1] Under a more declarative *Datalog* semantics, on the other hand, P_{tc} indeed *is* correct, since the result does *not* depend on rule or subgoal order. The flip side, however, is that effective and practically useful

[1] For $I = \{\mathsf{e}(\mathsf{a},\mathsf{b}), \mathsf{e}(\mathsf{b},\mathsf{a})\}$ the query ?-tc(c,X) correctly returns "No", while the similar ?-tc(X,c) will not terminate! Prolog's behavior gets worse when swapping rules r_1 and r_2, or when using left- or doubly-recursive variants P^l_{tc}, P^d_{tc}, respectively.

procedural debugging techniques for Prolog, based on the *box model* [19], are not available in Datalog. Instead, new debugging techniques are needed that are solely based on the declarative reading of rules. In this paper, we develop such a framework for declarative debugging and logic profiling.

Let $M = P(I)$ be the model of P on input I. Bugs in P (or I) manifest themselves through unexpected answers (ground atoms) $A \in M$, or expected but missing $A \notin M$. The key idea of our approach is to rewrite P into a *provenance-enriched* program $\hat{P}$, which records the derivation history of $M = P(I)$ in an extended model $\hat{M} = \hat{P}(I)$. A provenance graph G is then extracted from $\hat{M}$, which the user can explore further via predefined views and ad-hoc queries.

Use Cases Overview. Given an IDB atom A, our approach allows to answer questions such as the following: What is the *data lineage* of A, i.e., the set of EDB facts that were used in a derivation of A, and what is the *rule lineage*, i.e., the set of rules used to derive A? When chasing a bug or trying to locate a source of inefficiency, a user can explore further details: What is the graph structure G_A of all derivations of A? What is the *length* of A, i.e., of shortest derivations, and what is the *weight*, i.e., number of simple derivations (proof trees) of A?

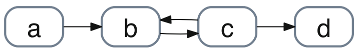
For another example, assume the user encounters two "suspicous" atoms A and B. It is easy to compute the *common lineage* $G_{AB} = G_A \cap G_B$ shared by A and B, or the *lowest common ancestors* of A and B, i.e., the rule firings and ground atoms that occur "closest" to A and B in G_{AB}, thus triangulating possible sources of error, somewhat similar to ideas used in delta debugging [22].

Since nodes in G_A are associated with relation symbols and rules, a user might also want to compute other aggregates, i.e., not only at the level of G_A (ground atoms and firings), but at the level of (non-ground) rules and relation symbols, respectively. Through this *schema-level profiling*, a user can quickly find the hot (cold) spots in P, e.g., rules having the most (least) number of firings.

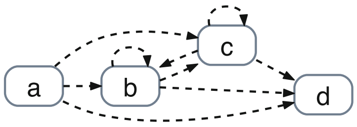
Running Example. Figure 1 gives an overview using a very simple example: (a) depicts an input graph e, while (b) shows its transitive closure $\mathsf{tc} := \mathsf{e}^+$. The structure and number of distinct derivations of tc atoms from base edges in e can be very different, e.g., when comparing the right-recursive P_{tc} $(=P^r_{\mathsf{tc}})$ above, with left-recursive or doubly-recursive variants P^l_{tc} or P^d_{tc}, respectively.

The provenance graph G (or the relevant subgraph $G_A \subseteq G$, given a goal A) provides crucial information to answer the above use cases. Fig. 1(c) shows the provenance graph for the computation of tc via P^r_{tc} from above. Box nodes represent *rule firings*, i.e., individual applications of the *immediate consequence operator* T_P, and connect all body atoms to the head atom via a unique firing node. For the debug goal $A = \mathsf{tc}(\mathsf{a}, \mathsf{b})$ the subgraph G_A, capturing all possible derivations of A, is highlighted (through filled nodes and bold edges).

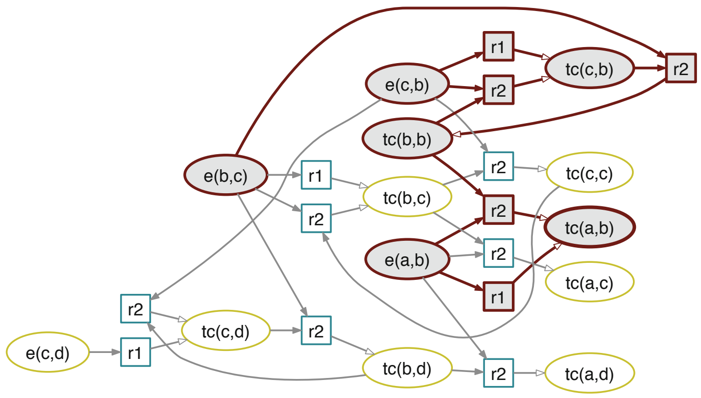
Overview and Contributions. We present a simple method for debugging and logically profiling a Datalog program P via a provenance-enriched rewriting $\hat{P}$. The key idea is to extract from the extended model $\hat{M}$ a provenance graph G which is then queried, analyzed, and visualized by the user. Given a debug goal A, relevant subgraphs G_A can be obtained easily and further analyzed via a library of common debug views and ad-hoc user queries. At the core of our approach



(a) Input graph: edge relation e



(b) Output: transitive closure $\mathsf{tc} = \mathsf{e}^+$



(c) Provenance graph G; highlighted subgraph $G_{\mathsf{tc(a,b)}}$; firing nodes r (boxes), atom nodes A (ovals); edge types $A \overset{\mathsf{in}}{\to} r$ and $r \overset{\mathsf{out}}{\to} A$ (edge labels not shown)

Fig. 1. P^r_{tc}-provenance graph for input e, with derivations of tc(a,b) highlighted in (c)

are rewritings that (i) capture rule firings, then (ii) reify them, i.e., turn them into nodes in G (via Skolem functions), while (iii) keeping track of derivation lengths using Statelog [11], a Datalog variant with states.[2] The rewritten Statelog program $\hat{P}$ is state-stratified [13] and has PTIME *data complexity*. We view the simplicity and system-independence as an important benefit of our approach. We have rapidly prototyped this approach for rather different Datalog engines, i.e., DLV [12] and LogicBlox [14], and are currently developing improved versions. We also note a close relationship of our provenance graphs with *provenance semirings* [8] (a detailed account is beyond the scope of this paper). Here our focus is on presenting a simple, effective method for debugging and profiling declarative rules for "mere mortals".

2 Provenance Rewritings for Datalog

In this section, we present three Datalog rewritings $P \overset{F}{\leadsto} \cdot \overset{G}{\leadsto} \cdot \overset{S}{\leadsto} \hat{P}$ for capturing rule firings, graph generation, and Statelog evaluation, respectively. We assume the reader is familiar with Datalog (e.g., see [1, 17]); the resulting Statelog program $\hat{P}$ has PTIME data complexity and involves a limited (i.e., safe) form of Skolem functions and state terms [13]. In the sequel, $\bar{X} = X_1, \ldots, X_n$ denotes a variable vector; lower case terms $a, b, \ldots$ denote constants.

[2] Other state-oriented Datalog extensions include Datalog$_{nS}$ [6] and XY-Datalog [21].

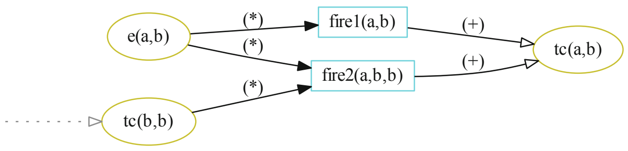


Fig. 2. Subgraph with two rule firings $\mathsf{fire}_1(\mathsf{a},\mathsf{b})$ and $\mathsf{fire}_2(\mathsf{a},\mathsf{b},\mathsf{b})$, both deriving $\mathsf{tc}(\mathsf{a},\mathsf{b})$

2.1 Recording Rule Firings: $P \overset{F}{\rightsquigarrow} P^F$

The first rewriting (cf. Green et al. [9]), captures the provenance of rule firings. Let r be a unique identifier of a rule in P. We assume r to be *safe*, i.e., every variable in r must also occur positively in the body:

$$r: \quad H(\bar{Y}) \text{ :- } B_1(\bar{X}_1), \ldots, B_n(\bar{X}_n)$$

Let $\bar{X} := \bigcup_i \bar{X}_i$ include all variables in r, ordered, e.g., by occurrence in the body. Since r is safe, $\bar{Y} \subseteq \bar{X}$, i.e., the head variables are among the $\bar{X}$. The rule r is now replaced by two new rules in the rewritten program P^F:

$$\begin{aligned} r_{in}: \mathsf{fire}_r(\bar{X}) &\text{ :- } B_1(\bar{X}_1), \ldots, B_n(\bar{X}_n) \\ r_{out}: \quad H(\bar{Y}) &\text{ :- } \mathsf{fire}_r(\bar{X}) \end{aligned}$$

Thus P^F records, for each r-satisfying instance $\bar{x}$ of $\bar{X}$, a unique fact: $\mathsf{fire}_r(\bar{x})$.

2.2 Graph Reification of Firings: $P^F \overset{G}{\rightsquigarrow} P^G$

To facilitate querying the results of the previous step, we *reify* ground atoms and firings as nodes in a *labeled provenance graph* G. For each pair of rules r_{in}, r_{out} above, we add n rules ($i = 1, \ldots, n$) to generate the in-labeled edges in G:

$$\mathsf{g}(\ B_i(\bar{X}_i),\ \mathsf{in},\ \mathsf{fire}_r(\bar{X})\ ) \text{ :- } \mathsf{fire}_r(\bar{X})$$

and one more rule for generating out-labeled edges in G as well:

$$\mathsf{g}(\ \mathsf{fire}_r(\bar{X}),\ \mathsf{out},\ H(\bar{Y})\ ) \text{ :- } \mathsf{fire}_r(\bar{X})$$

Note the safe use of atoms as *Skolem terms* in the rule heads: for finitely many rule firings $\mathsf{fire}_r(\bar{x})$, we obtain a finite number of in- and out-edges in G.

Example. After applying both transformations $\cdot \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot$ to P_{tc} from above, the rewritten program P^G_{tc} can be executed, yielding a directed graph with labeled edges $\mathsf{g}(v_1, \ell, v_2)$ in the enriched model $\hat{M}$. Figure 2 shows a subgraph with two rule firings, both deriving the atom $\mathsf{tc}(\mathsf{a},\mathsf{b})$. Oval (yellow) nodes represent atoms A and boxed (blue) nodes represent firings F. Arrows with solid heads and label (*) are in-edges, while those with empty heads and label (+), represent out-edges. Note that according to the declarative semantics, in (out) edges, model logical conjunction "$\wedge$" (logical disjunction "$\vee$"), respectively. Thus, w.r.t. their incoming edges, boxed nodes are AND-nodes, while oval nodes are OR-nodes.[3]

[3] In semiring parlance, they are product "$\otimes$" and sum "$\oplus$" nodes, respectively.

2.3 Statelog Rewriting: $P^G \overset{S}{\rightsquigarrow} P^S$

Statelog [11, 13] is a state-oriented Datalog variant for expressing active update rules and declarative rules in a unified framework. The next rewriting simulates a Statelog derivation in Datalog via a limited (safe) form of "state-generation". The key idea is to keep track of the firing rounds $I_{n+1} := T_P(I_n)$ of the T_P operator ($I_0 := I$ is the input database). This provides a simple yet powerful means to detect tuple rederivations, to identify unfounded derivations, etc.

First, replace all rules r_{in}, r_{out} above with their state-oriented counterparts:

$$\begin{aligned} r_{in} : \mathsf{fire}_r(\mathsf{S1}, \bar{X}) \ &{:-}\ B_1(\mathsf{S}, \bar{X}_1), \ldots, B_n(\mathsf{S}, \bar{X}_n),\ \mathsf{next}(\mathsf{S}, \mathsf{S1}). \\ r_{out} :\quad H(\mathsf{S}, \bar{Y}) \ &{:-}\ \mathsf{fire}_r(\mathsf{S}, \bar{X}). \end{aligned}$$

The goal $\mathsf{next}(\mathsf{S}, \mathsf{S1})$ is used for the safe generation of new states: The next state $s{+}1$ is generated only if at least one atom A was new in state s :

$$\begin{aligned} \mathsf{next}(0, 1) \ &{:-}\ \mathsf{true}. \\ \mathsf{next}(\mathsf{S}, \mathsf{S1}) \ &{:-}\ \mathsf{next}(\_, \mathsf{S}),\ \mathsf{new}(\mathsf{S}, A),\ \mathsf{S1} := \mathsf{S} + 1. \end{aligned}$$

An atom A is newly derived if it is true in $s{+}1$, but not in the previous state s:

$$\mathsf{newAtom}(\mathsf{S1}, A) \ {:-}\ \mathsf{next}(\mathsf{S}, \mathsf{S1}),\ \mathsf{g}(\mathsf{S1}, \_, \mathsf{out}, A),\ \neg\, \mathsf{g}(\mathsf{S}, \_, \mathsf{out}, A).$$

Similarly, rule firing F is new if it is true in $s{+}1$, but not previously in s:

$$\mathsf{newFiring}(\mathsf{S1}, F) \ {:-}\ \mathsf{next}(\mathsf{S}, \mathsf{S1}),\ \mathsf{g}(\mathsf{S1}, F, \mathsf{out}, \_),\ \neg\, \mathsf{g}(\mathsf{S}, F, \mathsf{out}, \_).$$

The n rules for generating in-edges are replaced with state-oriented versions:

$$\mathsf{g}(\ \mathsf{S},\ B_i(\bar{X}_i),\ \mathsf{in},\ \mathsf{fire}_r(\bar{X})\ ) \ {:-}\ \mathsf{fire}_r(\mathsf{S},\ \bar{X})$$

and similarly, for the out-edge generating rules:

$$\mathsf{g}(\ \mathsf{S},\ \mathsf{fire}_r(\bar{X}),\ \mathsf{out},\ H(\bar{Y})\ ) \ {:-}\ \mathsf{fire}_r(\mathsf{S},\ \bar{X}).$$

It is not difficult to see that the above rules are state-stratified (a form of local stratification) and that the resulting program terminates after polynomially many steps [13]: When no more new atoms (or firings) are derived in a state, then the above rules for next can no longer generate new states, thus in turn preventing rules of type r_{in} from generating new $\mathsf{fire}_r(\mathsf{S1}, \bar{X})$ atoms.

Example. When applying the transformations $\cdot \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot \overset{S}{\rightsquigarrow} \cdot$ to the transitive closure program P^r_{tc}, a 4-ary graph g is created, with the additional T_p-round counter in the first (state) argument position. Figure 3 shows the graphical representation of g for our running example (observe the cycle in g, caused by the cycle in the input e).

3 Debugging and Profiling Using Provenance Graphs

When debugging and profiling Datalog programs we typically employ all program transformations $P \overset{F}{\rightsquigarrow} \cdot \overset{G}{\rightsquigarrow} \cdot \overset{S}{\rightsquigarrow} \hat{P}$, i.e., the enriched model $\hat{M}$ contains the full provenance graph relation g with state annotations.

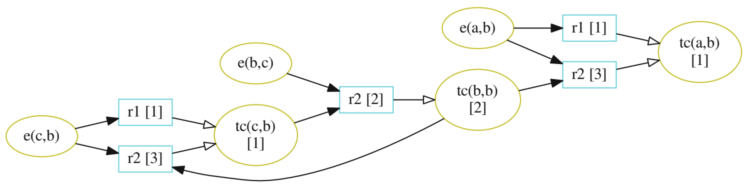


Fig. 3. State-annotated provenance graph g for the derivation of tc(a,b). Annotations [in brackets] show the firing round (i.e., state number) in which an atom was first derived. To avoid clutter, we often depict firing nodes without their variable bindings.

3.1 Debugging Declarative Rules

If a Datalog program does not compute the expected model, it is very helpful to understand how the model was derived, focusing in particular on certain goal atoms during the debugging process. Since relation g captures all possible derivations of the given program, we can define various queries and views on g to support debugging.

Provenance Graph. The complete description of how a program was evaluated can be derived by just visualizing the whole provenance graph g, optionally removing the state argument through projection:

```
ProvGraph(X,L,Y) :− g(_,X,L,Y).
```

Figure 1(c) shows the provenance graph for a transitive closure computation. One can easily see how all transitive edges were derived and—by following the edges backwards—one can see on which EDB facts and rules each edge depends.

Provenance Views. Since provenance graphs are large in practice, it is often desirable to just visualize subgraphs of interest. The following debug view returns all "upstream" edges, i.e., the provenance subgraph relevant for debug atom Q:

```
ProvView(Q,X,out,Q) :− ProvGraph(X,out,Q).
ProvView(Q,X,L,Y) :− ProvView(Q,Y,_,_), ProvGraph(X,L,Y).
```

Figure 3 shows the result of this view for the debug goal $Q = \mathsf{tc}(\mathsf{a}, \mathsf{b})$; the large (goal-irrelevant) remainder of the graph is excluded. Fig. 1(c), in contrast, shows the same query but now in the context of the whole provenance graph.

Computing the Length of Derivations. A typical question during debugging is when and from which other facts a debug goal was derived. Such temporal questions can be explained using a Statelog rewriting of the program. We annotate atoms and firings with a *length* attribute to record in which round they were first derived. The length is defined as follows:

$$\mathsf{len}(F) := 1 + \max\{\ \mathsf{len}(A) \mid (A \overset{\mathsf{in}}{\to} F) \in \mathsf{g}\ \}\ ; \quad \text{if } F \text{ is a firing node}$$

$$\mathsf{len}(A) := \begin{cases} \min\{\ \mathsf{len}(F) \mid (F \overset{\mathsf{out}}{\to} A) \in \mathsf{g}\ \} & ; \quad \text{if } A \text{ is an IDB atom} \\ 0 & ; \quad \text{if } A \text{ is an EDB atom} \end{cases}$$

A rule firing F can only succeed one round after the *last* body atom (i.e., having maximal length) has been derived. Conversely, the length of an atom A is determined by its *first* derivation (i.e., having minimal length). The Statelog rewriting captures evaluation rounds, so the state associated with a *new* firing determines the length of the firing:

```
len(F,LenF) :− newFiring(S,F), LenF=S.
```

Similarly, the length of an atom is equal to the first round it was derived:

```
len(A,LenA) :− newAtom(S,A), LenA=S.
```

Fig. 4 shows a provenance graph with such length annotations.

3.2 Logic-Based Profiling

There are multiple ways to write a Datalog program that computes a desired query result and the performance of these programs may vary significantly. For example, consider an EDB with a linear graph e having 10 nodes. In addition to the right-recursive program P^r_{tc}, we consider the doubly-recursive variant P^d_{tc} with the rules: (1) $\mathsf{tc}(X, Y) :- \mathsf{e}(X, Y)$ and (2) $\mathsf{tc}(X, Y) :- \mathsf{tc}(X, Z), \mathsf{tc}(Z, Y)$. When computing tc, the two programs perform differently and we would like to find the cause. In this section, we present queries for profiling measures of Datalog programs that can help to answer such questions.

Counting Facts. When evaluating a Datalog program on an input EDB, a number of IDB atoms are derived. We assume here that the resulting model only contains desired facts, i.e., the program was already debugged with the methods described earlier. We can use, e.g., the number of derived IDB atoms as a baseline for profiling a program. It can be computed easily via aggregation:

```
DerivedFact(H) :− ProvGraph(_,out,H).
DerivedHeadCount(C) :− C = count{ H : DerivedFact(H) }.
```

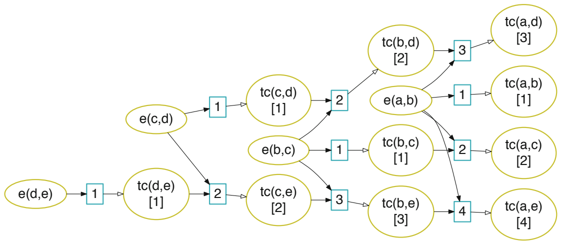
Both P^r_{tc} and P^d_{tc} derive 45 facts for our small graph example, which is exactly the number of transitive edges we are looking for.

Counting Firings. An important measure in declarative profiling is the number of rule firings needed to produce the final model. It can be computed from the out-edges and another simple aggregation:

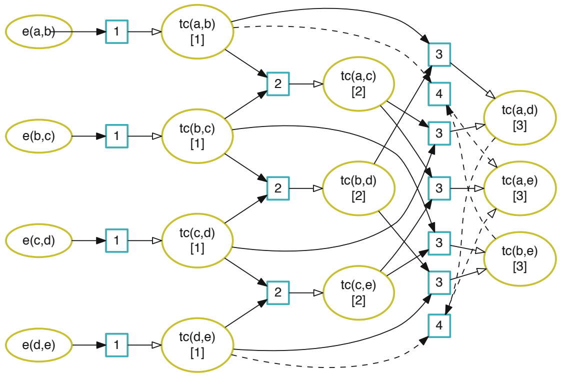
```
Firing(F) :− ProvGraph(F,out,_).
FiringCount(C) :− C = count{ F : Firing(F) }.
```

Using this measure we can see a clear difference between the two variants of the transitive closure program. While the right-recursive program P^r_{tc} uses 45 rule firings to compute the model, the doubly-recursive variant P^d_{tc} causes 129 rule firings to derive the same 45 transitive edges. The reason is that P^d_{tc} will use all combinations of edges to derive a fact, while P^r_{tc} extends paths only in one direction, one edge at a time.

For better readability, Figure 4 shows the provenance graph for a smaller input graph consisting of a 5-node linear chain. Nodes are annotated with their



(a) P^r_{tc}: right-recursive



(b) P^d_{tc}: doubly-recursive

Fig. 4. Provenance graphs with annotations for profiling P^r_{tc} and P^d_{tc} on a 5-node linear graph P^d_{tc} causes more rule firings than P^r_{tc} and also derives facts in multiple ways. Numbers denote $\mathsf{len}(F)$ (in firing nodes) and $\mathsf{len}(A)$ (in atom nodes), respectively.

length, i.e., earliest possible derivation round. Note how some atom nodes in the graph for P^d_{tc} in Fig. 4(b) have more incoming edges (and derivations) than the corresponding nodes in the P^r_{tc} variant shown in Fig. 4(a).

Computing the Maximum Round. Another measure is the number of states (T_P rounds), needed to derive all conclusions. From the rewriting P^S, we can simply determine the final state:

$$\mathsf{MaxRound(MR)} :- \mathsf{MR} = \mathsf{max}\{ \mathsf{S} : \mathsf{g(S,\_,\_,\_)} \}.$$

This measure shows another clear difference between P^r_{tc} and P^d_{tc}: While P^r_{tc} requires 10 rounds to compute all transitive edges in our sample graph, P^d_{tc} only needs 6 rounds. Generally, the doubly-recursive variant requires significantly fewer rounds, i.e., logarithmic in the size of the longest simple path in the graph versus linear for the right-recursive implementation.

Counting Rederivations. To analyze the number of derivations in more detail, we can use the Statelog rewriting P^S to also capture temporals aspects. With each application of the T_P operator, some facts might be rederived. Note that, if a fact is derived via different variable bindings in the body of a rule (or different rules), the rederivation is captured already in firings. Rederivations occurring during the fixpoint computation can be captured using the Statelog rewriting:

```
ReDerivation(S,F) :– g(S,F,out,A), len(A,LenA), LenA < S.
ReDerivationCount(S,C) :– C = count{ F : ReDerivation(S,F) }.
ReDerivationTotal(T) :– T = sum{ C : ReDerivationCount(S,C) }.
```

When comparing the rederivation counts, the difference between the P_{tc} variants is illuminated further. P^r_{tc} rederives facts 285 times until the fixpoint is reached. The double-recursive program P^d_{tc} causes 325 rederivations.

Schema-Level Profiling. We can easily determine how many new facts per *relation* are used in each round to derive new facts:

```
FactsInRound(S,R,A) :– g(S,A,in,_), RelationName(A,R).
FactsInRound(S1,R,A) :– g(S,_,out,A), next(S,S1), RelationName(A,R).
NewFacts(S,R,A) :– g(S,_,out,A), ¬ FactsInRound(S,R,A), RelationName(A,R).
NewFactsCount(S,R,C) :– C = count{ A : NewFacts(S,R,A) }.
```

This allows us to "plot" the temporal evolution for our result relation $\mathsf{R}(=\mathsf{tc})$: For P^r_{tc} and $\mathsf{S} = 1, \ldots, 9$ the counts are $\mathsf{C} = 9, 8, 7, 6, 5, 4, 3, 2, 1$, while for P^r_{tc} and $\mathsf{S} = 1, \ldots, 5$ we have $\mathsf{C} = 9, 8, 13, 14, 1$. Although P^d_{tc} requires fewer rounds than P^r_{tc}, its new fact count mostly increases over time, while for P^r_{tc}, it decreases.

Confronting the Real-World. In practical implementations, the doubly-recursive version P^d_{tc} has horrible performance. For a representative, realistic graph[4] with 1710 nodes and 3936 edges, the right-recursive P^r_{tc} runs in 2.6 sec, while the doubly-recursive P^d_{tc} takes 15.4 sec. Our metrics can easily explain the discrepancy. The tc-fact count for both versions is 304,000, but the rule firing count varies widely. Using our program rewriting and the profiling view FiringCount(C), we discover that P^d_{tc} has over 64 million different rule firings, while P^r_{tc} has under 566 thousand. One reason is that the doubly-recursive rule $\mathsf{tc}(X,Y)$:– $\mathsf{tc}(X,Z), \mathsf{tc}(Z,Y)$ derives the same $\mathsf{tc}(X,Y)$ fact many times over. This practical example also illustrates the burden of declarative debugging: Adding the profiling calculation to the right-recursive version, P^r_{tc} only grows the running time slightly, to 3 sec. Adding it to the doubly-recursive P^d_{tc} however, yields a running time of 51.3 sec. This is due to the cost of storing more than 64 million combinations of variables for rule firings.

4 Gpad Prototype Implementations

By design, our method of provenance-based debugging and profiling only relies on the declarative reading of rules, i.e., is agnostic about implementation details or evaluation techniques specific to the underlying Datalog engine. Indeed, parallel with the development of the method, we have implemented two incarnations of

[4] The specifics are secondary to our argument, but we list them for completeness. The graph is the application-level call-graph and edges indicating whether a method can call another) for the `pmd` program from the DaCapo benchmark suite, as produced by a precise low-level program analysis. Timings are on a quad-core Xeon E5530 2.4GHz 64-bit machine (only one thread was active at a time) with plentiful RAM (24GB) for the analysis, using LogicBlox Datalog ver. 3.7.10.

a ***G**raph-based **P**rovenance **A**nalyzer and **D**ebugger*, i.e., prototypes GPAD/DLV and GPAD/LB, for declarative debugging with the DLV [12] and LogicBlox [14] engines, respectively. Both prototypes "wrap" the underlying Datalog engine, and outsource some processing aspects to a host language.

For example, GPAD/DLV uses SWI-PROLOG [20] as a "glue" to automate (1) rule rewritings, (2) invocation of DLV, followed by (3) result post-processing, and (4) result visualization using Graphviz. We are actively developing GPAD further and plan a public release in the near future.

5 Related Work

Work on declarative debugging, in particular in the form of *algorithmic debugging* goes as far back as the 1980's [18, 7]. Algorithmic debugging is an interactive process where the user is asked to differentiate between the actual model of the (presumeably buggy) program and the user's intended model. Based on the user's input, the system then tries to locate the faulty rules in an interactive session. Our approach differs in a number of aspects. First, algorithmic debugging is usually based on a specific operational semantics, i.e., SLDNF resolution, a top-down, left-to-right strategy with backtracking and negation-as-failure, which differs significantly from the declarative Datalog semantics (cf. Section 1). Moreover, while our approach is applicable, in principle, in an interactive way, this suggests a tighter coupling between the debugger and the underlying rule engine. In contrast, our approach and its GPAD implementations do not require such tight coupling, but instead treat the rule engine as a black box. In this way, debugging becomes a post-mortem analysis of the provenance-enriched model $\hat{M} = \hat{P}(I)$ via simple yet powerful graph queries and aggregations.

Another approach, more closely related to ours, is the Datalog debugger [3], developed for the DES system. Unlike prior work, and similar to ours, they do not view derivations as SLD proof trees, but rather use a *computation graph*, similar to our labeled provenance graph. Our approach differs in a number of ways, e.g., our reification of derivations in a labeled graph allows us to use regular path queries to navigate the provenance graph, locate (least) common ancestors of buggy atoms, etc. Another difference is our use of Statelog for keeping track of derivation rounds, which facilitates profiling of the model computation over time (per firing round, identify the rules fired, the number of (re-)derivations per atom or relation, etc.) Recent related work also includes work on trace visualization for ASP [4], step-by-step execution of ASP programs [15], and an integrated debugging environment for DLV [16].

Debugging and Provenance. Chiticariu et al. [5] present a tool for debugging database schema mappings. They focus on the computation of derivation routes from source facts to a target. The method includes the computation of minimal routes, similar to shortest derivations in our graph. However, their approach seems less conducive to profiling since, e.g., provenance information on firing rounds is not available in their approach.

There is an intruigingly close relationship between *provenance semirings*, i.e., provenance polynomials and formal power series [8], and our labeled provenance graphs G. The semiring provenance of atom A is represented in the structure of G_A. Consider, e.g., Figure 2: the in-edges of rule firings correspond to a logical conjunction "$\wedge$", or more abstractly, the product operator "$\otimes$" of the semiring. Similarly, out-edges represent a disjunction "$\vee$", i.e., an abstract sum operator "$\oplus$", mirroring the fact that atoms in general have multiple derivations. It is easy to see that a fact A has an infinite number of derivations (proof trees) iff there is a cycle in G_A: e.g., the derivation of $A = \mathsf{tc}(\mathsf{a}, \mathsf{b})$ in Figures 1 and 2 involves a cycle through tc(b,b), tc(c,b), via two firings of r_2. This also explains Prolog's non-termination (Section 1), which "nicely" mirrors the fact that there are infinitely many proof trees. On the other hand, such cycles are not problematic in the original Datalog evaluation of $M = P(I)$ or in our extended provenance model $\hat{M} = \hat{P}(I)$, both of which can be shown to converge in polynomial time.

6 Conclusions

We have presented a framework for declarative debugging and profiling of Datalog programs. The key idea is to rewrite a program P into $\hat{P}$, which records the derivation history of $M = P(I)$ in an extended model $\hat{M} = \hat{P}(I)$. $\hat{P}$ is obtained from three simple rewritings for (1) recording rule firings, (2) reifying those into a labeled graph, while (3) keeping track of derivation rounds in the style of Statelog. After the rewritten program is evaluated, the resulting provenance graph can be queried and visualized for debugging and profiling purposes.

We have illustrated the declarative profiling approach by analyzing different, logically equivalent versions of the transitive closure program P_{tc}. The measures obtained through logic profiling correlate with runtime measures for a large, real-world example. Two prototypical systems GPAD/DLV and GPAD/LB have been implemented, for DLV and LogicBlox, respectively; a public release is planned for the near future. While we have presented our approach for positive Datalog only, it is not difficult to see how it can be extended, e.g., for well-founded Datalog. Indeed, the GPAD prototypes already support the handling of well-founded negation through a simple Statelog encoding [11, 13].

Acknowledgments. Work supported in part by NSF awards OCI-0722079, IIS-1118088, and a gift from LogicBlox, Inc.

References

[1] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)

[2] Bancilhon, F., Ramakrishnan, R.: An Amateur's Introduction to Recursive Query Processing Strategies. In: Readings in Database Systems, pp. 507–555 (1988)

[3] Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: A Theoretical Framework for the Declarative Debugging of Datalog Programs. In: Schewe, K.-D., Thalheim, B. (eds.) SDKB 2008. LNCS, vol. 4925, pp. 143–159. Springer, Heidelberg (2008)

[4] Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Workshop on Software Engineering for Answer Set Programming, SEA (2009)
[5] Chiticariu, L., Tan, W.C.: Debugging Schema Mappings with Routes. In: VLDB, pp. 79–90 (2006)
[6] Chomicki, J., Imieliński, T.: Finite representation of infinite query answers. ACM Transactions on Database Systems (TODS) 18(2), 181–223 (1993)
[7] Drabent, L., Nadjm-Tehrani, S.: Algorithmic Debugging with Assertions. In: Meta-programming in Logic Programming (1989)
[8] Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS (2007)
[9] Green, T.J., Karvounarakis, G., Ives, Z.G., Tannen, V.: Update Exchange with Mappings and Provenance. In: VLDB, pp. 675–686 (2007)
[10] Kunen, K.: Declarative Semantics of Logic Programming. Bulletin of the EATCS 44, 147–167 (1991)
[11] Lausen, G., Ludäscher, B., May, W.: On Active Deductive Databases: The Statelog Approach. In: Kifer, M., Voronkov, A., Freitag, B., Decker, H. (eds.) Transactions and Change in Logic DBs. LNCS, vol. 1472, pp. 69–106. Springer, Heidelberg (1998)
[12] Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (TOCL) 7(3), 499–562 (2006)
[13] Ludäscher, B.: Integration of Active and Deductive Database Rules. Ph.D. thesis, Albert-Ludwigs Universität, Freiburg, Germany (1998)
[14] Marczak, W.R., Huang, S.S., Bravenboer, M., Sherr, M., Loo, B.T., Aref, M.: SecureBlox: Customizable Secure Distributed Data Processing. In: SIGMOD (2010)
[15] Oetsch, J., Pührer, J., Tompits, H.: Stepping through an Answer-Set Program. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 134–147. Springer, Heidelberg (2011)
[16] Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. In: Workshop on Software Engineering for Answer Set Programming, SEA (2007)
[17] Ramakrishnan, R., Ullman, J.: A Survey of Deductive Database Systems. Journal of Logic Programming 23(2), 125–149 (1995)
[18] Shapiro, E.: Algorithmic program debugging. Dissertation Abstracts International Part B: Science and Engineering 43(5) (1982)
[19] Tobermann, G., Beckstein, C.: What's in a Trace: The Box Model Revisited. In: Adsul, B. (ed.) AADEBUG 1993. LNCS, vol. 749, pp. 171–187. Springer, Heidelberg (1993)
[20] Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. CoRR abs/1011.5332 (2010)
[21] Zaniolo, C., Arni, N., Ong, K.: Negation and Aggregates in Recursive rules: the LDL++ Aprrpach. In: Ceri, S., Tsur, S., Tanaka, K. (eds.) DOOD 1993. LNCS, vol. 760, pp. 204–221. Springer, Heidelberg (1993)
[22] Zeller, A.: Isolating cause-effect chains from computer programs. In: SIGSOFT Symposium on Foundations of Software Engineering (2002)

Inconsistency-Tolerant Query Rewriting for Linear Datalog+/–

Thomas Lukasiewicz, Maria Vanina Martinez, and Gerardo I. Simari

Department of Computer Science, University of Oxford, UK
firstname.lastname@cs.ox.ac.uk

Abstract. Inconsistency management in knowledge bases is an important problem that has been studied for a long time. During the recent years, additional interest in this topic has been sparked with the advent of the Semantic Web, which has made this problem even more relevant, since inconsistencies are very likely to occur in open environments such as the Web. Inconsistency-tolerant semantics to query answering have therefore become of special interest for representation and reasoning formalisms for the Semantic Web. Datalog+/– is a family of ontology languages that is in particular useful for representing and reasoning over lightweight ontologies in the Semantic Web. In this paper, we focus on inconsistency-tolerant query answering under the intersection semantics in linear Datalog+/–, a sublanguage of Datalog+/– that generalizes the *DL-Lite* family of tractable description logics (DLs). In particular, we show that query answering in linear Datalog+/– is first-order rewritable under this inconsistency-tolerant semantics, and therefore very efficiently computable in the data complexity.

1 Introduction

It has been widely acknowledged in both the database and the Semantic Web community that inconsistency is an issue that cannot be ignored. Knowledge bases in databases and the Semantic Web in the form of ontologies are becoming increasingly popular, and when integrating data from many different sources, either as a means to populate an ontology or simply to answer queries, integrity constraints are very likely to be violated in practice. In this paper, we address the problem of handling inconsistency in ontologies in databases and the Semantic Web, where scalability is an important issue.

We adopt the recently developed Datalog+/– family of ontology languages [5]. In particular, we focus on the *linear* sublanguage of Datalog+/–, which guarantees the first-order rewritability of conjunctive queries. Datalog+/– enables a modular rule-based style of knowledge representation, and it can represent syntactical fragments of first-order logic so that answering a BCQ Q under a set of Datalog+/– rules Σ for an input database D is equivalent to the classical entailment check $D \cup \Sigma \models Q$. Furthermore, since we can realistically assume that the database D is the only really large object in the input, we can leverage Datalog+/–'s properties of decidability of query answering and good query answering complexity in the data complexity. These properties, together with its expressive power, make Datalog+/– very useful in modeling real applications such as ontology querying, Web data extraction, data exchange, ontology-based data access, and data integration.

The following example shows a simple Datalog+/– ontology; the language and standard semantics for query answering in Datalog+/– is recalled in the next section.

Example 1. Consider A (linear) Datalog+/– ontology $KB = (D, \Sigma_T \cup \Sigma_E \cup \Sigma_{NC})$ is given by:

$$\begin{aligned} D \;&= \{\textit{directs}(john, d_1), \textit{directs}(tom, d_1), \textit{directs}(tom, d_2),\\ &\quad \textit{supervises}(tom, john), \textit{works\_in}(john, d_1), \textit{works\_in}(tom, d_1)\};\\ \Sigma_T \;&= \{\sigma_1 : \textit{works\_in}(X, D) \rightarrow \textit{emp}(X), \sigma_2 : \textit{directs}(X, D) \rightarrow \textit{emp}(X),\\ &\quad \sigma_3 : \textit{directs}(X, D) \rightarrow \textit{manager}(X)\};\\ \Sigma_{NC} &= \{\upsilon_1 : \textit{supervises}(X, Y) \wedge \textit{manager}(Y) \rightarrow \bot,\\ &\quad \upsilon_2 : \textit{supervises}(X, Y) \wedge \textit{works\_in}(X, D) \wedge \textit{directs}(Y, D) \rightarrow \bot\};\\ \Sigma_E \;&= \{\upsilon_3 : directs(X, D) \wedge directs(X, D') \rightarrow D = D'\}. \end{aligned}$$

Here, the formulas in Σ_T say that every person working for a department is an employee (σ_1), that every person directing a department is an employee (σ_2), and that somebody directing a department is a manager (σ_3). The formula υ_1 in Σ_{NC} states that if X supervises Y, then Y cannot be a manager, while υ_2 says that if Y is supervised by someone in a department, then Y cannot direct that department. The formula υ_3 in Σ_E states that the same person cannot direct two different departments. As we show later, this ontology is inconsistent. For instance, the atom *directs*$(john, d_1)$ triggers the application of σ_3, producing *manager*$(john)$, but that together with the atom *supervises*$(tom, john)$ (which belongs to D) violates the formula υ_1. ■

The most accepted approach to query answering over possibly inconsistent databases consists of finding *consistent answers*; this can be done *on the fly* during the query answering process, or over a database that has been previously treated to excise the pieces of information causing the inconsistencies. In general, consistent answers are very hard to compute, even for restricted sets of integrity constraints [7], or for lightweight ontology languages such as the description logics (DLs) in the *DL-Lite* family [14]. In this paper, we focus on a particular inconsistency-tolerant semantics to query answering in linear Datalog+/–, called the *intersection semantics*, which was introduced in [12] for *DL-Lite* as a tractable sound approximation to consistent answers.

The main contribution of this paper is an algorithm for inconsistency-tolerant query answering for linear Datalog+/– under the intersection semantics. Our approach is based on *query rewriting*, i.e., a given query is rewritten into another query, which fully embeds any underlying ontological knowledge, and which, evaluated on the data, returns the consistent answers under the intersection semantics. The result of this rewriting process is a first-order (FO) query. FO-rewritability of queries is an important property, since the rewritten query can immediately be translated into SQL. In this way, we reduce the problem of query answering over an ontology to the standard evaluation of an SQL query in (possibly highly optimized) relational DBMSs.

2 Preliminaries

We briefly recall some basics on Datalog+/– [5], namely, on relational databases, (Boolean) conjunctive queries ((B)CQs), tuple- and equality-generating dependencies (TGDs and EGDs, respectively), negative constraints, and ontologies in Datalog+/–.

Databases and Queries. We assume (i) an infinite universe of *(data) constants* Δ (which constitute the "normal" domain of a database), (ii) an infinite set of *(labeled) nulls* Δ_N (used as "fresh" Skolem terms, which are placeholders for unknown values, and can thus be seen as variables), and (iii) an infinite set of variables $\mathcal{V}$ (used in queries, dependencies, and constraints). Different constants represent different values (*unique name assumption*), while different nulls may represent the same value. We assume a lexicographic order on $\Delta \cup \Delta_N$, with every symbol in Δ_N following all symbols in Δ. We denote by $\mathbf{X}$ sequences of variables $X_1, \ldots, X_k$ with $k \geqslant 0$.

We assume a *relational schema* $\mathcal{R}$, which is a finite set of *predicate symbols* (or simply *predicates*). A *term* t is a constant, null, or variable. An *atomic formula* (or *atom*) $\mathbf{a}$ has the form $P(t_1, ..., t_n)$, where P is an n-ary predicate, and $t_1, ..., t_n$ are terms. We denote by $var(\mathbf{a})$ the set of all variables in atom $\mathbf{a}$ and naturally extend this notation to sets of atoms. A conjunction of atoms is often identified with the set of all its atoms.

A *database (instance)* D for a relational schema $\mathcal{R}$ is a (possibly infinite) set of atoms with predicates from $\mathcal{R}$ and arguments from Δ. A *conjunctive query (CQ)* over $\mathcal{R}$ has the form $Q(\mathbf{X}) = \exists \mathbf{Y}\, \Phi(\mathbf{X}, \mathbf{Y})$, where $\Phi(\mathbf{X}, \mathbf{Y})$ is a conjunction of atoms (possibly equalities, but not inequalities) with the variables $\mathbf{X}$ and $\mathbf{Y}$, and possibly constants, but without nulls. A *Boolean CQ (BCQ)* over $\mathcal{R}$ is a CQ of the form $Q()$, often written as the set of all its atoms, without quantifiers. Answers are defined via *homomorphisms*, which are mappings $\mu\colon \Delta \cup \Delta_N \cup \mathcal{V} \to \Delta \cup \Delta_N \cup \mathcal{V}$ such that (i) $c \in \Delta$ implies $\mu(c) = c$, (ii) $c \in \Delta_N$ implies $\mu(c) \in \Delta \cup \Delta_N$, and (iii) μ is naturally extended to atoms, sets of atoms, and conjunctions of atoms. The set of all *answers* to a CQ $Q(\mathbf{X}) = \exists \mathbf{Y}\, \Phi(\mathbf{X}, \mathbf{Y})$ over a database D, denoted $Q(D)$, is the set of all tuples $\mathbf{t}$ over Δ for which there exists a homomorphism $\mu\colon \mathbf{X} \cup \mathbf{Y} \to \Delta \cup \Delta_N$ such that $\mu(\Phi(\mathbf{X}, \mathbf{Y})) \subseteq D$ and $\mu(\mathbf{X}) = \mathbf{t}$. The *answer* to a BCQ $Q()$ over a database D is *Yes*, denoted $D \models Q$, iff $Q(D) \neq \emptyset$.

Tuple-Generating Dependencies. Tuple-generating dependencies describe constraints on databases in the form of generalized Datalog rules with existentially quantified conjunctions of atoms in rule heads. Given a relational schema $\mathcal{R}$, a *tuple-generating dependency (TGD)* σ is a first-order formula of the form $\forall \mathbf{X} \forall \mathbf{Y}\, \Phi(\mathbf{X}, \mathbf{Y}) \to \exists \mathbf{Z}\, \Psi(\mathbf{X}, \mathbf{Z})$, where $\Phi(\mathbf{X}, \mathbf{Y})$ and $\Psi(\mathbf{X}, \mathbf{Z})$ are conjunctions of atoms over $\mathcal{R}$ (without nulls), called the *body* and the *head* of σ, denoted $body(\sigma)$ and $head(\sigma)$, respectively. Such σ is satisfied in a database D for $\mathcal{R}$ iff, whenever there exists a homomorphism h that maps the atoms of $\Phi(\mathbf{X}, \mathbf{Y})$ to atoms of D, there exists an extension h' of h that maps the atoms of $\Psi(\mathbf{X}, \mathbf{Z})$ to atoms of D. We usually omit the universal quantifiers in TGDs. Since TGDs can be reduced to TGDs with only single atoms in their heads, in the sequel, every TGD has w.l.o.g. a single atom in its head [3]. As we will see later, this assumption is also valid for our treatment of inconsistency since it is solely based on query answering. A TGD σ is *guarded* iff it contains an atom in its body that contains all universally quantified variables of σ. A TGD σ is *linear* iff it contains only a single atom in its body.

Query answering under TGDs, i.e., the evaluation of CQs and BCQs on databases under a set of TGDs is defined as follows. For a database D for $\mathcal{R}$, and a set of TGDs Σ on $\mathcal{R}$, the set of *models* of D and Σ, denoted $mods(D, \Sigma)$, is the set of all (possibly infinite) databases B such that (i) $D \subseteq B$ and (ii) every $\sigma \in \Sigma$ is satisfied in B. The set of *answers* for a CQ Q to D and Σ, denoted $ans(Q, D, \Sigma)$, is the set of all tuples $\mathbf{a}$ such that $\mathbf{a} \in Q(B)$ for all $B \in mods(D, \Sigma)$. The *answer* for a BCQ Q to D and Σ is *Yes*, denoted $D \cup \Sigma \models Q$, iff $ans(Q, D, \Sigma) \neq \emptyset$. Note that query answering under general TGDs is undecidable [2], even when the schema and TGDs are fixed [3]. The two problems of CQ and BCQ evaluation under TGDs are LOGSPACE-equivalent [10,9]. Moreover, the query output tuple (QOT) problem (as a decision version of CQ evaluation) and BCQ evaluation are AC_0-reducible to each other. Henceforth, we thus focus only on BCQ evaluation, and any complexity results carry over to the other problems. Decidability of query answering for the guarded case follows from a bounded tree-width property. The data complexity of query answering in this case is P-complete.

Negative Constraints. Another crucial ingredient of Datalog+/– for ontological modeling are *negative constraints* (or simply *constraints*) γ, which are first-order formulas $\forall \mathbf{X} \Phi(\mathbf{X}) \rightarrow \bot$, where $\Phi(\mathbf{X})$ (called the *body* of γ) is a conjunction of atoms (without nulls and not necessarily guarded). We usually omit the universal quantifiers. Under the standard semantics of query answering of BCQs Q in Datalog+/– with TGDs, adding negative constraints is computationally easy, as for each $\forall \mathbf{X} \Phi(\mathbf{X}) \rightarrow \bot$ we only have to check that the BCQ $\Phi(\mathbf{X})$ evaluates to false in D under Σ; if one of these checks fails, then the answer to the original BCQ Q is true, otherwise the constraints can simply be ignored when answering the BCQ Q.

Equality-Generating Dependencies (EGDs). A further important ingredient of Datalog+/– for modeling ontologies are *equality-generating dependencies* (or *EGDs*) σ, which are first-order formulas $\forall \mathbf{X}\, \Phi(\mathbf{X}) \rightarrow X_i = X_j$, where $\Phi(\mathbf{X})$, called the *body* of σ, denoted $body(\sigma)$, is a conjunction of atoms (without nulls and not necessarily guarded) , and X_i and X_j are variables from $\mathbf{X}$. Such σ is satisfied in a database D for $\mathcal{R}$ iff, whenever there exists a homomorphism h such that $h(\Phi(\mathbf{X}, \mathbf{Y})) \subseteq D$, it holds that $h(X_i) = h(X_j)$. We usually omit the universal quantifiers in EGDs. Adding EGDs over databases with guarded TGDs along with negative constraints does not increase the complexity of BCQ query answering as long as they are *non-conflicting* [5]. Intuitively, this ensures that, if the chase (not covered here for reasons of space; cf. [5] for details) fails (due to strong violations of EGDs), then it already fails on the database D, and if it does not fail, then whenever "new" atoms (from the logical point of view) are created in the chase by the application of the EGD chase rule, atoms that are logically equivalent to the new ones are guaranteed to be generated also in the absence of the EGDs. This guarantees that EGDs do not have any impact on the chase with respect to query answering. Non-conflicting EGDs can be expressed as negative constraints of the form $\forall \mathbf{X}\, \Phi(\mathbf{X}), X_i \neq X_j \rightarrow \bot$. In the following, for ease of presentation, all non-conflicting EGDs are expressed as such special forms of negative constraints.

Datalog+/– Ontologies. A *Datalog+/– ontology* $KB = (D, \Sigma)$, where $\Sigma = \Sigma_T \cup \Sigma_{NC}$, consists of a database D, a finite set of TGDs Σ_T, and a finite set of negative constraints and non-conflicting EGDs Σ_{NC}. We say KB is linear iff Σ_T is linear. Example 1 illustrates a simple linear Datalog+/– ontology, which is used in the sequel as a running example.

3 Inconsistency in Datalog+/– Ontologies

In this section, we define the notions of consistency for Datalog+/– ontologies and of data repairs for inconsistent Datalog+/– ontologies. We also define consistent answers based on such repairs and on the intersection of such repairs in Datalog+/–. We show that BCQ answering under the consistent answer semantics is co-NP-complete in the data complexity for guarded and linear Datalog+/– ontologies and that BCQ answering under the intersection semantics is co-NP-complete in the data complexity for guarded Datalog+/– ontologies. We also characterize the intersection semantics via the notion of culprits, which are minimal inconsistent subsets of the database.

Definition 1 (Consistency). A Datalog+/– ontology $KB = (D, \Sigma)$ is *consistent* iff $mods(D, \Sigma) \neq \emptyset$.

Note that if $\Sigma_{NC} = \emptyset$, then $mods(D, \Sigma)$ is not empty. Different works on inconsistency handling in DLs allow for inconsistency to occur for different reasons. Depending on the expressive power of the underlying formalism, some works allow for both terminological axioms (TBox) and assertional axioms (ABox) to be inconsistent. In this work, following the idea from database theory in which formulas in Σ are interpreted as integrity constraints expressing the semantics of the data contained in D, we assume that Σ is itself consistent, and inconsistencies can only arise when D and Σ are considered together.

In the area of relational databases, the notion of *repair* was used in order to identify the consistent part of a possibly inconsistent database. A repair is a model of the integrity constraints that is minimal, i.e., “as close as possible” to the original database. We now extend the notion of repairs from the database literature to Datalog+/– ontologies. Intuitively, *data repairs* for $KB = (D, \Sigma)$ are subsets of D that satisfy all the constraints in Σ and minimally differ from it in the set-inclusion sense.

Definition 2 (Data Repair). Let $KB = (D, \Sigma)$ be a Datalog+/– ontology. Then, the set D' is a *data repair* for KB iff (i) $D' \subseteq D$, (ii) $mods(D', \Sigma) \neq \emptyset$, and (iii) there is no other set $D'' \subseteq D$ such that $D' \subset D''$ and $mods(D'', \Sigma) \neq \emptyset$. We denote by $DRep(KB)$ the set of all data repairs for KB.

Example 2. The Datalog+/– ontology KB in Example 1 has four data repairs:

$$r_1 = \{\mathit{directs}(john, d_1), \mathit{directs}(tom, d_1), \mathit{works\_in}(john, d_1), \mathit{works\_in}(tom, d_1)\},$$
$$r_2 = \{\mathit{directs}(john, d_1), \mathit{directs}(tom, d_2), \mathit{works\_in}(john, d_1), \mathit{works\_in}(tom, d_1)\},$$
$$r_3 = \{\mathit{supervises}(tom, john), \mathit{directs}(tom, d_1), \mathit{works\_in}(john, d_1), \mathit{works\_in}(tom, d_1)\},$$
$$r_4 = \{\mathit{supervises}(tom, john), \mathit{directs}(tom, d_2), \mathit{works\_in}(john, d_1), \mathit{works\_in}(tom, d_1)\}. \quad \blacksquare$$

In general, there can be several data repairs. The most widely accepted semantics for querying a possibly inconsistent database is that of *consistent answers* [1], which are the answers entailed in every ontology built from a data repair.

Definition 3 (Consistent Answers). Let $KB = (D, \Sigma)$ be a Datalog+/– ontology, and Q be a BCQ. Then, *Yes* is a *consistent answer* for Q to KB, denoted $KB \models_{\mathit{Cons}} Q$, iff it is an answer for Q to each $KB' = (D', \Sigma)$ with $D' \in DRep(KB)$.

Example 3. Consider again the linear Datalog+/– ontology KB in Example 1 and the set of its data repairs in Example 2. The atom $emp(john)$ can be derived from every data repair (since each contains either $works\_in(john, d_1)$ or $directs(john, d_1)$). Hence, the BCQ $Q = emp(john)$ evaluates to true under the consistent answer semantics. ■

Answering a query under the consistent answer semantics over a database with arbitrary negative constraints (but without taking into account TGDs) is co-NP-complete in the data complexity [8]. The same complexity result holds for BCQ query answering in the *DL-Lite* families of ontologies [12]. So, since linear Datalog+/– generalizes *DL-Lite* [5], the problem is at least as hard in our case. The next result shows that deciding consistent answers on (guarded and) linear Datalog+/– ontologies is co-NP-complete in the data complexity.

Theorem 1. *Given a (guarded or) linear Datalog+/– ontology KB and a BCQ Q, deciding whether KB* $\models_{Cons}$ *Q is* co-NP-*complete in the data complexity.*

To avoid this intractability result, an alternative semantics that considers only the atoms that are in the *intersection* of all data repairs has been presented in [12]. It yields a unique way of repairing inconsistent *DL-Lite* ontologies, and the consistent answers are intuitively the answers that can be obtained from that unique set. In the following, we extend the intersection semantics to Datalog+/– ontologies.

Definition 4 (Intersection Semantics). Let $KB = (D, \Sigma)$ be a Datalog+/– ontology, and Q be a BCQ. Then, *Yes* is a *consistent answer* for Q to KB *under the intersection semantics*, denoted $KB \models_{ICons} Q$, iff it is an answer for Q to $KB_I = (D_I, \Sigma)$, where $D_I = \bigcap\{D' \mid D' \in DRep(KB)\}$.

Example 4. Consider again the ontology $KB = (D, \Sigma)$ of the running example. Analyzing the set of all its data repairs, it is not difficult to verify that $D_I = \{works\_in(john, d_1), works\_in(tom, d_1)\}$. Thus, $Q = emp(tom) \wedge works\_in(john, d_1)$ evaluates to true. As shown in Example 3, *Yes* is a consistent answer for query $Q = emp(john)$, however, under the intersection semantics Q evaluates to false. ■

The following result shows that BCQ answering for guarded Datalog+/– ontologies under the intersection semantics is co-NP-complete in the data complexity, and thus harder than for *DL-Lite*, where it is polynomial [12].

Theorem 2. *Given a guarded Datalog+/– ontology KB and a BCQ Q, deciding whether KB* $\models_{ICons}$ *Q is* co-NP-*complete in the data complexity.*

We now characterize the intersection semantics through the notion of *culprits* relative to a set of negative constraints $NC \subseteq \Sigma_{NC}$, which is informally a minimal (under set inclusion) inconsistent subset of the database relative to NC.

Definition 5 (Culprit). Given a Datalog+/– ontology $KB = (D, \Sigma_T \cup \Sigma_{NC})$ and $IC \subseteq \Sigma_{NC}$, a *culprit* in KB relative to IC is a set $c \subseteq D$ such that $mods(c, \Sigma_T \cup IC) = \emptyset$, and there is no $c' \subset c$ such that $mods(c', \Sigma_T \cup IC) = \emptyset$. We denote by $culprits(KB, IC)$ (resp., $culprits(KB)$) the set of culprits in KB relative to IC (resp., $IC = \Sigma_{NC}$).

Theorem 3 states that the answers to a query under the intersection semantics correspond exactly to the (standard) answers obtained from $D - \bigcup_{c \in culprits(KB)} c$.

Theorem 3. *Let $KB = (D, \Sigma)$ with $\Sigma = \Sigma_T \cup \Sigma_{NC}$ be a Datalog+/– ontology, and Q be a BCQ. Then, $KB \models_{ICons} Q$ iff $(D - \bigcup_{c \in culprits(KB)} c, \Sigma_T) \models Q$.*

4 Inconsistency-Tolerant Query Rewriting for Linear Datalog+/–

The first-order (FO) rewritability of queries over an ontology allows to transform them into FO queries that can be executed over the database alone, i.e., the new queries embed the dependencies and constraints of the ontology. Since an FO query can be translated into an equivalent SQL expression, query answering can be delegated to a traditional relational DBMS, thus exploiting any underlying optimizations. The sublanguage of Datalog+/– with linear TGDs is FO-rewritable [4]. Recently, [6] presents a rewriting algorithm, inspired by resolution in logic programming, which deals with so-called *sticky-join* sets of TGDs, a class including sets of linear TGDs. However, this algorithm corresponds to the standard (non-inconsistency-tolerant) semantics for query answering. More recently, [13] presents a rewriting procedure for inconsistency-tolerant query answering in *DL-Lite*$_A$ ontologies under the intersection semantics; *DL-Lite*$_A$ belongs to the *DL-Lite* family of tractable DLs, which can all be expressed in linear Datalog+/– (with negative constraints). We now present a rewriting procedure for inconsistency-tolerant query answering in linear Datalog+/– under the intersection semantics.

4.1 Preliminaries

Under standard query answering in Datalog+/–, a class of TGDs is FO-rewritable iff for every set of TGDs Σ in that class, and every BCQ Q, there exists an FO query Q_Σ such that $(D, \Sigma) \models Q$ iff $D \models Q_\Sigma$, for every database D. In this work, we show that the class of linear TGDs is FO-rewritable for consistent query answering under the intersection semantics. This means that, for every set Σ composed of arbitrary sets Σ_T and Σ_{NC} of linear TGDs and negative constraints, respectively, and every BCQ Q, there exists an FO query Q_Σ such that $(D, \Sigma) \models_{ICons} Q$ iff $D \models Q_\Sigma$, for every database D. Here, Q_Σ encodes the set of linear TGDs and the *enforcement* of the negative constraints so that they reflect the intersection semantics for query answering.

The rewriting of a BCQ Q relative to a set Σ of linear TGDs and negative constraints is accomplished in two steps. First, we analyze how to rewrite the negative constraints into Q in a way that the rewriting enforces the intersection semantics, i.e., $KB \models_{ICons} Q$ iff $D \models Q'$, where Q' is the rewriting of Q obtained by enforcing the constraints in Σ_{NC}. This is done independently of the set of linear TGDs. Second, both the query and the negative constraints in Σ_{NC} may need to be rewritten relative to the set of TGDs. For this second part, we use Algorithm TGD-rewrite from [6].

Two atoms a and b *unify* iff there exists a substitution γ (called a *unifier* for a and b) such that $\gamma(a) = \gamma(b)$. A *most general unifier (mgu)* is a unifier for a and b, denoted $\gamma_{a,b}$, such that for any other unifier γ for a and b, there exists a substitution γ' such that $\gamma = \gamma' \circ \gamma_{a,b}$. Note that mgus are unique up to variable renaming.

4.2 TGD-Free Case

We first focus on the FO rewriting of a BCQ Q relative to an ontology without TGDs Σ_{NC} by enforcing the negative constraints in Σ_{NC}, i.e., on obtaining $(D, \Sigma_{NC}) \models_{ICons} Q$

iff $D \models Q'$, where Q' is the enforcement of Σ_{NC} in Q. To capture the intersection semantics, we have to establish a correspondence between the minimization of negative constraints in the rewriting Q' of Q and the minimization inherently encoded in culprits. To this end, we introduce the *normalization* of negative constraints.

Definition 6. Let Σ_{NC} be a set of negative constraints, $\upsilon \in \Sigma_{NC}$, and Q be a BCQ. Let $\sim_\upsilon$ be an equivalence relation on the arguments in the body of υ and the constants in Σ_{NC} and Q such that every equivalence class contains at most one constant. A *normalization instance* of υ relative to such $\sim_\upsilon$ is obtained from υ by replacing every argument in the body of υ by a representative of its equivalence class (which is a constant if the equivalence class contains a constant) and adding to the body the conjunction of all $s \neq t$ for any two different representatives s and t such that s is a variable occurring in the instance, and t is either a variable occurring in the instance or a constant in Σ_{NC} and Q. The *normalization* of υ, denoted $\mathcal{N}(\upsilon)$, is the set of all such instances of υ subject to all equivalence relations $\sim_\upsilon$. The *normalization* of Σ_{NC} is $\mathcal{N}(\Sigma_{NC}) = \bigcup_{\upsilon \in \Sigma_{NC}} N(\upsilon)$.

Example 5. Consider the set of negative constraints $\Sigma_{NC} = \{\upsilon_1 : p(U,U) \rightarrow \bot,\ \upsilon_2 : p(X,Y) \wedge q(X) \rightarrow \bot\}$ and the BCQ $Q = \exists X q(X)$. Its normalization is $\mathcal{N}(\Sigma_{NC}) = \{\upsilon'_1 : p(U,U) \rightarrow \bot, \upsilon'_2 : p(X,X) \wedge q(X) \rightarrow \bot, \upsilon'_3 : p(X,Y) \wedge q(X) \wedge X \neq Y \rightarrow \bot\}$. ■

The following result shows that the normalization of negative constraints does not change the culprits of an ontology, even in the additional presence of a set of TGDs Σ_T (where the constants in Σ_T are considered in the same way as those in Σ_{NC} and Q).

Lemma 1. *Let $KB = (D, \Sigma_{NC} \cup \Sigma_T)$ be a Datalog+/– ontology, and Q be a BCQ. Then, culprits$(KB, \Sigma_{NC} \cup \Sigma_T) =$ culprits$(KB, \mathcal{N}(\Sigma_{NC}) \cup \Sigma_T)$.*

The second step in the FO rewriting of a BCQ Q by enforcing the negative constraints is to identify the set of normalized negative constraints that must be *enforced* in Q, i.e., the normalized negative constraints that must be satisfied so that only the consistent answers under the intersection semantics are entailed from D.

Definition 7. Let Σ_{NC} be a set of negative constraints, $\upsilon \in \mathcal{N}(\Sigma_{NC})$, and Q be a BCQ. Then, υ *needs to be enforced* in Q iff there exists $C \subseteq Q$, $C \neq \emptyset$, such that C unifies with $B \subseteq body(\upsilon)$ via some mgu $\gamma_{C,B}$, and there exists no $\upsilon' \in \mathcal{N}(\Sigma_{NC})$ such that $body(\upsilon')$ maps isomorphically to $B' \subset body(\upsilon)$.

Example 6. Consider the linear Datalog+/– ontology KB from Example 1 and the BCQ $Q = supervises(tom, john)$. Then, $\upsilon'_1 : supervises(tom, john) \wedge manager(john) \rightarrow \bot$ is a normalized instance of υ_1 that must be enforced, and $C_1 = \{supervises(tom, john)\}$ unifies with $B_1 = \{supervises(tom, john)\}$ via the mgu $\gamma_{C_1,B_1} = \{\}$. ■

Example 7. Consider $\mathcal{N}(\Sigma_{NC})$ from Example 5, and the BCQ $Q = \exists X q(X)$. Then, neither υ'_1 nor υ'_2 need to be enforced in Q, while υ'_3 needs to be enforced in Q. ■

Algorithm Enforcement (see Fig. 1) performs the rewriting of a query Q by embedding all negative constraints that must be enforced. The following example shows how the algorithm works for the ontology KB and the BCQ Q from Example 1.

```
Algorithm Enforcement(BCQ $Q = \exists G$, normalized set of negative constraints IC)
 Here, $G$ is a quantifier-free formula, and $\exists G$ is the existential closure of $G$.
 1. $F := G$;
 2. for every $\upsilon \in IC$ do
 3.     for every $C \subseteq Q$, $C \neq \emptyset$, that unifies with $B \subseteq body(\upsilon)$ via mgu $\gamma_{C,B}$ do
 4.         if there is no $\upsilon' \in IC$ such that $body(\upsilon')$ maps isomorphically to $B' \subset body(\upsilon)$ then
 5.             $F := F \wedge \neg\exists_{\overline{G}}\,((\bigwedge_{X \in var(C)} X = \gamma_{C,B}(X)) \wedge \gamma_{C,B}(body(\upsilon)))$ (where $\exists_{\overline{G}} R$ is
                    the existential closure of $R$ relative to all variables in $R$ that are not in $G$);
 6. return $\exists F$.
```

Fig. 1. Computing the enforcement of a normalized set of NCs *IC* relative to a BCQ Q

Example 8. Coming back to Example 6, the enforcement of Σ_{NC} in $Q = supervises(tom, john) \wedge works\_in(tom, d_1)$, returned by Algorithm Enforcement in Fig. 1, is given by $(Q \wedge \neg manager(john) \wedge \neg directs(john, d_1))$. Ignoring Σ_T, we have $(D, \Sigma_{NC}) \not\models_{ICons} Q$, since $Q \notin D_I = \{directs(john, d_1), works\_in(john, d_1)\}$; D_I is the intersection of all data repairs in (D, Σ_{NC}). As expected, $D \not\models$ Enforcement$(Q, \mathcal{N}(\Sigma_{NC}))$. ■

We now establish the correctness of Algorithm Enforcement. The following proposition is used in the proof of the main correctness result in Theorem 4 below. It states that to answer a query under the intersection semantics, it is only necessary to look at the set of normalized negative constraints that need to be enforced in Q.

Proposition 1. *Let $KB = (D, \Sigma_{NC})$ be a Datalog+/– ontology without TGDs, and Q be a BCQ. Let $\Sigma_Q \subseteq \mathcal{N}(\Sigma_{NC})$ be the set of constraints that must be enforced in Q. Then, $KB \models_{ICons} Q$ iff $(D, \Sigma_Q) \models_{ICons} Q$.*

Theorem 4 shows the correctness of Algorithm Enforcement. It is important to note that here we are now only assuming a Datalog+/– ontology of the form $KB = (D, \Sigma)$ where Σ contains only negative constraints, i.e., no rewriting relative to TGDs is needed. Though the above results are valid for general Datalog+/– ontologies, Theorem 4 only holds for Datalog+/– ontologies that do not have TGDs.

Theorem 4. *Let $KB = (D, \Sigma_{NC})$ be a Datalog+/– ontology without TGDs, and Q be a BCQ. Then, $KB \models_{ICons} Q$ iff $D \models$ Enforcement$(Q, \mathcal{N}(\Sigma_{NC}))$.*

4.3 General Case

We now concentrate on the general problem of rewriting a BCQ Q relative to a set of negative constraints and linear TGDs $\Sigma_{NC} \cup \Sigma_T$. To this end, we have to generalize the enforcement of Σ_{NC} in Q described in the previous section. The following result is used to show that to enforce Σ_{NC} in Q, it is possible to rewrite the body of the negative constraints first and then to enforce the new set of negative constraints (containing all possible rewritings of the negative constraints relative to Σ_T) in Q. It follows immediately from the soundness and completeness of Algorithm TGD-rewrite from [6].

Lemma 2. *Let $KB = (D, \Sigma)$ with $\Sigma = \Sigma_{NC} \cup \Sigma_T$ be a linear Datalog+/– ontology. Let Σ_{Rew} be the set of all negative constraints $F \to \bot$ such that $F \in$ TGD-rewrite$(body(\upsilon), \Sigma_T)$ for some $\upsilon \in \Sigma_{NC}$. Then, $culprits(KB, \Sigma) = culprits(KB, \Sigma_{Rew})$.*

```
Algorithm rewriteICons(BCQ $Q$, set of negative constraints and linear TGDs $\Sigma_{NC} \cup \Sigma_T$)
 1. $\Sigma_{Rew} := \{F \rightarrow \bot \mid F \in \text{TGD-rewrite}(body(\upsilon), \Sigma_T)$ for some $\upsilon \in \Sigma_{NC}\}$;
 2. $Q_{rw} := \text{TGD-rewrite}(Q, \Sigma_T)$;
 3. $out := \emptyset$;
 4. for each $Q \in Q_{rw}$ do
 5.     $out := out \cup \text{Enforcement}(Q, \mathcal{N}(\Sigma_{Rew}))$;
 6. return $out$.
```

Fig. 2. Rewriting a BCQ Q relative to a set of negative constraints and linear TGDs Σ under the intersection semantics; see Fig. 1 for Algorithm Enforcement

As an immediate consequence, query answering in linear Datalog+/– under the intersection semantics is invariant to rewriting the negative constraints relative to the linear TGDs. This result follows immediately from Lemma 2 and Theorem 3.

Proposition 2. *Let $KB = (D, \Sigma)$, $\Sigma = \Sigma_{NC} \cup \Sigma_T$, be a linear Datalog+/– ontology. Let Σ_{Rew} be the set of all negative constraints $F \rightarrow \bot$ such that $F \in$ TGD-rewrite$(body(\upsilon)$, $\Sigma_T)$ for some $\upsilon \in \Sigma_{NC}$. Then, $KB \models_{ICons} Q$ iff $(D, \Sigma_{Rew} \cup \Sigma_T) \models_{ICons} Q$.*

The following example illustrates the rewriting of the set of negative constraints in the running example relative to a corresponding set of linear TGDs.

Example 9. Consider the negative constraint $\upsilon_1 = supervises(X, Y) \wedge manager(Y) \rightarrow \bot$ from Σ_{NC} in Example 1. Then, the rewriting of $body(\upsilon_1)$ relative to Σ_T is given by:

$$\begin{aligned}\text{TGD-rewrite}(body(\upsilon_1), \Sigma_T) = \{&rw_1 : supervises(X, Y) \wedge manager(Y),\\ &rw_2 : supervises(X, Y) \wedge directs(Y, D)\}\,.\end{aligned}$$

Similarly, TGD-rewrite$(body(\upsilon_1), \Sigma_T) = \{body(\upsilon_2)\}$, and TGD-rewrite$(body(\upsilon_3)$, $\Sigma_T) = \{body(\upsilon_3)\}$. (Recall that υ_3 is treated as the negative constraint $directs(X, D) \wedge directs(X, D') \wedge D \neq D' \rightarrow \bot$.) Hence, $\Sigma_{Rew} = \{rw_1 \rightarrow \bot, rw_2 \rightarrow \bot, \upsilon_2, \upsilon_3\}$. ■

Algorithm rewriteICons in Fig. 2 computes the rewriting of a BCQ Q relative to a set of negative constraints and linear TGDs $\Sigma = \Sigma_{NC} \cup \Sigma_T$. The algorithm works as follows. First, the rewriting of the bodies of the negative constraints in Σ_{NC} are computed relative to Σ_T, using algorithm TGD-rewrite from [6]. Then, similarly, the rewriting of Q is computed relative to Σ_T. Finally, for each query in the rewriting of Q, the algorithm enforces the normalization of the rewritten set of negative constraints. The following example illustrates how Algorithm rewriteICons works.

Example 10. Consider again the BCQ $Q = manager(john)$ to the linear Datalog+/– ontology $\Sigma = \Sigma_{NC} \cup \Sigma_T$ from Example 1. First, algorithm rewriteICons computes Σ_{Rew}, the rewriting of Σ_{NC} relative to Σ_T, as shown in Example 9. Second, Q is rewritten relative to Σ_T, obtaining $Q_{rw} = \{\{manager(john)\}, \{directs(john, D)\}\}$. Finally, for every $Q' \in Q_{rw}$, the algorithm computes Enforcement(Q', Σ_{Rew}): for $Q' = manager(john)$, we get Enforcement$(Q', \Sigma_{Rew}) = \{Q_1 = manager(john) \wedge \forall Y \neg supervises(john, Y)\}$ and for $Q' = \{directs(john, D)\}$, Enforcement$(Q', \Sigma_{Rew}) = \{Q_2 = directs(john, D) \wedge \forall Y, X, D' \neg supervises(john, Y) \wedge \neg(supervises(X, john) \wedge works_in(X, D)) \wedge \neg(directs(john, D') \wedge D \neq D')\}$. Then, the output is $out = \{Q_1, Q_2\}$ $(= Q_1 \vee Q_2)$.

Both Q_1 and Q_2 are false on the database D from Example 1, and so the consistent answer for Q under the intersection semantics is *false*, which is correct, since $D_I = \bigcap_{D' \in DRep(KB)} D' = \{works\_in(john, d_1), works\_in(tom, d_1)\}$, and thus $KB \not\models_{ICons} Q$.■

The following theorem shows the correctness of Algorithm rewriteICons.

Theorem 5. *Let $KB = (D, \Sigma)$ with $\Sigma = \Sigma_{NC} \cup \Sigma_T$ be a linear Datalog+/– ontology, and Q be a BCQ. Then, $KB \models_{ICons} Q$ iff $D \models \bigvee_{F \in \mathsf{rewriteICons}(Q,\Sigma)} F$.*

5 Related Work

In the database community, the fields of *database repairing* and *consistent query answering* (CQA for short) have gained much attention since the work [1], which provided a model-theoretic construct of a database *repair*: a repair of an inconsistent database is a model of the set of integrity constraints that is minimal, i.e., “as close as possible” to the original database. Arenas et al. [1] propose a method to compute consistent query answers based on query rewriting that applies to FO queries without disjunction or quantification, and databases with only binary universal integrity constraints without TGDs. The rewriting applies to and produces FO queries. When a literal in the query can be resolved with an integrity constraint, the resolvent forms a residue. All such residues are then conjoined with the literal to form its expanded version. If a literal that has been expanded appears in a residue, the residue has to be further expanded until no more changes occur. A rewriting method for a larger subset of conjunctive queries but that is limited to primary key FDs (a special case of EGDs) was proposed in [11]. The work [7] summarizes the results of the area of CQA.

More recently, the work [12] studies the adaptation of CQA for *DL-Lite* ontologies. In that work, the *intersection* semantics is presented as a sound approximation of consistent answers that for the *DL-Lite* family is easier to compute, as well as the *closed ABox* version for both of them, which considers the closure of the set of assertional axioms (ABox, or extensional database) by the terminological axioms (TBox, or intensional database). Later, in [13], the authors explore FO-rewritability of *DL-Lite* ontologies under the intersection and closed ABox semantics. The data and combined complexity of the semantics were studied in [14] for a wider spectrum of DLs. The rewritability results obtained in this paper for consistent answers under the intersection semantics for linear Datalog+/– ontologies significantly generalize the previous work for *DL-Lite*$_A$. In [14], intractability results for query answering were found for $\mathcal{EL}_\perp$ under the intersection semantics, and a non-recursive segment of that language was proved to be computable in polynomial time. Note that these languages are incomparable with linear Datalog+/– in the sense that neither subsumes the other.

6 Conclusion

In this paper, we have introduced inconsistency-tolerant query answering under the intersection semantics in linear Datalog+/– ontologies. We have shown that query answering in linear Datalog+/– is first-order rewritable under this inconsistency-tolerant semantics, and therefore very efficiently computable in the data complexity.

Acknowledgements. This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/J008346/1 "PrOQAW: Probabilistic Ontological Query Answering on the Web", the European Research Council (FP7/2007-2013)/ERC grant 246858 (DIADEM), by a Google Research Award, and by a Yahoo! Research Fellowship.

References

1. Arenas, M., Bertossi, L.E., Chomicki, J.: Consistent query answers in inconsistent databases. In: Proc. of PODS, pp. 68–79 (1999)
2. Beeri, C., Vardi, M.Y.: The Implication Problem for Data Dependencies. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 73–85. Springer, Heidelberg (1981)
3. Calì, A., Gottlob, G., Kifer, M.: Taming the infinite chase: Query answering under expressive relational constraints. In: Proc. of KR, pp. 70–80 (2008)
4. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. In: Proc. of PODS, pp. 77–86 (2009)
5. Calì, A., Gottlob, G., Lukasiewicz, T.: A general Datalog-based framework for tractable query answering over ontologies. Journal of Web Semantics (2012)
6. Calì, A., Gottlob, G., Pieris, A.: Query rewriting under non-guarded rules. In: Proc. of AMW (2010)
7. Chomicki, J.: Consistent query answering: Five easy pieces. In: Proc. of ICDT, pp. 1–17 (2007)
8. Chomicki, J., Marcinkowski, J.: On the Computational Complexity of Minimal-Change Integrity Maintenance in Relational Databases. In: Bertossi, L., Hunter, A., Schaub, T. (eds.) Inconsistency Tolerance. LNCS, vol. 3300, pp. 119–150. Springer, Heidelberg (2005)
9. Deutsch, A., Nash, A., Remmel, J.B.: The chase revisited. In: Proc. of PODS, pp. 149–158 (2008)
10. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data exchange: Semantics and query answering. Theor. Comput. Sci. 336(1), 89–124 (2005)
11. Fuxman, A., Miller, R.J.: First-order query rewriting for inconsistent databases. J. Comput. Syst. Sci. 73(4), 610–635 (2007)
12. Lembo, D., Lenzerini, M., Rosati, R., Ruzzi, M., Savo, D.F.: Inconsistency-Tolerant Semantics for Description Logics. In: Hitzler, P., Lukasiewicz, T. (eds.) RR 2010. LNCS, vol. 6333, pp. 103–117. Springer, Heidelberg (2010)
13. Lembo, D., Lenzerini, M., Rosati, R., Ruzzi, M., Savo, D.F.: Query Rewriting for Inconsistent DL-Lite Ontologies. In: Rudolph, S., Gutierrez, C. (eds.) RR 2011. LNCS, vol. 6902, pp. 155–169. Springer, Heidelberg (2011)
14. Rosati, R.: On the complexity of dealing with inconsistency in description logic ontologies. In: Proc. of IJCAI, pp. 1057–1062 (2011)

Confluence Analysis for Distributed Programs: A Model-Theoretic Approach

William R. Marczak[1], Peter Alvaro[1], Neil Conway[1],
Joseph M. Hellerstein[1], and David Maier[2]

[1] University of California, Berkeley
[2] Portland State University

Abstract. Building on recent interest in distributed logic programming, we take a model-theoretic approach to analyzing confluence of asynchronous distributed programs. We begin with a model-theoretic semantics for Dedalus and introduce the *ultimate model*, which captures non-deterministic eventual outcomes of distributed programs. After showing the question of confluence undecidable for Dedalus, we identify restricted sub-languages that guarantee confluence while providing adequate expressivity. We observe that the semipositive restriction Dedalus$^+$ guarantees confluence while capturing PTIME, but show that its restriction of negation makes certain simple and practical programs difficult to write. To remedy this, we introduce Dedalus^S, a restriction of Dedalus that allows a kind of stratified negation, but retains the confluence of Dedalus$^+$ and similarly captures PTIME.

1 Introduction

In recent years there has been optimism that declarative languages grounded in Datalog can provide a clean foundation for distributed programming [1]. This has led to activity in language and system design (e.g., [2–5]), as well as formal models for distributed computation using such languages (e.g., [6–8]).

The bulk of this work has presented or assumed a formal operational semantics based on transition systems and traces of input events. A model-theoretic semantics for these languages has been notably absent. In a related paper [9], we developed a model-theoretic semantics for Dedalus, a distributed logic language based on Datalog, in which the "meaning" of a program is precisely the set of stable models [10] corresponding to all possible temporal permutations of messages. In [9], we show these models equivalent to all possible executions in an operational semantics akin to those in prior literature.

In this paper we take advantage of the availability of declarative semantics to explore the correctness of distributed programs. Specifically, we address the desire to ensure deterministic program outcomes—confluence—in the face of inherently non-deterministic timings of computation and messaging.

Using our model-theoretic semantics for Dedalus, we can reason about the set of possible outcomes of a distributed program, based on what we define as its *ultimate models*. We also identify restricted sub-languages of Dedalus that ensure a model-theoretic notion of confluence: the existence of a unique ultimate model for any program in that sub-language. The next question then is to identify a sub-language that ensures confluence without unduly constraining expressivity—both in terms of both computational power, and the ability to employ familiar programming constructs.

A natural step in this direction is to restrict Dedalus to its semi-positive subset, a language we call Dedalus^{+}. This is inspired in part by the CALM theorem [11, 1, 12], which established a connection between confluence and monotonicity. However, we note that this restriction makes common distributed systems tasks difficult to achieve.

We achieve a more comfortable balance between expressive power, ease of programming and guarantees of confluence in Dedalus^{S}, which admits a controlled use of negation that draws inspiration from both stratified negation in logic programming, and coordination protocols from distributed computing. We present the model-theoretic semantics of Dedalus^{S}, and give it an operational semantics by compiling Dedalus^{S} programs into stylized Dedalus programs that augment the original code with "coordination" rules that effectively implement distributed stratified evaluation. We believe the result is practically useful—indeed, Dedalus^{S} corresponds closely to Bloom, a practical programming we have used to implement a broad array of distributed systems [13].

Due to space restrictions, proofs and additional examples are available in [14].

2 Dedalus

Dedalus extends Datalog to model the critical semantic issue from asynchronous distributed computing: asynchrony across nodes. We use a restricted version of Sacca and Zaniolo's *choice* construct [10], interpreted under the stable model semantics, to model program behaviors. Our use of the stable model semantics induces a potentially infinite number of distinctions that are not meaningful in an "eventual" sense. Thus, we introduce the *ultimate model* semantics as a representation of program output.

We begin this section by reviewing the syntax of Dedalus first presented in Alvaro *et al.* [15]. We then review the model-theoretic semantics for Dedalus [9].

2.1 Syntax

Preliminaries. Let $\mathcal{U}$ be an infinite universe of values, with $\mathbb{N} = \{0, 1, 2, \ldots\} \subset \mathcal{U}$.

A *relation schema* is a pair $R^{(k)}$ where R is a relation name and k its arity. A *database schema* $\mathcal{S}$ is a set of relation schemas. Any relation name occurs at most once in a database schema. We assume the existence of an order: every database schema contains the relation schema $<^{(2)}$. Later, we will explain how $<$ is populated.

A *fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an n-tuple $(c_1, \ldots, c_n)$, where each $c_i \in \mathcal{U}$. We denote a fact over $R^{(n)}$ by `R(c`$_1$`, ..., c`$_n$`)`. A *relation instance* for relation schema $R^{(n)}$ is a set of facts whose relation is R. A *database instance* maps each relation schema $R^{(n)}$ to a corresponding relation instance for $R^{(n)}$.

A *rule* φ consists of several distinct components: a head atom $head(\varphi)$, a set $pos(\varphi)$ of *positive body atoms*, a set $neg(\varphi)$ of *negative body atoms*, a set of inequalities $ineq(\varphi)$ of the form $x < y$, and a set of choice operators $cho(\varphi)$ applied to the variables. The conventional syntax for a rule is:

`head(`φ`)` $\leftarrow f\_1, \ldots, f\_n, \neg g\_1, \ldots, \neg g\_m,$ `ineq(`φ`), cho(`φ`).`

where $f_i \in pos(\varphi)$ for $i = 1, \ldots, n$ and $g_i \in neg(\varphi)$ for $i = 1, \ldots, m$.

Dedalus maintains the usual Datalog safety restrictions: every variable symbol `V` in a rule must appear in some atom in *pos*. For readability, we will use the underscore symbol (_) to represent a variable that appears only once in a rule.

Spatial and Temporal Extensions. Given a database schema $\mathcal{S}$, we use $\mathcal{S}^+$ to denote the schema obtained as follows. For each relation schema $\texttt{r}^{(n)} \in (\mathcal{S} \setminus \{<\})$, we include a relation schema $\texttt{r}^{n+1}$ in $\mathcal{S}^+$. The additional column added to each relation schema is the *location specifier*. By convention, the location specifier is the first column of the relation. $\mathcal{S}^+$ also includes $<^{(2)}$, and a relation schema $\texttt{node}^{(1)}$: the finite set of node identifiers that represents all of the nodes in the distributed system. We call $\mathcal{S}^+$ a *spatial schema*.

A *spatial fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an $(n + 1)$-tuple $(d, c_1, \ldots, c_n)$ where each $c_i \in \mathcal{U}$, $d \in \mathcal{U}$, and `node(d)`. We denote a spatial fact over $R^{(n)}$ by $\texttt{R(d, c}_1\texttt{, ..., c}_n\texttt{)}$. A *spatial relation instance* for a relation schema $\texttt{r}^{(n)}$ is a set of spatial facts for $\texttt{r}^{(n+1)}$. A *spatial database instance* is defined similarly to a database instance.

Given a database schema $\mathcal{S}$, we use $\mathcal{S}^*$ to denote the schema obtained as follows. For each relation schema $\texttt{r}^{(n)} \in (\mathcal{S} \setminus \{<\})$ we include a relation schema $\texttt{r}^{(n+2)}$ in $\mathcal{S}^*$. The first additional column added is the location specifier, and the second is the *timestamp*. By convention, the location specifier is the first column of every relation in $\mathcal{S}^*$, and the timestamp is the second. $\mathcal{S}^*$ also includes $<^{(2)}$ (finite), $\texttt{node}^{(1)}$ (finite), $\texttt{time}^{(1)}$ (infinite) and $\texttt{timeSucc}^{(2)}$ (infinite), We call $\mathcal{S}^*$ a *spatio-temporal schema*.

A *spatio-temporal fact* over a relation schema $R^{(n)}$ is a pair consisting of the relation name R and an $(n + 2)$-tuple $(d, t, c_1, \ldots, c_n)$ where each $c_i \in \mathcal{U}$, $d \in \mathcal{U}, t \in \mathcal{U}$, `node(d)`, and `time(t)`. We denote a spatial fact over $R^{(n)}$ by $\texttt{R(d, t, c}_1\texttt{, ..., c}_n\texttt{)}$.

A *spatio-temporal relation instance* for relation schema $\texttt{r}^{(n)}$ is a set of spatio-temporal facts for $\texttt{r}^{(n+2)}$. A *spatio-temporal database instance* is defined similarly to a database instance; in any spatio-temporal database instance, $\texttt{time}^{(1)}$ is mapped to the set containing a `time(t)` fact for all $\texttt{t} \in \mathbb{N}$, and $\texttt{timeSucc}^{(2)}$ to the set containing a `timeSucc(x,y)` fact for all $y = x + 1$, $(x, y \in \mathbb{N})$.

We will use the notation `f@t` to mean the spatio-temporal fact obtained from the spatial fact `f` by adding a timestamp column with the constant `t`.

A *spatio-temporal rule* over a spatio-temporal schema $\mathcal{S}^*$ is a rule of one of the following three forms:

$$\begin{aligned}
&\texttt{p(L,T,}\overline{\texttt{W}}\texttt{)} \leftarrow \texttt{b}_1\texttt{(L,T,}\overline{\texttt{X}_1}\texttt{), ..., b}_l\texttt{(L,T,}\overline{\texttt{X}_l}\texttt{), }\neg\texttt{c}_1\texttt{(L,T,}\overline{\texttt{Y}_1}\texttt{), ..., }\neg\texttt{c}_m\texttt{(L,T,}\overline{\texttt{Y}_m}\texttt{),}\\
&\qquad\qquad \texttt{node(L), time(T), } ineq(\varphi)\texttt{.}\\
&\texttt{p(L,S,}\overline{\texttt{W}}\texttt{)} \leftarrow \texttt{b}_1\texttt{(L,T,}\overline{\texttt{X}_1}\texttt{), ..., b}_l\texttt{(L,T,}\overline{\texttt{X}_l}\texttt{), }\neg\texttt{c}_1\texttt{(L,T,}\overline{\texttt{Y}_1}\texttt{), ..., }\neg\texttt{c}_m\texttt{(L,T,}\overline{\texttt{Y}_m}\texttt{),}\\
&\qquad\qquad \texttt{node(L), time(T), timeSucc(T,S), } ineq(\varphi)\texttt{.}\\
&\texttt{p(D,S,}\overline{\texttt{W}}\texttt{)} \leftarrow \texttt{b}_1\texttt{(L,T,}\overline{\texttt{X}_1}\texttt{), ..., b}_l\texttt{(L,T,}\overline{\texttt{X}_l}\texttt{), }\neg\texttt{c}_1\texttt{(L,T,}\overline{\texttt{Y}_1}\texttt{), ..., }\neg\texttt{c}_m\texttt{(L,T,}\overline{\texttt{Y}_m}\texttt{),}\\
&\qquad\qquad \texttt{node(L), time(T), time(S), choice((L, T, }\overline{\texttt{B}}\texttt{),(S)), node(D), } ineq(\varphi)\texttt{.}
\end{aligned}$$

In the rules above, $\overline{\texttt{B}}$ is a tuple containing all the distinct variable symbols in $\overline{\texttt{X}_1}, \ldots, \overline{\texttt{X}_l}, \overline{\texttt{Y}_1}, \ldots, \overline{\texttt{Y}_m}$. The variable symbols `D` and `L` may appear in any of $\overline{\texttt{W}}, \overline{\texttt{X}_1}, \ldots, \overline{\texttt{X}_l}, \overline{\texttt{Y}_1}, \ldots, \overline{\texttt{Y}_m}$, whereas `T` and `S` may not. Head relation name `p` may not be `time`, `timeSucc`, or `node`. Relations $\texttt{b}_1, \ldots, \texttt{b}_l, \texttt{c}_1, \ldots, \texttt{c}_m$ may not be `timeSucc`, `time`, or `<`.

The first kind is a *deductive* rule, the second an *inductive* rule, and the last an *asynchronous rule*. The last two kinds of rules are collectively called *temporal* rules.

The use of a single location specifier and timestamp in rule bodies corresponds to considering deductions that can be evaluated at a single node at a single point in time.

The `choice` construct is from Saccà and Zaniolo [10] and is used to model the fact that the network may arbitrarily delay, re-order, and batch messages. We use the causality rewrite of Alvaro *et al.* [9], which restricts `choice` in the following way: a message sent by a node x at local timestamp s cannot cause another message to arrive in the past of node x (i.e., at a time before local timestamp s). Intuitively, the causality constraint rules out models corresponding to impossible executions, in which effects are perceived before their causes. Full details about `choice` and the causality constraint are available in a companion paper [9].

A Dedalus *program* is a finite set of causally rewritten spatio-temporal rules over some spatio-temporal schema $\mathcal{S}^*$.

Syntactic Sugar. The restrictions on timestamps and location specifiers suggest a natural syntactic sugar that omits boilerplate usage of timestamp attributes and location specificers, as well as the use of `node`, `time`, `timeSucc`, and `choice` in rule bodies. Example deductive, inductive, and asynchronous rules are shown below.

$$\texttt{p}(\overline{\texttt{W}}) \leftarrow \texttt{b}_1(\overline{\texttt{X}_1}),\ \ldots,\ \texttt{b}_l(\overline{\texttt{X}_l}),\ \neg\texttt{c}_1(\overline{\texttt{Y}_1}),\ \ldots,\ \neg\texttt{c}_m(\overline{\texttt{Y}_m}).$$
$$\texttt{p}(\overline{\texttt{W}})\texttt{@next} \leftarrow \texttt{b}_1(\overline{\texttt{X}_1}),\ \ldots,\ \texttt{b}_l(\overline{\texttt{X}_l}),\ \neg\texttt{c}_1(\overline{\texttt{Y}_1}),\ \ldots,\ \neg\texttt{c}_m(\overline{\texttt{Y}_m}).$$
$$\texttt{p}(\overline{\texttt{W}})\texttt{@async} \leftarrow \texttt{b}_1(\overline{\texttt{X}_1}),\ \ldots,\ \texttt{b}_l(\overline{\texttt{X}_l}),\ \neg\texttt{c}_1(\overline{\texttt{Y}_1}),\ \ldots,\ \neg\texttt{c}_m(\overline{\texttt{Y}_m}).$$

In any rule, the body location specifier can be accessed by including a variable symbol or constant prefixed with `#` as any body atom's first argument. In asynchronous rules, the head location specifier can be accessed in a similar manner in the head atom, as shown in the following rule.

```
p(#D,L,W)@async ← b(#L,D,W), ¬c(#L,L).
```

The head and body location specifiers are `D` and `L` respectively. `D` may appear in the body, `L` may appear in the head, and `L` may appear duplicated in the body.

2.2 Semantics

We only consider Dedalus programs whose deductive rules are syntactically stratified.

An *input schema* $\mathcal{S}^I$ for a Dedalus program P with spatio-temporal schema $\mathcal{S}^*$ is a subset of P's spatial schema $\mathcal{S}^+$. Every input schema contains the `node` relation; we will not explicitly mention the presence of `node` when detailing an input schema. A relation in $\mathcal{S}^I$ is called an *EDB relation*. All other relations are called *IDB*.

An *EDB instance* $\mathcal{E}$ is a spatial database instance that maps each EDB relation `r` to a finite spatial relation instance for `r`. The *active domain* of an EDB instance $\mathcal{E}$ for a program P is the set of constants appearing in $\mathcal{E}$ and P. Every EDB instance maps the $<$ relation to a total order over its active domain. We can view an EDB instance as a spatio-temporal database instance $\mathcal{K}$. For every $\texttt{r(d,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{E}$, the fact $\texttt{r(d,t,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{K}$ for all $\texttt{t} \in \mathbb{N}$. Intuitively, EDB facts "exist at all timesteps."

We refer to a Dedalus program together with an EDB instance as a Dedalus *instance*. The behavior of a Dedalus program can be viewed as a mapping from EDB instances to spatio-temporal database instances. We use the *stable model semantics* to describe this mapping. Intuitively, there is a one to one correspondence between stable models and values for timestamps for all messages that obey the causality rewrite [9].

Example 1. Take the following Dedalus program with input schema {q}. Assume the EDB instance is {node(n1), q(n1)}.

```
p(#L)@async ← q(#L).
```

Let the power set of X be denoted $\mathcal{P}(X)$. For each $S \in \mathcal{P}(\mathbb{N} \setminus \{0\})$, where $|S| = |\mathbb{N}|$, the following are exactly the stable models: $\{\texttt{node(n1)}\} \cup \{\texttt{p(n1,i)} \mid \texttt{i} \in S\} \cup \{\texttt{q(n1,i)} \mid \texttt{i} \in \mathbb{N}\}$.

Since q is part of the input schema, it is true at every time. Every time involves a separate choice of time for p, which must be later than time 0. The causality constraint rules out elements of the power set with finite cardinality [9].

Ultimate Models. Stable models highlight uninteresting temporal differences that may not be "eventually" significant. Intuitively, there would be different stable models for different message orderings, even when those orderings were not meaningful because they represented some commutative operation. An example appears in Appendix F of [14]. In order to rule out such behaviors from the output, we will define the concept of an *ultimate model* to represent a program's "output."

An *output schema* for a Dedalus program P with spatio-temporal schema $\mathcal{S}^*$ is a subset of P's spatial schema $\mathcal{S}^+$. We denote the output schema as $\mathcal{S}^O$.

Let $\Diamond\Box$ map spatio-temporal database instances $\mathcal{T}$ to spatial database instances. For every spatio-temporal fact $\texttt{r(p,t,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{T}$, the spatial fact $\texttt{r(p,c}_1\texttt{,...,c}_n\texttt{)} \in \Diamond\Box\mathcal{T}$ if relation r is in $\mathcal{S}^O$ and $\forall \texttt{s}\,.\,(\texttt{s} \in \mathbb{N} \wedge \texttt{t} < \texttt{s}) \Rightarrow (\texttt{r(p,s,c}_1\texttt{,...,c}_n\texttt{)} \in \mathcal{T})$. The set of *ultimate models* of a Dedalus instance I is $\{\Diamond\Box(\mathcal{T}) \mid \mathcal{T}$ is a stable model of I$\}$. Intuitively, an ultimate model contains exactly the facts in relations in the output schema that are eventually always true in a stable model.

Note that an ultimate model is always finite because of the finiteness of the EDB, the safety conditions on rules, the restrictions on the use of timeSucc and time, and the prohibition on binding timestamps to non-timestamp attributes. A Dedalus program only has a finite number of ultimate models for the same reason.

Example 2. For Example 1 with $\mathcal{S}^O = \{\texttt{p}\}$, there are two ultimate models: {} and {p(n1)}. The latter corresponds to an element of the power set S such that $\exists x\,.\,\forall y\,.\,(y > x) \Rightarrow (y \in S)$. The former corresponds to an element S that does not have this property.

3 Refining Dedalus

Dedalus can express a broad class of distributed systems but this flexibility comes at a cost. As we have shown, a Dedalus program may have multiple ultimate models. However, it is often desirable to ensure that a program has a single, deterministic output, regardless of non-determinism in its behavior.

Example 3. A simple asynchronous marriage ceremony:

```
i_do(X)@async ← i_do_edb(X).
runaway() ← ¬i_do(bride), i_do(groom).
runaway() ← ¬i_do(groom), i_do(bride).
runaway()@next ← runaway().
i_do(X)@next ← i_do(X).
```

The intended meaning of the program is that the marriage is off (`runaway()` is true) if one party says "I do" and the other does not. However, the Dedalus program as given does not match this specification. Any stable model where `i_do(groom)` and `i_do(bride)` disagree in their first chosen timestamps yields an ultimate model containing `runaway()`. If the votes are assigned the same timestamp, the ultimate model does not contain `runaway()`. See Appendix A of [14] for a version of this example involving asynchrony.

In this case, there is a preferred model where negation is not applied to a set until its content has been fully determined. This is akin go the notion of a perfect model in Datalog. Typically, a programmer would induce this preferred model by inserting *coordination* code (e.g., voting or consensus between all communicating agents) to ensure that there are no outstanding messages in flight, before applying a summarizing operation like negation.

In the remainder of this section, we explore the aspects of Dedalus that allow such ambiguities and propose a restricted language Dedalus$^+$ that rules them out (but complicates the specification of programs). In Section 4, we consider a different language—Dedalus^S—that allows relatively intuitive program specifications like our examples, but narrows their interpretation to a single, "preferred" model.

3.1 Problems with Dedalus

A Dedalus program is *confluent* if, for every EDB instance, it has a single ultimate model. A program that is not confluent is *diffluent*. Confluence is a desirable, albeit conservative, correctness property for a distributed program. A program that is confluent produces deterministic output despite any non-deterministic behaviors that might occur during its execution. For example, if we could show that a data replication protocol was confluent, we could prove a version of the commonly desired property that all replicas be "eventually consistent" after all messages have been delivered [16, 17]. Confluence may be viewed as a specialization of the more general notion of consistency of distributed state.

Lemma 1. *Confluence of* Dedalus *programs is undecidable.*

This result is hardly surprising, as confluence is defined over all EDB instances. Another symptom of Dedalus being "too big" a language is its expressive power.

Lemma 2. Dedalus *subsumes PSPACE.*

3.2 Dedalus$^+$

Distributed programs that produce non-deterministic output or have exponential runtimes are often undesirable. Since checking for confluence in Dedalus is undecidable, we present a restriction of Dedalus that allows only confluent programs and prove that it captures exactly PTIME.

A Dedalus program is *semipositive* if the $\neg$ symbol is only used on EDB relations. A Dedalus program P has *guarded asynchrony* if for every relation `p` appearing in the head of an asynchronous rule, the program P has a rule `p(`$\bar{X}$`)@next` $\leftarrow$ `p(`$\bar{X}$`)`. The language of semipositive Dedalus programs with guarded asynchrony is called Dedalus$^+$.

To show that all DEDALUS$^+$ programs are confluent, we begin by showing that DEDALUS$^+$ programs are *temporally inflationary*: if a stable model of a DEDALUS$^+$ instance contains a spatio-temporal fact `f@t`, then it also contains `f@t+1` (and thus the ultimate model contains `f`).

Lemma 3. DEDALUS$^+$ *programs are temporally inflationary.*

Theorem 1. DEDALUS$^+$ *programs are confluent.*

Since a DEDALUS$^+$ program has a unique ultimate model, the specific choice of values for timestamps does not affect the ultimate model. In particular, we can assume that the `timeSucc` of the body timestamp is always chosen.

Corollary 1. *Define $\mathcal{A}(P)$ to be the program transformation that converts every asynchronous rule φ of* DEDALUS$^+$ *program P into an inductive rule by undoing the causality and* `choice` *rewrites, dropping the* `choice` *operator, and adding* `timeSucc(T,S)` *to $pos(\varphi)$. Then, the ultimate model of $\mathcal{A}(P)$ is the same as the ultimate model of P.*

Of course, there are confluent DEDALUS programs not in DEDALUS$^+$ (see Appendix E of [14]). Not only are DEDALUS$^+$ programs confluent, but they also capture exactly PTIME.

Lemma 4. *Define the program transformation $\mathcal{I}(P)$ in the following way: in every inductive rule of* DEDALUS$^+$ *program P—except any basic persistence rule for a relation that appears in the head of an asynchronous rule—remove the* `timeSucc(T,S)` *body atom, and replace all instances of the variable* `S` *with the variable* `T`*. The ultimate model of $\mathcal{I}(P)$ is the same as the ultimate model of P.*

Theorem 2. DEDALUS$^+$ *captures exactly PTIME.*

4 DEDALUS^S

The marriage program from Example 3 uses IDB negation to determine the truth value of `runaway`, and is thus not directly expressible in DEDALUS$^+$. To avoid using IDB negation, we can rewrite the program to “push down” negation to the EDB relations `groom_i_do` and `bride_i_do`, and then derive the `runaway` IDB relation positively as shown in Example 4.

While the rewrite is straightforward, a majority of the program’s rules need to be modified. Note that since Example 4 is in DEDALUS$^+$, it is confluent; therefore, it is not subject to the non-deterministic output observed in the original program (Example 3).

Example 4. An asynchronous marriage ceremony without IDB negation:

```
i_dont(X)@async ← ¬i_do_edb(X).
runaway() ← i_dont(bride).
runaway() ← i_dont(groom).
runaway()@next ← runaway().
i_dont(X)@next ← i_dont(X).
```

Programs involving negation of recursion, such as the distributed garbage collection program presented in Appendix B of [14], present a more difficult problem, as negation must be pushed down through recursion. The best known techniques for this may result in unacceptable overhead as they involve doubling the arity of relations.

In general, the restriction of negation to EDB relations presents a significant barrier to expressing practical programs. In this section, we introduce DEDALUS^S, an extension of DEDALUS$^+$ that allows stratified IDB negation. As one might expect, DEDALUS^S retains the benefits of DEDALUS$^+$. We provide an operational semantics for DEDALUS^S, based on the one for DEDALUS [9], inspired by coordination protocols from distributed systems.

4.1 Safe IDB Negation

The stratified semantics for logic programs with negation is both intuitive and corresponds to common distributed systems practices: negation is not applied until the negated relation is "done" being computed. After some preliminary definitions, we introduce a semantics for stratifiable DEDALUS programs.

The PDG of a DEDALUS program P with spatio-temporal schema $\mathcal{S}^*$ is a directed graph with one node per relation; each node i has a label $L(i)$. If node i represents relation `p`, then $L(i) = $ `p`. There is an edge from the node with label `q` to the node with label `p` if relation `p` appears in the head of a rule with `q` in its body. If some rule with `p` in the head and `q` in the body is asynchronous (resp. inductive), then the edge is said to be *asynchronous* (resp. *inductive*). If some rule with `p` in the head has `¬q` in its body, then the edge is said to be *negated*. Collectively, asynchronous and inductive edges are referred to as *temporal edges*. The PDG does not contain nodes for the `node`, `timeSucc`, or `time` relations, or any relation introduced in the causality [9] or `choice` [10] rewrites.

DEDALUS^S is the language of DEDALUS programs with guarded asynchrony whose PDG does not contain any cycles through negation. As is standard, a DEDALUS^S program can be partitioned into *strata*. The *stratum* of a relation `r` is the largest number of negated edges on any path from `r`. Each stratum of an n-stratum DEDALUS^S program can be viewed as a DEDALUS$^+$ program. Stratum i's program, P_i, consists of all rules whose head relation is in stratum i. The output schema of P_i contains all relations in stratum $i + 1$, and P_i's EDB contains all relations in stratum $j < i$. P_0's EDB contains all EDB relations. P_n's output schema contains all relations in P's output schema.

The *ultimate model* of a DEDALUS^S program is the ultimate model $P_n(\ldots P_1(P_0(E))\ldots)$, obtained by a stratum-order evaluation.

Since a DEDALUS^S program is a straightforward composition of DEDALUS$^+$ programs, we can apply several previous results. Note that DEDALUS^S programs are temporally inflationary.

Corollary 2. DEDALUS^S *programs are confluent.*

Note that every DEDALUS$^+$ program is a DEDALUS^S program, and every DEDALUS^S program has a constant number of strata in the size of its input. Thus we have:

Corollary 3. DEDALUS^S *programs capture exactly PTIME.*

4.2 Coordination Rewrite

While the model-theoretic semantics of Dedalus^S are clear, its negation semantics are different than those of Dedalus. Thus, we cannot directly apply the correspondence to a distributed operational semantics in Alvaro *et al.* [9]. Fortunately, we can rewrite any Dedalus^S program to a Dedalus program.

Given a Dedalus^S program S, the *coordination rewrite* $\mathcal{P}(S)$ of S is the Dedalus program obtained by adding `p_done()` to the body of any rule in S that contains a `¬p(...)` atom and adding rules to define `p_done()` as described below.

We will see that `p_done()` has the property that in any stable model $\mathcal{M}$ if `p_done(l,t)` $\in \mathcal{M}$, then `p_done(l,s)` $\in \mathcal{M}$ for all timestamps `s` $>$ `t`. Furthermore, if `p_done(l,t)` $\in \mathcal{M}$, then `p(l,s,c`$_1$`,...,c`$_n$`)` $\in \mathcal{M}$ implies that `p(l,t,c`$_1$`,...,c`$_n$`)` $\in \mathcal{M}$ for all timestamps `s` $>$ `t`. Intuitively, `p_done()` is true when the content of `p` is *sealed* (henceforth unchanging).

A *collapsed PDG* of a Dedalus program P is the graph obtained by replacing each strongly connected component of the PDG of P with a single node i, such that $L(i)$ comprises the set of all relations from the component. If a strongly connected component has any asynchronous edges, we call the resulting collapsed node *async recursive*. Each node in the collapsed PDG whose label contains a relation names in $\mathcal{S}^O$ is called an *output* node. Note that a collapsed PDG is acyclic.

For EDB relations `p`, the rule for `p_done` is `p_done()`. For IDB relations `p`, we present `p_done()` for non-async-recursive nodes and async recursive nodes separately.

Non-Async-Recursive Nodes. For non-async-recursive nodes, we compute a `done` fact for each rule, then collate these into a `done` fact for each relation. The `done` fact for a deductive rule is true when all of the relations in the body of the rule are henceforth unchanging. We assume guarded asynchrony applies to the rules in this section.

Let i be a non-async-recursive node. Repeat the following for each element of `p` $\in L(i)$. Assume the rules in P with head relation `p` are numbered $1, \dots, i_p$.

The rule for `p_done()` is: `p_done()` $\leftarrow$ `r`$_1$`_done(), ..., r`$_{i_p}$`_done()`.

Let the nodes in the collapsed PDG connected via incoming edges to node i be denoted by $E(i)$. Let the relations $\bigcup_{k \in E(i)} L(k)$ be named `p`$_1, \dots,$ `p`$_{i_q}$. For each rule $1 \leq j \leq i_p$ in P with head relation `p`, handle rule j according to the cases below.

Deductive: Add the rule: `r`$_j$`_done()` $\leftarrow$ `p`$_1$`_done(), ..., p`$_{i_q}$`_done()`.

Asynchronous: For an asynchronous rule, we need to ensure that there are no messages that have not yet been delivered, before we derive `r`$_j$`_done()`. We do this by adding rules to record all sent messages, and rules for receivers to send acknowledgements back to senders. When a sender has received an acknowledgement for each sent message, and there are no more messages to send, he indicates this to the receiver. In the vacuous case where a sender has no messages to send to a receiver, he also indicates this to the receiver. When a receiver has been notified by all nodes that there are no in-flight messages, he can derive `r`$_j$`_done()`. The rules to express this protocol are in Appendix D of [14].

Async Recursive Nodes. The difficulty with a relation `p` in an async recursive node is that `r` is done when all messages have been received in the node, and all messages have been received if `p` is done. To circumvent this circular dependency, we introduce a specialized two-phase voting protocol.

Consider an async recursive node i. Let the asynchronous rules with head relations in $L(i)$ be numbered $1, \dots, i_p$. Add the rule: $\texttt{all\_ack}_i\texttt{()} \leftarrow \texttt{r}_1\texttt{\_done()}, \dots, \texttt{r}_{i_p}\texttt{\_done()}$.

For each rule j, add the rules for asynchronous rules shown in Appendix D of [14], except for the last two rules. Instead write:

```
r_j_not_done() ← p_j_to_send(X̄), ¬p_j_ack(X̄).
r_j_done() ← ¬r_j_not_done().
```

We perform a two-round voting protocol among the nodes; the node with the minimum identifier is the master. We assume that guarded asynchrony does not apply to the relations in the head of any asynchronous rule in the following protocol. The rules shown below begin the first round of voting. Nodes vote $\texttt{complete\_1}_i$ if $\texttt{all\_ack}_i$ is true—if they have no outstanding unacknowledged messages. Votes are sent to the master.

```
not_node_min(L1) ← node(L1), node(L2), L2 < L1.
node_min(L) ← ¬not_node_min(L), node(L).
start_round_1_i() ← node_min(#L,L), ¬round_1_i().
round_1_i()@next ← start_round_1_i().
round_1_i()@next ← round_1_i(), ¬start_round_2_i().
vote_1_i(#N)@async ← start_round_1_i(), node(N).
complete_1_i(#M,N)@async ← vote_1_i(#N), all_ack_i(#N), node_min(#N,M).
incomplete_1_i(#M,N)@async ← vote_1_i(#N), ¬all_ack_i(#N), node_min(#N,M).
```

To persist votes until round 1 begins again, these rules are instantiated for $k = 1$ and 2.

```
complete_k_i(N)@next ← complete_k_i(N), ¬start_round_1_i().
incomplete_k_i(N)@next ← incomplete_k_i(N), ¬start_round_1_i().
```

To count votes, we assume the following rules are instantiated for $k = 1$ and 2. Round 1 is restarted if some node votes $\texttt{incomplete\_1}_i$ in round 1—i.e., it has an outstanding unacknowledged message – or $\texttt{incomplete\_2}_i$ in round 2.

```
recv_k_i(N) ← complete_k_i(N).
recv_k_i(N) ← incomplete_k_i(N).
not_all_recv_k_i() ← node(N), ¬recv_k_i(N).
not_all_comp_k_i() ← node(N), ¬complete_k_i(N).
start_round_1_i() ← ¬not_all_recv_k_i(), not_all_comp_k_i().
```

Once a node has received a $\texttt{vote\_1}_i$ vote solicitation, it starts keeping track of whether it has sent any messages in the async recursive component; this information is erased if another $\texttt{vote\_1}_i$ solicitation is received. The causality constraint ensures that $\neg\texttt{all\_ack}_i\texttt{()}$ is true if a message is sent, as messages cannot be instantly acknowledged.

```
sent_i() ← ¬all_ack_i().
sent_i()@next ← sent_i(), ¬vote_1_i().
```

Round 2 is started by the master if no node has an outstanding message.

```
start_round_2_i() ← ¬not_all_recv_1_i(), ¬not_all_comp_1_i(), node_min(#L,L).
```

The voting for round 2 is shown below. Nodes vote $\texttt{incomplete\_2}_i$ if they have sent any messages since the last $\texttt{vote\_1}_i$ solicitation. Recall that any $\texttt{incomplete\_2}_i$ votes result in the protocol restarting with round 1.

```
vote_2_i(#N)@async ← start_round_2_i(), node(N).
complete_2_i(#M,N)@async ← vote_2_i(#N), all_ack_i(#N), ¬sent_i(#N), node_min(#N,M).
incomplete_2_i(#M,N)@async ← vote_2_i(#N), sent_i(#N), node_min(#N,M).
```

The entire async recursive node i is done when all nodes have voted $\texttt{complete\_2}_i$.

```
done_recursion_i() ← ¬not_all_recv_2_i(), ¬not_all_comp_2_i().
```

Finally, for every relation $\texttt{p} \in L(i)$, add the rule: $\texttt{p\_done()} \leftarrow \texttt{done\_recursion}_i\texttt{()}$.

This program transformation produces a DEDALUS$^+$ program equivalent to any DEDALUS^S program. The rules for computing `p_done` have the desired effect.

Lemma 5 (Sealing). *Assume a* DEDALUS^S *program S with relation* `p`. *The* DEDALUS *program $\mathcal{P}(S)$ contains a relation* `p_done` *with the following property: in any of its stable models $\mathcal{M}$, if* `p_done(1,t)` $\in \mathcal{M}$, *then* `p_done(1,s)` $\in \mathcal{M}$ *for all timestamps* `s` $>$ `t`. *Furthermore, if* `p_done(1,t)` $\in \mathcal{M}$, *then* `p(1,s,`c_1`,...,`c_n`)` $\in \mathcal{M}$ *implies that* `p(1,t,`c_1`,...,`c_n`)` $\in \mathcal{M}$ *for all timestamps* `s` $>$ `t`.

The above Lemma implies that the ultimate model of DEDALUS^S program S is the same as the ultimate model of DEDALUS program $\mathcal{P}(S)$, as relations in lower strata are complete before higher strata rules are satisfiable. See Appendix C of [14] for an example of applying the program transformation $\mathcal{P}$.

In distributed systems, global computation barriers are commonly enforced by protocols based on voting: the two-phase commit protocol from distributed databases is a straightforward example [18]. In the protocol from the program transformation $\mathcal{P}$, every agent responsible for a fragment of the global state must "vote" that every message they send to the coordinator has been acknowledged. The coordinator must tally these votes and ensure that the vote is unanimous for all agents. If the protocol completes successfully, the coordinator may proceed past the barrier.

5 Related Work

The purely declarative semantics of DEDALUS, based on the reification of logical time into facts, are close in spirit and interpretation to Statelog [19], the languages proposed by Cleary and Liu [20–22], and work in temporal deductive databases [23].

Significant recent work ([2–5]) has focused on using deductive database languages extended with networking primitives to specify and implementing network protocols and distributed systems. Theorem 1 resembles the correctness proof of "pipelined semi-naive evaluation" for distributed Datalog presented by Loo *et al.* [24]. In general, however, the language extensions proposed in much of this prior work added expressivity and domain applicability but compromised declarativity, making formal analysis difficult [25, 7].

Recently, Ameloot *et al.* explored Hellerstein's CALM theorem using relational transducers [6]. They proved that monotonic first-order queries are exactly those queries that can be computed in a coordination-free fashion using transducers. Some of their assumptions differ from ours—for example, they assume that all messages sent by a node are multicast to a fixed neighbor set, whereas DEDALUS permits arbitrary unicast.

Abiteboul *et al.* recently proposed Webdamlog [12], another distributed variant of Datalog that bears many similarities to DEDALUS. They demonstrate that Webdamlog has an operational semantics similar to the operational semantics in DEDALUS [9], and provide conservative conditions for confluence based on a variant of (node-local) stratification. Our work additionally provides a model-theoretic semantics for DEDALUS^S that corresponds to the operational semantics. DEDALUS^S programs (which are guaranteed to be confluent) also admit a broader use of negation—ensured via a synthesized coordination protocol—than the stratification conditions of Webdamlog.

References

1. Hellerstein, J.M.: The Declarative Imperative: Experiences and Conjectures in Distributed Logic. SIGMOD Rec 39, 5–19 (2010)
2. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.C.: BOOM Analytics: Exploring Data-centric, Declarative Programming for the Cloud. In: EuroSys (2010)
3. Belaramani, N., Zheng, J., Nayate, A., Soulé, R., Dahlin, M., Grimm, R.: PADS: A policy architecture for distributed storage systems. In: NSDI (2009)
4. Chu, D.C., Popa, L., Tavakoli, A., Hellerstein, J.M., Levis, P., Shenker, S., Stoica, I.: The design and implementation of a declarative sensor network system. In: SenSys (2007)
5. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Communications of the ACM 52(11), 87–95 (2009)
6. Ameloot, T.J., Neven, F., Van den Bussche, J.: Relational Transducers for Declarative Networking. In: PODS (2011)
7. Navarro, J.A., Rybalchenko, A.: Operational Semantics for Declarative Networking. In: Gill, A., Swift, T. (eds.) PADL 2009. LNCS, vol. 5418, pp. 76–90. Springer, Heidelberg (2008)
8. Pérez, J.A.N., Rybalchenko, A., Singh, A.: Cardinality Abstraction for Declarative Networking Applications. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 584–598. Springer, Heidelberg (2009)
9. Alvaro, P., Ameloot, T.J., Hellerstein, J.M., Marczak, W., Van den Bussche, J.: A Declarative Semantics for Dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley (November 2011)
10. Saccà, D., Zaniolo, C.: Stable Models and Non-Determinism in Logic Programs with Negation. In: PODS, pp. 205–217 (1990)
11. Alvaro, P., Conway, N., Hellerstein, J.M., Marczak, W.R.: Consistency Analysis in Bloom: a CALM and Collected Approach. In: CIDR (2011)
12. Abiteboul, S., Bienvenu, M., Galland, A., Antoine, E.: A rule-based language for web data management. In: PODS (2011)
13. Bloom programming language, http://www.bloom-lang.org
14. Marczak, W., Alvaro, P., Conway, N., Hellerstein, J.M., Maier, D.: Confluence analysis for distributed programs: A model-theoretic approach. Technical report, EECS Department, University of California, Berkeley (June 2012)
15. Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in Time and Space. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 262–281. Springer, Heidelberg (2011)
16. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: SOSP (1995)
17. Terry, D.B., Demers, A.J., Petersen, K., Spreitzer, M., Theimer, M., Welch, B.W.: Session Guarantees for Weakly Consistent Replicated Data. In: Proceedings of the Third International Conference on Parallel and Distributed Information Systems, PDIS 1994, pp. 140–149. IEEE Computer Society, Washington, DC (1994)
18. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques (1993)
19. Lausen, G., Ludäscher, B., May, W.: On Active Deductive Databases: The Statelog Approach. In: Kifer, M., Voronkov, A., Freitag, B., Decker, H. (eds.) DYNAMICS 1997, and ILPS-WS 1997. LNCS, vol. 1472, pp. 69–106. Springer, Heidelberg (1998)
20. Cleary, J.G., Utting, M., Clayton, R.: Data Structures Considered Harmful. In: Australasian Workshop on Computational Logic (2000)

21. Liu, M., Cleary, J.: Declarative Updates in Deductive Databases. Journal of Computing and Information 1, 1435–1446 (1994)
22. Lu, L., Cleary, J.G.: An Operational Semantics of Starlog. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 131–162. Springer, Heidelberg (1999)
23. Chomicki, J., Imieliński, T.: Temporal Deductive Databases and Infinite Objects. In: PODS, pp. 61–73 (1988)
24. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative Networking: Language, Execution and Optimization. In: SIGMOD (2006)
25. Mao, Y.: On the declarativity of declarative networking. In: NetDB (2009)

Business Network Reconstruction Using Datalog

Daniel Ritter[1] and Till Westmann[2]

[1] SAP AG, Technology Development – Process and Network Integration,
Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
daniel.ritter@sap.com
[2] 28msec Inc,
200 Sheridan Avenue, Palo Alto CA 94306, USA
till.westmann@28msec.com

Abstract. The discovery, representation and reconstruction of Business Networks (BN) from Network Mining (NM) raw data is a difficult problem for enterprises. This is due to e.g. complex business processes within and across enterprise boundaries, heterogeneous technology stacks, and fragmented data. To remain competitive, visibility into the enterprise and partner networks on different, interrelated abstraction levels is desirable.

We present an approach to represent and reconstruct one part of the BN - the (technical) integration networks - from NM raw data using Datalog. The raw data expressed as integration network model is represented as Datalog facts, on which Datalog rules are applied to infer and thus reconstruct the network. We have built a system that is used to apply this approach to real-world enterprise landscapes and we report on our experience with this system.

Keywords: Business Network, Datalog, Knowledge Representation, Linked Data, Logic Programming, Network Inference, Network Mining.

1 Introduction

Enterprises are part of value chains consisting of business processes connecting intra- and inter-enterprise participants. The network that connects these participants with their technical, social and business relations is called a *Business Network*. Even though this network is very important for the enterprise, there are few - if any - people in the organization who understand this network as the relevant data is hidden in heterogeneous enterprise system landscapes. To change that, *Network Mining* (NM) systems are used to discover and extract raw data [15] - be it technical data (e.g. configurations of integration products like Enterprise Service Buses (ESB) [12]) or business data (e.g. information about a supplier in a Supplier Relationship Management (SRM) product). The task at hand is to reconstruct the "as-is" Business Networks from this incomplete, fragmented, cross-domain NM data.

In the system description we present an approach to represent and reconstruct the technical aspect of a Business Network, called *Integration Network*, from discovered raw data using Datalog. We describe how the knowledge hidden in the NM raw data can be represented independent of their original domains as integration model, for which we have chosen a relational representation. To compute the network we use Datalog rule-based inference programs, which contain rules to e.g. identify entity equivalences, compute edges and semantic references and deal with human additions. We have validated our approach on simulated integration network data and report our experience with the network inference Datalog system in real-world enterprise networks.

In section 2 design principles and decisions are discussed. Section 3 sketches the integration model and inference approach, and section 4 describes our experience with real-world customer landscape reconstruction. Section 5 concludes with related work, before we summarize and give an outlook in section 6.

2 Design Principles and Decisions

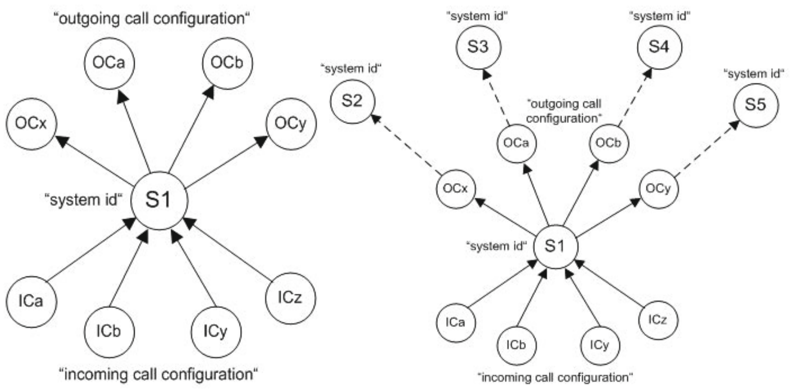
The major design decisions taken were about finding a representation for an integration model and a language to express inference algorithms. We needed to select (1) an approach, which does not require to modify the system when changing the inference programs or the integration model, (2) a well-understood representation for information suitable for the inference approach, and (3) a sufficiently powerful inference technique, simple enough to be used by our customers and partners to define their own inference programs.

The necessity of (1) is derived from developing the inference programs in the early prototypes. The domain of the data and the scope of inference evolved - and it will continue to do so as more data sources are integrated and inference is refined. Hence the lifecycle of the data model and of the inference programs needs to be decoupled from that of the system. Since system landscapes and business networks for large enterprises are very complex and many implementations need customer-specific modifications or extensions both (2) and (3) are required. As the relational model is a foundation for most business applications and is thus well-understood by customers, it is a natural choice for (2). Consequently, we initially considered SQL and its imperative extensions to express inference programs. However, as network analysis and inference are expressed more naturally using recursive rules we moved towards logic programming languages like Prolog or Datalog, choosing Datalog for its simpler semantics.

3 Network Representation and Inference Approach

The integration network model as virtual "as-is" enterprise landscape covers a representative intersection of entities from the enterprise integration middleware space [12]. Although this domain has many aspects, which are treated differently in different system implementations, we identified a common integration model. The basic entities relevant for the inference are nodes, i.e. logical entities like applications or tenants, called *System*, running on physical hosts, and edges representing the communication between systems via messages, the

MessageFlow. Technically, messages are exchanged over interfaces, *Interface*, and channels, containing e.g. services, bindings and operations, which we represent as *IncomingConfiguration* and *OutgoingConfiguration*. The inbound and outbound configurations are considered separate entities, since they carry important information about the message flows, thus helping to reconstruct the edges.


"outgoing call configuration"
OCa
OCb
OCx
OCy
"system id"
S1
ICa
ICb
ICy
ICz
"incoming call configuration"
"system id"
S2
S3
S4
S5
"outgoing call configuration"
OCa
OCb
OCx
OCy
"system id"
S1
ICa
ICb
ICy
ICz
"incoming call configuration"

(a) Config. call graph (b) Call graph extension

Fig. 1. Outgoing and incoming configuration call graphs

The algorithm for computing integration networks consists of multiple steps, which have been identified for a parallel analysis allowing it to scale across large datasets of mined data. The inference mechanism is independent of the specific integration and system domains. Unique systems and hosts are identified by equivalence algorithms and semantic links between hosts and systems are computed. Based on that, incoming and outgoing configurations are identified (see Fig. 1(a)) and then used to reconstruct message flows through building separate call graphs (see Fig. 1(b)) which are merged afterwards.

Listing 1.1. Message flow from outgoing configuration

```
msg_flow(?sys_id_snd, ?sys_id_recv) :-
    outgoing_disc(?sys_id_snd, ?RCONF),
    receiver_disc(?RCONF, ?sys_id_recv).
```

Listing 1.1 shows a Datalog rule used to reconstruct message flows (*msg_flow*) within the call graph exploiting relations of unique entity identifiers of outgoing call configurations of the sender (*outgoing_disc*) and the receiver systems (*receiver_disc*). Then message flows are linked with application and integration content and custom knowledge is integrated. With custom knowledge, the quality of the inference mechanism can be improved and information complemented or enriched.

4 Results and Experiences

For our system we developed a basic Datalog engine in Java/OSGi based on [18], that allows to evaluate recursive rules and supports basic data types, comparisons and expressions. We applied the system to real-world customer landscapes containing mediated communication through middleware systems, direct connectivity, e.g. web services, and system landscape information. This real-world validation allowed the evaluation of cross-middleware inference, combination of embedded and mediated communication and fragmented information registered in different domains. With that we showed that the NM auto-discovery and inference is feasible and resulted in highly reliable results. Moreover, our system is helpful in the everyday work of integration experts, since it gives an overview of the complete "as-is" network, which is very difficult using existing middleware tools. The system reduces the effort to document integration scenarios and helps to answer questions where answers are difficult to find today. For instance, when combining configuration and runtime data, it is possible to find unused and possibly obsolete interfaces and flows. Hence several customers plan to use this system in their upgrade projects of their middleware content, as it will substantially save migration time and effort.

5 Related Work

Our network represention and inference approach are based on Datalog, which is a well-researched topic [9, 18] that had its revival recently due to good parallelization capabilities, latest through [1, 11]. Even in the enterprise analytics domain, Datalog has been applied, mainly through work of [3–5]. However, these approaches address non-network inference domains.

In terms of the integration network model, [17] represents closest known related work, which defines a path algebra used to traverse arbitrary graphs. Similarly we define nodes and edges with inbound and outbound connectors, however different in terms of meaning and usage.

The linked (web) data research shares similar approaches and methodologies, which have so far neglected linked enterprise data and mainly focused on RDF-based approaches [7, 8]. Applications of Datalog in the area of linked data [6, 16] and semantic web [14] show that it is used in the inference domain, however not used for network inference.

6 Summary and Future Work

In this paper we define a modeling and inference approach to reconstruct integration networks from NM raw data using Datalog. The discovered raw data is represented as Datalog facts to create a domain independent knowledge base and applied rule-based inference representing a multi-step network inference approach. We applied our system to real-world enterprise networks.

Future work will be conducted in several areas, among them extensions to basic Datalog like constraints in Datalog [13] for model conformance checks, pruning for efficient evaluation [2] and probability [10] to express levels of certainty of model instances.

References

1. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.C.: BOOMAnalytics: Exploring Data-centric, Declarative Programming for the Cloud. In: EuroSys (2010)
2. Campagna, D., Sarna-Starosta, B., Schrijvers, T.: Approximating Constraint Propagation in Datalog. In: 11th International Colloquium on Implementation of Constraint LOgic Programming Systems (CICLOPS), Lexington, KY (2011)
3. Aref, M.: Datalog for Enterprise Applications – From Industrial Applications to Rese. In: Datalog 2.0 Workshop, Oxford (2010)
4. Aref, M.: LogicBlox for Enterprise Applications. Northern California Database Day (2011)
5. Huang, S.S., Green, T.J., Loo, B.T.: Datalog and Emerging Applications: An Interactive Tutorial. In: SIGMOD (2011)
6. Abiteboul, S.: Distributed data management on the web. In: Datalog 2.0 Workshop, Oxford (2010)
7. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data – The Story so Far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
8. Bizer, C.: The Emerging Web of Linked Data. IEEE Intelligent Systems 24(5), 87–92 (2009)
9. Gallaire, H., Minker, J. (eds.): Symposium on Logic and Data Bases. Advances in Data Base Theory. Plenum Press, New York (1978)
10. Gutmann, B., Thon, I., Kimmig, A., Bruynooghe, M., De Raedt, L.: The magic of logical inference in probabilistic programming. Theory and Practice of Logic Programming (2011)
11. Hellerstein, J.M.: The Declarative Imperative – Experiences and Conjectures in Distributed Logic. Technical Report, Berkeley (2010)
12. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman, Amsterdam (2003)
13. Li, N., Mitchell, J.C.: DATALOG with Constraints: A Foundation for Trust Management Languages. In: Dahl, V. (ed.) PADL 2003. LNCS, vol. 2562, pp. 58–73. Springer, Heidelberg (2002)
14. Motik, B.: Using Datalog on the Semantic Web. In: Datalog 2.0 Workshop, Oxford (2010)
15. Ritter, D.: From Network Mining to Large Scale Business Networks. In: International Workshop on Large Scale Network Analysis, LSNA, Lyon (2012)
16. Polleres, A.: Using Datalog for Rule-Based Reasoning over Web Data: Challenges and Next Steps. In: Datalog 2.0 Workshop, Oxford (2010)
17. Rodriguez, M.A., Neubauer, P.: A Path Algebra for Multi-Relational Graphs. In: International Workshop on Graph Data Management, GDM, Hannover (2011)
18. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. I. Computer Science Press (1988)

Data Exchange in Datalog Is Mainly a Matter of Choice

Domenico Saccà and Edoardo Serra

DEIS, Università della Calabria, 87036 Rende, Italy
sacca@unical.it, eserra@deis.unical.it

Abstract. Data exchange is the problem of migrating a data instance from a source schema to a target schema such that the materialized data on the target schema satisfies the integrity constraints specified by: TGDs (Tuple Generating Dependencies), which are universal quantified formulas with additional existential quantifiers, and EGDs (Equality Generating Dependencies), which are universal quantified formulas enforcing the equality of two variables. This paper presents a formulation of the data exchange problem using DATALOG with choice, which is a non deterministic construct based on stable model semantics. TGDs are represented by rules and a choice predicate is used to non-deterministically select values for the existential variables. Every EGD can be naturally represented by a goal rule. However, as in general it expresses a functional dependency, in this case the goal rule can be replaced by a choice predicate defining the functional dependency inside one of TGD rules. Although classical certainty semantics for query answering in a data exchange setting can be also defined for DATALOG with choice, this paper explores another direction: searching for a solution for which a number of given "sensible" queries have uncertainty-guaranteed answers. The paper discusses properties of privacy-preserving data exchange and illustrates its complexity. Finally, EGDs are extended to express count constraints (e.g, an employee may manage at most k departments instead of only one) and the choice construct is therefore extended to implement count constraints. The resulting setting can be used to define the exchange of aggregate data.

Keywords: Datalog, Nondeterminism, Choice, Data Exchange, Privacy, Aggregate Terms.

1 Introduction

Data exchange [6,1,7] is the problem of migrating a data instance from a source schema to a target schema such that the materialized data on the target schema satisfies the integrity constraints specified by it. The classical data exchange setting is: $(S, T, \Sigma_{st}, \Sigma_t)$, where S is the source relational database schema, T is the target schema, Σ_t are dependencies on the target scheme T and Σ_{st} are source-to-target dependencies.

The dependencies in Σ_{st} map data from the source to the target schema and are TGD (Tuple Generating Dependency), which have the following format: $\forall \mathbf{X}(\ \phi_S(\mathbf{X}) \rightarrow \exists \mathbf{Y}\ \ \psi_T(\mathbf{X}, \mathbf{Y})\ )$, where $\phi_S(\mathbf{X})$ and $\psi_T(\mathbf{X}, \mathbf{Y})$ are conjunctions of literals on S and T, respectively, and $\mathbf{X}, \mathbf{Y}$ are lists of variables.

Dependencies in Σ_t specify constraints on the target schema and can be either TGDs or EGDs (Equality Generating Dependencies) – the latter ones have the form $\forall \mathbf{X}(\,\psi_T(\mathbf{X}) \rightarrow x_1 = x_2\,)$, where x_1 and x_2 are variables in $\mathbf{X}$.

As an example, consider a source schema S with one relation $\texttt{DeptEmp}(\texttt{dpt\_id}, \texttt{mgr\_name}, \texttt{eid})$ listing departments with their managers and their employees. The target schema T has a relation $\texttt{Dept}(\texttt{dpt\_id}, \texttt{mgr\_id}, \texttt{mgr\_name})$ for departments and their managers, and a separate relation for department employees $\texttt{Emp}(\texttt{eid}, \texttt{dpt\_id})$. We assume that each (unique) manager of a department is employed by the same department and, therefore, cannot manage another department. The source-to-target and target dependencies are:

$$\Sigma_{st} = \{\,\texttt{DeptEmp}(\texttt{d},\texttt{n},\texttt{e}) \longrightarrow \exists \texttt{M}\,(\,\texttt{Dept}(\texttt{d},\texttt{M},\texttt{n}) \wedge \texttt{Emp}(\texttt{e},\texttt{d})\,)\,\}$$
$$\Sigma_t = \{\,\texttt{Dept}(\texttt{d},\texttt{m},\texttt{n}) \longrightarrow \texttt{Emp}(\texttt{m},\texttt{d});\;\; \texttt{Dept}(\texttt{d}_1,\texttt{m},\texttt{n}) \wedge \texttt{Dept}(\texttt{d}_2,\texttt{m},\texttt{n}) \longrightarrow \texttt{d}_1 = \texttt{d}_2;$$
$$\texttt{Dept}(\texttt{d},\texttt{m}_1,\texttt{n}_1) \wedge \texttt{Dept}(\texttt{d},\texttt{m}_2,\texttt{n}_2) \longrightarrow \texttt{m}_1 = \texttt{m}_2 \wedge \texttt{n}_1 = \texttt{n}_2\,\}$$

where all low-case letter variables are universally quantified. The dependency in Σ_{st} derives tuples in Dept for all departments, by adding suitable identifiers for their managers, and tuples in Emp for all employees who are not manager. The first dependency in Σ_t adds tuples in Emp for all managers, while the other two enforce the functional dependencies (FDs) $\texttt{mgr\_id} \rightarrow \texttt{dpt\_id}$ and $\texttt{dpt\_id} \rightarrow \texttt{mgr\_id}\;\texttt{mgr\_name}$.

Two problems must be dealt with in order to formulate data exchange dependencies in DATALOG: (1) handling existential quantifiers in TGDs and (2) implementing EGDs. In [18,15] existential quantifiers are removed using skolemization and in [11] EGDs are reformulated using recursive positive logic rules.

In this paper we propose to use the expressive power of DATALOG with choice ([17]) for recasting the data exchange setting in terms of a logic programming paradigm. In particular, we implement existential quantifiers by the choice construct and EGDs by goal rules. To get a flavor of our approach, we present the formulation of the above example:

$$\texttt{Dept}(\texttt{d},\texttt{m},\texttt{n}) \longleftarrow \texttt{DeptEmp}(\texttt{d},\texttt{n},\texttt{e}), \mathcal{D}_\texttt{M}(\texttt{m}), \texttt{choice}((\texttt{d},\texttt{n},\texttt{e}),(\texttt{m})).$$
$$\texttt{Emp}(\texttt{e},\texttt{d}) \longleftarrow \texttt{DeptEmp}(\texttt{d},\texttt{n},\texttt{e}).$$
$$\texttt{Emp}(\texttt{m},\texttt{d}) \longleftarrow \texttt{Dept}(\texttt{d},\texttt{m},\texttt{n}).$$
$$\longleftarrow \texttt{Dept}(\texttt{d}_1,\texttt{m},\texttt{n}), \texttt{Dept}(\texttt{d}_2,\texttt{m},\texttt{n}), \texttt{d}_1 \neq \texttt{d}_2. \tag{1}$$
$$\longleftarrow \texttt{Dept}(\texttt{d},\texttt{m}_1,\texttt{n}_1), \texttt{Dept}(\texttt{d},\texttt{m}_2,\texttt{n}_2), \texttt{m}_1 \neq \texttt{m}_2. \tag{2}$$
$$\longleftarrow \texttt{Dept}(\texttt{d},\texttt{m}_1,\texttt{n}_1), \texttt{Dept}(\texttt{d},\texttt{m}_2,\texttt{n}_2), \texttt{n}_1 \neq \texttt{n}_2. \tag{3}$$

where $\mathcal{D}_M$ is the (possibly countably infinite) domain for manager identifiers. The *choice* constructs implements the existential quantifier as it selects a unique manager id for each triple (d, n, e). The goal rule 1 enforces the FD $\texttt{mgr\_id} \rightarrow \texttt{dpt\_id}$ stated by the first EGD for it derives a contradiction if a same manager manages two distinct departments. Accordingly, the goal rules 2 and 3 enforce the FD $\texttt{dpt\_id} \rightarrow \texttt{mgr\_id}\;\texttt{mgr\_name}$. It is interesting to note that all goal rules can be removed by just rewriting the choice constructs in the first rule, that therefore becomes:

$$\texttt{Dept}(\texttt{d},\texttt{m},\texttt{n}) \longleftarrow \texttt{DeptEmp}(\texttt{d},\texttt{n},\texttt{e}), \mathcal{D}_\texttt{M}(\texttt{m}), \texttt{choice}((\texttt{d}),(\texttt{m},\texttt{n})), \texttt{choice}((\texttt{m}),(\texttt{d})).$$

We point out that the above rewriting is made possible because there exists exactly one rule deriving tuples in `Dept` and, therefore, the FDs can be locally expressed by the choice constructs.

The resulting DATALOG program is stratified modulo *choice* and, then, its so-called choice models are in general multiple but the existence of at least one with finite size as well as its computation in polynomial time is guaranteed [17,10,12]. The example confirms that the *choice* is a powerful construct to define FDs and, therefore, it may be very effective in implementing most of EGDs without using goal rules. In the case EGDs must be instead implemented by a goal rule which a-posteriori enforces the correct choice, the existence of a choice model is not anymore guaranteed.

It is interesting to remark the correspondence between DATALOG program with stratified choice and weakly acyclic set of TGDs. If this set is not weakly acyclic then the corresponding DATALOG program with choice is not stratified so that, not only existence of a choice model is not anymore guaranteed, but there could exist a model with infinite size.

We point out that the choice formulation does not provide a universal solution even though, using suitable technicalities (e.g., adding nulls into the domains), one of the choice model could be considered as the encoding of a universal solution. Note that a universal solution is relevant for answering queries under the certain semantics. But, in this paper we address the problem of privacy-preserving data exchange, so that we are rather interested in guaranteeing "uncertainty" in query answering. To this end, we analyze the complexity of the following problem for the case of finite domains: given a number of (sensible) queries, decide whether each of such queries is true in at most half of the choice models and false in the others. Not surprisingly, this problem is PP-hard and in PSPACE.

Another problem that is dealt with in this paper is extending EGDs to increase their expressive power. An EGD typically imposes a sort of functional dependency – for instance, an employee manages at most one department. A natural extension is to express that an employee manages at most k departments, where k is a given positive integer. We show that count constraints introduced in [16] can be though of as powerful extensions of EGDs and, in addition, can be implemented by a suitable extension of the choice, using set terms and the aggregate set count predicate. Obviously, the notion of universal solution cannot be in general applied to count constraints.

Finally we show that set terms and count predicates can be exploited to implement aggregate data exchange.

The paper is organized as follows. In Section 2 we present basic notation and background for DATALOG with choice. In Section 3 we illustrate how to define a data exchange setting using DATALOG with choice. We then concentrate on privacy-preserving data exchange in Section 4, illustrate the problem of guaranteeing uncertainty in answering sensible queries and prove its complexity. In Section 5 we show that count constraints can be seen as extensions of EGDs and propose an extension of choice that implements a typical count constraint. As count constraints are based on set terms, we also illustrate how to use such terms to deal with aggregate data exchange. Finally we draw the conclusion in Section 6.

2 DATALOG with Choice

We assume that the reader is familiar with basic notions of relational databases, logic programming and DATALOG [19,13,2].

We are given a *universe* U (which is a countable set of constant symbols) and a set S of relation symbols with given finite arities. Let r be any relation symbol in S, say with arity k: a *tuple on* r is any element of U^k, a *relation on* r is any finite set R of tuples on r, and the set of all relations on r is denoted by $inst(r)$. A (*relational*) *database scheme* $\mathbf{D}$ is a set of different relation symbols $r_1, \ldots, r_m\rangle$, with $m > 0$. A (*relational*) *database* D on $\mathbf{D}$ is a set of relations $D(r_1), \ldots, D(r_m)$, where for each i, $1 \leq i \leq m$, $D(r_i) \in inst(r_i)$. The *active domain* of a database D, denoted by U_D, is the set of all constants occurring in D.

A *logic program* P is a finite set of rules c of the form $H(c) \leftarrow B(c)$, where $H(c)$ is an atom (*head* of the rule) and $B(c)$ is a conjunction of literals (*body* of the rule). A rule with empty body is called a *fact*. The *ground instantiation* of P is denoted by $ground(P)$; the *Herbrand universe* and the *Herbrand base* of P are denoted by U_P and B_P, respectively. An *interpretation* is any subset of B_P.

A DATALOG program is a logic program without functions symbols. Predicate symbols can be either extensional (i.e., defined by the facts of a database — *EDB predicate symbols*) or intensional (i.e., defined by the rules of the program — *IDB predicate symbols*). A DATALOG program P has associated a relational database scheme $\mathbf{D}_P$, which consists of all EDB predicate symbols of P. Given a database D on $\mathbf{D}_P$, the tuples of D are seen as facts added to P; so P on D yields the following logic program $P_D = P \cup \{q(t). \ : \ q \in \mathbf{D}_P \wedge t \in D(q)\}$. Given an interpretation I and predicate symbol r in P_D, $I(r) = \{t : r(t) \in I\}$, i.e., if r is seen as a relation symbol, $I(r)$ is a relation on s.

A DATALOG$^\neg$ program is a DATALOG program with negation in the rule bodies. Given a program P and an interpretation I, $pos(P, I)$ denotes the positive DATALOG program that is obtained from $ground(P)$ by (i) removing all rules for which there exists a negative literal $\neg A$ in the body with A is in I, and (ii) by removing all negative literals from the remaining rules. Finally, I is a (*total*) *stable model* [9] if $I = \mathbf{T}^\infty_{pos(P,I)}(\emptyset)$, which is the *least fixpoint* of the classical *immediate consequence transformation* for the positive program $pos(P, I)$.

The complexity of computing a stable model of P_D is measured according to the *data complexity* approach of [3,20] for which the program is assumed to be constant while the database is variable. It is well known that computing a stable model of a DATALOG$^\neg$ program P requires exponential time (unless $\mathtt{P} = \mathtt{NP}$). Actually, deciding whether there exists a stable model or not is NP-complete [14].

A DATALOG$^\neg$ program with stratified negation (i.e. there is no recursion through negation) is called DATALOG$^{\neg s}$. Computing the unique stable model (coinciding with the stratified model) of a DATALOG$^{\neg s}$ program P on a database D can be done in time polynomial on the size of D.

A disciplined form of unstratified negation is the *choice* construct, which is used to enforce functional dependency (FDs) constraints on rules of a logic program and to introduce a form of nondeterminism. The formal semantics of the choice can be given in terms of stable model semantics [17]. A rule c with choice constructs, called a *choice rule*, has the following general format:

$$c: \ A \leftarrow B(Z), choice((X_1),(Y_1)), \ \ldots, choice((X_k),(Y_k)).$$

where, $B(Z)$ denotes the conjunction of all the literals in the body of c that are not choice constructs, and $X_i, \ Y_i, \ Z$, $1 \leq i \leq k$, denote vectors of variables occurring in the body of c such that $X_i \cap Y_i = \emptyset$ and $X_i, Y_i \subseteq Z$. Each construct $choice((X_i),(Y_i))$ prescribes that the set of all consequences derived from c, say R, must respect the FD $X_i \rightarrow Y_i$. Observe that the FD is local in the sense that it holds inside the portion of relation that is derived by the rule and not in other possible portions of it that could be defined by other rules.

The formal semantics of choice is given in terms of stable models by replacing the above choice c rule with the following rules:

1. Replace c with the rule (*modified choice rule*):

   $$A \leftarrow B(Z), \ chosen_c(W).$$

   where $W \subseteq Z$ is the list of all variables appearing in the choice goals, i.e. $W = \bigcup_{1 \leq j \leq k} X_j \cup Y_j$.
2. Add the new rule (*chosen rule*):

   $$chosen_c(W) \leftarrow B(Z), \ \neg diffChoice_c(W).$$

3. For each $choice((X_i),(Y_i))$ ($1 \leq i \leq k$), add a new rule (*diffChoice rule*):

   $$diffChoice_c(W) \leftarrow chosen_c(W'), \ Y_i \neq Y_i'.$$

   where (i) the list of variables W' is derived from W by replacing each $V \notin X_i$ with a new variable V' (e.g. by priming those variables), and (ii) $Y_i \neq Y_i'$ is true if $V \neq V'$, for some variable $V \in Y_i$ and its primed counterpart $V' \in Y_i'$.

A $\texttt{DATALOG}^{\neg}$ program P with choice rules is called a *choice program*. The *standard version* sv_P of P is the program obtained from P by applying the above transformation to every choice rule. Given a database D, any stable model of $sv_P(D)$ is called a *choice model* of $P(D)$. Moreover, we say that P is *stratified modulo choice* if, by considering choice atoms as extensional atoms, the program results stratified. If P is stratified modulo choice, then the choice models of $P(D)$ are in general multiple but the existence of at least one as well as its computation in polynomial time is guaranteed [17,10,12].

Let $\texttt{DATALOG}^{\neg s,c}$ denote the set of all $\texttt{DATALOG}^{\neg}$ programs that are stratified modulo choice. Observe that stable model semantics continues to hold also for a $\texttt{DATALOG}^{\neg}$ program with choice that is not in $\texttt{DATALOG}^{\neg s,c}$; however, both the existence of a stable model and polynomial computability of one of them is not anymore guaranteed.

Given a program P in $\texttt{DATALOG}^{\neg s,c}$, a (bound) query goal over P is any ground literal. In order to answer to a query several semantics are defined in [12] as for example certain and possibility semantics. The answer of a query under certain semantics will be true if such query is true in each model of the program, while under possibility semantic, it will be true if the query is true in at least one model. The complexity of the query response under two semantics are respectively coNP-complete and NP-complete.

3 Data Exchange in Datalog with Choice

The classical data exchange setting $(S, T, \Sigma_{st}, \Sigma_t)$ can be formulated by means of a DATALOG program P with choice such that the source relational scheme S is the set of EDB predicates, T is a subset of IDB predicates and the rules of P implement both Σ_{st} and Σ_t. In addition we assume that a number of additional EDB predicates are available. They store domain values that will be used in the target database (*domain predicates*). We assume that each domain predicate has in general a finite number of elements but it can also be a countably infinite set. Next we show how to represent Σ_{st} and Σ_t in the case that the TGDs are weakly acyclic – informally, TGDs are *weakly acyclic* if there is no recursion among positions passing through existentially quantified variables, where a position is a pair of relation scheme symbol R and an attribute A of R (see [6,7] for a formal definition).

Take any TGD in $\Sigma_{st} \cup \Sigma_t$, say

$$d : \forall \mathbf{X} \; (\, \phi(\mathbf{X}) \rightarrow \exists \, \mathbf{Z} \; \psi(\mathbf{X}, \mathbf{Z}) \,)$$

where $\phi(\mathbf{X})$ is a conjunctions of either EDB or IDB predicates, $\psi(\mathbf{X}, \mathbf{Z})$ is a conjunction of IDB predicates, say $\psi_1(\mathbf{X}, \mathbf{Z}) \wedge \cdots \wedge \psi_n(\mathbf{X}, \mathbf{Z})$, $\mathbf{X}$ and $\mathbf{Z}$ are two lists of distinct variables, every variable in $\mathbf{X}$ occurs in ϕ and possibly in ψ and every variable in $\mathbf{Z}$ occurs in ψ. The following rules are associated to the dependency d:

$$\begin{aligned} exists\_choice_d(\mathbf{X}, \mathbf{Z}) &\leftarrow \phi(\mathbf{X}), \mathcal{D}_{\mathbf{Z}}(\mathbf{Z}), choice((\mathbf{X}), (\mathbf{Z})). \\ \psi_i(\mathbf{X}, \mathbf{Z}) &\leftarrow exists\_choice_d(\mathbf{X}, \mathbf{Z}) \qquad\qquad 1 \leq i \leq n. \end{aligned}$$

where $exist\_choice_d$ is a new IDB predicate symbol (obviously, not included among the IDB predicates symbols of the target schema) and $\mathcal{D}_{\mathbf{Z}}$ is a conjunction of the domain predicates for the variables $\mathbf{Z}$.

For each EGD in Σ_t, say

$$\forall \mathbf{X} \; (\, \phi(\mathbf{X}) \rightarrow \; x_1 = x_2 \,)$$

where x_1 and x_2 are variables in $\mathbf{X}$ and $\phi(\mathbf{X})$ is a conjunctions of IDB predicates, we introduce the constraint:

$$\leftarrow \phi(\mathbf{X}) \wedge \; x_1 \neq x_2.$$

Take the example presented in Section 1. The associated DATALOG program with choice is:

$$\begin{aligned} \mathtt{exists\_choice}(\mathtt{d}, \mathtt{n}, \mathtt{e}, \hat{\mathtt{m}}) &\leftarrow \mathtt{DeptEmp}(\mathtt{d}, \mathtt{n}, \mathtt{e}), \mathcal{D}_{\mathtt{M}}(\hat{\mathtt{m}}), \mathtt{choice}((\mathtt{d}, \mathtt{n}, \mathtt{e}), (\hat{\mathtt{m}})). && (4) \\ \mathtt{Dept}(\mathtt{d}, \mathtt{m}, \mathtt{n}) &\leftarrow \mathtt{exists\_choice}(\mathtt{d}, \mathtt{n}, \mathtt{e}, \mathtt{m}). && (5) \\ \mathtt{Emp}(\mathtt{e}, \mathtt{d}) &\leftarrow \mathtt{exists\_choice}(\mathtt{d}, \mathtt{n}, \mathtt{e}, \mathtt{m}). && (6) \\ \mathtt{Emp}(\mathtt{m}, \mathtt{d}) &\leftarrow \mathtt{Dept}(\mathtt{d}, \mathtt{m}, \mathtt{n}). && (7) \\ &\leftarrow \mathtt{Dept}(\mathtt{d}_1, \mathtt{m}, \mathtt{n}), \mathtt{Dept}(\mathtt{d}_2, \mathtt{m}, \mathtt{n}), \mathtt{d}_1 ! = \mathtt{d}_2. && (8) \\ &\leftarrow \mathtt{Dept}(\mathtt{d}, \mathtt{m}_1, \mathtt{n}_1), \mathtt{Dept}(\mathtt{d}, \mathtt{m}_2, \mathtt{n}_2), \mathtt{m}_1 ! = \mathtt{m}_2. && (9) \\ &\leftarrow \mathtt{Dept}(\mathtt{d}, \mathtt{m}_1, \mathtt{n}_1), \mathtt{Dept}(\mathtt{d}, \mathtt{m}_2, \mathtt{n}_2), \mathtt{n}_1 ! = \mathtt{n}_2. && (10) \end{aligned}$$

where $\mathcal{D}_M$ is the domain of manager ids – this domain can be countably infinite. As the head of the rule 6 above does not contain any existential variable, it can be simply rewritten as: $\mathtt{Emp(e,d)} \leftarrow \mathtt{DeptEmp(d,n,e)}$. Then, we can fold the first two rules, thus obtaining the rule:

$$\mathtt{Dept(d,M,n)} \leftarrow \mathtt{DeptEmp(d,n,e)}, \mathcal{D}_{\mathtt{M}}(\mathtt{m}), \mathtt{choice((d,n,e),(m))}.$$

By performing some straightforward rewriting to push down the two goal rule constraints into the choice rule, we obtain the following $\mathtt{DATALOG}^{\neg s,c}$ program (with stratified choice):

$$\begin{aligned}
\mathtt{Dept(d,m,n)} &\leftarrow \mathtt{DeptEmp(d,n,e)}, \mathcal{D}_{\mathtt{M}}(\mathtt{m}), \mathtt{choice((d),(m))}, \mathtt{choice((m),(d))}.\\
\mathtt{Emp(e,d)} &\leftarrow \mathtt{DeptEmp(d,n,e)}.\\
\mathtt{Emp(m,d)} &\leftarrow \mathtt{Dept(d,m,n)}.
\end{aligned}$$

Consider now the case of a data exchange setting with TGDs that are not weakly acyclic. For instance, referring to our current example, assume that a manager is not anymore obliged to be employed by its managed department and s/he can manage more than one department. The constraint that any employee is employed by only one department must be now explicitly declared. The source-to-target dependency set Σ_{st} is unchanged whereas the target-to-target dependencies become:

$$\begin{aligned}
\Sigma_t = \{&\mathtt{Dept(d,m,n)} \rightarrow \exists \mathtt{D}\, \mathtt{Emp(m,D)};\ \ \mathtt{Emp(e,d)} \rightarrow \exists\, \mathtt{M}\, \exists\, \mathtt{N}\, (\, \mathtt{Dept(d,M,N)}\,);\\
&\mathtt{Dept(d,m_1,n_1)} \wedge \mathtt{Dept(d,m_2,n_2)} \rightarrow \mathtt{m_1} = \mathtt{m_2} \wedge \mathtt{n_1} = \mathtt{n_2};\\
&\mathtt{Emp(e,d_1)} \wedge \mathtt{Emp(e,d_2)} \rightarrow \mathtt{d_1} = \mathtt{d_2}\}
\end{aligned}$$

The TGDs are not weakly acyclic. Let us show how to formulate this data exchange example in DATALOG with choice.

$$\mathtt{Dept(d,m,n)} \leftarrow \mathtt{DeptEmp(d,n,e)}, \mathcal{D}_{\mathtt{M}}(\mathtt{m}), \mathtt{choice((d),(m))}. \tag{11}$$
$$\mathtt{Emp(e,d)} \leftarrow \mathtt{DeptEmp(d,n,e)}. \tag{12}$$
$$\mathtt{Emp(m,D)} \leftarrow \mathtt{Dept(d,m,n)}, \mathcal{D}_{\mathtt{D}}(\mathtt{D}), \mathtt{choice((m),(D))}. \tag{13}$$
$$\mathtt{ExDept(d)} \leftarrow \mathtt{Dept(d,n,e)}. \tag{14}$$
$$\mathtt{Dept(d,m,n)} \leftarrow \mathtt{Emp(e,d)}, \neg\mathtt{ExDept(d)}, \mathcal{D}_{\mathtt{M}}(\mathtt{m}), \mathcal{D}_{\mathtt{N}}(\mathtt{n}), \mathtt{choice((d),(m,n))}. \tag{15}$$

where $\mathcal{D}_D$ and $\mathcal{D}_N$ are the domains of department ids and of manager names, respectively. We have introduced negation in the last rule in order not to generate a new pair (manager id, manager name) for an existing department. This program is not stratified modulo choice and, then, and the theory of choice models does not apply. In summary, known undecidability problems in handling TGDs that are not weakly acyclic have a clear counterpart in the representation by DATALOG with choice: the program is not stratified modulo choice.

Proposition 1. *Let P be a* $\mathtt{DATALOG}^{\neg s,c}$ *program associated to a data exchange setting $ET = (S, T, \Sigma_{st}, \Sigma_t)$ with weakly acyclic TGDs. Then M is a choice model of P if and only if there exists a solution R_T of ET such that $M(T) = R_T$.*

As proved in [12], given a bound query q and $\mathtt{DATALOG}^{\neg s,c}$ program with finite domains:

- deciding whether q is true under certain semantics (i.e., it is true in all choice models) is coNP-complete;
- deciding whether q is true under possible semantics (i.e., it is true in at least one choice model) is NP-complete.

Recall that we are considering data complexity. We point out the results with weakly acyclic TGDs also hold when the attribute domains are countably infinite: every choice model is finite even though the number of choice models could be infinite.

4 Uncertainty-Guaranteed Query Answering for Privacy-Preserving Data Exchange

A crucial issue in data exchange research is answering certain queries and, under this respect, the notion of universal solution represents an important theoretical achievement. This notion is (probably) lost when DATALOG with choice is used to define data exchange. Nevertheless, at the risk of being accused of heresy, we believe that answering certain queries is not a "must" for data exchange. In fact, attracted by the dark ("don't care") side of nondeterminism (typical of choice), once data have been migrated, we accept to get answers from the transferred data without caring about certainty. Indeed, during privacy-preserving data exchange setting, rather aiming for certainty, one could have an opposite goal: defining a target schema for which answering a number of given "sensible" queries is "uncertain"!

As an example, consider a source relation schema consisting of two relation schemes: $\mathtt{TI}(\mathtt{T},\mathtt{I})$ and $\mathtt{TSC}(\mathtt{T},\mathtt{S},\mathtt{C})$ with attributes $\mathtt{T}$ (Transaction), $\mathtt{I}$ (Item), $\mathtt{S}$ (Store) and $\mathtt{C}$ (Customer). The FD $\mathtt{T} \rightarrow \mathtt{S}\,\mathtt{C}$ holds, thus a transaction is executed by exactly one costumer in exactly one store. On the other hand, a transaction consists of a number of items. The target relation schema comprises three relation schemes: $\widehat{\mathtt{TI}}(\mathtt{T},\mathtt{I})$, $\widehat{\mathtt{TSC}}(\mathtt{T},\mathtt{S},\mathtt{C})$ and $\mathtt{CM}(\mathtt{C},\mathtt{M})$, where the attribute $\mathtt{M}$ stands for (Customer Care) Manager and the FD $\mathtt{C} \rightarrow \mathtt{M}$ holds. We are also given domain $\mathcal{D}_M$ and $\mathcal{D}_C$ for managers and customers.

We want to move data from the source to the target scheme but, for privacy reasons, the associations between transactions and items must be perturbed (i.e., the transactions IDs of the same store are permuted) and customers must be renamed. In addition we want to assign a manager to each customer.

$$\begin{aligned}
\mathtt{ex\_choice}(\mathtt{t},\hat{\mathtt{t}}) \leftarrow & \mathtt{TSC}(\mathtt{t},\mathtt{s},\mathtt{c}), \mathtt{TSC}(\hat{\mathtt{t}},\mathtt{s},\mathtt{c}), \mathtt{t} \neq \hat{\mathtt{t}}, \\
& \mathtt{choice}((\mathtt{t}),(\hat{\mathtt{t}})), \mathtt{choice}((\hat{\mathtt{t}}),(\mathtt{t})). \qquad (16)
\end{aligned}$$

$$\widehat{\mathtt{TI}}(\hat{\mathtt{t}},\mathtt{i}) \leftarrow \mathtt{TI}(\mathtt{t},\mathtt{i}), \mathtt{ex\_choice}(\mathtt{t},\hat{\mathtt{t}}). \qquad (17)$$

$$\begin{aligned}
\widehat{\mathtt{TSC}}(\hat{\mathtt{t}},\mathtt{s},\hat{\mathtt{c}}) \leftarrow & \mathtt{TSC}(\mathtt{t},\mathtt{s},\mathtt{c}), \mathtt{ex\_choice}(\mathtt{t},\hat{\mathtt{t}}), \mathcal{D}_{\mathtt{C}}(\hat{\mathtt{c}}), \\
& \mathtt{choice}((\mathtt{c}),(\hat{\mathtt{c}})), \mathtt{choice}((\hat{\mathtt{c}}),(\mathtt{c})). \qquad (18)
\end{aligned}$$

$$\mathtt{CM}(\mathtt{c},\hat{\mathtt{m}}) \leftarrow \widehat{\mathtt{TSC}}(\mathtt{t},\mathtt{s},\mathtt{c}), \mathcal{D}_{\mathtt{M}}(\hat{\mathtt{m}}), \mathtt{choice}((\mathtt{c}),(\hat{\mathtt{m}})). \qquad (19)$$

In the example we are not interested in finding certain answers. We rather want to be guaranteed that "sensitive" queries do no give certain answers – for instance we have perturbed data so that customers are renamed and they cannot be eventually recognized from transaction ids.

Let us assume that each domain predicate has a finite number of elements. We define that a query q is *α-uncertain*, where α is a given non-negative value less than or equal to 0.5, if q is true in at least $(0.5 - \alpha) \times n$ and false in at least $(0.5 - \alpha) \times n$ of the n choice models - the smaller is α, the higher is uncertainty. It is not surprising that complexity of uncertainty is higher than the one of certainty – recall that PP is is the class of decision problems that can be solved by a nondeterministic Turing machine in polynomial time, where the acceptance condition is that a majority (more than half) of computation paths accept.

Proposition 2. *Deciding whether q is α-uncertain is in* PSPACE *and* PP*-hard.*

Proof. (Sketch) To see that the decision problem is in PSPACE, observe that we can compute in deterministic polynomial space and non-deterministic polynomial time a choice model. Then we can easily implement a mechanism to compute every choice model and to count the number of them that make true the query. The PP-hardness proof immediately derives from the capability of DATALOG with choice to express SAT under possibility semantics (see [12]): we only need to check that the majority of choice models give a positive answer. □

Again we consider data complexity. We also stress that with infinite domains PSPACE membership continues to hold whereas obviously PP hardness does not.

We are aware that data perturbation is not in general sufficient to guarantee that private information is not disclosed and there an increasing amount of research investigating this issue (see for instance [8,4]). Advanced privacy-preserving techniques can be suitably applied while defining the data exchange setting using DATALOG with choice.

Observe that the high cost of deciding uncertainty is paid only for designing the target relational scheme. Once data have been migrated, query answering becomes a typical "deterministic" operation on a database, mostly implementable in polynomial time.

5 Data Exchange with Count Aggregates

We now assume that, in addition to domain constants, the Herbrand universe includes constant set terms defined as follows: given any list S of attributes, a constant set term is a set of tuples (i.e., a table) over S. Given the attributes A and B with domains $\{a_1, a_2, a_3\}$ and $\{b_1, b_2\}$ respectively, examples of constant set terms on (A, B) are $\{[a_1, b_1], [a_2, b_1], [a_3, b_2]\}$ and $\{[a_2, b_1]\}$, while $\{[a_1], [a_3]\}$ and $\{[a_2]\}$ are constant set terms on (A).

A *set term* is either a constant set term or a *formula term*, defined as $\{\mathbf{X} : \exists \mathbf{Y} \psi\}$, where $\mathbf{X}$ and $\mathbf{Y}$ are disjoint list of variables and ψ is conjunction of literals in which variables in $\mathbf{X}$ occur free (similar notation for set terms and aggregate predicates has been used in the dlv system [5]).

There is an interpreted function symbol *count* (denoted by $\#$) that can be applied to a set term T to return the number of tuples in T (i.e., the cardinality of the table represented by T).

A *count constraint* C, as introduced in [16], is a formula of type:

$$\forall \mathbf{X} \ (\ \phi(\mathbf{X}) \ \rightarrow \ \#(\{ \ \mathbf{Y} : \exists \mathbf{Z} \psi(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) \ \}) \leq \beta \ )$$

where $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ are disjoint lists of variables, $\mathbf{X}$ and $\mathbf{Z}$ can be empty, and β is an integer term.

The above constraint can be written in $\mathtt{DATALOG}^{\neg s,c}$ as:

$$\leftarrow \ \phi(\mathbf{X}) \wedge \#(\{ \ \mathbf{Y} : \exists \mathbf{Z} \psi(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) \ \}) > \beta$$

Count constraints can be considered as extensions of EGDs. In fact, a generic EGD $\forall \mathbf{X} \ (\ \phi(\mathbf{X}) \rightarrow \ x_1 = x_2 \ )$ can be formulated by the following count constraint:

$$\forall \mathbf{X}' \ (\mathit{true} \rightarrow \#(\{ y : \phi(y, \mathbf{X}')\} \ ) \leq 1)$$

where $\mathbf{X}'$ contains all variables in $\mathbf{X}$ except x_1 and x_2; moreover, y replaces both x_1 and x_2 in ϕ.

We now extend the choice construct to enforce a count constraint while making a choice. To give a first intuition of our approach, we go back to the privacy-preserving data exchange example of the previous section. Suppose now that we want to enforce the constraint that each manager can take care of at most 10 customers:

$$\leftarrow \#(\{c : \mathtt{CM}(\mathtt{c}, \mathtt{m})\}) > \mathtt{10}.$$

The above goal rule can be removed and the count constraint pushed down into the body of rule 19 by adding a new choice construct:

$$\mathtt{CM}(\mathtt{c}, \mathtt{m}) \leftarrow \widehat{\mathtt{TSC}}(\mathtt{t}, \mathtt{s}, \mathtt{c}), \mathcal{D}_{\mathtt{M}}(\mathtt{m}), \mathtt{choice}((\mathtt{c}), (\mathtt{m})), \mathtt{choice}((\mathtt{m}), (\mathtt{c}))[\mathtt{10}]. \qquad (20)$$

The construct $\mathtt{choice}((\mathtt{M}), (\mathtt{c}))[\mathtt{10}]$ is implemented by the following different version of the *diffChoice rule*:

$$\mathit{diffChoice}(\mathtt{c}, \mathtt{m}) \leftarrow \mathit{chosen}(\mathtt{c}_1, \mathtt{m}), \ldots, \mathit{chosen}(\mathtt{c}_{10}, \mathtt{m}), \ \mathtt{c} \neq \mathtt{c}_1 \neq \cdots \neq \mathtt{c}_{10}.$$

The availability of count aggregate operator can be exploited to perform aggregate data exchange. For instance, in our running privacy-preserving data exchange example, we may realize that disclosing the associations between customers and transactions can cause a privacy breach despite data perturbation. So we may decide to exchange just the quantity (number) of transactions made by a customer in every store, rather than to give the whole list of renamed transactions. Then the relation scheme $\widehat{\mathtt{TSC}}(\mathtt{T}, \mathtt{S}, \mathtt{C})$ is replaced by $\mathtt{NSC}(\mathtt{N}, \mathtt{S}, \mathtt{C})$, where the attribute $\mathtt{N}$ stores the number of transactions executed by a customer in a store, and a new relation scheme $\mathtt{TS}(\mathtt{T}, \mathtt{S})$ is added to preserve the association between transactions and stores. Then rule 18 is replaced by the following two rules:

$$\begin{aligned}\mathtt{NSC}(\mathtt{n}, \mathtt{s}, \hat{\mathtt{c}}) \leftarrow & \mathtt{n} = \#(\{\mathtt{t} : \mathtt{TSC}(\mathtt{t}, \mathtt{s}, \mathtt{c})\}), \mathcal{D}_{\mathtt{C}}(\hat{\mathtt{c}}), \mathtt{choice}((\mathtt{c}), (\hat{\mathtt{c}})), \mathtt{choice}((\hat{\mathtt{c}}), (\mathtt{c})).\\ \mathtt{TS}(\hat{\mathtt{t}}, \mathtt{s}) \leftarrow & \mathtt{TSC}(\mathtt{t}, \mathtt{s}, \mathtt{c}), \mathtt{ex\_choice}(\mathtt{t}, \hat{\mathtt{t}}).\end{aligned}$$

The example confirms that the combination of choice and count constraints represents a powerful framework for defining aggregate privacy-preserving data exchange using DATALOG. The choice by itself searches for “any” solution rather than for a general solution. The introduction of count constraints worsens the problem – extending the notion of universal solution to data exchange with count constraints is a complicated problem even for the classical setting and, probably, it is not feasible at all. But, as we have argued throughout the paper, a universal solution is not the panacea for performing data exchange and the more prosaic "don't care" non determinism can be very effective for solving the problem in some specific contexts such privacy-preserving data exchange.

6 Conclusion

In this paper we have proposed to use DATALOG with choice to formulate a data exchange problem from a source to a target database schema. Classical data exchange setting includes TGDs (Tuple Generating Dependencies), which are universal quantified formulas with additional existential quantifiers, and EGDs (Equality Generating Dependencies), which are universal quantified formulas enforcing the equality of two of the quantified variables. Existential quantifiers in TGDs are implemented by the choice construct, which non-deterministically select values for the existential variables. To express general EGDs we have used goal rules. However, as an EGD often expresses functional dependencies, in this case the goal rule can be removed and the constraint is alternatively expressed as an additional choice predicate into the body of some rule implementing a TGD.

Concerning the semantics of query answering in the context of data exchange, we have explored a direction opposite to the classical one, which is based on certainty semantics: our goal is instead to find a mapping from the source to the target schema such that a number of given "sensible" queries have uncertainty-guaranteed answers. We have discussed properties of privacy-preserving data exchange and illustrated its complexity. Finally, we have shown that EGDs are special cases of count constraints (e.g, an employee can manage at most three departments instead of one) and we have extended the choice construct to implement such constraints. Finally, as count constraints are based on set terms and the aggregate count operator on them, we have illustrated how the overall setting can be used also to define aggregate data exchange.

Acknowledgments. The authors thank an anonymous reviewer for many useful and constructive comments.

References

1. Arenas, M., Barceló, P., Fagin, R., Libkin, L.: Locally consistent transformations and query answering in data exchange. In: Beeri, C., Deutsch, A. (eds.) PODS, pp. 229–240. ACM (2004)
2. Ceri, S., Gottlob, G., Tanca, L.: Logic programming and databases. Springer-Verlag New York, Inc., New York (1990)
3. Chandra, A., Harel, D.: Structure and complexity of relational queries. Journal of Computer and System Sciences 25, 99–128 (1982)

4. Clifton, C., Kantarcioğlu, M., Doan, A., Schadow, G., Vaidya, J., Elmagarmid, A., Suciu, D.: Privacy-preserving data integration and sharing. In: DMKD 2004: Proceedings of the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, pp. 19–26. ACM, New York (2004)
5. Faber, W., Pfeifer, G., Leone, N., Dell'Armi, T., Ielpa, G.: Design and implementation of aggregate functions in the dlv system. TPLP 8(5-6), 545–580 (2008)
6. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. In: Neven, F., Beeri, C., Milo, T. (eds.) PODS, pp. 90–101. ACM (2003)
7. Fagin, R., Kolaitis, P.G., Popa, L.: Data exchange: getting to the core. ACM Trans. Database Syst. 30(1), 174–210 (2005)
8. Fung, B.C.M., Wang, K., Chen, R., Yu, P.S.: Privacy-preserving data publishing: A survey of recent developments. ACM Comput. Surv. 42(4), 14:1–14:53 (2010)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, pp. 1070–1080 (1988)
10. Giannotti, F., Pedreschi, D., Saccà, D., Zaniolo, C.: Non-Determinism in Deductive Databases. In: Delobel, C., Masunaga, Y., Kifer, M. (eds.) DOOD 1991. LNCS, vol. 566, pp. 129–146. Springer, Heidelberg (1991)
11. Gottlob, G., Nash, A.: Data exchange: computing cores in polynomial time. In: Vansummeren, S. (ed.) PODS, pp. 40–49. ACM (2006)
12. Greco, S., Saccà, D., Zaniolo, C.: Extending stratified datalog to capture complexity classes ranging from P to QH. Acta Inf. 37(10), 699–725 (2001)
13. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer-Verlag New York, Inc., Secaucus (1993)
14. Marek, W., Truszczyński, M.: Autoepistemic logic. J. ACM 38(3), 587–618 (1991)
15. Marnette, B.: Generalized schema-mappings: from termination to tractability. In: Paredaens, J., Su, J. (eds.) PODS, pp. 13–22. ACM (2009)
16. Saccà, D., Serra, E., Guzzo, A.: Count Constraints and the Inverse OLAP Problem: Definition, Complexity and a Step toward Aggregate Data Exchange. In: Lukasiewicz, T., Sali, A. (eds.) FoIKS 2012. LNCS, vol. 7153, pp. 352–369. Springer, Heidelberg (2012)
17. Saccà, D., Zaniolo, C.: Stable models and non-determinism in logic programs with negation. In: Rosenkrantz, D.J., Sagiv, Y. (eds.) PODS, pp. 205–217. ACM Press (1990)
18. ten Cate, B., Chiticariu, L., Kolaitis, P.G., Tan, W.C.: Laconic schema mappings: Computing the core with sql queries. PVLDB 2(1), 1006–1017 (2009)
19. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. The New Technologies, vol. II. W. H. Freeman & Co., New York (1990)
20. Vardi, M.Y.: The complexity of relational query languages. In: Lewis, H.R., Simons, B.B., Burkhard, W.A., Landweber, L.H. (eds.) STOC, pp. 137–146. ACM (1982)

Optimizing Large-Scale Semi-Naïve Datalog Evaluation in Hadoop

Marianne Shaw, Paraschos Koutris, Bill Howe, and Dan Suciu

University of Washington
{mar,pkoutris,billhowe,suciu}@cs.washington.edu

Abstract. We explore the design and implementation of a scalable Datalog system using Hadoop as the underlying runtime system. Observing that several successful projects provide a relational algebra-based programming interface to Hadoop, we argue that a natural extension is to add recursion to support scalable social network analysis, internet traffic analysis, and general graph query. We implement semi-naive evaluation in Hadoop, then apply a series of optimizations spanning fundamental changes to the Hadoop infrastructure to basic configuration guidelines that collectively offer a 10x improvement in our experiments. This work lays the foundation for a more comprehensive cost-based algebraic optimization framework for parallel recursive Datalog queries.

1 Introduction

The MapReduce programming model has had a transformative impact on data-intensive computing, enabling a single programmer to harness hundreds or thousands of computers for a single task, often after only a few hours of development. When processing with thousands of computers, a different set of design considerations can dominate: I/O scalability, fault tolerance, and programming flexibility. The MapReduce model itself, and especially the open source implementation Hadoop [11], have become very successful by optimizing for these considerations.

A critical success factor for MapReduce has been its ability to turn a "mere mortal" java programmer into a distributed systems programmer. It raised the level of abstraction for parallel, highly scalable data-oriented programming. But it did not raise the level of abstraction high enough, evidently, because some of the earliest and most successful projects in the Hadoop ecosystem provided declarative languages on top of Hadoop. For example, HIVE provided an SQL interface, and Yahoo's Pig provided a language that closely resembles the relational algebra[1].

MapReduce (as implemented in Hadoop) has proven successful as a common runtime for non-recursive relational algebra-based languages. Our thesis is

[1] Relational algebra is not traditionally considered declarative, but Pig programs, while syntactically imperative, can be optimized by the system prior to execution and generally provide a significantly higher level of abstraction than MapReduce, so we consider the term applicable.

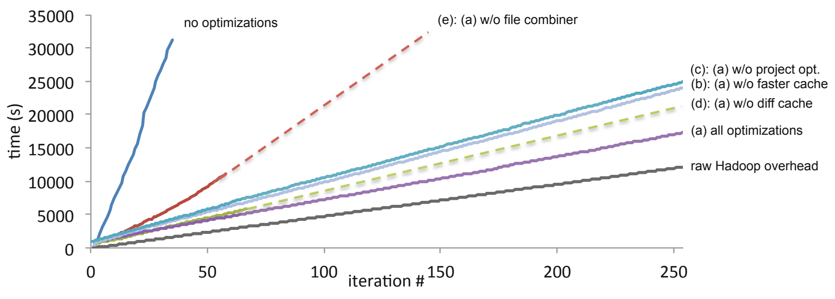


Fig. 1. The cumulative effect of our optimizations on overall runtime. (a) All optimizations applied. (b) Relaxing the specialization of the cache for inner joins. (c) Relaxing the optimization to eliminate the project operator. (d) Relaxing the diff cache optimization (extrapolated).

that Hadoop, suitably extended, can also be successful as a common runtime for recursive languages as required for graph analytics [18,7], AI and planning applications [9], networking [14]. In previous work, our group extended Hadoop with caching and scheduling features to avoid reprocessing loop-invariant data on every iteration and to afford expression of multi-step loop bodies and various termination conditions. In this paper, we describe how this underlying framework, appropriately extended and configured, can be used to implement a scalable Datalog engine.

In this paper we present some engineering solutions for the semi-naive evaluation of a linear Datalog on Hadoop-based systems. The input to our system is a Datalog query. The query is parsed, optimized, and compiled into a series of MapReduce jobs, using an extended implemenation that directly supports iteration [7]. Our semi-naive algorithm requires three relational operators: join, duplicate elimination, and set difference, and each of them requires a separate MR job. We consider a series of optimizations to improve on this basic strategy.

Figure 1 summarizes the cumulative effect of a series of optimizations designed to improve on this basic strategy using a simple reachability query as a test case. The dataset is a social network graph dataset with 1.4 billion unique edges. The x-axis is the iteration number and the y-axis is the cumulative runtime: the highest point reached by a line indicates the total runtime of the job. Dashed lines indicate extrapolation from incomplete experiments. Figure 1(a) gives the runtime when all optimizations are applied, which is 10X-13X faster than Hadoop itself (labeled no optimizations in the figure), and only about 40% higher than the raw Hadoop overhead required to run two no-op jobs with degenerate Map and Reduce functions (labeled raw Hadoop overhead in the figure.)

The optimizations are as follows. First, for join, we notice that one of the relations in the join is loop invariant: using a previously developed cache for Hadoop [7], we store this invariant relation at the reducers, thus avoiding the expensive step of scanning and shuffling this relation at every datalog iteration. The join cache is the most significant optimization; all experiments (a)-(e) in Figure 1 use some form of join cache. Second, by specializing and indexing the cache to implement inner joins, we can avoid many unnecessary reduce calls required by the original MapReduce semantics (Figure 1(b) show the effect of relaxing this optimization). Third, we notice that the duplicate elimination and difference can be folded into a single MR job: the newly generated tuples are shuffled to the reducers, and the same reduce job both eliminates duplicates, and checks if these tuples have already been discovered at previous iterations (Figure 1(c) shows the effect of relaxing this optimization). Fourth, with appropriate extensions to support cache insertions during execution, we observe that the cache framework can also be used to improve performance of the set difference operator (Figure 1(d) shows the effect of relaxing this optimization). Finally, we found it necessary to combine files after every job to minimize the number of map tasks in the subsequent job (Figure 1(e) shows the effect of relaxing this optimization).

In Section 3, we describe the implementation of semi-naive evaluation and the optimizations we have applied. In Section 5, we analyze these optimizations to understand their relative importance.

2 Related Work

This work is based on the MapReduce framework, which was first introduced in [8]. In this project we build our system using the popular open source implementation of MapReduce, Hadoop.

There has been an extensive line of research on providing a higher level interface to the MapReduce programming model and its implementation in Hadoop, including Dryad [12], DryadLINQ, Hyracks [6], Boom [3] and PigLatin [16]. Of these, only Hyracks provides some support for iterative queries, and they do not expose a Datalog programming interface and do not explore logical optimizations for iterative programs.

Parallel evaluation of logic systems including Datalog has been studied extensively. Balduccini et al. explore vertical (data) and horizontal (rules) parallelism, but evaluate their techniques only on small, artificial datasets [4]. Perri et al. consider parallelism at three levels: components (strata), rules, and within a single rule and show how to exploit these opportunities using modern SMP and multicore machines [17]. We target several orders of magnitude larger datasets (billions of nodes rather than tens of thousands) and have a very general model of parallelism that subsumes all three levels explore by Perri.

Damásio and Ferreira consider algorithms for improving transitive closure on large data, seeking to reduce the number of operations and iterations required. They apply these techniques to semantic web datasets and evaluate them on a single-site PostgreSQL system. The datasets contain up to millions of nodes, still three orders of magnitude smaller than our target applications.

In addition to our work on HaLoop [7], there have been several other systems providing iterative capabilities over a parallel data flow system. Pregel [15], a system developed for graph processing using the BSP model, also supports recursive operations. The Twister system [10] retains a MapReduce programming model, but uses a pub-sub system as the runtime to make better use of main memory and improve performance. The Spark system [19] also makes better use of main memory, but introduces a relational algebra-like programming model along with basic loop constructs to control iteration. The Piccolo project supports a message-passing programming model with global synchronization barriers. The programmer can provide locality hints to ensure multiple tables are co-partitioned. The Daytona project [5] at Microsoft Research provides iterative capabilities over a MapReduce system implemented on the Azure cloud computing platform. These projects all assume an imperative programming model. We explore a declarative programming model to reduce effort and expose automatic optimization opportunities. The BOOM project [3] is a distributed datalog system that provides extensions for temporal logic and distributed protocols. All of these systems are potential runtimes for Datalog. Some, but not all, of our optimizations will be applicable in these contexts as well.

A recent line of research examines recursive computations on MapReduce theoretically. In this work [2], the authors discuss issues, problems and solutions about an implementation of Datalog on the MapReduce framework. On a continuation of this work, the authors find a class of Datalog queries where it is possible to drastically reduce (to a logarithmic number) the number of recursion steps without significantly increasing the communication/replication cost.

3 Optimizing Semi-Naive Evaluation in Hadoop

Our basic execution model for Datalog in Hadoop is semi-naive evaluation. For illustration, consider a simple reachability query:

```
A(x,y) :- R(x,y), x=1234
A(x,y) :- A(x,z), R(z,y)
```

A (fully) naïve execution plan MapReduce is intuitively very expensive. On each iteration, the naïve plan requires one MR job for the join to find the next generation of results, a second job to project the result of the join and remove duplicate answers, a third MR job to compute the union (with duplicate elimination) with results discovered in previous iterations, and a fourth MR job to test for fixpoint. The inputs to each of these jobs are potentially large, distributed datasets.

An improvement is semi-naive evaluation, captured as follows.

```
$\Delta A^0 = \sigma_{x=1234}(R), i = 1$
while $\Delta A^{i-1}$ is not empty:
  $A^i = (\Delta A^0 \cup \cdots \cup \Delta A^{i-1})$
  $\Delta A^i = \pi_{xy}(\Delta A^{i-1} \bowtie_{z=z} R) - A^i, \quad i \leftarrow i+1$
```

The final result is the concatenation of the results of all previous iterations ΔA^i. There is no need to remove duplicates.

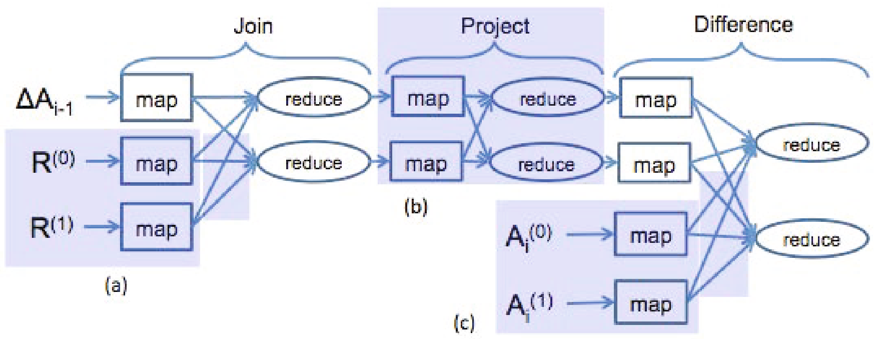


Fig. 2. Semi-naive evaluation implemented in Hadoop. (a) R is loop invariant, but gets loaded and shuffled on each iteration. (b) The extra mapreduce step to implement the project operator can be avoided by extending the join and difference operators appropriately. (c) A_i grows slowly and monotonically, but is loaded and shuffled on each iteration.

In Hadoop, this execution plan involves only three MR jobs: one for the join $\bowtie_{z=z}$, one to compute the projection π_{xy} and remove duplicates, and one to compute the difference of the new results and all previous results. Computing the union of all previous results does *not* require an independent MR job, and in fact requires no work at all. The reason is that the input to a MR job is a set of files that are logically concatenated, which is just what we need.

Figure 2 illustrates the optimization opportunities for each operator. At (a), the EDB R is scanned and shuffled on every iteration even though it never changes. At (b), the overhead of an extra MapReduce step to implement the project operator can be folding duplicate elimination into the difference operator. At (c), the result relation A grows slowly and monotonically, but is scanned and shuffled on every iteration. In the remainder of this section, we describe how to optimize these operations.

3.1 Join

The join $\Delta A^{i-1} \bowtie_{z=z} R$ is implemented as a *reduce-side join* as is typical in Hadoop. The map phase hashes the tuples of both relations by the join key (and optionally applies a selection condition if the query calls for it). The reduce phase then computes the join for each unique key (the cross product of $\sigma_{z=k}\Delta A^{i-1}$ and $\sigma_{z=k}R$ for each key k). The join uses Hadoop's secondary sort capabilities to ensure that only $\sigma_{z=k}\Delta A^{i-1}$ incurs memory overhead; $\sigma_{z=k}R$ can be pipelined. Skew issues can be addressed by using an out-of-core algorithm such as hybrid hash join or by other common techniques.

The critical bottleneck with the join operation is that the entire relation R must be scanned and shuffled on each and every iteration. In previous work on HaLoop [7], we added a *Reducer Input Cache* (RIC) to avoid scanning and shuffling loop-invariant relations (problem (a)). Specifically, HaLoop will cache the reducer inputs across all reduce nodes and create an index for the cached data and stores it on local disk. Reducer inputs are cached during reduce function invocation, so the tuples in the reducer input cache are sorted and grouped by reducer input key. When a reducer processes a shuffled key and its values, it searches the appropriate local reducer input cache to find corresponding keys and values. An iterator that combines the shuffled data and the cached data is then passed to the user-defined reduce function for processing. In the physical layout of this cache, keys and values are separated into two files, and each key has an associated pointer to its corresponding values. Since the cache is sorted and accessed in sorted order, only one sequential scan must be made in the worst case. Fault-tolerance was preserved by arranging for caches to be rebuilt when failures occurred without having to rerun all previous iterations.

To be fully transparent with the original MapReduce semantics, all cached keys, regardless of whether they appear in the mapper or not, should be passed to the reducer for processing. However, the equijoin semantics used in Datalog expose an optimization opportunity: Only those cached values that match a value in the incoming mapper output need be extracted and passed to the reducer, for significant savings (Figure 1(b)).

3.2 Difference

In Figure 2(b), the set difference operator compares the generated join output to the loop's accumulated result to ensure that only the newly discovered tuples are output in the operation's result. By default, the difference operation requires the scanning and shuffling of all previous iterations' results on every iteration. As the number of iterations increases, both the number of (possibly small) files accessed and the amount of data being reshuffled increases. Each file requires a separate map task, so it is important to group the many small files before computing the difference. The performance improvement of this configuration detail is significant (Figure 2(d)).

Like the join operation, the difference operation in Figure 2(b) can benefit from a cache. The previously discovered results need not be scanned and shuffled on each iteration; instead, these values can be maintained in a cache and updated on each iteration. We extended the original HaLoop caching framework to support insertions during iterative processing, generalizing it for use with the difference operator.

The difference operator uses the cache as follows. Each tuple is stored in the cache as a key-value pair (t, i), where the key is the tuple t discovered by the previous join operator and the value is the iteration number i for which that tuple was discovered. On each iteration, the map phase of the difference operator hashes the incoming tuples as keys with values indicating the current iteration number. During the reduce phase, for each incoming tuple (from the map phase), the cache is probed to find all instances of the tuples previously discovered across all iterations. Both the incoming and cached data are passed to the user-defined reduce function. Any tuples that were previously discovered are suppressed in the output.

For example, consider a reachability query. If a node t was discovered on iteration 1, 4, 5, and 8, there would be four instances of t in the cache when the reduce phase of iteration 8 executes. The reducer then would receive a list of five key-value pairs: the pair $(t, 8)$ from the map phase, and the pairs $(t, 1)$, $(t, 4)$, $(t, 5)$, $(t, 8)$ from the cache. The reducer can then determine that the tuple had been seen before, and therefore emit nothing. If the tuple had never before been seen, then it would receive a singleton list $(t,8)$ and would recognize that this tuple should be included in ΔA^9 and emit the tuple.

To avoid storing duplicate values in the cache, we leverage Haloop's cache-filtering functionality. When a reduce operation processes an individual value associated with a key, Haloop invokes the user-defined $isCache()$ routine to first consult a hashtable of all tuples previously written to the cache during this iteration. We reduce the number of duplicates that must be considered by this mechanism by using a combiner function on the map side. Combiners are a common Hadoop idiom used to reduce the amount of communication between mappers and reducers by pre-aggregating values. In this case, the combiner ensures that each mapper only produces each value once.

A limitation in the current Haloop cache implementation is that the cache is rewritten completely on every iteration during which new tuples are discovered, incurring significant IO overhead in some cases. A redesigned cache that avoids this step is straightforward, but remains future work.

3.3 Project

The project operator at Figure 2(b) is implemented as a separate MapReduce job. The two tasks accomplished in this job — column elimination and duplicate row elimination — can be delegated to the join operator and the difference operator, respectively.

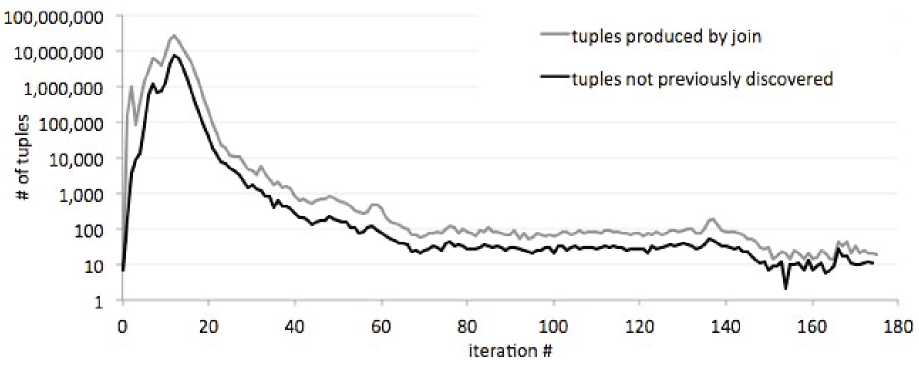


Fig. 3. The "endgame" of a recursive computation. The number of new tuples discovered peaks early (y-axis, log scale). Beyond iteration 25, less than 1000 new nodes are discovered on each iteration though execution continues for hundreds of iterations.

The difference operator naturally removes duplicates as a side effect of processing. To remove columns, we have made a straightforward extension to the join operator to provide a *column-selecting join* that is capable of removing columns. Since it does not need to remove duplicates, this step can be performed entirely in parallel and therefore incurs very little overhead.

By replacing sequences of Join→Project→Difference operations with sequences of ColumnSelectingJoin→Difference, the system is able to eliminate a map-reduce job *per iteration* for significant savings (Figure 1(c)).

4 Implementation

We have implemented a Datalog interpreter that converts schema and rules to a sequence of MapReduce jobs. The Datalog query is converted into the relational algebra, optimized (if desired), and then translated into a set of MapReduce jobs that are configured through JobConf configuration parameters. A sample input to the interpreter is seen below:

```
1) backend R[long, long, long] "btc".
2) res(x) :- R(1570650593L, b, x) .
3) res(y) :- res(z), R(z,b,y) .
4) ans res(y)
```

Line (1) specifies the schema for the R dataset, which consists of three columns of type long, and backed by the Hadoop directory "btc." Input directories can contain either delimited text files or SequenceFiles. Lines (2–3) define rules available for evaluating the query answer, specified on line (4).

5 Analysis and Evaluation

We evaluate our datalog system on the 2010 Billion Triple Challenge (BTC) dataset, a large graph dataset (625GB uncompressed) with 2B nodes and 3.2B quads. Queries are executed on a local 21-node cluster, consisting of 1 master node and 20 slave nodes. The cluster consists of Dual Quad Core 2.66GHz and 2.00GHz machines with 16GB RAM, all running 64bit RedHat Enterprise Linux. Individual MapReduce jobs are given a maximum Java heap space of 1GB.

The source of the BTC data is a web crawl of semantic data, and the majority of the nodes are associated with social network information. The graph is disconnected, but Joslyn et al [13] found that the largest component accounts for 99.8% of the distinct vertices in the graph.

As a result of these properties, recursive queries over this graph are challenging to optimize. In particular, they exhibit the endgame problem articulated by Ullman et al [2]. To illustrate the problem, consider Figure 3 which shows the number of nodes encountered by iteration number for a simple reachability query (i.e., find all nodes connected to a particular node.) The x-axis is the iteration number and the y-axis is the number of new nodes discovered in log-scale. The two datasets represent the size of the results of the join and difference operators. The nodes encountered by the join operation are in red and the nodes that are determined to not have not been previously discovered are in blue. The overall shape of the curve is striking. In iteration 17, over 10 million new nodes are discovered as the frontier passes through a well-connected region of the graph. By iteration 21, however, the number of new nodes produced has dropped precipitously. In ongoing work, we are exploring dynamic reoptimization techniques to make both regimes efficient. In this paper, we focus on the core hadoop-related system optimizations required to lay a foundation for that work.

In this evaluation, we consider the following questions:

– Join Cache: How much improvement can we expect from the reducer input cache in the context of long-running recursive queries and semi-naive evaluation?
– Diff Cache: How much improvement can we expect from an extended cache subsystem suitable for use with the difference step of semi-naive evaluation?
– Equijoin semantics: If we optimize for equi-joins in the underlying subsystem, how much improvement can we expect over the original MapReduce semantics where every key from both relations must be processed?

To answer these questions, we consider a simple reachability query

```
A(y) :- R(startnode,y)
A(y) :- A(x),R(x,y)
```

where *startnode* is a literal value that refers to a single node id. This query returns 29 million nodes in the result and runs for 254 iterations for the *startnode* we have selected. The query is simple, but clearly demonstrates the challenges of optimizing recursive queries in this context.

Unless otherwise noted in the text, the individual identifiers in the BTC2010 data set are hashed to long integer values and stored as SequenceFiles.

After 33 iterations, our cumulative running time for (2) Join $\rightarrow$ Proj $\rightarrow$ Diff is less than 15% of the query time when no caches are used. Additionally, modifying our operators to eliminate an unnecessary Hadoop job to implement the Project operator reduces iteration time by more than 26 seconds per iteration. Over the course of 254 iterations, this adds up to over 100 minutes of query time. After 33 iterations, the cumulative query time for (3) ColumnSelectingJoin $\rightarrow$ Diff is less than 10% of the cacheless query's execution time.

Figure 4 presents the per-operation contribution of the cumulative iteration time for iterations 1-33. The three scenarios are (1) ColumnSelectingJoin $\rightarrow$ Diff with no caches, (2) Join $\rightarrow$ Proj $\rightarrow$ Diff with both a join cache and a diff cache, (3) ColumnSelectingJoin $\rightarrow$ Diff with both a join cache and a diff cache, and (4) and estimate of the minimum job overhead for running two no-op Hadoop jobs on every iteration. With no caches used, the time is dominated by the join operation. With optimizations applied, the time approaches the minimum overhead of Hadoop (about 18 seconds per job for our cluster and current configuration).

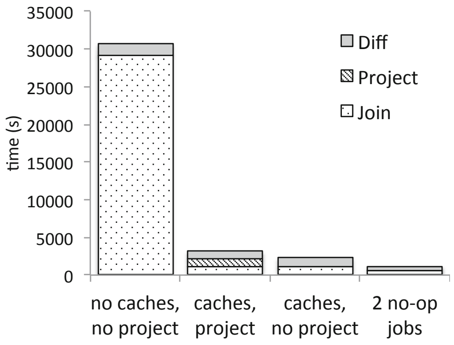


Fig. 4. The total time spent in each operation across iterations 1-33 (excluding 0) for three evaluation strategies. Without caching, the join operation dominates execution time.

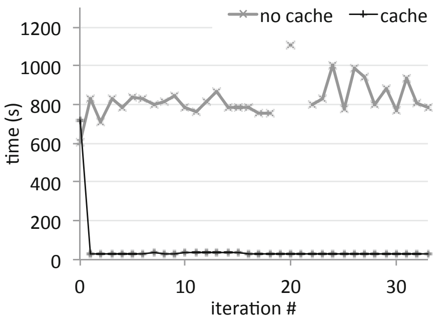


Fig. 5. Time to execute the join only. The use of a cache to avoid re-scanning and re-shuffling the loop-invariant data on each iteration results in significant improvement (and also appears to decrease variability). Gaps represent failed jobs.

Figure 5 shows the runtime of the join step only by iteration, both with and without the cache enabled. On the first iteration, populating the cache incurs some overhead. (We plot successful jobs only for clarity; gaps in the data represent inaccurate runtime measurements due to job failures.) On subsequent iterations, the invariant graph relation need not be scanned and shuffled, resulting in considerable savings. For our test query, the time for the join per iteration went from approximately 700 seconds to approximately 30 seconds, and the overall runtime of the query went from 50 hours to 5 hours (making these kind of jobs feasible!)

In Figure 6, the cache subsystem has been extended to allow new nodes to be added to it on each iteration. The y-axis shows the runtime of the difference operator, and the x-axis is the iteration number. This capability was not available

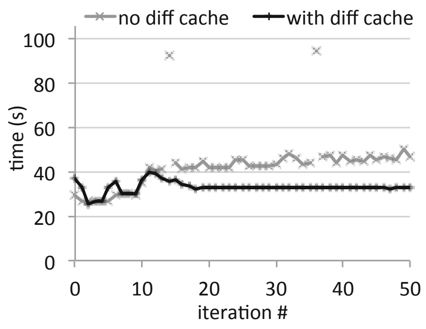


Fig. 6. Time per iteration for the difference operator only. The cache improves performance by about 20%.

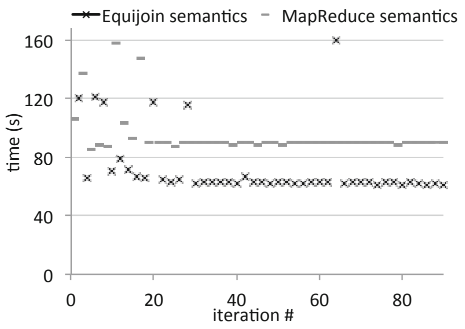


Fig. 7. The per-iteration impact of specializing the cache for equijoin, which is safe given our target lanaguage of datalog.

in HaLoop and is used to improve semi-naive evaluation. Without the cache, each iteration must scan more and more data, resulting in a (slow) increase in the per-iteration runtime. With the cache, the increase in size has no discernible effect on performance, and the performance of the operator improves by about 20%. The outlier values were the result of failures. The fact that no failures occurred when using the cache is not statistically significant.

The cache is specialized for the equijoin case, improving performance over the original MapReduce semantics. Specifically, the original semantics dictates that the reduce function must be called for every unique key, including all those keys in the cache that do not have a corresponding joining tuple in ΔA_i during semi-naive evaluation. By exposing a datalog interface rather than raw MapReduce, we free ourselves from these semantics and can only call the reduce function once for each incoming key. In Figure 7, the effect of this specialization is measured for the first few iterations.

6 Conclusions and Future Work

Informed by Hadoop's success as a runtime for relational algebra-based languages, and bulding on our previous work on the HaLoop system for iterative processing, we explore the suitability of Hadoop as a runtime for recursive Datalog. We find that caching loop invariant data delivers an order of magnitude speedup, while specialized implementations of the operators, careful configuration of Hadoop for iterative queries, and extensions to the cache to support set difference delivers another factor of 2 speedup.

We also find that the overhead of an individual Hadoop job is significant in this context, as it amplified by the iterative processing (500+ Hadoop jobs are executed to evaluate one query!) This overhead accounts for approximately half of time of each iteration step after all optimizations are applied.

In future work on system optimizations, we are considering extensions to Hadoop to optimize the caching mechanism and avoid unnecessary IO by using a full-featured disk-based indexing libraryon each node in a manner similar to HadoopDB [1]. We are also exploring the literature for new ways to mitigate the startup overheaad of each Hadoop job by sharing VMs across jobs.

Perhaps more importantly, we are aggressively exploring cost-based algebraic dynamic re-optimization techniques for parallel recursive queries, motivated in particular by the endgame problem (Figure 3). While this paper explored Hadoop in particular, our ongoing work is designed to produce optimization strategies that will transcend any particular distributed runtime. To demonstrate this, we are planning experiments that use more recent parallel runtimes that directly support recursion [10,19].

References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. Proc. VLDB Endow. 2(1), 922–933 (2009)

2. Afrati, F.N., Borkar, V., Carey, M., Polyzotis, N., Ullman, J.D.: Map-reduce extensions and recursive queries. In: Proceedings of the 14th International Conference on Extending Database Technology, EDBT/ICDT 2011, pp. 1–8. ACM, New York (2011)
3. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.: Boom analytics: exploring data-centric, declarative programming for the cloud. In: Morin, C., Muller, G. (eds.) EuroSys, pp. 223–236. ACM (2010)
4. Balduccini, M., Pontelli, E., Elkhatib, O., Le, H.: Issues in parallel execution of non-monotonic reasoning systems. Parallel Comput. 31(6), 608–647 (2005)
5. Barga, R., Ekanayake, J., Jackson, J., Lu, W.: Daytona: Iterative mapreduce on windows azure, http://research.microsoft.com/en-us/projects/daytona/
6. Borkar, V.R., Carey, M.J., Grover, R., Onose, N., Vernica, R.: Hyracks: A flexible and extensible foundation for data-intensive computing. In: Abiteboul, S., Böhm, K., Koch, C., Tan, K.-L. (eds.) ICDE, pp. 1151–1162. IEEE Computer Society (2011)
7. Bu, Y., Howe, B., Balazinska, M., Ernst, M.: Haloop: Efficient iterative data processing on large clusters. In: Proc. of International Conf. on Very Large Databases, VLDB (2010)
8. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
9. Eisner, J., Filardo, N.W.: Dyna: Extending Datalog for Modern AI. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 181–220. Springer, Heidelberg (2011)
10. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative mapreduce. In: HPDC, pp. 810–818 (2010)
11. Hadoop, http://hadoop.apache.org/
12. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Ferreira, P., Gross, T.R., Veiga, L. (eds.) EuroSys, pp. 59–72. ACM (2007)
13. Joslyn, C., Adolf, R., Al Saffar, S., Feo, J., Goodman, E., Haglin, D., Mackey, G., Mizell, D.: High performance semantic factoring of giga-scale semantic graph databases. In: Semantic Web Challenge Billion Triple Challenge (2010)
14. Loo, B.T., Gill, H., Liu, C., Mao, Y., Marczak, W.R., Sherr, M., Wang, A., Zhou, W.: Recent Advances in Declarative Networking. In: Russo, C., Zhou, N.-F. (eds.) PADL 2012. LNCS, vol. 7149, pp. 1–16. Springer, Heidelberg (2012)
15. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Elmagarmid, A.K., Agrawal, D. (eds.) SIGMOD Conference, pp. 135–146. ACM (2010)
16. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Wang, J.T.-L. (ed.) SIGMOD Conference, pp. 1099–1110. ACM (2008)
17. Perri, S., Ricca, F., Sirianni, M.: A parallel asp instantiator based on dlv. In: Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, pp. 73–82. ACM, New York (2010)
18. Shaw, M., Detwiler, L., Noy, N., Brinkley, J.: S.D.: vsparql: A view definition language for the semantic web. Journal of Biomedical Informatics (2010), doi:10.1016/j.jbi, 08.008
19. Zaharia, M., Chowdhury, N.M.M., Franklin, M., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley (May 2010)

Logical Foundations of Continuous Query Languages for Data Streams

Carlo Zaniolo

University of California at Los Angeles
zaniolo@cs.ucla.edu

Abstract. Data Stream Management Systems (DSMS) have attracted much interest from the database community, and extensions of relational database languages were proposed for expressing continuous queries on data streams. However, while relational databases were built on the solid bedrock of logic, the same cannot be said for DSMS. Thus, a logic-based reconstruction of DSMS languages and their unique computational model is long overdue. Indeed, the banning of blocking queries and the fact that stream data are ordered by their arrival timestamps represent major new aspects that have yet to be characterized by simple theories. In this paper, we show that these new requirements can be modeled using the familiar deductive database concepts of closed-world assumption and explicit local stratification. Besides its obvious theoretical interest, this approach leads to the design of a powerful version of Datalog for data streams. This language is called Streamlog and takes the query and application languages of DSMS to new levels of expressive power, by removing the unnecessary limitations that severely impair current commercial systems and research prototypes.

1 Introduction

Data stream management systems represent a vibrant area of new technology for which researchers have extended database query languages to support continuous queries on data streams [4,3,8,10,18,7,12,20]. These database-inspired approaches have produced remarkable systems and applications, but have yet to deliver solid theoretical foundations for their data models and query languages—particularly if we compare with the extraordinary ones on which the success of relational databases was built. Logic provided the theoretical bedrock for relational databases from the very time in which they were introduced by E.F. Codd, and this foundation was then refined, generalized and strengthened by the work on database and logic, and Datalog, which delivered concepts and models of great power and elegance [21,1,22].

Until now, DSMS researchers have made little use of logic-based concepts, although these provide a natural formalism and simple solutions for many of the difficult problems besetting this area, as we will show in this paper. In particular, we show that Reiter's Closed World assumption [19] provides a natural basis on which to study and formalize the blocking behavior of continuous query operators, whereby concepts such as local stratification can be used to achieve a natural and efficient expression of recursive rules with non-monotonic constructs.

The paper is organized as follows. In the next section, we present a short discussion of related previous work and then, in Section 3, we explore the problem of supporting order and recursion on single stream queries for both monotonic and non-monotonic constructs. Thus, in Section 4, we introduce Streamlog, which is basically Datalog with modified well-formedness rules for negation. These rules guarantee both simple declarative semantics and efficient execution (Section 5). Because of possible skews between their timestamps, multiple streams pose complex challenges at the logical and implementation levels. We study this problem in Section 6, where we propose a backtrack-oriented solution and show that its benefits extend well beyond union.

2 Continuous Queries on Relational Data Streams

As described in various surveys [4,12], data streams can be modeled as append-only relations on which the DSMS is asked to support standing queries (i.e., continuous queries). As soon as tuples arrive in the input stream, the DSMS is expected to decide, in real time or quasi real-time, which additional results belong to the query answer and promptly append them to the output stream. This is an incremental computation model, where no output can be taken back; therefore, the DSMS might have to delay returning an output tuple until it is sure that the tuple belongs to the final output—a certainty that for many queries is only reached after the DSMS has seen the whole input. The queries showing this behavior, and operators causing it, are called *blocking*, and have been characterized in [4] as follows: *A blocking query operator is one that is unable to produce the first tuple of the output until it has seen the entire input.* Clearly, blocking query operators are incompatible with the computation model of DSMS and should be disallowed, whereas all non-blocking queries should instead be allowed. However, many queries and operators, including essential ones such as union, fall in-between and are only partially blocking; currently, we lack simple rules to decide when, and to which extent, partially blocking operators should be allowed and how they should be treated. Therefore, better understanding and formal characterizations are badly needed.

The main previous results on blocking queries proved that non-monotonic query operators are blocking, whereas monotonic operators are non-blocking [17,13]. Given that negation and traditional aggregates are non-monotonic, most current DSMS simply disallow them in queries, although this exclusion causes major losses in expressive power [17]. However, a more sophisticated analysis suggests that these losses are avoidable, since (i) the monotonicity notion used in [17] is not the subset ordering used in databases and Horn clauses, and (ii) previous research on deductive databases made great strides in coping with non-monotonicity via concepts such as stratification and stable models [22].

Therefore, in this paper, we provide a reasoned reconstruction of the basic non-monotonic theory of logic languages in the context of data streams, leading to the

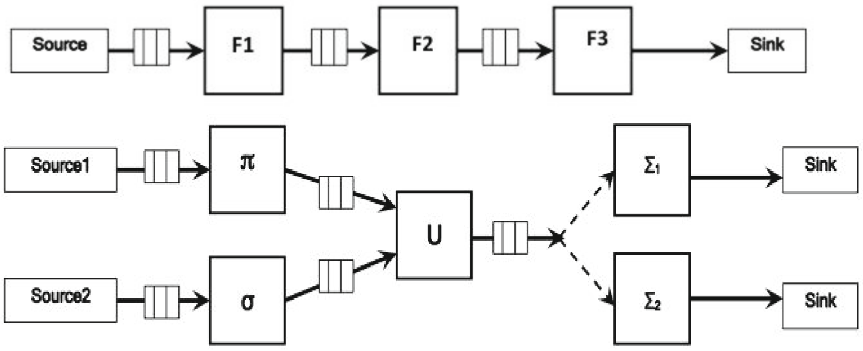


Fig. 1. Continuous Query Graphs

design of a concrete language called Streamlog. We will revisit the closed-world assumption and adapt well-known concepts such as stratification to discover, much to our surprise, that non-monotonic constructs dovetail with data stream languages, enabling Streamlog to achieve great expressive power.

Queries on data streams are commonly visualized using workflow models such as that of Figure 1, that show the pipelined execution used by the DSMS to implement continuous queries. The boxes labelled `Source` at the left of our graph, depict tuples coming from an external stream source or a database relation. For instance in the first query, the source feeds incoming tuples to a buffer; then query operator `F1` takes the tuples from this buffer and feeds them to its output buffer that supplies operator `F2`, and so on. As shown in Figure 1, some boxes might consist of very simple operators, e.g., the relational algebra operators of projection, selection and union. In general, however, the boxes can implement much more complex functions, including pattern search operators or data mining tasks [20]. Complex functions can been written as user-defined aggregates written natively in SQL [20], but Streamlog can also be quite effective in this role.

A key assumption is that operators are order-preserving. Thus, each operator takes tuples from the front of its input queue and add the tuple(s) it produces, if any to the tail of its output buffer. Thus, buffers might delay but not alter the functions computed by simply feeding the output of one operator directly into the input of the next. Thus, when the operators are defined by Streamlog rules, then the semantics of our continuous query is defined by the logic program consisting of (i) the goal defined by the `Sink` node (ii) the rules in the boxes feeding, directly or indirectly, into the sink node, and (iii) the facts streaming from the `source` nodes into said boxes and rules.

In this paper, we focus on data streams whose tuples are explicitly timestamped. More specifically, we will assume that the first column of our tuples contain a timestamp that either (i) was created by the external device that created the tuple (external timestamp) or (ii) it was added by the DSMS at the time it received the tuple (internal timestamp). In either case, tuples are arranged and processed by increasing values of their timestamps. Extending these results to data streams that are not timestamped will be discussed in future papers.

3 Single Stream Processing

The top query graph of Figure 1 shows the processing flow for a single stream, while the one below it shows the processing of multiple streams. In both cases we assume that timestamped data streams (i) enter each query operator in increasing timestamp order and (ii) leave the query operator in the same order. As we shall see, although (i) and (ii) represent two facets of the same problem, the technical issues and opportunities they bring about are quite different. For (i) consider the example of a stream of messages of the form `msg(Time, MsgCode)` and say that we are looking for repeated occurrences of code "red" messages. Then the following Datalog rule can be used to define multiple occurrences of the same alarm code "X":

Example 1. Repeated occurrences of the same alarm.

$$\texttt{repeated}(\texttt{T},\texttt{X}) \leftarrow \texttt{msg}(\texttt{T},\texttt{X}), \texttt{msg}(\texttt{T0},\texttt{X}), \texttt{T} > \texttt{T0}.$$

Thus, the final query goal ?$\texttt{repeated}(\texttt{T},\texttt{red})$ will detect repeated occurrences of code "red", whereby an application might sound an alarm, which is triggered for all but the first occurrence of code red.

The semantics of query Q on a stream, such as `msg`, is defined by the cumulative answer that Q has returned until time τ. This cumulative answer at time τ must be equal to the answer computed upon the database containing all the data stream tuples with timestamp $\leq \tau$. In a blocking query, this equality only holds at the end of the input, whereas for a continuous non-blocking query it must hold for every instant in time.

Massive data streams over long periods can exceed the system storage capacity. In DSMS, this problem is addressed with windows or other synopses [3]. Queries involving windows can be easily expressed using rules. For instance if $\texttt{wsize}(\texttt{W})$ defines the window within which we detect repetitions, Example 1 becomes:

Example 2. Multiple occurrences within a window

$$\begin{aligned}\texttt{windoweg}(\texttt{T2},\texttt{red}) \leftarrow\ & \texttt{msg}(\texttt{T2},\texttt{red}), \texttt{msg}(\texttt{T1},\texttt{red}),\\ & \texttt{T1} < \texttt{T2}, \texttt{wsize}(\texttt{W}), \texttt{T2} \leq \texttt{T1} + \texttt{W}.\end{aligned}$$

But, unlike in other DSMS [3], windows do not play a key role in our semantics.

The Importance of Order. Since query operators return sequences of tuples that are fed into the next query operator, assuring the correct order of their output sequences becomes critical. To illustrate this point, say that we modify Example 1, above, by keeping the body of the rule unchanged; but then we change the head of the rule so that the timestamp of the former occurrence is used, rather than the current one:

Example 3. Time-warped repetitions ?$\texttt{wrepeated}(\texttt{Time},\texttt{X})$

$$\texttt{wrepeated}(\texttt{T0},\texttt{X}) \leftarrow \texttt{msg}(\texttt{T},\texttt{X}), \texttt{msg}(\texttt{T0},\texttt{X}), \texttt{T} > \texttt{T0}.$$

We immediately realize that there is a problem, since repetitions normally arrive in an order that is different from that of their previous occurrences. For instance, we might have that a message with code α arrives at time t_α, followed by a message with code β, which is then repeated immediately, while the first repetition of α arrives 10 minutes later. Then, to produce tuples by increasing timestamps, we will need to hold up the output for 10 minutes. Here too punctuation marks and windows are effective palliatives to control the problem, but in general the delay required can be unbound. The situation of *unbound wait* can be as bad as that of blocking queries. For instance say that at some point, a rare color shows up in our input stream, never to show up again. Then for any new color that has its first occurrence after our rare color, no output can ever be generated until the very end of the input. In a nutshell, rules such as that of Example 3 must be disallowed, although they contain no negation or other nonmonotonic operators.

Negated Goals: The addition of order-inducing constraints in the rules offers unexpected major benefits when dealing with negated goals. Say that we want to detect the first occurrence of "code red" warning. For that, we only need to make sure that once we receive such a message there is no identical other message preceding it:

Example 4. First occurrence of code red: ?$\texttt{first}(\texttt{T},\texttt{red})$.

$$\begin{aligned}\texttt{first}(\texttt{T},\texttt{X}) &\leftarrow \texttt{msg}(\texttt{T},\texttt{X}), \neg\texttt{previous}(\texttt{T},\texttt{X}).\\ \texttt{previous}(\texttt{T},\texttt{X}) &\leftarrow \texttt{msg}(\texttt{T0},\texttt{X}), \texttt{T0} < \texttt{T}.\end{aligned}$$

To find the second occurrence of code red we will start by finding one that follows the first. Moreover there cannot be any other occurrence between this and the first one:

Example 5. Second occurrence of code red ?$\texttt{second}(\texttt{T},\texttt{red})$.

$$\begin{aligned}\texttt{second}(\texttt{T2},\texttt{Y}) &\leftarrow \texttt{first}(\texttt{T1},\texttt{Y}), \texttt{msg}(\texttt{T2},\texttt{Y}), \texttt{T2} > \texttt{T1}, \neg\texttt{befr}(\texttt{T2},\texttt{Y}).\\ \texttt{befr}(\texttt{T2},\texttt{Y}) &\leftarrow \texttt{first}(\texttt{T1},\texttt{Y}), \texttt{T1} < \texttt{T2}, \texttt{msg}(\texttt{Tb},\texttt{Y}), \texttt{Tb} < \texttt{T2}, \texttt{T1} < \texttt{Tb}.\end{aligned}$$

These queries only use negation on events that, according to their timestamps, are past event. Thus the queries can be answered in the present: they are non-blocking. Therefore, they should be allowed by a DSMS compiler, which must therefore be able to set them apart from other queries with negation which are instead blocking.

For instance, a blocking query is the following one that finds the last occurrence of code-red alert:

Example 6. Last occurrence of code red: ?$\texttt{last}(\texttt{T},\texttt{red})$.

$$\begin{aligned}\texttt{last}(\texttt{T},\texttt{Z}) &\leftarrow \texttt{msg}(\texttt{T},\texttt{Z}), \neg\texttt{next}(\texttt{T},\texttt{Z}).\\ \texttt{next}(\texttt{T},\texttt{Z}) &\leftarrow \texttt{msg}(\texttt{T1},\texttt{Z}), \texttt{T1} > \texttt{T}.\end{aligned}$$

This is obviously a blocking query, inasmuch as we do not have the information needed to decide whether the current red-alert message is actually the final one, while messages are still arriving. Only when the data stream ends, we can make such an inference: to answer this query correctly, we will have to wait till the input stream has completed its arrival, and then we can use the standard CWA to entail the negation that allows us to answer our query. But the standard CWA assumption will not help us to conclude that our query is non-blocking. We will instead exploit the timestamp ordering of the data streams to define a Progressive Closing World Assumption (PCWA) that can be used in the task. In our definition, we will also include traditional database facts and rules, since these might also be used in continuous queries.

Progressive Closing World Assumption (PCWA): We consider a world consisting of one timestamped-ordered stream and database facts. Once a fact `stream`(`T`, ...) is observed in the input stream, the PCWA allows us to assume ¬`stream`(`T1`, ...), provided that `T1` < `T`, and `stream`(`T1`, ...) is not entailed by our fact base augmented with the `stream` facts having timestamp ≤ `T`.

Therefore, our PCWA for a single data stream revises the standard CWA of deductive databases with the provision that the world is in fact expanding according to its timestamps. Therefore, we will also allow the standard notions of entailment that guarantee consistency: besides the least models of Horn Clauses these also include the perfect models of (locally) stratified programs.

In the next section, we derive from the PCWA simple conditions that ensure syntactic well-formedness and efficient implementation for our programs.

4 Streamlog

In Streamlog, base predicates, derived predicates, and the query goal are all timestamped in their first arguments. These will be called temporal, to distinguish them from non-timestamped database facts and predicates that might also be used in the programs.

The same safety criteria used in Datalog can be used in Streamlog. Furthermore, we assume that timestamp variables are made safe by equality chains equating their values to the timestamps in the base stream predicates. Therefore, even if $T1$ is safe, expressions such as $T2 = f(T1)$ or $T2 = T1 + 1$ cannot be used to deduce the safety of $T2$. Only equality can be used for timestamp arguments.

We can now propose obvious syntactic rules that will avoid blocking behavior in the temporal rules of safe Streamlog programs.

– *Strictly Sequential:* A rule is said to be *Strictly sequential* when the timestamp of its head is > than every timestamp in the body of the rule. A predicate is strictly sequential when all the rules defining it are strictly sequential.
– *Sequential:* A rule is said to be sequential when it satisfies the following three conditions:
(i) the timestamp of its head is equal to the timestamp of some positve goal,
(ii) the timestamp of its head is > or ≥ than the timestamps of the remaining goals, and

(iii) its negated goals are strictly sequential or have a timestamp that is $<$ than the timestamp of the head.

– A program is said to be sequential when all its rules are sequential or strictly sequential.

The programs in Examples 4, and 5 are sequential, given that the predicates `previous` and `befr` in their negated goals are strictly sequential.

Next observe that the programs in Examples 4, and 5 are stratified with respect to negation with `previous` occupying a lower stratum than `first`, which is in a stratum not higher than `befr`, which is in a stratum lower than `second`.

Stratified Datalog programs have a syntactic structure that is easy for a compiler to recognize and turn into an efficient implementation [22]. In fact, the unique stable model of these programs, called the perfect model, can be computed efficiently using a stratified iterated fixpoint [22]. Unfortunately stratified programs do not allow negation or aggregates in recursive rules, and therefore, are not conducive to efficient expression of algorithms such as shortest path. Much previous research was devoted to overcoming this limitation. In particular, there is a class of programs called locally stratified programs that have a unique stable model, called perfect model. Unfortunately, the stratification for a locally stratified programs can only be verified against its instantiated version, whereby supporting perfect models is, in general, an $\Pi^1{}_1$-complete problem [9].

Overcoming this limitation and supporting negation or aggregates in recursion has thus provided a major focus of topical research where progress has been very slow. Therefore, we were pleasantly surprised to find out that the simple notion of sequential programs for Streamlog avoids the non-monotonicity problems that have hamstrung Datalog and frustrated generations of researchers. To illustrate this point, let us first use the stratified program of Example 7 to express the well-known shortest path problem. In this example, we use the paths computed for previous timestamps to discard longer arcs in the incoming stream. We also use a simple predicate `lgr(T1, T2, T)` whereby `T` is equal to the larger of the first two arguments:

Example 7. Continuous shortest paths in graphs defined by stream of arcs.

$$\begin{array}{ll}
\mathtt{path(T,X,Y,D)} \leftarrow & \mathtt{arc(T,X,Y,D)}, \neg\mathtt{shorter(T,X,Y,D)}.\\
\mathtt{shorter(T,X,Y,D)} \leftarrow & \mathtt{path(T1,X,Y,D1)}, \mathtt{T1} < \mathtt{T}, \mathtt{D1} \leq \mathtt{D}.\\
\\
\mathtt{path(T,X,Z,D)} \leftarrow & \mathtt{path(T1,X,Y,D1)}, \mathtt{path(T2,Y,Z,D2)},\\
 & \mathtt{lgr(T1,T2,T)}, \mathtt{D} = \mathtt{D1{+}D2}.
\end{array}$$

According to these rules, the arrival of one or more new arc will trigger addition of new edges in `path`. Then these new edges can trigger the addition of additional ones in recursive rule, where `path` appears twice. The differential fixpoint used in this computation [22] will result in at least one of these two `path` goals to have a timstamp equal to `T`—i.e., the larger of the two values is used to timestamp new fact generated in the head. This quadratic expression of transitive closure requires only the memorization of `path`; it is thus preferable to a linear rule that uses both `arc` and `path`, both of which then require memorization.

5 Declarative Semantics and Operational Semantics

Example 8 above, shows how to improve our rules by *pushing negation into recursion.* The program so obtained is sequential, and therefore it has a formal semantics and efficient implementation that are discussed after the example.

Example 8. Negation can be pushed inside recursion.

$$\begin{aligned}
\texttt{minpath(T,X,Y,D)} &\leftarrow \texttt{arc(T,X,Y,D)}, \neg\texttt{shorter(T,X,Y,D)}.\\
\texttt{minpath(T,X,Z,D)} &\leftarrow \texttt{minpath(T1,X,Y,D1)}, \texttt{minpath(T2,Y,Z,D1)}, \texttt{lgr(T1,T2,T)},\\
&\quad\; \neg\texttt{shorter(T,X,Z,D)}, \texttt{D} = \texttt{D1} + \texttt{D2}.\\
\texttt{shorter(T,X,Z,D)} &\leftarrow \texttt{minpath(T1,X,Z,D1)}, \texttt{D} \leq \texttt{D1}, \texttt{T1} < \texttt{T}.
\end{aligned}$$

The timestamps in our data stream form a sequence that is unbound but finite. We can denote them by their position in the sequence, and talk about the n^{th} timestamp, without fear of ambiguity. Then, sequential programs are locally stratified by their timestamp values as discussed next. To prove this we will construct the *bistate* equivalent of our program. The first step is a temporal expansion, where a rule with a condition $\leq$ ($\geq$) between temporal arguments, is replaced by two rules: in the first rule, the temporal arguments are set to equal, and in the other they are set to $<$ ($> S$). Likewise, a rule with `lgr` is replaced by three rules, resp. for $=$, $<$ and$>$. Then we have the following rewriting:

1. In each rule, rename with the suffix `new_` the head predicate and the body predicates that have a timestamp equal to the that of the head,
2. Rename all the predicates in the body whose temporal argument is less than that of the head by the suffix `_old`
3. Remove the temporal arguments from the rules.

Thus, for Example 8 we obtain:

Example 9. Bistate represention for the program of Eample 8

$$\begin{aligned}
\texttt{minpath\_new(X,Y,D)} &\leftarrow \texttt{arc\_new(X,Y,D)}, \neg\texttt{shorter\_new(X,Y,D)}.\\
\texttt{minpath\_new(X,Z,D)} &\leftarrow \texttt{minpath\_new(X,Y,D1)}, \texttt{minpath\_new(Y,Z,D1)},\\
&\quad\; \neg\texttt{shorter\_new(X,Z,D)}, \texttt{D} = \texttt{D1} + \texttt{D2}.\\
\texttt{minpath\_new(X,Z,D)} &\leftarrow \texttt{minpath\_old(X,Y,D1)}, \texttt{minpath\_new(Y,Z,D1)},\\
&\quad\; \neg\texttt{shorter\_new(X,Z,D)}, \texttt{D} = \texttt{D1} + \texttt{D2}.\\
\texttt{minpath\_new(X,Z,D)} &\leftarrow \texttt{minpath\_new(X,Y,D1)}, \texttt{minpath\_old(Y,Z,D1)},\\
&\quad\; \neg\texttt{shorter\_new(X,Z,D)}, \texttt{D} = \texttt{D1} + \texttt{D2}.\\
\\
\texttt{shorter\_new(X,Z,D)} &\leftarrow \texttt{minpath\_old(X,Z,D1)}, \texttt{D} \leq \texttt{D1}.
\end{aligned}$$

The program so obtained is stratifiable in several ways, including the following one: we assign to stratum 0 all and only the predicates with suffix `old`, and the predicates with suffix `new` are all in higher strata. For instance, we will assign `minpath_old` to level 0, and then `shorter_new`, and `arc_new` to level 1, and `minpath_new` to level 2. Thus here we have one stratum with `_old` predicates, and $S = 2$ strata for `_new` predicates.

Now we can generate a local stratification based on the distinct temporal values of the timestamps which form a finite sequence $\tau_1, \ldots, \tau_\mathtt{n}$. In our case $\mathtt{minpath}(0, \ldots)$ will be asigned to stratum 0, $\mathtt{shorter}(\tau_1, \ldots)$ and $\mathtt{arc}(\tau_1, \ldots)$ are assigned to stratum 1, and $\mathtt{minpath}(\tau_1, \ldots)$ are assigned to stratum 2. Then the process repeats with with temporal argument T2 being assigned as follows: $\mathtt{shorter}(\tau_2, \ldots)$ and $\mathtt{arc}(\tau_2, \ldots)$ to stratum 3, and $\mathtt{minpath}(\tau_2, \ldots)$ to stratum 4. Thus, for our sequence $\tau_1, \ldots, \tau_n$ we have $1 + n \times S$ strata, where a $\mathtt{p\_new}(\tau_\mathtt{j}, \ldots)$ that belonged to state k in the bistate version will now be assigned to stratum $j \times S + k$ in the local stratification.

The computation of perfect model for the locally stratified program now becomes straightforward. Basically, we iterate over the following two steps *for each set of tuples arriving with a new timestamp*: (i) the stratified bistate version of the program is used to derive additional new values for the predicates, and (ii) the old version is incremented with this newly derived atoms.

6 Multiple Streams

A much studied DSMS problem is how to best ensure that binary query operators, such as unions or joins, generate outputs sorted by increasing timestamp values [14,5,6]. To derive a logic-based characterization of this problem, assume that our msg stream is in fact built by combining the two message streams sensr1 and sensr2. For stored data, this operation requires a simple disjunction as follows:

Example 10. Disjunction expressing the union of two streams.

$$\mathtt{msg(T1, S1)} \leftarrow \mathtt{sensr1(T1, S1)}.$$
$$\mathtt{msg(T2, S2)} \leftarrow \mathtt{sensr2(T2, S2)}.$$

However even if sensr1 and sensr2 are ordered by their timestamps, this disjunction says nothing about the fact that the output should be ordered. Indeed, assuring such an order represents a serious problem for a DSMS, due to the time-skews that normally occur between different data streams. Thus, for the union in Figure 1, when one of the two input buffers is empty, we cannot take the first item from the other buffer, until we know what its timestamp value will be. This problem has been extensively studied, but only at the implementation level [14,5,6]. At the logical level the problem can be solved as follows:

Example 11. Union of synchronized streams.

$$\mathtt{msg(T1, S1)} \leftarrow \mathtt{sensr1(T1, S1)}, \neg\mathtt{missing2(T1)}.$$
$$\mathtt{msg(T2, S2)} \leftarrow \mathtt{sensr2(T2, S2)}, \neg\mathtt{missing1(T2)}.$$

Now we check that all the stream2 tuples (resp. the stream1 tuples) with timestamp less than T1 (resp. less than T2) added to msg:

$$\mathtt{missing2(T1)} \leftarrow \mathtt{sensr2(T2, S2)}, \mathtt{T2} < \mathtt{T1}, \neg\mathtt{msg(T2, S2)}.$$
$$\mathtt{missing1(T2)} \leftarrow \mathtt{sensr1(T1, S)}, \mathtt{T1} < \mathtt{T2}, \neg\mathtt{msg(T1, S1)}.$$

The expression given in Example 11 is clearly better than the sort-merge approach proposed in the literature that can be described as follows:

Example 12. Union of unsynchronized streams by sort merging.

$$\mathtt{msg(T1,S1)} \leftarrow \mathtt{sensr1(T1,S1),sensr2(T2,\_),T2} \geq \mathtt{T1}.$$
$$\mathtt{msg(T2,S2)} \leftarrow \mathtt{sensr2(T2,S2),sensr1(T1,\_),T1} \geq \mathtt{T2}.$$

This expression is correct but not complete[1]. As a result, this operator might have to enter an idle-waiting state that is akin to temporary blocking [5].

From the viewpoint of users, neither the solution in Example 11 nor that in Example 12 are satisfactory. What users instead want is to write the simple rules shown in Example 10 and let the system take care of time-skews. Therefore in Streamlog, we will allow users to work under the *Perfect Syncronization Assumption (PSA)*, whereby the data streams of interest are perfectly synchronized. Under PSA, we ca now extend the PCWA we had previously defined for a single data stream to a collection of N data streams $\mathtt{stream_j(T,\ldots)}$, $j = 1, \ldots, N$ as follows: We can assume $\neg\mathtt{stream_j(T_j,\ldots)}$ iff for some i, $1 \leq i \leq N$ $\mathtt{stream_i(T_i,\ldots)}$ with $\mathtt{T_i} > \mathtt{T_j}$, and $\mathtt{stream(T_j,\ldots)}$ is not entailed by our fact base augmented with all the stream facts with timestamp $\leq \mathtt{T_i}$ Since in reality PSA does not hold, the DSMS is given the responsibility to enforce efficient policies and conditions needed to ensure that queries return the same answers as those produced under PSA. For instance, for the query at hand, the DSMS system might in fact enforces conditions such as $\neg\mathtt{missing2(T1)}$ in the first rule of our union. Efficient support of these PSA-emulating conditions requires the use of sophisticated techniques, such as *intelligent backtracking* [5]. For instance in Figure 1, say that the lower buffer feeding the union has a tuple with timestamp t_1, while the other buffer is empty. Rather than waiting idly for the arrival of some tuples in the empty buffer, we can backtrack to the previous operators feeding the buffer. If a tuple with timestamp $< t_1$ is found, it must be moved quickly through the operators since this is the one that must pass through to the union next. But if only tuples with timestamps $> t_1$ are found, then the union operator will be signalled (e.g., via punctuation marks) to let the tuple with timestamp t_1 to go through. Finally, if the buffer is empty, an additional backtracking step will be performed to visit the buffer supplied by `Source1`, and so on. As discussed in [5], this backtracking approach can lead to significant improvements in the response time of our DSMS. Although space limitations prevent us from discussing this approach further, we observe that (i) while Streamlog is clearly inspired by Datalog, its execution exploits Prolog's backtracking mechanism, and (ii) the techniques used to support PSA are also very useful to control and expedite execution of single-stream queries.

To illustrate (ii), say that $\mathtt{tempr(Time,Locat,Celcius)}$ is the stream containing the temperature readings of our sensors, from various locations. Then the following rules could be used to continuously return each new temperature maximum:

[1] For instance, the `sensr2` stream might have ended and the current clock is past `T1`.

$$\begin{aligned}&\texttt{max(T, Loc, Cel)} \leftarrow \texttt{tempr(T, Loc, Cel)}, \neg\texttt{hotter(T, Cel)}.\\ &\texttt{hotter(T1, C1)} \leftarrow \ \texttt{tempr(T2,, C2)}, \texttt{C2} \geq \texttt{C1}, \texttt{T2} \leq \texttt{T1}.\end{aligned}$$

Here the backtracking technique used to support NSA for union can be used to detect that all the the `tempr` tuples with timestamp $\leq$ `T` have already arrived and thus the query ?`max(T, Loc, Celcius)` can be answered at once without awaiting for tuples with timestamps larger than `T`. Also, if we construct the bistate equivalent of our rules we see that we obtain a stratified program, whereby the original program is locally stratified and the efficient execution techniques previously discussed remain valid. Therefore we can relax the definition of Strictly Sequential rules as follows:
Strictly Sequential: A rule is said to be *Strictly sequential* when the timestamp of the head of the rule is $>$ than the timestamp of each recursive goal and $\geq$ the timestamps of the non-recursive goals.

This extension does not compromise the key properties of our sequential programs, for which the following properties hold[2]:

Theorem 1. *If P is a Sequential Program then: (i) P is locally stratified, and (ii) the unique stable model of P can be computed by repeating the iterated fixpoint of its bistate version for each timestamp value.*

For an additional example illustrating the uses of this generalization, let us return to our shortest-path program in Example 8. When several arcs arrive with the same timestamp, they might result in the addition of multiple paths between the same node pair. Thus we could add an additional rule (and stratum) to select the shortest among such paths that share the same timestamp.

7 Conclusion

While the results presented here are still preliminary, they show that logic can bring sound theoretical foundations and superior expressive power to DSMS languages which, currently, are dreadfully lacking in both. In terms of syntax, Streamlog is just standard Datalog over timestamed predicates; however Streamlog obtains the greater level of expressive power that negation (and aggregates) in recursive rules entail by guaranteeing that simple sequentiality conditions holds between the timestamped predicates in the rules. The use of standard Datalog implies that the implementation techniques developed for XY-stratification [23] can be used for Streamlog, and similar results are at hand for the many DSMS that use continuous versions of SQL. This also sets our approach apart from that proposed in [2] that relies on an explicit sequencing operator SEQ and an operational semantics that is realized through a Prolog-based implementation.

[2] The outline of the proof of this property is similar to that outlined in previous section and it is based on a similar proof for XY-stratification presented in [22]. In fact, the temporal arguments define an explicit stratification that is similar to that of XY-stratified programs [23] and Statelog programs [15].

Acknowledgements. This work was supported in part by NSF (Grant No. IIS 1118107). Thanks are due to the reviewers for many useful comments.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Anicic, D., Fodor, P., Rudolph, S., Stühmer, R., Stojanovic, N., Studer, R.: A Rule-Based Language for Complex Event Processing and Reasoning. In: Hitzler, P., Lukasiewicz, T. (eds.) RR 2010. LNCS, vol. 6333, pp. 42–57. Springer, Heidelberg (2010)
3. Arasu, A., Babu, S., Widom, J.: CQL: A Language for Continuous Queries over Streams and Relations. In: Lausen, G., Suciu, D. (eds.) DBPL 2003. LNCS, vol. 2921, pp. 1–19. Springer, Heidelberg (2004)
4. Babcock, B., et al.: Models and issues in data stream systems. In: PODS (2002)
5. Bai, Y., Thakkar, H., Wang, H., Zaniolo, C.: Optimizing timestamp management in data stream management systems. In: ICDE (2007)
6. Bai, Y., Thakkar, H., Wang, H., Zaniolo, C.: Time-stamp management and query execution in data stream management systems. IEEE Internet Computing 12(6), 13–21 (2008)
7. Carney, D., et al.: Monitoring streams - a new class of data management applications. In: VLDB, Hong Kong, China (2002)
8. Chandrasekaran, S., Franklin, M.: Streaming queries over streaming data. In: VLDB (2002)
9. Cholak, P., Blair, H.A.: The complexity of local stratification. Fundam. Inform. 21(4), 333–344 (1994)
10. Cranor, C., Gao, Y., Johnson, T., Shkapenyuk, V., Spatscheck, O.: Gigascope: High performance network monitoring with an sql interface. In: SIGMOD, p. 623. ACM Press (2002)
11. Gallaire, H., Nicolas, J.-M., Minker, J. (eds.): Advances in Data Base Theory, vol. 1. Plemum Press (1981)
12. Golab, L., Tamer Özsu, M.: Issues in data stream management. ACM SIGMOD Record 32(2), 5–14 (2003)
13. Gurevich, Y., Leinders, D., Van den Bussche, J.: A Theory of Stream Queries. In: Arenas, M. (ed.) DBPL 2007. LNCS, vol. 4797, pp. 153–168. Springer, Heidelberg (2007)
14. Johnson, T., Muthukrishnan, S., Shkapenyuk, V., Spatscheck, O.: A heartbeat mechanism and its application in gigascope. In: VLDB, pp. 1079–1088 (2005)
15. Lausen, G., Ludäscher, B., May, W.: On Active Deductive Databases: The Statelog Approach. In: Freitag, B., Decker, H., Kifer, M., Voronkov, A. (eds.) DYNAMICS 1997, and ILPS-WS 1997. LNCS, vol. 1472, pp. 69–106. Springer, Heidelberg (1998)
16. Law, Y.-N., Wang, H., Zaniolo, C.: Data models and query language for data streams. In: VLDB, pp. 492–503 (2004)
17. Law, Y.-N., Wang, H., Zaniolo, C.: Relational languages and data models for continuous queries on sequences and data streams. ACM Trans. Database Syst. 36, 8:1–8:32 (2011)
18. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously Adaptive Continuous Queries over Streams. In: SIGMOD, pp. 49–61 (2002)
19. Reiter, R.: On closed world data bases. In: Logic and Data Bases, pp. 55–76 (1977)

20. Thakkar, H., Laptev, N., Mousavi, H., Mozafari, B., Russo, V., Zaniolo, C.: Smm: A data stream management system for knowledge discovery. In: ICDE, pp. 757–768 (2011)
21. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. I. Computer Science Press (1988)
22. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann (1997)
23. Zaniolo, C., Arni, N., Ong, K.: Negation and Aggregates in Recursive Rules: The LDL++ Approach. In: Ceri, S., Tsur, S., Tanaka, K. (eds.) DOOD 1993. LNCS, vol. 760, pp. 204–221. Springer, Heidelberg (1993)

Author Index

Alvaro, Peter 135
Alviano, Mario 31
Ameloot, Tom J. 44
Aref, Molham 1

Bertossi, Leopoldo 86
Brass, Stefan 56

Civili, Cristina 68
Conway, Neil 135

Eiter, Thomas IX

Febbraro, Onofrio 81

Gardezi, Jaffer 86
Grasso, Giovanni 81
Green, Todd J. 1
Gurevich, Yuri 9

Hellerstein, Joseph M. 135
Howe, Bill 165

Interlandi, Matteo 99

Karvounarakis, Grigoris 1
Köhler, Sven 111
Kolaitis, Phokion G. XI
Koutris, Paraschos 165

Leone, Nicola 31, 81
Ludäscher, Bertram 111
Lukasiewicz, Thomas 123

Maier, David 135
Manna, Marco 31
Marczak, William R. 135
Martinez, Maria Vanina 123
Mugnier, Marie-Laure 21

Polleres, Axel 27

Reale, Kristian 81
Ricca, Francesco 81
Ritter, Daniel 148
Rosati, Riccardo 68

Saccà, Domenico 153
Serra, Edoardo 153
Shaw, Marianne 165
Simari, Gerardo I. 123
Smaragdakis, Yannis 111
Suciu, Dan 165

Terracina, Giorgio 31

Van den Bussche, Jan 44
Veltri, Pierfrancesco 31

Westmann, Till 148

Zaniolo, Carlo 177