# JowsDatalog: an Ontology Based Data Access Tool for Jointly-Weakly-Sticky Datalog$^{\pm}$ Programs

by

Pourya Saljoughi Badlou

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Computer Science

in

Computer Science

Carleton University
Ottawa, Ontario

© 2019
Pourya Saljoughi Badlou

# Abstract

*Jointly-Weakly-Sticky* (*JWS*) Datalog$^\pm$ is an expressive member of the family of Datalog$^\pm$ programs. It is characterized by a marking procedure, the *existential dependency graph*, and joint acyclicity. Query-answering (QA) can be done in polynomial-time in data complexity through $\mathsf{SChQA}^{\mathcal{S}}$, a chase-based, bottom-up QA algorithm for *JWS* Datalog$^\pm$ programs. The QA algorithm can be optimized by using a *magic-sets* query rewriting technique, $\mathsf{MagicD}^{+}$, for *JWS* programs. $\mathsf{MagicD}^{+}$ takes a Datalog$^\pm$ program and a query, and rewrites the combination into a new Datalog$^\pm$ program that becomes an input to $\mathsf{SChQA}^{\mathcal{S}}$. With the new program, $\mathsf{SChQA}^{\mathcal{S}}$ avoids generating irrelevant facts. The main contributions of this thesis are the design and implementation of an *ontology-based data access* (OBDA) tool, JowsDatalog, in which $\mathsf{SChQA}^{\mathcal{S}}$ and $\mathsf{MagicD}^{+}$ are implemented.

## Acknowledgements

I would like to thank my supervisor, Professor Leopoldo Bertossi for his support, advice and lots of good ideas throughout my thesis. His guidance and technical questions have been eyes opener for me on many occasions.

I also would like to thank Dr. Mostafa Milani whose thoughtful consideration, guidance, involvement and constructive criticism have been invaluable.

I would like to thank my parents Hayedeh and Ahmad, my sister Leila, and my brother Davoud for their unconditional love, and support.

Last but not least, I would like to give special thanks to Vanessa A Peterson, my caring friend, for her help with reviewing my thesis and emotional support. Finally, I wish to thank my friends Omid K, Omid M, Arash, Sania, Zahra, Kamyar, Amir, Reza, and Anis whom I have great memories with.

# Contents

**Abstract** i

**Acknowledgements** ii

**1 Introduction** 1

1.1 Overview . . . 1

1.2 $\mathsf{SChQA}^{\mathcal{S}}$ . . . 7

1.3 $\mathsf{MagicD}^{+}$ . . . 9

1.4 N-adaptive $\mathsf{SChQA}^{\mathcal{S}}$ . . . 11

1.5 JowsDatalog . . . 11

1.6 Contributions . . . 17

**2 Background** 18

2.1 Relational Databases . . . 18

2.2 Datalog . . . 20

2.3 Datalog$^{\pm}$ . . . 21

2.4 Program Classes . . . 25

2.4.1 Weakly-Acyclic Programs . . . 26

2.4.2 Jointly-Acyclic Programs . . . 28

2.4.3 Sticky Programs . . . 29

2.4.4 Weakly-Sticky Programs . . . . . . . . . . 31

2.4.5 Jointly-Weakly-Sticky Programs . . . . . . . . . . 32

2.5 SChQA$^{\mathcal{S}}$: A Chase-Based QA Algorithm for *JWS* Datalog$^{\pm}$ Programs 34

2.6 The MagicD$^{+}$ Rewriting Algorithm . . . . . . . . . . 38

**3 State of the Art 43**

3.1 ChaseFun . . . . . . . . . . 44

3.2 DEMo . . . . . . . . . . 44

3.3 Llunatic . . . . . . . . . . 44

3.4 PDQ . . . . . . . . . . 45

3.5 Benchmarking the Chase . . . . . . . . . . 45

**4 JowsDatalog 48**

4.1 An Overview of JowsDatalog . . . . . . . . . . 48

4.2 JDParser . . . . . . . . . . 49

4.3 Basic Approach . . . . . . . . . . 52

4.3.1 The Evaluator . . . . . . . . . . 52

4.3.2 The Fact-Generator . . . . . . . . . . 54

4.3.3 Basic Approach Workflow . . . . . . . . . . 55

4.4 In-DB Approach . . . . . . . . . . 56

4.4.1 Building the Database . . . . . . . . . . 57

4.4.2 In-DB's Evaluator . . . . . . . . . . 60

4.4.3 In-DB's Fact-Generator . . . . . . . . . . 61

4.4.4 In-DB Algorithm . . . . . . . . . . 62

4.5 MS-Rewriter . . . . . . . . . . 72

4.6 N-adaptive SChQA$^{\mathcal{S}}$ . . . . . . . . . . 74

5 Experimental Evaluation 76

5.1 Dataset and Programs . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

5.2 JWS Test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

5.2.1 In-DB with MS-Rewriter vs. In-DB without MS-Rewriter . . 78

5.3 Comparison Test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

5.3.1 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

6 Conclusions and Future Work 82

Bibliography 82

References 83

A A Weakly-Acyclic Program and its Queries 88

B A Jointly-Weakly-Sticky Program and its Queries 91

# List of Tables

3.1 Tested Tools Specification in ChaseBench . . . . . . . . . . . . . . . 46

3.2 The Chase Benchmark . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

5.1 *JWS* Test Results (1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

5.2 *JWS* Test Results (2) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

5.3 Comparison Test Results for JowsDatalog . . . . . . . . . . . . . . 80

# List of Figures


1.1 Querying Data Sources through an Ontology . . . . . . . . . . . . . . 2

1.2 An ER Diagram . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Extending the EDB through the Chase . . . . . . . . . . . . . . . . . 5

1.4 An Overview of JowsDatalog . . . . . . . . . . . . . . . . . . . . . . 12

2.1 DG for a Weakly-Acyclic Datalog$^{\pm}$ Program (1) . . . . . . . . . . . 27

2.2 DG for a Weakly-Acyclic Datalog$^{\pm}$ Program (2) . . . . . . . . . . . 27

2.3 The EDG Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.4 The DG of $\mathcal{P}$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.1 JowsDatalog Workflow . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.2 The Flowchart for *Fact-Generator* . . . . . . . . . . . . . . . . . . . . 54

4.3 The Flowchart for Storing the Database . . . . . . . . . . . . . . . . 57

4.4 The New Version of *Evaluator* . . . . . . . . . . . . . . . . . . . . . . 60

4.5 The New Version of the *Fact-Generator* . . . . . . . . . . . . . . . . . 62

4.6 The Flowchart of *In-DB* . . . . . . . . . . . . . . . . . . . . . . . . . 64

# Chapter 1

# Introduction

## 1.1 Overview

*Metadata* are data about data, and provide semantics about other data. Metadata are schema, data types, domains, and *integrity constraints* (ICs) in relational databases. ICs capture semantics of data [Borgida and Mylopoulos, 2004, Harel and Rumpe, 2004], and help to maintain the data quality, to decrease uncertainty by filtering out inadmissible instances, and support the ideas of consistent *query answering* (QA) and repairs in relational databases [Arenas et al., 1999, Bertossi, 2011].

An *entity-relationship* (ER) model represents an external, data-related reality, in terms of relationships among entities that are eventually stored in a database. The ER model is closer to the outside reality (OR) than the relational database. The ER model is an abstract, simplified description or representation of the OR, which leaves out irrelevant or contingent aspects and details. These ER models can encompass multiple semantic constraints. Without these constraints, there are too many possible ORs applicable that can conform to the ER model. This would make the model too ambiguous or uncertain. A relational model is produced from the ER model by using the languages of predicate logic and set theory. The relational ICs become part of the model, and are also utilized as semantic constraints. Some

of them come from semantic constraints of the original ER model. Although the ER model is usually forgotten after the relational database is created, the ER model could be used as metadata. The ER model can be utilized as a semantic layer that is used with the database, and becomes closer to OR, and what the user understands. This can be done by borrowing languages that have been designed for the semantic web initiative [Berners-Lee et al., 2001, Hitzler et al., 2009, Shadbolt et al., 2006].

Logical languages to express metadata can interact with the logical data model (database). Since the ER model is a diagrammatic model, it can be better reconstructed as a symbolic and logic-based ontology. In turn, *ontology* is a logical description of a set of concepts and their relationships [Chandrasekaran et al., 1999], and therefore, it becomes metadata as an explicit and formal ER model.

*Ontology-based data access* (OBDA) [Lenzerini, 2011] allows access through a conceptual layer that takes the form of an ontology, the underlying data that is usually stored in a relational database. OBDA is a new paradigm, based on the use of knowledge representation and reasoning techniques, for governing the resources (data, metadata, services, processes, etc.) of modern information systems.

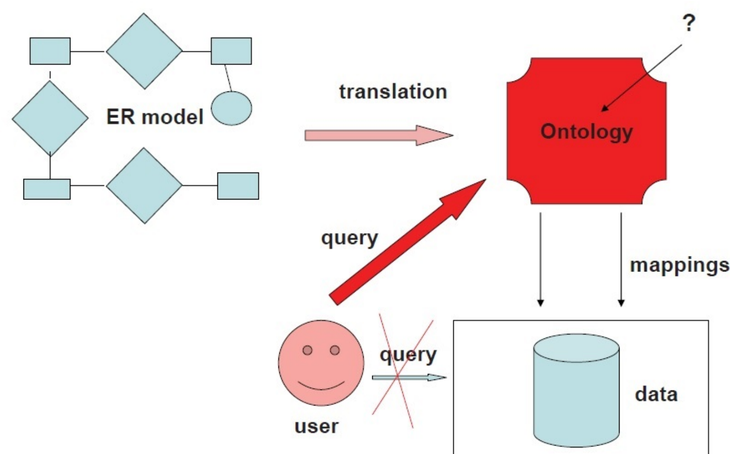


Figure 1.1: Querying data sources through an ontology

Queries can be expressed in terms of the ontology language, but are answered by eventually appealing to the extensional data underneath. Ontology queries

are internally translated into database queries by using the mappings between the ontology and the underlying data sources. This process is shown in Figure 1.1 and illustrated in Example 1.1. This ontology-based approach enables conceptually simpler and more flexible integration of data management and data sources with higher-level reasoning systems. These ontologies can be useful for interoperability and integration purposes [Maedche et al., 2003].

**Example 1.1.** Consider the following ER diagram in Figure 1.2.

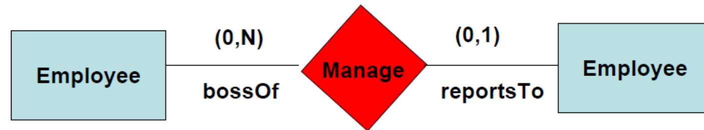


Figure 1.2: An ER diagram

The basic predicates for the ontology are as follows: $Employee(\cdot)$, $BossOf(\cdot,\cdot)$, and $ReportsTo(\cdot,\cdot)$. Capturing the (0, 1) constraint on the ER's *reportsTo*: "Every employee reports to at most one employee" is as follows:

$\forall\, x(Employee(x) \rightarrow \forall x \forall y_1 y_2\ ((Employee(y1) \wedge ReportsTo(x, y1)$

$\wedge Employee(y2) \wedge ReportsTo(x, y2)) \rightarrow y1 = y2).$ ■

Common languages of choice for representing ontologies are certain classes (or dialects) of *description logic* (DL) [Artale et al., 2009], Datalog [Abiteboul et al., 1995, Ceri et al., 1990], and more recently, Datalog$^\pm$ [Calì et al., 2009, Cali et al., 2012, Calì et al., 2012a]. The restricted syntax of DL makes automated reasoning feasible, and efficient. Datalog has been around for some years in the database community, and used for query and view definition in relational databases. Datalog is a declarative query language for relational databases that is based on the logic programming paradigm, and allows to define recursive views as opposed to relational algebra/calculus, and older versions of SQL [Abiteboul et al., 1995, Ceri et al., 1990]. A Datalog program is shown in Example 1.2 and more information is available in Section 2.2.

**Example 1.2.** A Datalog program $\mathcal{P}$:

$$path(X, Y) \leftarrow arc(X, Y). \quad (1.1)$$

$$path(X, Z) \leftarrow path(X, Y), Arc(Y, Z). \quad (1.2)$$

Here, $path(X, Y)$ and $arc(X, Y)$ are called atom. The left hand side of the arrow in (1.1) and (1.2) is called *head*. The right hand side in those equations is called *body*. Some of the predicates in $\mathcal{P}$ are *extensional*, i.e. they do not appear in rule heads, and the extensions for them (i.e. their sets of tuples) are given by a complete database instance $D$, which is called *the extensional database* (EDB). In this example, predicate $Arc$ is extensional (cf. Chapter 2).

■

Due to the limited expressive power of Datalog, an extension of Datalog named Datalog$^\pm$ with new kinds of rules and constraints was introduced [Calì et al., 2011, Cali et al., 2012]. Datalog$^\pm$ programs are expected to be both sufficiently expressive and computationally well-behaved in relation to *conjunctive query answering* (CQA). This work employs the expressive power of Datalog$^\pm$. Datalog$^\pm$ ontologies can represent ER models [Calí et al., 2012], semantic web languages/ontologies [Arenas et al., 2014, Calì et al., 2012a], UML with object classes [Calì et al., 2012b], etc. It also helps to represent the navigation in multidimensional data models for data quality assessment and cleaning [Milani et al., 2014].

Datalog$^\pm$ allows representing ontological axioms and ICs that cannot be expressed in Datalog. Datalog$^\pm$ tries to keep the good properties of Datalog, like declarative, clear logical semantics, efficiency, and tractable query answering. Rules in Datalog$^\pm$ admit existentially qualified variables ($\exists$-variables) in their head [Ceri et al., 1990] that can be seen as *tuple-generating dependencies* (*tgds*) [Abiteboul et al., 1995]. The "+" in Datalog$^\pm$ allows for those extensions, while the "−" in Datalog$^\pm$ refers to syntactic restrictions we impose on the rules and their (syntactic) interactions to achieve tractability. We should mention that Datalog$^\pm$

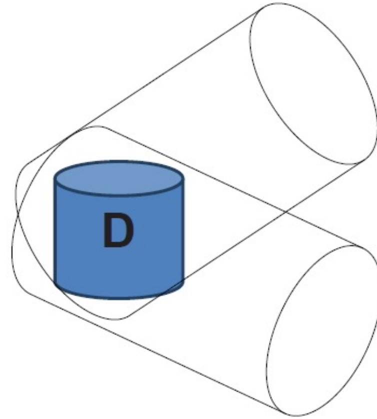


Figure 1.3: Generating new tuples for EDB predicates and full extensions for intensional predicates; depending on the kind of rules, possibly several extensions

is a subclass of Datalog$^+$ (or Datalog$^\exists$) [Baget et al., 2009, Calì et al., 2009, Leone et al., 2012, Krötzsch and Rudolph, 2011] whose programs do not have syntactic restrictions.

A Datalog$^\pm$ program, with the new kind of rules and classical ones, is combined with an EDB, $D$, that is considered to be *incomplete*, but this EDB is extended through the Datalog$^\pm$ program. The *chase* of the rules on the EDB generates an instance that both extends the EDB and represents the whole class of extensions. This process is known as the *chase procedure*. The extensions that are shown in figure 1.3 are databases that extend the EDB and satisfy the rules as classical logical formulas. It turns out that what is certain, as expressed by *conjunctive queries* (CQs), is what is true in the extension produced by the chase. The chase starts from the data in $D$, applies the rules in the ontology (mapping the body of a rule to the instance) generates a new tuple for the head, and creates new values (nulls) corresponding to existentially quantified variables in the rule's head. The example 1.3 informally illustrates this process and the notions involved.

**Example 1.3.** Consider a Datalog$^+$ program consisting of the set $\mathcal{P}$ of rules below and the EDB $D = \{person(John)\}$:

$$\exists Z\, father(Z, X) \leftarrow person(X). \quad (1.3)$$

$$person(X) \leftarrow father(X, Y). \quad (1.4)$$

The program's schema has a unary (i.e. one-argument) predicate $person$, with a position $person[1]$ (the argument in predicate $person$), and a binary predicate $father$, with positions $father[1]$ and $father[2]$. The initial instance $D$ makes the antecedent of rule (1.3) true, but not its head. So, a new tuple, $father(\zeta_1, John)$, is generated by the chase in which $\zeta_1$ is a labeled null value. Now, the body of rule (1.4) becomes true, generating a tuple $person(\zeta_1)$. Continuing in this way, the extension of $D$ produced by the chase includes the following tuples (among infinitely many others due to further rule enforcements):

$$\begin{aligned} chase(D, \mathcal{P}) = \{&father(\zeta_1, John), person(\zeta_1),\\ &father(\zeta_2, \zeta_1), person(\zeta_2),\\ &father(\zeta_3, \zeta_2), person(\zeta_3), ...\} \end{aligned}$$ ■

As it is shown in Example 1.3, the chase may create non-terminating loops and may not terminate for Datalog$^+$ programs, creating an infinite chase instance. The problem of query answering (QA) is about answering a CQ on the combination of $D$ and the program. QA is undecidable [Johnson and Klug, 1984] and cannot be done in polynomial-time (in data complexity) for Datalog$^+$ programs [Deutsch et al., 2008,Fagin et al., 2005,Leone et al., 2012]. The chase procedure for Datalog$^+$ programs may produce an infinite extension. However, in some cases, even with an infinite chase, imposing syntactic restrictions to Datalog$^+$ programs can guarantee the decidability and tractability of QA. For example, QA is decidable for *Sticky, Weakly-Sticky* (*WS*), and more recently *Jointly-Weakly-Sticky* (*JWS*) Datalog$^\pm$ programs.

The class of WS programs (cf. Section 2.4.4) is a generalization of *Sticky* Datalog$^\pm$ programs. The latter is a syntactic class of programs characterized by the restrictions on variables participating in a body join. Stickiness can be checked syntactically, through a two-step marking procedure on a set of *tgds* (cf.

Section 2.4.4). WS Datalog$^{\pm}$ extends *Sticky* Datalog$^{\pm}$ by also capturing the well-known class of Weakly-Acyclic (*WA*) programs (cf. Section 2.4.1) [Fagin et al., 2005].

There are two general approaches for QA over a Datalog$^{\pm}$ program [Calì et al., 2013,Gottlob et al., 2014]. One approach is bottom-up chasing, in which we obtain an instance satisfying *tgds* that is used for QA by expanding the rules. The other one is a query rewriting technique used in a way that obtains correct answers by evaluating the new query directly on the initial extensional data. This second approach has limited applicability with Datalog$^{\pm}$ programs.

*JWS* programs (cf. Section 2.4.5) extend both *Sticky* and *WS* Datalog$^{\pm}$ programs [Milani and Bertossi, 2016]. *JWS* is characterized by a marking procedure, the *existential dependency graph* (*EDG*) and the joint acyclicity [Krötzsch and Rudolph, 2011]. QA can be done in polynomial-time in data through $\mathsf{SChQA}^{\mathcal{S}}$, a chase-based, bottom-up QA algorithm [Milani and Bertossi, 2016] for *JWS* Datalog$^{\pm}$ programs. The design and implementation of this algorithm are the main contributions of this thesis.

## 1.2 $\mathsf{SChQA}^{\mathcal{S}}$

Designing an OBDA tool with which QA can be done in polynomial-time in data complexity for *JWS* Datalog$^{\pm}$ programs is an open area of research. In order to achieve this goal, the OBDA tool can use $\mathsf{SChQA}^{\mathcal{S}}$, a practical chase-based algorithm [Milani and Bertossi, 2016]. $\mathsf{SChQA}^{\mathcal{S}}$ is informally described in Example 1.4 and a deeper look into $\mathsf{SChQA}^{\mathcal{S}}$ can be found in Section 2.5. Before we proceed to the example, we need to introduce *homomorphism*, *freezing a null*, and *resumption* of the chase.

Given two ground atoms $A$ and $B$ (containing only constants and nulls) with the same predicate $P$ and a set $\Pi$ of positions (i.e. arguments) of $P$, $A$ is $\Pi$-*homomorphic* to $B$ if there is a homomorphism $h$, such that $h(A) = B$ and $h$ is

the identity on terms in positions in $\Pi$. We say $A$ is *homomorphic* to $B$ if $A$ is $\Pi$-homomorphic to $B$ with $\Pi = \emptyset$. Intuitively speaking, when we are finding a homomorphism between two atoms, a constant matches with itself or a null label and a null matches with another null or a constant (cf. Example 2.9).

*Freezing a null*, $\zeta_i$, is treating it as a constant (i.e. as John would be treated in Example 1.3. *Resumption* is *freezing* every null in a chase instance and continuing with the chase.

$\mathsf{SChQA}^{\mathcal{S}}$ algorithm gets a *JWS* program $\mathcal{P}$, an *EDB* $D$, a CQ $\mathcal{Q}$, possibly with free variables, and $\mathcal{S}^{\exists}(\mathcal{P})$ which is the set of predicate positions (cf. Section 2.3). $\mathsf{SChQA}^{\mathcal{S}}$ starts from $\mathcal{I} := D$, applies the rules in $\mathcal{P}$, and finds pairs of rule/assignments, $\sigma$ and $\theta$ with $\sigma \in \mathcal{P}$, that are applicable over $\mathcal{I}$, i.e.: (a) $\mathcal{I} \models (body(\sigma))[\theta]$; and (b) there is an assignment $\theta'$ that extends $\theta$, maps the $\exists$-variable of $\sigma$ into *fresh nulls*, and $\theta'(head(\sigma))$ is not $\mathcal{S}^{\exists}(\mathcal{P})$-homomorphic to any atom in $I$. The chase adds applicable pairs to the instance $I$. After all applicable pairs are processed, the algorithm *freezes* every null in instance $I$ and performs *resumption*. This algorithm has as many *resumptions* as $\exists$-variables in the query. At the end, the algorithm returns the set of answers to the query by obtaining it from $I$.

**Example 1.4.** Consider the program $\mathcal{P}$ with the EDB $D = \{p(a,b), u(b)\}$, $\mathcal{S}^{\exists}(\mathcal{P}) = \{u[1]\}$ (These positions syntactically depend on the program and we will discuss them and their relevance in Section 2.5.), and a CQ $\mathcal{Q}(X) : \exists Y\, p(X,Y)$.

$$\begin{array}{lccl} \sigma_1 : & \exists Z\, p(Y,Z) & \leftarrow & p(X,Y). \\ \sigma_2 : & t(Y) & \leftarrow & u(X), p(X,Y), p(Y,W). \end{array}$$

The algorithm starts from $I := D$ and tries to find applicable pairs. Then, $(\sigma_1, \theta_1)$ with $\theta_1 : X \mapsto a, Y \mapsto b$ is applicable; and $\mathsf{SChQA}^{\mathcal{S}}$ adds $p(b, \zeta_1)$ to $I$ in which $\zeta_1$ is a new null value for variable $Z$.

There are no more applicable pairs. Notice that $(\sigma_1, \theta_2)$ with $\theta_2 :\ X \mapsto b, Y \mapsto \zeta_1$ is not applicable since any $\theta_2' :\ \theta_2 \cup \{Z \mapsto \zeta_2\}$ generates $p(\zeta_1, \zeta_2)$ that is $\mathcal{S}^{\exists}(\mathcal{P})$-homomorphic to $p(a,b)$ and $p(b,\zeta_1)$ in $I$. $\mathsf{SChQA}^{\mathcal{S}}$ is resumed once since $\mathcal{Q}$ has one

$\exists$-variable, $Y$. This is done by freezing $\zeta_1$ and going back to apply the rules again. Now, $p(\zeta_1, \zeta_2)$ is not $\mathcal{S}^{\exists}(\mathcal{P})$-homomorphic anymore and $(\sigma_1, \theta_2)$ is applied, which results in $p(\zeta_1, \zeta_2)$. As a consequence, $(\sigma_2, \theta_3)$ with $\theta_3 :\ X \mapsto b, Y \mapsto \zeta_1, W \mapsto \zeta_2$ is applicable, which generates $t(\zeta_1)$. The algorithm stops with instance $I$:

$$I = D \cup \{p(b, \zeta_1), p(\zeta_1, \zeta_2), t(\zeta_1)\}$$

The query is posed to this instance and the algorithm returns $\mathcal{Q}(I) = \{a, b\}$. Since, we expect the answers that are logically entailed by $D \cup \mathcal{P}$, the query answers can not contain nulls. ■

## 1.3 MagicD$^+$

The QA algorithm can be optimized by using a *magic-sets* query rewriting technique, MagicD$^+$ [Milani and Bertossi, 2016], for *JWS* and its subclasses, *WS* and *Sticky* programs. MagicD$^+$ extends the classical magic-sets rewriting technique for Datalog$^\pm$ programs [Alviano et al., 2012]. It is explained in Section 2.6. The class of *WS* Datalog$^\pm$ programs is not closed under this rewriting technique. On the other hand, MagicD$^+$ is closed for *JWS* programs. MagicD$^+$ [Milani and Bertossi, 2016] takes a Datalog$^\pm$ program and a query, and rewrites it into a new Datalog$^\pm$ program. The new program can be the input for SChQA$^{\mathcal{S}}$. With the new program, SChQA$^{\mathcal{S}}$ avoids generating irrelevant facts. The MagicD$^+$ algorithm is informally described in Example 1.5. A comprehensive example of this algorithm, containing constants in the query, is available in Example 2.13.

**Example 1.5.** Consider the program $\mathcal{P}$ with the EDB $D = \{p(a, b), u(b)\}$, $\mathcal{S}^{\exists}(\mathcal{P}) = \{u[1]\}$, and the CQ $\mathcal{Q}(X) : \exists Y\, p(X, Y)$.

$$\sigma_1 : \quad \exists Z\, p(Y, Z) \ \leftarrow\ p(X, Y).$$

$$\sigma_2 : \quad t(Y) \ \leftarrow\ u(X), p(X, Y), p(Y, W).$$

$\mathsf{MagicD}^+$ starts from $\mathcal{Q}$ and generates adorned predicates by annotating predicates in $\mathcal{Q}$ with strings of $b$'s (stands for "bound") and $f$'s (stands for "free") in the positions that contain constants and variables, respectively. Therefore, we get $p^{ff}(X, Y)$.

The next step is for every newly generated adorned predicate $P^\alpha$, $\mathsf{MagicD}^+$ finds every rule $\sigma$ with the head predicate $P$ and it generates an adorned rule $\sigma'$ by replacing every body predicate in $\sigma$ with its adorned predicate and the head of $\sigma$ with $P^\alpha$. Here:

$$\sigma'_1: \qquad \exists Z\, p^{ff}(Y, Z) \ \leftarrow \ p^{ff}(X, Y).$$

The result of $\mathsf{MagicD}^+$ is a program with EDB, $D$, and $\mathcal{P}'$ with only one rule, $\sigma'_1$. Answers can be obtained by executing $\mathsf{SChQA}^{\mathcal{S}}$ over $\mathcal{P}'$.

$\mathsf{SChQA}^{\mathcal{S}}$ starts from $I := D$ and tries to find applicable pairs. Then, $(\sigma'_1, \theta_1)$ with $\theta_1 : X \mapsto a, Y \mapsto b$ is applicable; and $\mathsf{SChQA}^{\mathcal{S}}$ adds $p^{ff}(b, \zeta_1)$ to $I$ in which $\zeta_1$ is a new null value for variable $Z$.

There are no more applicable pairs. Notice that $(\sigma_1, \theta_2)$ with $\theta_2 : \ X \mapsto b, Y \mapsto \zeta_1$ is not applicable since any $\theta'_2 : \ \theta_2 \cup \{Z \mapsto \zeta_2\}$ generates $p(\zeta_1, \zeta_2)$ that is $\mathcal{S}^\exists(\mathcal{P}')$-homomorphic to $p^{ff}(a, b)$ and $p^{ff}(b, \zeta_1)$ in $I$. $\mathsf{SChQA}^{\mathcal{S}}$ is resumed once since $\mathcal{Q}'$ has one $\exists$-variable $Y$. This is done by freezing $\zeta_1$ and going back to apply the rules again. Now, $p^{ff}(\zeta_1, \zeta_2)$ is not $\mathcal{S}^\exists(\mathcal{P}')$-homomorphic anymore and $(\sigma_1, \theta_2)$ is applied which results in $p^{ff}(\zeta_1, \zeta_2)$.

The algorithm stops with instance $I$:

$$I = D \cup \{p^{ff}(b, \zeta_1), p^{ff}(\zeta_1, \zeta_2)\}$$

It returns $\mathcal{Q}'(I) = \{a, b\}$. Since, we expect the answers that are logically entailed by $D \cup \mathcal{P}$, the query answers can not contain nulls. ■

## 1.4 N-adaptive $\mathsf{SChQA}^{\mathcal{S}}$

As it is shown in Example 1.5, the chase is optimized based on the query, since the program has only one rule now. The limitation of $\mathsf{MagicD}^{+}$ is that this algorithm generates a program which is specifically designed to answer the given query. If we want to answer many queries in an application, we have to execute the chase (in general, not specifically $\mathsf{SChQA}^{\mathcal{S}}$) every time for each query. The chase is a time-consuming process and if we use a chase-based algorithm that is independent from the query, we can run the chase algorithm once and provide the answers to different queries by using the materialized instance $I$ produced by the chase.

$\mathsf{SChQA}^{\mathcal{S}}$ creates a chase instance that is dependent on the query only in terms of the number of $\exists$-variables in the query. If we give an upper bound $N$ for this value, then the result of $\mathsf{SChQA}^{\mathcal{S}}$ could be used for answering queries with at most $N$ $\exists$-variables. We call this method as *N-adaptive* $\mathsf{SChQA}^{\mathcal{S}}$ and this method is independent from the queries with at most $N$ $\exists$-variables.

In this thesis, we investigate the trade-off between using $\mathsf{MagicD}^{+}$ with $\mathsf{SChQA}^{\mathcal{S}}$ and *N-adaptive* $\mathsf{SChQA}^{\mathcal{S}}$. This optimization of the $\mathsf{SChQA}^{\mathcal{S}}$ is one of the contributions of this thesis.

## 1.5 JowsDatalog

The main objective of the work presented in this thesis is the design and implementation of an OBDA tool, JowsDatalog, in which $\mathsf{SChQA}^{\mathcal{S}}$ and $\mathsf{MagicD}^{+}$ [Milani and Bertossi, 2016] have been implemented. JowsDatalog is a Java implementation of $\mathsf{SChQA}^{\mathcal{S}}$ and $\mathsf{MagicD}^{+}$. In this section, we keep referring to Example 1.4. This tool gets the input, a Datalog$^{\pm}$ program, query(s), and the EDB, from the user, and translates the input into Java programming language objects. After that, JowsDatalog executes the $\mathsf{SChQA}^{\mathcal{S}}$and provides the answers to the queries and the materialized instance $I$. This process is shown in Figure 1.4.

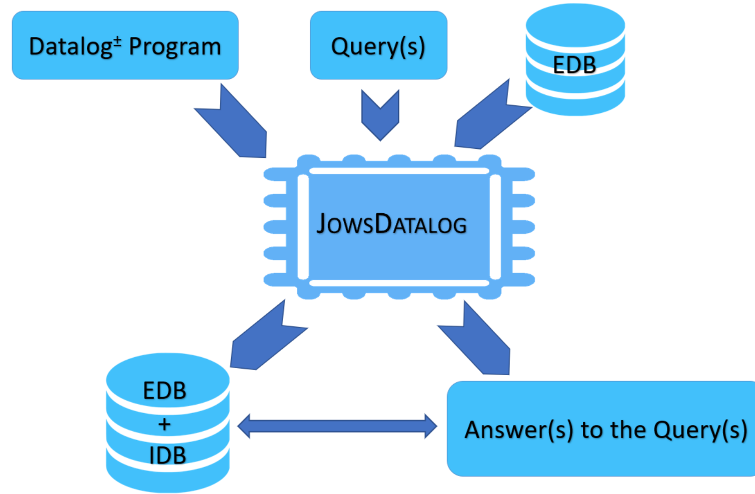


Figure 1.4: An Overview of JowsDatalog

MagicD$^+$ is implemented through the MS-Rewriter module. This module gets the program, $\mathcal{P}$, and the CQ, $\mathcal{Q}$, and applies MagicD$^+$ rewriting technique on them. This module generates a new program, $\mathcal{P}'$, which is optimized based on the query.

We consider two approaches for implementing the SChQA$^{\mathcal{S}}$:

1. *Basic* approach: A module in which a plain in-main-memory implementation of the chase procedure and query-answering (QA) is provided. We call it simply *Basic* in the following.

2. *In-DB* approach: A module that is using the power of a *relational database management system* (RDBMS) and the SQL language for implementing the chase procedure and QA. We call it simply *In-DB* in the following.

As we could not provide a stable implementation of *Basic*, we focus on *In-DB*. This approach uses the RDBMS's optimized algorithms (joins, indexing, etc.) for implementing SChQA$^{\mathcal{S}}$.

*In-DB* creates tables for every predicate in the program, and stores both the EDB and facts that are generated during the chase, inside the database. *Tgds*' bodies, in particular, their body joins, become queries to the database. We add an attribute, *ITR*, to each table to make sure that JowsDatalog processes a

rule-assignment only once. The idea of using iteration number, $ITR$, is related to the semi-naive evaluation of Datalog$^{\pm}$ programs and can be applied independent of the application of the *In-DB* or *Basic*. The idea is to use iteration number to be able to generate only relevant and fresh rule/assignments at each iteration. JowsDatalog processes *tgds* in parallel during the chase. The full description of this module is available in Section 4.4. We informally explain our method in Example 1.6.

**Example 1.6.** Consider the program $\mathcal{P}$ with the EDB $D = \{p(a,b), u(b)\}$, the CQ $\mathcal{Q}(X) : \exists Y\, p(X,Y)$, and $\mathcal{S}^{\exists}(\mathcal{P}) = \{u[1]\}$. Our implementation computes these positions from the program (cf. Section 4.2).

$$\sigma_1 : \quad \exists Z\, p(Y,Z) \;\leftarrow\; p(X,Y).$$

$$\sigma_2 : \quad t(Y) \;\leftarrow\; u(X), p(X,Y), p(Y,W).$$

JowsDatalog starts by creating tables for $p$, $u$, and $t$. We use $Xi$ for naming the attributes in the database. For example, if $p$ has two argument, we use $X0$ and $X1$ as the name for arguments in position p[1] and p[2], resp. Then, it inserts the EDB facts into the database. Each table has an additional field, $ITR$. The initial value of $ITR$ is zero for the EDB tuples. The database looks like as follows:

| **p** | | | |
|---|---|---|---|
| ID | X0 | X1 | ITR |
| 1 | a | b | 0 |

| **u** | | |
|---|---|---|
| ID | X0 | ITR |
| 1 | b | 0 |

| **t** | | |
|---|---|---|
| ID | X0 | ITR |

After that, *In-DB* starts executing the $\mathsf{SChQA}^{\mathcal{S}}$, and computing the chase. It initializes *nullSet*, a hash-set for storing fresh null labels, $itr = 0$, a variable for keeping the track of the iteration, and $firstItr = itr$, a memory variable which helps us during the resumption stage. Then, the module does a loop over *tgds*,

applies them, and increases *itr* by one at the end of each iteration. JowsDatalog processes *tgds* in parallel during each iteration. This loop terminates when no more tuples have been inserted into the database during the last iteration.

In this example, the algorithm starts the loop and processes *tgds* in parallel. The module generates a query for $\sigma_1$'s body:

```
SELECT p.X0, p.X1 from p
  where  (p.ITR = 0)
```

This query gets tuples for $p[1]$ and $p[2]$ from $p$ where their $ITR$ is zero. (The value zero corresponds to the value of *itr*). This condition enforce the participation of at least one new tuple in the join. The result-set is $\{X \mapsto a, Y \mapsto b\}$, creating tuple $p(b, \zeta_1)$, where $\zeta_1$ is a new null value for variable $Z$ in the head. Now, it checks if this tuple has any homomorphism in the database:

```
SELECT * from p where  X0 = 'b';
```

This query does not have any results, and $p(b, \zeta_1)$ is inserted into $P$, with $ITR = itr + 1$. Also, $\zeta_1$ is added to *nullSet*. $\sigma_2$ does not have any applicable rule-assignments since the result-set of the following query is empty:

```
SELECT u.X0, p1.X1, p2.X1 from
  u, p as p1, p as p2 where
   u.X0 = p1.X0 and
   p1.X1 = p2.X0 and
   ((u.ITR = 0) or
         (p1.ITR = 0) or
         (p2.ITR = 0));
```

This query basically joins $p$ and $u$ based on values in positions ($u[1]$, $p[1]$), and ($p[2]$,$p[1]$). Now, the first iteration over *tgds* is finished, and *itr* is incremented by one ($itr = 1$).

*In-DB* repeats the loop again. While $\sigma_2$ does not have any applicable rule-

assignments, the potential tuple for $\sigma_1$ is $p(\zeta_1, \zeta_2)$. This tuple has a homomorphism inside the database to $p(a, b)$, and therefore the module does not add any tuples to the database during this iteration. The second iteration over *tgds* is finished ($itr = 2$).

The last iteration has not updated the instance. The database looks as follows:

| **p** | | | |
|---|---|---|---|
| ID | X0 | X1 | ITR |
| 1 | a | b | 0 |
| 2 | b | $\zeta_1$ | 1 |

| **u** | | |
|---|---|---|
| ID | X0 | ITR |
| 1 | b | 0 |

| **t** | | |
|---|---|---|
| ID | X0 | ITR |

*In-DB* performs the *resumption* stage once since $\mathcal{Q}$ has one $\exists$-variable, $Y$. We freeze the null values in *nullSet* by removing them from the hash-set. Then, we apply *tgds* until we don't get any new tuples.

JowsDatalog starts applying *tgds*. Basically, the mappings for each *tgd* needs at least one tuple in which $ITR$ is equal to *itr* ($itr = 2$). We don't have such a tuple in the database. This causes the termination of *In-DB* with an incomplete instance. Notice that the instance used for QA in Example 1.4 contains more tuples.

To overcome this problem, we modify the *resumption* as follows: (a) we freeze every null in the instance by removing them from *nullSet*. (b) *In-DB* applies *tgds* once, in which the lower bound for $ITR$ is the value of $firstItr$ and the upper bound is *itr*. If the mapping generates new tuples, $ITR$ will be $itr + 1$. (c) *In-DB* updates the value for *itr* to $itr + 1$, and $firstItr$ to *itr*.

Therefore the new *resumption* for this example is as follows: (a) it clears the *nullSet*. (b) *In-DB* sets the condition for $ITR$ from $ITR = 0$ to $ITR = 2$. The

algorithm applies *tgds* one more time. The mapping for $\sigma_1$'s body is:

```
SELECT p.X0, p.X1 from p
           where   ((p.ITR <= 2 and p.ITR >= 0))
```

The mapping for $\sigma_1$ adds $p(\zeta_1, \zeta_2)$ to $I$ with $ITR = 3$ as $\zeta_1$ is treated as a constant. The new mapping for $\sigma_2$ is obtained by executing the following query:

```
SELECT u.X0, p1.X1, p2.X1 from
  u, p as p1, p as p2 where
            u.X0 = p1.X0 and
            p1.X1 = p2.X0 and
            ((u.ITR <= 0 and u.ITR >= 2) or
             (p1.ITR <= 0 and p1.ITR >= 2) or
             (p2.ITR <= 0 and p2.ITR >= 2));
```

This query does not have any result. (c) $itr = 3$ and $firstItr = 3$.

Now, we should repeat the loop one more time. *In-DB* starts it and $t(\zeta_1)$ is added to $I$ with $ITR = 4$. Then, it increases *itr* by one and performs another loop which does not update the instance and the algorithm stops. The answer to the query is determined by executing the following query:

```
SELECT p.X0 from p;
```

JowsDatalog removes all answers containing a null symbol from the answer set and returns $\{a, b\}$. Also, JowsDatalog stores the materialized instance $I$ inside PostgreSQL.

| p | | | |
|---|---|---|---|
| ID | X1 | X2 | ITR |
| 1 | a | b | 0 |
| 2 | b | $\zeta_1$ | 1 |
| 3 | $\zeta_1$ | $\zeta_2$ | 3 |

| u | | |
|---|---|---|
| ID | X1 | ITR |
| 1 | b | 0 |

| t | | |
|---|---|---|
| ID | X1 | ITR |
| 1 | $\zeta_1$ | 4 |

■

## 1.6 Contributions

The key features of our research are as follows:

1. We introduce JowsDatalog, an OBDA tool in which $\mathsf{SChQA}^{\mathcal{S}}$ and $\mathsf{MagicD}^{+}$ are implemented.

2. To the best of our knowledge, JowsDatalog is the only OBDA tool which can guarantee that the chase terminates, and QA can be done in polynomial-time in data complexity for *Sticky*, *WS*, and *JWS* Datalog$^{\pm}$ programs.

3. We design an implementation of the $\mathsf{SChQA}^{\mathcal{S}}$ that makes use of a database, *PostgreSQL*, and performs the chase by using a *relational database management system* (RDBMS). We also optimize some parts of the $\mathsf{SChQA}^{\mathcal{S}}$ which do not have a in detail explanation, like checking the homomorphism and finding applicable pairs.

4. We introduce *N-adaptive* $\mathsf{SChQA}^{\mathcal{S}}$ which makes $\mathsf{SChQA}^{\mathcal{S}}$ partially independent from the query.

# Chapter 2

# Background

## 2.1 Relational Databases

A relational database is a digital database, which is a location to store and retrieve data, based on the relational model of data [Codd, 1970]. In the context of relational databases, we assume that we have a relational schema $\mathcal{R}$ containing two disjoint domains: $\Gamma^C$, with possibly infinitely many *constants*, and $\Gamma^N$, of infinitely many *labeled nulls*. $\mathcal{R}$ also contains predicates of fixed finite arities. If $P$ is an $n$-ary predicate (i.e. with $n$ arguments) and $1 \leq i \leq n$, $P[i]$ denotes its $i$-th position. $\mathcal{R}$ gives rise to a language $\mathfrak{L}(\mathcal{R})$ of first-order (FO) predicate logic with equality (=). Variables are usually denoted with $x, y, z, ...$, and finite sequences thereof by $\bar{x}, ...$. Constants are usually denoted with $a, b, c, ...$; and nulls are denoted with $\zeta, \zeta_1, ...$. An *atom* is of the form $P(t_1, \ldots, t_n)$, with $P$ an $n$-ary predicate and $t_1, \ldots, t_n$ *terms*, i.e. constants, nulls, or variables. The atom is *ground* (aka. a tuple) if it contains no variables. An *instance* $I$ for schema $\mathcal{R}$ is a possibly infinite set of ground atoms. The *active domain* of an instance $I$, denoted $Adom(I)$, is the set of constants and nulls that appear in atoms of $I$. Instances can be used as interpretation structures for language $\mathfrak{L}(\mathcal{R})$. A *database instance* is a finite instance that contains no nulls.

A *homomorphism* from instance $I$ to instance $I'$ for the same schema is a

structure-preserving mapping, $Adom(I) \rightarrow Adom(I')$, such that: (a) $t \in \Gamma^C$ implies $h(t) = t$, and (b) for every ground atom $P(\bar{t}) \in I$, it holds $P(h(\bar{t})) \in I'$. $h(\bar{t})$ is defined component-wise.

A *conjunctive query* (CQ) is a FO formula, $\mathcal{Q}(\bar{x})$, of the form:

$$\exists \bar{y}\ (P_1(\bar{x}_1) \wedge \cdots \wedge P_n(\bar{x}_n)), \tag{2.1}$$

with (distinct) free variables $\bar{x} := (\bigcup \bar{x}_i) \smallsetminus \bar{y}$. If $\mathcal{Q}$ has $m$ (free) variables, for an instance $I$, $\bar{t} \in (\Gamma^C \cup \Gamma^N)^m$ is an *answer* to $\mathcal{Q}$ if $I \models \mathcal{Q}[\bar{t}]$, meaning that $Q[\bar{t}]$ becomes true in $I$ when the variables in $\bar{x}$ are component-wise replaced by the values in $\bar{t}$. $\mathcal{Q}(I)$ denotes the set of answers to $\mathcal{Q}$ in $I$. $\mathcal{Q}$ is a *boolean conjunctive query* (BCQ) when $\bar{x}$ is empty; and if it is *true* in $I$, $\mathcal{Q}(I) := \{\mathit{true}\}$. Otherwise, $\mathcal{Q}(I) = \emptyset$, and we say it is *false*.

A *tuple-generating dependency* (*tgd*), also called a *rule*, is an implicitly universally quantified sentence of $\mathfrak{L}(\mathcal{R})$ of the form:

$$\sigma\colon\ \ \exists \bar{y}\ P(\bar{x}, \bar{y}) \leftarrow\ P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n), \tag{2.2}$$

with $\bar{x} \subseteq \bigcup_i \bar{x}_i$, and the dots in the antecedent standing for conjunctions. The variables in $\bar{y}$ (that could be empty) are the *existential variables*. We assume $\bar{y} \cap \cup \bar{x}_i = \emptyset$. With $\mathit{head}(\sigma)$ and $\mathit{body}(\sigma)$ we denote the atom in the consequent and the set of atoms in the antecedent of $\sigma$, respectively. A *tgd* may contain constants from $\Gamma^C$ in predicate positions.

A *constraint* is an *equality-generating dependency* (*egd*) or a *negative constraint* (*nc*), which are also sentences of $\mathfrak{L}(\mathcal{R})$, respectively, of the forms:

$$x = x' \leftarrow P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n), \tag{2.3}$$

$$\bot \leftarrow P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n), \tag{2.4}$$

where $x, x' \in \bigcup_i \bar{x}_i$, and $\bot$ is a symbol that denotes the boolean constant (propositional variable) that is always false. Satisfaction of constraints by an instance is as in FO logic. *Tgds*, *egds*, and *ncs* are particular kinds of relational *integrity constraints* (ICs) [Abiteboul et al., 1995]. In particular, *egds* include *key constraints*

and *functional dependencies* (FDs). ICs also include *inclusion dependencies* (IDs) that are subsumed by *tgds*.

Relational databases work under the *closed world assumption* (CWA) [Abiteboul et al., 1995]: ground atoms not belonging to a database instance are assumed to be false. As a result of this form of data-completeness assumption, an IC is always true or false when checked for satisfaction on a database instance, never undetermined. If instances are allowed to be incomplete or open, i.e. with undetermined or missing ground atoms, ICs can be used, by enforcing them, to generate new tuples.

## 2.2 Datalog

Datalog is a declarative query language for relational databases that is based on the logic programming paradigm, and allows to define recursive views [Abiteboul et al., 1995, Ceri et al., 1990]. A Datalog program $\mathcal{P}$ for schema $\mathcal{R}$ is a finite set of non-existential rules, i.e. as in (2.2) above but without $\exists$-variables. Some of the predicates in $\mathcal{P}$ are *extensional*, i.e. they do not appear in rule heads, and the extensions for them (i.e. their sets of tuples) are given by a complete database instance $D$ (for the extensional subschema of $\mathcal{R}$), which is called *the extensional database* (EDB).[1] The other, *intentional*, predicates are defined by rules that have them in their heads. For Datalog programs, we may assume, without loss of generality, that intentional predicates appear only in rules, but not in the EDB.

The *minimal-model semantics* of a Datalog program with respect to an extensional database instance $D$ is given by a fixed-point semantics: the extensions of the intentional predicates are obtained by, starting from $D$, iteratively enforcing the rules and creating tuples for the intentional predicates, i.e. whenever a ground (or instantiated) rule body becomes true in the extension obtained so far, but not

[1] That is, the *closed-world assumption* (CWA) applies to the extensional atoms in $D$: If a ground atom for the extensional subschema is not explicitly a member of $D$, it is assumed to be *false*.

the head, the corresponding ground head atom is added to the extension under computation. If the set of initial ground atoms is finite, the process reaches a fixed-point after a number of steps that is polynomially bounded in the size of $D$.

A CQ as in (2.1) can be expressed as a Datalog rule of the form:

$$ans_{\mathcal{Q}}(\bar{x}) \ \leftarrow \ P_1(\bar{x}_1), ..., P_n(\bar{x}_n), \tag{2.5}$$

where $ans_{\mathcal{Q}}$ is an auxiliary, answer-collecting predicate. The answers to query $\mathcal{Q}$ form the extension of predicate $ans_{\mathcal{Q}}(\cdot)$ in the minimal model. When $\mathcal{Q}$ is a BCQ, $ans_{\mathcal{Q}}$ is a propositional atom; and $\mathcal{Q}$ is true in the undelying instance exactly when the atom $ans_{\mathcal{Q}}$ belongs to the minimal model of the program.

**Example 2.1.** A Datalog program $\mathcal{P}$ containing the rules

$$R(x,y) \ \leftarrow \ P(x,y),$$
$$R(x,z) \ \leftarrow \ P(x,y), R(y,z)$$

recursively defines, on top of the extensional relation $P$, the intentional predicate $R$ as the transitive closure of $P$. For $D = \{P(a,b), P(b,d)\}$ as extensional database, the extension of $R$ can be computed by iteratively adding tuples enforcing the program rules, which results in $\{R(a,b), R(b,d), R(a,d)\}$.

The CQ $\mathcal{Q}(x)\colon\ R(x,b) \wedge R(x,d)$ can be expressed by the rule:

$$ans_{\mathcal{Q}}(x) \ \leftarrow \ R(x,b), R(x,d)$$

The set of answer is the computed extension for $ans_{\mathcal{Q}}(x)$, namely $\{a\}$. ■

## 2.3 Datalog$^\pm$

Datalog$^\pm$ is an extension of Datalog. The "+" in Datalog$^\pm$ allows for $\exists$-variables. The "−" refers to the syntactic restrictions we impose on the rules and their syntactic interactions. We will refer to some of those restrictions in Section 2.4. Accordingly, until then we will consider Datalog$^+$ programs.

A Datalog$^+$ program may contain, in addition to (non-existential) Datalog rules, existential rules of the form (2.2), constraints of the forms (2.3) and (2.4), and a finite extensional database $D$ that may be incomplete and contains ground atoms for the extensional predicates, i.e. those that do not appear in rule heads, and possibly also for the intensional predicates, i.e. those appearing in rule heads. *We will usually denote with $\mathcal{P}$ the set of rules and constraints, and with $D$ the extensional database (EDB). Accordingly a program is of the form $\mathcal{P} \cup D$. When no possible confusion arises, we simply refer to $\mathcal{P}$ as the "program".* A program has an associated schema formed by the predicates in it. The set of positions (for the predicates) in a program $\mathcal{P}$ is denoted with $Pos(\mathcal{P})$. (We may safely assume all the predicates in an EDB for program $\mathcal{P}$ also appear in $\mathcal{P}$.)

The semantics of a Datalog$^+$ program $\mathcal{P} \cup D$ is model-theoretic, and given by the class $Mod(\mathcal{P}, D)$ of all instances $D'$ for the program's schema that extend $D$ and make $\mathcal{P}$ true. In particular, given an $n$-ary CQ $\mathcal{Q}(\bar{x})$, $\bar{t} \in (\Gamma^C \cup \Gamma^N)^n$ is an *answer* wrt. $\mathcal{P}$ and $D$ iff $D' \models \mathcal{Q}[\bar{t}]$ for every $D' \in Mod(\mathcal{P}, D)$. This is *certain answer* semantics that requests truth in *all* models. Without any restrictions on the program, and even for programs without constraints, *conjunctive query answering* (CQA) may be undecidable [Beeri and Vardi, 1981].

CQA appeals to all possible models of the program. However, the *chase procedure* [Maier et al., 1979] can be used to generate a single instance that represents the class $Mod(\mathcal{P}, D)$ for this purpose. We show it by means of an example containing only *tgds*.

**Example 2.2.** Consider a program $\mathcal{P}$ with the set of rules:

$$\sigma: \quad \exists z\, R(y, z) \leftarrow R(x, y).$$
$$\sigma': \quad S(x, y, z) \leftarrow R(x, y), R(y, z).$$

and an extensional instance $D = \{R(a, b)\}$, providing an incomplete extension for the program's schema. With the $I := D$, the pair $(\sigma, \theta_1)$, with (value) *assignment* (for variables) $\theta_1:\ x \mapsto a, y \mapsto b$, is *applicable*: $\theta_1(body(\sigma)) = \{R(a, b)\} \subseteq I$.

The chase *enforces* $\sigma$ by inserting a new tuple $R(b, \zeta_1)$ into $I$, with $\zeta_1$ a *fresh* null, resulting in instance $I = \{R(a, b), R(b, \zeta_1)\}$.

Now, $(\sigma', \theta_2)$, with $\theta_2 : \; x \mapsto a, y \mapsto b, z \mapsto \zeta_1$, is applicable in $I$, because $\theta_2(body(\sigma')) = \{R(a, b), R(b, \zeta_1)\} \subseteq I$. The chase adds $S(a, b, \zeta_1)$ into $I$. The chase continues, without stopping, creating an infinite instance, usually called *the chase* (instance):

$$\begin{aligned} chase(\mathcal{P}, D) = \{&R(a, b), R(b, \zeta_1),\\ &S(a, b, \zeta_1), R(\zeta_1, \zeta_2),\\ &R(\zeta_2, \zeta_3), S(b, \zeta_1, \zeta_2), \ldots\}. \end{aligned}$$

■

Depending on the programs and instances, the chase may be finite or infinite; and different orders of chase steps may result in different sequences and instances. However, it is possible to define a *canonical chase procedure* that determines a canonical sequence of chase steps, and consequently, a canonical chase instance [Calì et al., 2013]. In this work, we call it simply the chase in the following.

Given a program $\mathcal{P}$ and an EDB $D$, the chase (instance) is a *universal model* [Fagin et al., 2005]: For every $I \in Mod(\mathcal{P}, D)$, there is a homomorphism from the chase into $I$. For this reason, the (certain) answers to a CQ $\mathcal{Q}$ under $\mathcal{P}$ and $D$ can be computed by evaluating $\mathcal{Q}$ over the chase instance (discarding the answers containing nulls) [Fagin et al., 2005].

If the program $\mathcal{P} \cup D$ has *ncs*, they are expected to be satisfied by the chase. That is, the BCQ associated to the *nc* (2.4), i.e. $\mathcal{Q}_\eta : \; \exists \bar{x}_1 \cdots \bar{x}_n (P_1(\bar{x}_1) \wedge \cdots \wedge P_n(\bar{x}_n)$, obtained from the body of (2.4), must be false. If this is not the case, we say $\mathcal{P}$ is *inconsistent*. If $\mathcal{P}$ has *egds*, they are also expected to be satisfied by the chase. However, one can modify the chase in order to enforce the *egds*, which may be possible or not. In the latter case, we say the *chase fails*. It is possible to define a canonical chase that involves *egds* [Calì et al., 2013].

**Example 2.3.** Consider a program $\mathcal{P}$ with $D = \{R(a,b)\}$ with two rules and an *egd*:

$$\exists z, w\ S(y,z,w) \ \leftarrow\ R(x,y). \tag{2.6}$$

$$P(x,y) \ \leftarrow\ S(x,y,y). \tag{2.7}$$

$$y = z \ \leftarrow\ S(x,y,z). \tag{2.8}$$

The chase of $\mathcal{P}$ first applies (2.6) and results in $I = \{R(a,b), S(b,\zeta_1,\zeta_2)\}$. There are no more *tgd*/assignment applicable pairs. But, if we enforce the *egd* (2.8), equating $\zeta_1$ and $\zeta_2$, we obtain $I' = \{R(a,b), S(b,\zeta_1,\zeta_1)\}$. Now, (2.7) and $\theta' : x \mapsto b, y \mapsto \zeta_1$ are applicable, so we add $P(b,\zeta_1)$ to $I'$, therefore, $I' = \{R(a,b), S(b,\zeta_1,\zeta_1),\ P(b,\zeta_1)\}$. The chase terminates (no applicable *tgds* or *egds*), obtaining $chase(\mathcal{P}, D) = I'$. Notice that the program consisting only of (2.6) and (2.7) produces $I$ as the chase, which makes the BCQ $\exists x, y\ P(x,y)$ evaluate to *false*. With the program also including the *egd* (2.8) the answer is now *true*.

Now consider program $\mathcal{P}'$ that is $\mathcal{P}$ with the extra rule:

$$\exists z\ S(z,x,y) \leftarrow\ R(x,y). \tag{2.9}$$

which enforced on $I'$ results in $I'' = \{R(a,b), S(b,\zeta_1,\zeta_1), P(b,\zeta_1), S(\zeta_3,a,b)\}$. Now (2.8) is applied, which creates a chase failure as it tries to equate constants $a$ and $b$. This is the case where the set of *tgds* and the *egd* are mutually inconsistent. ■

Characterizations of computationally well-behaved classes of Datalog$^\pm$ programs usually do not consider any kind of *egds* and *ncs* in the program, but only the *tgds*. However, considering *ncs* is not complicated for these characterizations since they may have a trivial effect of QA or no effect at all. More precisely, if a program $\mathcal{P}$ consists of a set of *tgds* $\mathcal{P}^R$ and a set of *ncs* $\mathcal{P}^C$, then CQA amounts to deciding if $\mathcal{P}^R \cup \mathcal{P}^C \cup D \models \mathcal{Q}$, for which the following result holds.

**Proposition 2.1.** *[Cali et al., 2012, theo. 6.1]* $\mathcal{P}^R \cup \mathcal{P}^C \cup D \models \mathcal{Q}$ *iff (a)* $\mathcal{P}^R \cup D \models \mathcal{Q}$*, or (b) for some* $\eta \in \mathcal{P}^C$*,* $\mathcal{P}^R \cup D \models \mathcal{Q}_\eta$*, where* $\mathcal{Q}_\eta$ *is the BCQ*

*obtained as the existential closure of the body of $\eta$.*

Case (b) above holds exactly when $\mathcal{P} \cup D$ is inconsistent, and $\mathcal{Q}$ becomes trivially true. This shows that CQA evaluation under *ncs* can be reduced to the same problem without *ncs*, and the data complexity of CQA does not change. Furthermore, *ncs* may have an effect on CQA only if they are mutually inconsistent with the rest of the program, in which case every BCQ becomes trivially true. The presence of *egds* may have a more dramatic effect of QA, which can become undecidable, and the presence of *egds* may also change query answers, as in Example 2.3 (cf. [Bertossi and Milani, 2018, sec. 2] for a more detailed discussion). As a consequence, *we assume in the rest of this thesis that programs do not have* egds *or* ncs.

## 2.4 Program Classes

CQ answering over Datalog$^+$ programs with arbitrary sets of *tgds* is undecidable [Beeri and Vardi, 1981]. Actually, it is undecidable whether the chase terminates, even for a fixed instance [Beeri and Vardi, 1981, Deutsch et al., 2008]. Several sufficient conditions, syntactic [Deutsch et al., 2008, Fagin et al., 2005, Krötzsch and Rudolph, 2011, Marnette, 2009] or data-dependent [Meier et al., 2009], that guarantee chase termination have been identified. *Weak-Acyclicity* [Fagin et al., 2005] and *Joint-Acyclicity* [Krötzsch and Rudolph, 2011] are syntactic conditions that use a static analysis of a dependency graph of predicate positions in the program.

A non-terminating chase does not imply that CQ answering is undecidable. Several program classes are identified for which the chase may be infinite, but QA is still decidable. That is the case for *Linear*, *Guarded*, *Sticky*, *Weakly-Sticky Datalog*$^\pm$ [Calì et al., 2009, Calì et al., 2010, Calì et al., 2011, Cali et al., 2012], *shy Datalog*$^\exists$ [Leone et al., 2012], and *finite expansion sets* (fes), *finite unification sets*

(fus), *bounded-tree width sets* (bts) [Baget et al., 2009, Baget et al., 2011b, Baget et al., 2011a]. Each program class defines conditions on the program rules that lead to good computational properties for QA. In this section, we focus on *Sticky*, *WS*, and *JWS* Datalog$^\pm$ programs.

### 2.4.1 Weakly-Acyclic Programs

The *dependency graph* (DG) of a program $\mathcal{P}$ with schema $\mathcal{R}$ is a directed graph whose vertices are the positions (of predicates) in $\mathcal{P}$. The edges are defined as follows: for every $\sigma \in \mathcal{P}$, and every universally quantified variable ($\forall$-variable)[2] $x$ in $head(\sigma)$ in position $p$ in $body(\sigma)$ (among possibly other positions where $x$ appears in $body(\sigma)$): (a) for each occurrence of $x$ in position $p'$ in $head(\sigma)$, create an edge from $p$ to $p'$, (b) for each $\exists$-variable $z$ in position $p''$ in $head(\sigma)$, create a *special (dashed) edge* from $p$ to $p''$.

The *rank of a position* $p$ in the graph, denoted by $rank(p)$, is the maximum number of special edges over all (finite or infinite) paths ending at $p$. $\Pi_F(\mathcal{P})$ and $\Pi_\infty(\mathcal{P})$ denote the sets of finite-rank and infinite-rank positions in $\mathcal{P}$, resp. It is possible to prove that finite-rank positions are finite positions, i.e. they belong to $FinPos(\mathcal{P} \cup D)$ for every EDB $D$ [Fagin et al., 2005]. A program is *Weakly-Acyclic* (*WA*) if all of the positions belong to $\Pi_F(\mathcal{P})$ [Fagin et al., 2005].

**Example 2.4.** Let $\mathcal{P}$ be a program with rules:

$$R(x, z) \leftarrow R(x, y), R(y, z).$$
$$\exists z\, P(y, z) \leftarrow R(x, y).$$

The DG of $\mathcal{P}$ is shown in Figure 2.1. Positions $R[1]$, $R[2]$ and $P[1]$ have rank 0; and $P[2]$, rank 1. $\mathcal{P}$ is *WA* since all positions have finite-rank. There is a cycle in the DG of $\mathcal{P}$, but it does not involve any special edge.

Now, let $\mathcal{P}'$ be *WS* program with rules:

[2] Every variable that is not existentially quantified is implicitly universally quantified.

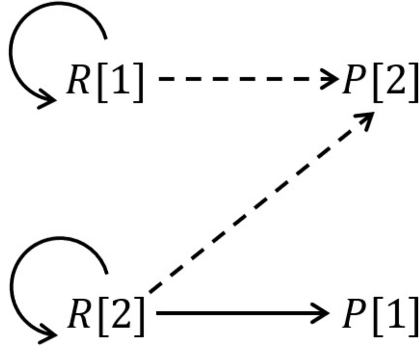


Figure 2.1: DG of $\mathcal{P}$

$$P(y,x) \leftarrow R(x,y).$$

$$\exists z\ R(y,z) \leftarrow P(x,y).$$

The DG of $\mathcal{P}'$ is shown in Figure 2.2. Positions $R[1]$ and $P[2]$ have rank 0. The

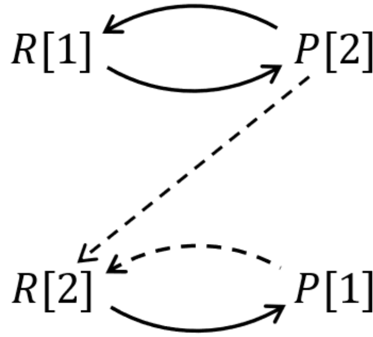


Figure 2.2: DG of $\mathcal{P}'$

program is not *WA* since $R[2]$ and $P[1]$ have infinite rank. $\mathcal{P}'$ is not *WS*, because its DG graph has a cycle with a special edge. ■

The problem of BCQ answering over a *WA* program is PTIME-complete in data complexity [Fagin et al., 2005]. This is because the chase for these programs stops in polynomial time in the size of the data [Fagin et al., 2005]. The same problem is 2EXPTIME-complete in combined complexity, i.e. in the size of both program rules and the data [Kolaitis et al., 2006].

### 2.4.2 Jointly-Acyclic Programs

The definition of the class of *Jointly-Acyclic* (*JA*) programs appeals to the *existential dependency graph* (EDG) of a program [Krötzsch and Rudolph, 2011] that we briefly review here.

Let $\mathcal{P}$ be a program with rules standardized apart, i.e. no variable appears in more than one rule. For a variable $x$ in rule $\sigma$, let $B_x$ and $H_x$ be the sets of positions where $x$ occurs in the body, resp. in the head, of $\sigma$. For an $\exists$-variable $z$, the set of *target positions* of $z$, denoted by $T_z$, is the smallest set of positions such that: (a) $H_z \subseteq T_z$, and (b) $H_x \subseteq T_z$ for every $\forall$-variable $x$ with $B_x \subseteq T_z$. Roughly speaking, $T_z$ is the set of positions where the invented (fresh) null values for the $\exists$-variable $z$ may appear during the chase.

The EDG of $\mathcal{P}$ is a directed graph with the $\exists$-variables of $\mathcal{P}$ as its nodes. There is an edge from $z \in \sigma$ to $z' \in \sigma'$ if there is a body variable $x$ in $\sigma'$ such that $B_x \subseteq T_z$. Intuitively, the edge shows that the values invented by $z$ may appear in the body of $\sigma'$, and cause (null) value invention for $z'$. Therefore, a cycle represents the possibility of inventing infinitely many null values for the $\exists$-variables in the cycle. A program is *Jointly-Acyclic* (*JA*) if its EDG is acyclic.

**Example 2.5.** Consider a program $\mathcal{P}$ with the following rules:

$$\exists z_1 \; R(y_1, z_1) \;\leftarrow\; P(x_1, y_1). \tag{2.10}$$

$$\exists z_2 \; P(y_2, z_2) \;\leftarrow\; R(x_2, y_2), U(x_2), U(y_2). \tag{2.11}$$

$$\exists z_3 \; S(x_3, y_3, z_3) \;\leftarrow\; P(x_3, y_3). \tag{2.12}$$

$B_{y_1} = \{P[2]\}$ and $H_{y_1} = \{R[1]\}$ are sets of positions where the variable $y_1$ appears in the body and, resp. the head of rule (2.10). Similarly, $B_{x_2} = \{R[1], U[1]\}$, $B_{y_2} = \{R[2], U[1]\}$, and $B_{y_3} = \{P[2]\}$. $T_{z_1} = \{R[2]\}$ and $T_{z_2} = \{P[2], R[1], S[2]\}$ are sets of target positions of $z_1$ and resp. $z_2$.

In the EDG of $\mathcal{P}$ in Figure 2.3 there is an edge from $z_2$ to $z_1$ since for the

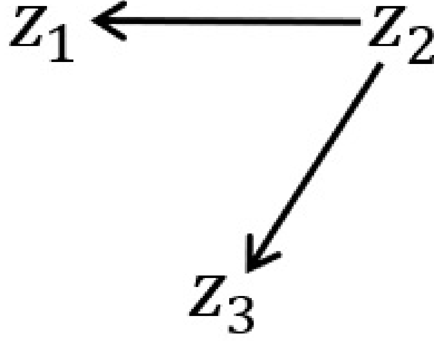


Figure 2.3: The EDG of $\mathcal{P}$

body variable $y_1$ in rule (2.10), where $z_1$ appears, $B_{y_1} \subseteq T_{z_2}$ holds, which means $y_1$ occurs only in the target positions of $z_2$. Similarly, there is an edge from $z_2$ to $z_3$ since for the body variable $y_3$ in rule (2.12), where $z_3$ appears, $B_{y_3} \subseteq T_{z_2}$ holds, which means $y_3$ occurs only in the target positions of $z_2$. There is no edge from $z_1$ to $z_2$ since, in rule (2.11), $B_{x_2} \nsubseteq T_{z_1}$ and $B_{y_2} \nsubseteq T_{z_1}$. For a similar reason, there is no self-loop for $z_2$. The graph is acyclic, and then, $\mathcal{P}$ is *JA*. ■

*JA* programs have polynomial size (finite) chase wrt. the size of the extensional data, and properly extend *WA* programs. BCQ answering over *JA* programs is PTIME-complete in data complexity, and 2EXPTIME-complete is combined complexity [Krötzsch and Rudolph, 2011].

### 2.4.3 Sticky Programs

They are characterized through a body variable *marking procedure* whose input is the set of rules of a program $\mathcal{P}$ (the extensional data do not participate in it). The procedure has two steps:

(a) *Preliminary step*: For each $\sigma \in \mathcal{P}$ and variable $x$ in *body*$(\sigma)$, if there is an atom $A$ in *head*$(\sigma)$ where $x$ does not appear, mark each occurrence of $x$ in *body*$(\sigma)$.

(b) *Propagation step*: For each $\sigma \in \mathcal{P}$, if a marked variable in *body*$(\sigma)$ appears in position $p$, then for every $\sigma' \in \mathcal{P}$ (including $\sigma$), mark the variables in

$body(\sigma')$ that appear in $head(\sigma')$ in position $p$.

We say that $\mathcal{P}$ is *Sticky* (or belongs to the program class *Sticky*) when, after applying the marking procedure, there is no rule with a marked variable appearing more than once in its body (i.e. not a join variable). Notice that a variable never appears both marked and unmarked in a same body.

**Example 2.6.** Consider programs $\mathcal{P}$ and $\mathcal{P}'$ below, both with EDB $D = \{R(a, b)\}$.

| $\mathcal{P}$ | $\mathcal{P}'$ |
|---|---|
| $\exists z\ R(y, z) \leftarrow R(x, y).$ | $\exists z\ R(y, z) \leftarrow R(x, y).$ |
| $S(x, y, z) \leftarrow R(x, y), R(y, z).$ | $S(x, y, z) \leftarrow R(x, y), R(y, z).$ |
| | $P(x, z) \leftarrow S(x, y, z).$ |

For program $\mathcal{P}$ on the left-hand side below, the first rule already shows marked variable $x$ (with a hat) after the preliminary step. The set of rules on the right-hand side is the final result of the marking procedure applied to $\mathcal{P}$:

| | |
|---|---|
| $\exists z\ R(y, z) \leftarrow R(\hat{x}, y).$ | $\exists z\ R(y, z) \leftarrow R(\hat{x}, \hat{y}).$ |
| $S(x, y, z) \leftarrow R(x, y), R(y, z).$ | $S(x, y, z) \leftarrow R(x, y), R(y, z).$ |

Variable $y$ in the first rule-body ends up marked after the propagation step: it appears in the rule head, in position $R[1]$, where marked variable $x$ appears in the same rule. Accordingly, $\mathcal{P}$ is Sticky: there is no marked variable that appears more than once in a rule body.

For program $\mathcal{P}'$, the result of the marking procedure is as follows:

$$\begin{aligned}
\exists z\ R(y, z) &\leftarrow\ R(\hat{x}, \hat{y}).\\
S(x, y, z) &\leftarrow\ R(x, \hat{y}), R(\hat{y}, z).\\
P(x, z) &\leftarrow\ S(x, \hat{y}, z).
\end{aligned}$$

$\mathcal{P}'$ is *not* Sticky: $y$ in the second rule body is marked and occurs twice in it (in $R[2]$ and $R[1]$). ■

The syntactic stickiness condition guarantees that QA can be done in PTIME in data complexity; and is EXPTIME-complete in combined complexity [Cali et al., 2012]. The chase of a *Sticky* program may not terminate, as shown in Example 2.6. However, a CQ can be answered by rewriting it into an FO query, actually a union of CQs, doing *backward-chaining* through the rules, and answering the FO query directly on the EDB. The rewriting depends only on the rules and the query; and the size of the rewriting is independent from the EDB [Cali et al., 2012, Gottlob et al., 2014].

### 2.4.4 Weakly-Sticky Programs

They form a syntactic class that extends those of *WA* and *Sticky* programs. Its characterization does not depend on the extensional data, and uses the notions of finite-rank and marked variable introduced in Sections 2.4.1 and 2.4.3, resp.: A set of rules $\mathcal{P}$ is *Weakly-Sticky* (*WS*) if, for every rule in it and every repeated variable in its body, the variable is either non-marked or appears in a position in $\Pi_F(\mathcal{P})$.

**Example 2.7.** Consider $\mathcal{P}$ with the set of rules:

$$\exists z\ R(y,z) \ \leftarrow R(x,y).$$
$$R(x,z) \ \leftarrow R(x,y), U(y), R(y,z).$$

for which $\Pi_F(\mathcal{P}) = \{U[1]\}$ and $\Pi_\infty(\mathcal{P}) = \{R[1], R[2]\}$. After applying the marking procedure, every body variable in $\mathcal{P}$ becomes marked. $\mathcal{P}$ is *WS* since the only repeated marked variable is $y$, in the second rule, and it appears in $U[1] \in \Pi_F(\mathcal{P})$.

Now, let $\mathcal{P}'$ be the program with the first rule of $\mathcal{P}$ and the second rule as

follows:

$$R(x,z) \leftarrow R(x,y), R(y,z).$$

Now, $\Pi_F(\mathcal{P}') = \emptyset$ and $\Pi_\infty(\mathcal{P}') = \{R[1], R[2]\}$. After applying the marking procedure, every body variable in $\mathcal{P}'$ is marked. $\mathcal{P}'$ is *not WS* since $y$ in the second rule is repeated, marked and appears in $R[1]$ and $R[2]$, both in $\Pi_\infty(\mathcal{P})$. ■

The *WS* condition guarantees tractability of CQ answering wrt. the size of the EDB [Calí et al., 2012]. Intuitively, *WS* generalizes the syntactic condition of sticky rules by preventing a repeated, marked variable from appearing only in infinite-rank positions, where it has no bound on the values it can take. However, appearing at least once in a finite position propagates boundedness to its other occurrences. For *WS* programs, QA can be done by rewriting a CQ into a union of CQs, and answering the resulting query over the EDB [Calí et al., 2012]. However, unlike *Sticky* programs, for *WS* programs the rewritten query and its size may depend on the EDB, but the latter is polynomially bounded by the size of the EDB.

### 2.4.5 Jointly-Weakly-Sticky Programs

Jointly-Weakly-Sticky (*JWS*) Datalog$^\pm$ programs form a syntactic class that extends those of *WS*, *WA*, and *Sticky* programs [Milani and Bertossi, 2016]. This class has a tractable QA problem, and is also closed under magic-set optimization, *MagicD*$^+$ [Milani and Bertossi, 2016]. The concept of *selection function* $\mathcal{S}$ is used to identify sets of finite positions of programs $\mathcal{P} \cup D$.

A selection function $\mathcal{S}$ associates every program $\mathcal{P} \cup D$ with a subset $\mathcal{S}(\mathcal{P} \cup D)$ of $\mathit{FinPos}(\mathcal{P} \cup D)$, which is the set of positions that take finitely many values in $\mathit{chase}(\mathcal{P}, D)$.

The selection functions $\mathcal{S}^\perp$, $\mathcal{S}^\top$ and $\mathcal{S}^{\mathit{rank}}$ are defined by: $\mathcal{S}^\perp(\mathcal{P} \cup D) := \emptyset$,

$\mathcal{S}^{\top}(\mathcal{P} \cup D) := \mathit{FinPos}(\mathcal{P} \cup D)$, and $\mathcal{S}^{rank}(\mathcal{P} \cup D) := \Pi_F(\mathcal{P})$, the set of finite-rank positions of $\mathcal{P}$, resp.

The class of *JWS* programs is based on the syntactic selection function $\mathcal{S}^{\exists}$, which in its turn appeals to the *existential dependency graph* of a program [Krötzsch and Rudolph, 2011], given in Section 2.4.2.

For a program $\mathcal{P}$, the set of *finite-existential positions*, denoted by $\Pi_F^{\exists}(\mathcal{P})$, is the set of positions that are not in the target set of any $\exists$-variable in a cycle in EDG($\mathcal{P}$).

Intuitively, a position in $\Pi_F^{\exists}(\mathcal{P})$ is not in the target of any $\exists$-variable that may be used to invent infinitely many nulls. Therefore, it specifies a subset of $\mathit{FinPos}(\mathcal{P} \cup D)$, for every EDB $D$ for $\mathcal{P}$. Accordingly, $\Pi_F^{\exists}(\mathcal{P})$ determines a syntactic selection function $\mathcal{S}^{\exists}$, defined by: $\mathcal{S}^{\exists}(\mathcal{P} \cup D) := \Pi_F^{\exists}(\mathcal{P})$. *For every program* $\mathcal{P} \cup D$*:* $\Pi_F(\mathcal{P}) \subseteq \Pi_F^{\exists}(\mathcal{P}) \subseteq \mathit{FinPos}(\mathcal{P} \cup D)$.

A program $\mathcal{P}$ is *JWS* if for every rule in $\mathcal{P}$ and every variable in its body that occurs more than once, the variable is either non-marked or appears in some positions in $\Pi_F^{\exists}(\mathcal{P})$. The class of *JWS* programs extend *WS*. This is illustrated in the program in Example 2.8.

**Example 2.8.** Let $\mathcal{P}$ be the program below.

$$\exists z\, R(y,z) \leftarrow R(x,y), U(y).$$

$$R(x,z) \leftarrow R(x,y), R(y,z).$$

Its DG is shown in Figure 2.4, and its EDG has only one node, $z$, without any edge. Then, $\Pi_F(\mathcal{P}) = \{U[1]\}$ and $\Pi_F^{\exists}(\mathcal{P}) = \{U[1], R[1], R[2]\}$. It is easy to check that the marking procedure leaves every body variable marked. As a consequence, $\mathcal{P}$ is not *WS*, because marked $y$ in the second rule body does not appear in $U[1]$. However, it is *JWS*, because all the body positions are finite-existential. ■

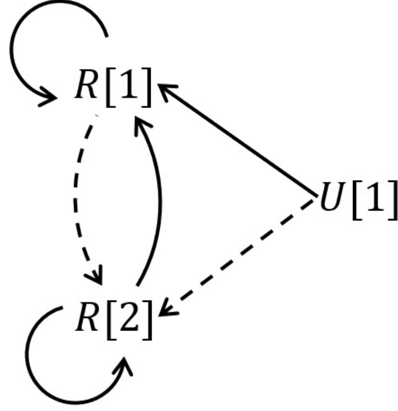


Figure 2.4: The DG of $\mathcal{P}$

## 2.5 $\mathsf{SChQA}^{\mathcal{S}}$: A Chase-Based QA Algorithm for *JWS* Datalog$^{\pm}$ Programs

$\mathsf{SChQA}^{\mathcal{S}}$ [Milani and Bertossi, 2016] is a bottom-up chase-based QA algorithm that is applicable to any Datalog$^{+}$ program and any computable selection function, and returns sound answers. However, completeness is guaranteed only when $\mathsf{SChQA}^{\mathcal{S}}$ is applied to *JWS* programs and its subclasses with a computable $\mathcal{S}^{\exists}$.

$\mathsf{SChQA}^{\mathcal{S}}$ uses the computable selection function $\mathcal{S}$, and calls as input a program $\mathcal{P}$ with schema $\mathcal{R}$, and a CQ $\mathcal{Q}$ over the same schema $\mathcal{R}$ and returns the set of answers $ans(\mathcal{Q}, \mathcal{P} \cup D)$. The algorithm relies on the notions of homomorphic atoms and freezing nulls.

**Definition 2.1.** (a) Given two ground atoms $A$ and $B$ (containing only constants and nulls) with the same predicate $P$ and a set $\Pi$ of positions of $P$, $A$ is $\Pi$-*homomorphic to* $B$ if there is a homomorphism $h$, such that $h(A) = B$ and $h$ is the identity on terms in positions in $\Pi$. We say $A$ is *homomorphic* to $B$ if $\Pi = \emptyset$. (b) *Freezing a null* $\zeta \in \Gamma^N$ in an atom $A$ means moving it to $\Gamma^C$, the set of constants. ■

**Example 2.9.** The atom $A = S(a, \zeta, \zeta)$ is homomorphic to $B = S(a, b, b)$ because $h(A) = B$ with $h = \{a \mapsto a, \zeta \mapsto b\}$. It is not $\Pi$-homomorphic to $B$ if $\Pi = \{s[2]\}$

since $h$ is not identity on $\zeta$ in $S[2]$. The atom $A$ is not homomorphic to $B' = S(a, b, c)$ since there is no homomorphism $h'$ such that $h'(A) = B'$.

Freezing the null value $\zeta$ in $A$ means treating $\zeta$ in $A(a, \zeta, \zeta)$ as a constant. This may have an impact on homomorphisms between atoms that contain the nulls. Now, with $\zeta$ frozen, $A(a, \zeta, \zeta)$ is not homomorphic to $B$ anymore. ■

**Definition 2.2.** Consider a Datalog$^+$ program $\mathcal{P}$ with the EDB $D$, and an instance $I$ of $\mathcal{P}$. A pair of rule/assignment $(\sigma, \theta)$, with $\sigma \in \mathcal{P}$, is $\mathcal{S}(\mathcal{P} \cup D)$-*applicable* over $I$ if: (a) $\theta(body(\sigma)) \subseteq I$; and (b) there is an assignment $\theta'$ that extends $\theta$, maps the $\exists$-variables of $\sigma$ into fresh nulls, and $\theta'(head(\sigma))$ is *not* $\mathcal{S}(\mathcal{P} \cup D)$-*homomorphic* to any atom in $I$. ■

$\mathcal{S}(\mathcal{P} \cup D)$ is a set of positions. When $\mathcal{P} \cup D$ is clear from the context, a pair of rule/assignment $(\sigma, \theta)$ becomes "applicable". Typically, $I$ will be a portion of the chase with $\mathcal{P} \cup D$. For an instance $I$ and a program $\mathcal{P}$, the applicable rule/assignment pairs are computed by first finding $\sigma \in \mathcal{P}$ for which $body(\sigma)$ is satisfied by $I$. That gives an assignment $\theta$ for which $\theta(body(\sigma)) \in I$. Then, we construct $\theta'$ as in Definition 2.2 and we iterate over atoms in $I$ and we check if they are $\mathcal{S}(\mathcal{P})$-homomorphic to $\theta'(head(\sigma))$.

The $\mathsf{SChQA}^{\mathcal{S}}$ algorithm executes the steps in Algorithm 2.1. In Step 4, *resumption* means freezing every null in every atom of the current instance $I$, and continuing with the algorithm steps. Here, freezing may cause new applicable pairs of rule/assignment because it changes homomorphic atoms. Notice that a pair of rule/assignment is applied only once in Step 2. Moreover, if there are more than one applicable pairs, then $\mathsf{SChQA}^{\mathcal{S}}$ chooses the pair as in the chase using the notion of level and then lexicographic order (cf. Section 2.3). Notice that the first three steps of the algorithm can be seen as the construction of a modified chase, which produces a finite instance and appeals to the notion of freezing nulls. Freezing nulls and chase resumption are first used in [Leone et al., 2012] for QA over

*shy* Datalog$^+$ programs. The algorithm is demonstrated through Example 2.10 and 2.11.

---

**Algorithm 2.1** The $\mathsf{SChQA}^{\mathcal{S}}$ algorithm [Milani and Bertossi, 2016]

**Inputs:** A selection function $\mathcal{S}$, a Datalog$^+$ program $\mathcal{P}$, an EDB $D$, and a CQ $\mathcal{Q}$ over $\mathcal{P}$, possibly with free variables

**Output:** $ans(\mathcal{Q}, \mathcal{P} \cup D)$

**Step 1:** Initialize instance $I$ with $D$

**Step 2:** Choose an applicable rule/assignment pair $(\sigma, \theta)$ over $I$, add $\theta'(head(\sigma))$ to $I$ (c.f. Definition 2.2)

**Step 3:** Iteratively apply Step 2 until all applicable pairs are applied

**Step 4:** Resume Step 2, i.e. freeze nulls in $I$, and continue with Steps 2-3. Repeat resumption $M_{\mathcal{Q}}$ times where $M_{\mathcal{Q}}$ is the number of $\exists$-variables in $\mathcal{Q}$

**Step 5:** Return the tuples in $\mathcal{Q}(I)$ that do not have any null values, frozen or not

---

In this thesis, the program is *JWS* and the selection function $\mathcal{S}$ is $\mathcal{S}^{\exists}$.

**Example 2.10.** Consider a program $\mathcal{P}$ with the EDB $D = \{S(a,b,c), V(b), U(c)\}$, and a CQ $\mathcal{Q}(x) : \exists y\ P(x,y)$. The hat signs show the marked variables according to the marking procedure explained in Section 2.4.3.

$$\sigma_1: \quad \exists w\ S(y,z,w) \ \leftarrow\ S(\hat{x},\hat{y},\hat{z}).$$

$$\sigma_2: \quad \exists y,z\ S(x,y,z) \ \leftarrow\ U(\hat{x}).$$

$$\sigma_3: \quad P(y,z) \ \leftarrow\ S(\hat{x},y,z), V(\hat{x}), S(y,z,\hat{w}).$$

The set of finite rank positions is $\mathcal{S}^{\exists}(\mathcal{P}) = \Pi_F(\mathcal{P}) = \{V[1], U[1]\}$. The program is *WS*. Specifically in $\sigma_3$, $x$ occurs in $V[1]$ which is in $\mathcal{S}^{\exists}(\mathcal{P})$ and $y$ and $z$ are not marked.

The algorithm starts from $I := D$. At Step 2, $(\sigma_1, \theta_1)$ with $\theta_1 : x \mapsto a, y \mapsto b, z \mapsto c$ are applicable; and $\mathsf{SChQA}^{\mathcal{S}}$ adds $S(b,c,\zeta_1)$ to $I$. Next, applicable pair $(\sigma_2, \theta_2)$ with $\theta_2 : x \mapsto c$ is applied and it adds $S(c,\zeta_2,\zeta_3)$ to $I$.

There is no more applicable pairs and we continue with Step 4. Notice that $(\sigma_1, \theta_3)$ with $\theta_3 : x \mapsto b, y \mapsto c, y \mapsto \zeta_1$ is not applicable since any $\theta_3' : \theta_3 \cup \{w \mapsto \zeta_4\}$ generates $S(c, \zeta_1, \zeta_4)$ that is $\mathcal{S}^{\exists}(\mathcal{P})$-homomorphic to $S(c, \zeta_2, \zeta_3)$ in $I$. $\mathsf{SChQA}^{\mathcal{S}}$ is resumed once since $\mathcal{Q}$ has one $\exists$-variable $y$. This is done by freezing $\zeta_1, \zeta_2, \zeta_3$ and returning to Step 2. Now, $S(c, \zeta_1, \zeta_4)$ and $S(c, \zeta_2, \zeta_3)$ are not $\mathcal{S}^{\exists}(\mathcal{P})$-homomorphic anymore and $(\sigma_1, \theta_3)$ is applied which results in $S(c, \zeta_1, \zeta_4)$. As a consequence, $(\sigma_3, \theta_4)$ with $\theta_4 : x \mapsto b, y \mapsto c, z \mapsto \zeta_1, w \mapsto \zeta_4$ is applicable, which generates $P(c, \zeta_1)$. The instance $I$ in Step 5 is:
$I = D \cup \{S(b, c, \zeta_1), S(c, \zeta_2, \zeta_3), S(c, \zeta_1, \zeta_4), P(c, \zeta_1), S(\zeta_2, \zeta_3, \zeta_5), S(\zeta_1, \zeta_4, \zeta_6)\}$,
and the algorithm returns $\mathcal{Q}(I) = \{c\}$.

The free CQ $\mathcal{Q}'(y) : \exists x\ P(x, y)$ (that asks $y$ values rather than the values $x$ in $\mathcal{Q}$) can be answered on the same instance $I$ since it also has one $\exists$-variable. However, the returned set of answers to $\mathcal{Q}'$ is $\emptyset$ since $\mathcal{Q}'(I) = \{\zeta_1\}$ and the query answers does not contain nulls. ■

**Example 2.11.** Consider program $\mathcal{P}$ with rules below, the EDB:
$D = \{R(a, b), S(a, c), M(b), T(a, b)\}$,
and the free CQ $\mathcal{Q}(x) : \exists y, z\ P(x, y, z)$. The hat signs show the marked variables according to the marking procedure explained in Section 2.4.3.

$$\sigma_1 : \quad \exists Z\ R(Y, Z) \leftarrow T(\hat{X}, \hat{Y}).$$

$$\sigma_2 : \quad \exists Z\ T(Y, Z) \leftarrow R(\hat{X},\ \hat{Y}),\ S(\hat{X},\ \hat{W}),\ M(\hat{Y}).$$

$$\sigma_3 : \quad \exists Z\ P(X,\ Y,\ Z) \leftarrow T(X,\ Y).$$

The set of finite rank positions is:
$\mathcal{S}^{\exists}(\mathcal{P}) = \Pi_F(\mathcal{P}) = \{S[1], S[2], M[1], P[1], P[2], P[3]\}$ , and the program is in *WS*. Specifically, in $\sigma_2$, variables $X$ and $Y$ are marked and repeated twice, which shows that the program is not sticky. $M[1]$ and S[1] belongs to the finite-rank and we can conclude that the program is *WS*.

The algorithm starts from $I := D$. The applicable pairs are listed below, and

SChQA$^{\mathcal{S}}$ adds them to $I$ :

$(\sigma_1, \theta_1)$ with $\theta_1 : X \mapsto a, Y \mapsto b$; add $R(b, \zeta_1)$ to $I$.

$(\sigma_2, \theta_2)$ with $\theta_2 : X \mapsto a, Y \mapsto b, W \mapsto c$; add $T(b, \zeta_2)$ to $I$.

$(\sigma_3, \theta_3)$ with $\theta_3 : X \mapsto a, Y \mapsto b$; add $P(a, b, \zeta_3)$ to $I$.

$(\sigma_3, \theta_4)$ with $\theta_4 : X \mapsto b, Y \mapsto \zeta_2$; add $P(b, \zeta_2, \zeta_4)$ to $I$.

SChQA$^{\mathcal{S}}$ does not freeze $\zeta_1$, $\zeta_2$, $\zeta_3$, and $\zeta_4$ since they are not in $\mathcal{S}^{\exists}(\mathcal{P})$. Notice that $(\sigma_1, \theta_5)$ with $\theta_5 : X \mapsto b, Y \mapsto \zeta_2$ is not applicable since any $\theta'_5 :\ \theta_5 \cup \{z \mapsto \zeta_5\}$ generates $R(\zeta_2, \zeta_5)$ that is $\mathcal{S}^{\exists}(\mathcal{P})$-homomorphic to $R(a, b)$ (and also $R(b, \zeta_1)$) in $I$.

Now, the algorithm freezes all nulls, $\zeta_1, \zeta_2, \zeta_3, \zeta_4$, and SChQA$^{\mathcal{S}}$ is resumed twice since $\mathcal{Q}$ has two $\exists$-variable, $X, Y$. During the second resumption, $(\sigma_1, \theta_5)$ with $\theta_5 : X \mapsto b, Y \mapsto \zeta_2$ becomes applicable as $R(\zeta_2, \zeta_5)$ and $R(a, b)$ (or also $R(b, \zeta_1)$) are not $\mathcal{S}^{\exists}(\mathcal{P})$-homomorphic anymore, and $(\sigma_1, \theta_5)$ is applied which results in $R(\zeta_2, \zeta_5)$. There aren't any new applicable pairs, and the algorithm freezes $\theta_5$.

During the final resumption, nothing is added to $I$ as there are not any new applicable pairs. The instance $I$ is:

$$I = D \cup \{R(b, \zeta_1), R(\zeta_2, \zeta_5), T(b, \zeta_2), P(a, b, \zeta_3), P(b, \zeta_2, \zeta_4)\},$$

and the algorithm returns $\mathcal{Q}(I) = \{a, b\}$. ■

## 2.6 The MagicD$^+$ Rewriting Algorithm

Magic-sets is a general technique for rewriting logical rules so that they may be implemented bottom-up in a way that avoids the generation of irrelevant facts [Beeri and Ramakrishnan, 1991, Ceri et al., 1990]. The advantage of such a rewriting technique is that, by working bottom-up, we can make use of the structure of the query and the data values in it, optimizing the data generation process. In this section, we review the MagicD$^+$ rewriting for Datalog$^+$ programs [Milani and Bertossi, 2016]. We should mention that in contradiction to *JWS* programs, *WS*

programs are not closed under MagicD$^+$.

MagicD$^+$ [Milani and Bertossi, 2016] takes a Datalog$^+$ program and rewrites it, using a given query, into a new Datalog$^+$ program. It has two changes regarding the technique in [Ceri et al., 1990] in order to: (a) works with ∃-variables in *tgd*s, and (b) considers the extensional data of the predicates that also have intensional data defined by the rules. For (a), the solution proposed in [Alviano et al., 2012] is applied. As (b) is specifically relevant for Datalog$^+$ programs that allow predicates with both extensional and intentional data, MagicD$^+$ is proposed.

MagicD$^+$ relies on the *adornments*, a convenient way for representing binding information for intentional predicates [Ceri et al., 1990]. Binding information can be propagated in rule bodies according to a *side-way information passing strategy* [Beeri and Ramakrishnan, 1991]. The MagicD$^+$ algorithm is explained through Example 2.13.

**Definition 2.3.** Let $P$ be a predicate of arity $k$ in a program $\mathcal{P}$. An adornment for $P$ is a string $\alpha = \alpha_1, ..., \alpha_k$ over the alphabet $\{b, f\}$. The $i$-th position of $P$ is considered *bound* if $\alpha_i = b$, or *free* if $\alpha_i = f$. For an atom $A = P(a_1, ..., a_k)$ and an adornment $\alpha$ for $P$, the magic atom of $A$ wrt. $\alpha$ is the atom $mg_P^{\alpha}(\bar{t})$, where $mg_P^{\alpha}$ is a predicate not in $\mathcal{P}$, and $\bar{t}$ contains all the terms in $a_1...a_k$ that correspond to bound positions according to $\alpha$. ■

**Example 2.12.** "*bfb*" is a possible adornment for ternary predicate $S$, and $mg_S^{bfb}(x, z)$ is the magic atom of $S(x, y, z)$ wrt. "*bfb*". ■

**Definition 2.4.** Let $\sigma$ be a *tgd* and $\alpha$ be an adornment for the predicate of $P$ in $head(\sigma)$. A *side-way information passing strategy* (sips) for $\sigma$ wrt. $\alpha$ is a pair $(\prec^{\alpha}_{\sigma}, f^{\alpha}_{\sigma})$, where:

- $\prec^{\alpha}_{\sigma}$ is a strict partial order over the set of atoms in $\sigma$, such that if $A = head(\sigma)$ and $B \in body(\sigma)$, then $B \prec^{\alpha}_{\sigma} A$.

- $f^\alpha_\sigma$ is a function assigning to each atom $A$ in $\sigma$, a subset of the variables in $A$ that are bound after processing $A$. $f^\alpha_\sigma$ must guarantee that if $A = head(\sigma)$, then $f^\alpha_\sigma(A)$ contains only all variables in $head(\sigma)$ that correspond to the bound arguments of $\alpha$. ■

The MagicD$^+$ rewriting technique takes a Datalog$^+$ program $\mathcal{P}$ and a CQ $\mathcal{Q}$ of schema $\mathcal{R}$, and returns a program $\mathcal{P}_m$ and a CQ $\mathcal{Q}_m$ of schema $\mathcal{R}_m$, such that $ans_{\mathcal{Q}}(\mathcal{Q}, \mathcal{P}) = ans_{\mathcal{Q}_m}(\mathcal{Q}_m, \mathcal{P}_m)$. We explain the algorithm by using the program in Example 2.13.

**Example 2.13.** Let $\mathcal{P}$ be a program with $D = \{U(b), R(a, b)\}$ and the rules,

$$P(x, z) \ \leftarrow \ R(x, y), R(y, z). \tag{2.13}$$

$$\exists z \ R(y, z) \leftarrow U(y), R(x, y). \tag{2.14}$$

and consider CQ $\mathcal{Q} : \exists x \ P(a, x)$ imposed on $\mathcal{P}$. ■

MagicD$^+$ has the following steps:

**1. Generation of adorned rules:** MagicD$^+$ starts from $\mathcal{Q}$ and generates adorned predicates by annotating predicates in $\mathcal{Q}$ with strings of $b$'s and $f$'s in the positions that contain constants and variables resp. For every newly generated adorned predicate $P^\alpha$, MagicD$^+$ finds every rule $\sigma$ with the head predicate $P$ and it generates an adorned rule $\sigma'$ as follows and adds it to $\mathcal{P}_m$. According to a predetermined sips, MagicD$^+$ replaces every body atom in $\sigma$ with its adorned atom and the head of $\sigma$ with $P^\alpha$. The adornment of the body atoms is obtained from the sips and its function $f^\alpha_\sigma$. This possibly generates new adorned predicates for which the MagicD$^+$ goes through this step again.

**Example 2.14.** (ex. 2.13 cont.) $P^{bf}$ is the new adorned predicate obtained from $\mathcal{Q}$. MagicD$^+$ considers $P^{bf}$ and (2.13). It generates the rule,

$$P^{bf}(x,z) \leftarrow R^{bf}(x,y), R^{bf}(y,z). \tag{2.15}$$

and adds it to $\mathcal{P}_m$. In this rule, $R^{bf}$ is a new adorned predicate. MagicD$^+$ generates the adorned rule,

$$\exists z\ R^{bf}(y,z) \leftarrow U(y), R^{fb}(x,y). \tag{2.16}$$

and adds it to $\mathcal{P}_m$. Here, (2.14) is not adorned wrt. $R^{fb}$, because this bounds the position $R[2]$ that holds the $\exists$-variable $z$. The following are the result of adorned rules:

$$P^{bf}(x,z) \leftarrow R^{bf}(x,y), R^{bf}(y,z). \tag{2.17}$$

$$\exists z\ R^{bf}(y,z) \leftarrow U(y), R^{fb}(x,y). \tag{2.18}$$

■

**2. Adding magic atoms and magic rules:** Let $\sigma$ be an adorned rule in $\mathcal{P}_m$ with the head predicate $P^{\alpha}$. MagicD$^+$ adds magic atom of $head(\sigma)$ (cf. Definition 2.3) to the body of $\sigma$. Additionally, it generates magic rules as follows. For every occurrence of an adorned predicate $P^{\alpha}$ in $\sigma$, it constructs a magic rule $\sigma'$ that defines $mg_P^{\alpha}$ (a magic predicate might have more than one definition). It is assumed that the atoms in $\sigma'$ are ordered according to the partial order in the sips of $\sigma$ and $\alpha$. If the occurrence of $P^{\alpha}$ is in atom $A$ and there are $A_1, ..., A_n$ on the left hand side of $A$ in $\sigma$, the body of $\sigma'$ contains $A_1, ..., A_n$ and the magic atom of $A$ in the head. We also create a seed for the magic predicates, in the form of a fact, obtained from the query.

**Example 2.15.** (ex. 2.14 cont.) Adding the magic atoms to the adorned rules, the following rules would be generated:

$$P^{bf}(x,z) \leftarrow mg_P^{bf}(x), R^{bf}(x,y), R^{bf}(y,z). \tag{2.19}$$

$$\exists z\ R^{bf}(y,z) \leftarrow mg_R^{bf}(y), U(y), R^{fb}(x,y). \tag{2.20}$$

The following magic rules define the magic predicates:

$$mg_R^{bf}(x) \leftarrow mg_P^{bf}(x). \quad (2.21)$$

$$mg_R^{bf}(y) \leftarrow mg_R^{bf}(x), R^{bf}(x, y). \quad (2.22)$$

■

**3. Adding rules to load extensional data:** This step applies only if $\mathcal{P}$ has intentional predicates with extensional data in $D$. The MagicD$^+$ algorithm adds rules to load the data from $D$ when such a predicate gets adorned.

**Example 2.16.** (ex. 2.15 cont.) In Example 2.14, $R$ is an intentional predicates that is adorned and has extensional data $R(a, b)$. MagicD$^+$ adds the following rules to load its extensional data for $R^{bf}$ and $R^{fb}$:

$$R^{bf}(x) \leftarrow mg_R^{bf}(x), R(x, y). \quad (2.23)$$

$$R^{fb}(x) \leftarrow mg_R^{fb}(x), R(x, y). \quad (2.24)$$

■

# Chapter 3

# State of the Art

We want to review OBDA tools that can deal with Datalog$^{\pm}$ programs, and their chase procedure is integrated with an RDBMS. The interesting research that we found is [Benedikt et al., 2017] which provides a benchmark, ChaseBench, by running a variety range of experiments over different tools, ChaseFun [Bonifati et al., 2016], DEMo [Pichler and Savenkov, 2009], Llunatic [Geerts et al., 2013], and PDQ [Benedikt et al., 2015]. The primary reason that some of these tools are proposed is solving the problem of *data exchange* (DE).

DE [Fagin et al., 2005, Geerts et al., 2014b, Benedikt et al., 2017] is the process of materializing an instance of a target schema by translating data, coming from one or more relational sources, and satisfying two sets of dependencies: (a) $\sum_{st}$, source to target ($s$-$t$) *tgds* where we deal with a new form of *tgds*, in which we can have conjunction in the head of the rule, and all body atoms use relations of the source schema, and all head atoms use relations of the target schema. Whenever we use $TGDs$, we are referring to this kind. (b) $\sum_{t}$, ICs (i.e. $TGDs$ or *egds*) on target schema.

## 3.1 ChaseFun

This paper [Bonifati et al., 2016] presents a novel bottom-up chase-based tool, ChaseFun, for DE that efficiently handle arbitrary functional dependencies (FDs) on the target. This approach relies on exploiting the interactions between $\sum_{st}$ and target FDs. This approach lets these dependencies to manage the size of the intermediate chase results, by playing on a careful ordering of chase steps interleaving FDs and (chosen) $TGDs$. In addition, reasoning on dependency interaction leads to interesting parallelization opportunities and yields additional scalability gains. Also, this tool can work with *WA* programs.

## 3.2 DEMo

DEMo [Pichler and Savenkov, 2009] is a DE tools. It optimizes the chase for dealing with DE. This tool supports *s-t* and target, $TGDs$ and *egds*. This system is applicable in designing the mappings, target dependencies, and in selection of a chase order. The key difference of DEMo with other former systems for mapping analysis, is the focus on redundancy aspect of DE. This paper does not have any experiments related to scalability.

## 3.3 Llunatic

Llunatic [Geerts et al., 2013] is initially developed for data cleaning (or data repairing) which is viewed as an essential issue in many applications related to relational database. Data cleaning solutions provide a consistent instance of a given database, with respect to the set of ICs. This paper introduce a novice algorithm to solve database repairing problems. The major component of the algorithm is computing minimal solutions with a parallel chase-based engine, Llunatic, which runs over the RDBMS to compute repairs and adopts a variant of the traditional

chase procedure for *egds* [Fagin et al., 2005].

LLUNATIC cleaning system is redesigned as an open-source data exchange framework [Geerts et al., 2014a]. In order to generate solutions for mapping and cleaning scenarios, LLUNATIC has revised and extended the traditional chase procedure for *egds* and *TGDs* [Fagin et al., 2005]. Several variants of the chase, including one with fully homomorphism check, have been implemented on top of PostgreSQL, and this tool provides certain answer in polynomial time for *WA* programs.

## 3.4 PDQ

PDQ [Benedikt et al., 2015] is a system based on query rewriting, and manages query reformulation with constraints and access patterns, by utilizing chase-based proofs. Required inputs for this tool are a query, a set of ICs, and a set of interface descriptions (e.g., views or access methods). It rewrites the query that refers only to the interfaces. This process is done by using an underlying implementation of a chase in which the chase does a full homomorphism check. This tool can deal with *WA* programs as it is shown in [Benedikt et al., 2017].

## 3.5 Benchmarking the Chase

CHASEBENCH [Benedikt et al., 2017] gives the extensive benchmark for assessing the chase implementations over a wide scope of assumptions about the dependency sets, the combination of *TGDs* and *egds*. The tools that are compared may not support the same class of Datalog$^{+}$ programs. As the designated frameworks works with varying sorts of dependencies, multiple datasets with various sizes and programs are prepared for the experiments. We are interested in the results of CHASEFUN, DEMO, LLUNATIC, and PDQ. This paper has categorized the chase algorithm that is used in these tools, into following groups: (a) *unrestricted chase*,

(b) *restricted chase*, (c) *unrestricted skolem chase*, and (d) *single-TGD-parallel (or 1-parallel) chase.*

The *unrestricted* (or oblivious) *chase* simply eliminates the homomorphism check which is introduced in Section 2.5. This version of the chase applies *TGDs*, and *egds*, and may face the problem of non-terminating chase for *WA* programs. The *restricted chase* is the same as performing step 1 to 3 of Algorithm 2.1, and it does the homomorphism check, so that the chase terminates in polynomial time for *WA* programs. We should mention that this form $freezes$ a null at step 2 and it does not have *resumption.*

The *unrestricted skolem chase* extends unrestricted chase by skolemizing the *TGDs*. This can be done by replacing the $\exists$-variable $Z$ in *TGD* $\sigma$ with a function term $f(\sim Z)$, where $f(\sim Z)$ is a fresh function symbol, and $\sim Z$ contains all variables occurring in both the head and the body of $\sigma$. Then, the chase only finds applicable pairs without checking homomorphism. Although cases exist where the restricted chase terminates, the Skolem chase does not. This can be determined by checking syntactic analysis (acyclicity [20]) of the program. The *single-TGD-parallel (or 1-parallel) chase* is executing each *TGDs* in parallel.

One of the primarily test is finding out if each tool produces certain answers. For this, six small scenarios that produce small chase results, are prepared. The specification of tools that we are interested in their test results, and the certain answer test's results are listed in Table 3.1.

| System | *s-t TGDs* | t *TGDs* | *EGDs* | Cert. Ans. | Chase Strategy |
|---|---|---|---|---|---|
| ChaseFun | y | n | FDs only | n | unrestricted Skolem |
| DEMo | y | y | y | n | restricted chase |
| Llunatic | y | y | y | y | all four forms |
| PDQ | y | y | y | y | restricted chase |

Table 3.1: Tested tools specification; 'y'= yes, and 'n' = no. [Benedikt et al., 2017]

From 23 scenarios designed for building the benchmark, we are interested in the results of DOCTORS DE task. Dependencies in this scenario are *tgds* (old version, the one introduced in Chapter 2). To be more precise, this scenario contains 9

queries and a schema mapping with 5 *tgds*, that is classified as a *WA* Datalog$^{\pm}$ program. The dataset size is 10K and 100K and it is the same dataset that is used in [Geerts et al., 2014a]. The results of the scalability test is shown in Table 3.2[1].

| System | 10K | 100K |
|---|---|---|
| ChaseFun | 0.35 | 0.62 |
| DEMo | 31 | 1777 |
| Llunatic | 0.08 | 0.35 |
| PDQ | 12 | 388 |

Table 3.2: The chase benchmark for our desired tools [Benedikt et al., 2017]; The time is in second.

[1]We collected these results from their GitHub repository: `https://dbunibas.github.io/chasebench/times.pdf`

# Chapter 4

# JowsDatalog

In this chapter, we describe the architecture, design, and implementation of JowsDatalog, a Datalog$^{\pm}$ tool for building and querying ontologies. JowsDatalog is a Java implementation of $\mathsf{SChQA}^{\mathcal{S}}$ that is introduced in [Milani and Bertossi, 2016]. $\mathsf{SChQA}^{\mathcal{S}}$ guarantees that the chase terminates for *JWS* Datalog$^{\pm}$ programs, and QA can be done in polynomial time in data complexity.

The structure of this chapter is as follows: first, we explain the architecture of JowsDatalog. Secondly, we describe how JowsDatalog starts the process by parsing the input. After that, we explain two approaches for implementing the $\mathsf{SChQA}^{\mathcal{S}}$. Then, we go through the module that we design for $\mathsf{MagicD}^{+}$ [Milani and Bertossi, 2016]. Finally, we explain how we can make $\mathsf{SChQA}^{\mathcal{S}}$ adaptive, in the sense that it becomes partially independent from the query.

## 4.1 An Overview of JowsDatalog

JowsDatalog gets as inputs a Datalog$^{\pm}$ program, query(s), and the EDB, from the user, and translates the input into Java programming language's objects. After that, JowsDatalog executes the $\mathsf{SChQA}^{\mathcal{S}}$, and provides sets of answers for the queries. This process is shown in Figure 1.4.

JowsDatalog has the following modules:

1. *JDParser*: This module is the first gate of JowsDatalog. It gets the input and stores it into Java objects.

2. *Basic*: A plain in-main-memory implementation of the chase procedure and query-answering (QA) is provided in this module.

3. *In-DB*: A module that uses the power of an RDBMS and the SQL language for implementing the chase procedure.

4. *MS-Rewriter*: MagicD$^+$ [Milani and Bertossi, 2016] is implemented through this module, and this performs the rewriting technique.

JowsDatalog starts the process by parsing the input through *JDParser*. Then, the first option is passing the output of the parser to: (a) *In-DB* or *Basic*, and getting the output; (b) *MS-Rewriter*, and then forward it to either *In-DB* or *Basic*, and getting the output. The last step provides the answers for the query to the user, and the extended EDB with tuples generated during the chase. The workflow of the JowsDatalog is shown in Figure 4.1.

## 4.2 JDParser

Parsing is the process of breaking a string of symbols, either in natural language, computer languages, or data structures into smaller elements for easy translation into another language. We use *JDParser* to analyze the Datalog$^\pm$ programs and represent it in terms of Java objects.

*JDParser* gets a Datalog$^\pm$ program and represents it as Java programming language objects. We make use of JFlex [Klein et al., 2001] which is a lexical analyzer generator in Java. JFLex is a mediator to reduce the complexity of the parser. Also, it increases the flexibility of the *JDParser*. We define the Datalog$^\pm$ grammar in format of regular expressions in a JFLex program, and the output, *JDParser* is a Java class in which we can generates Java objects. *JDParser* is integrated inside JowsDatalog.

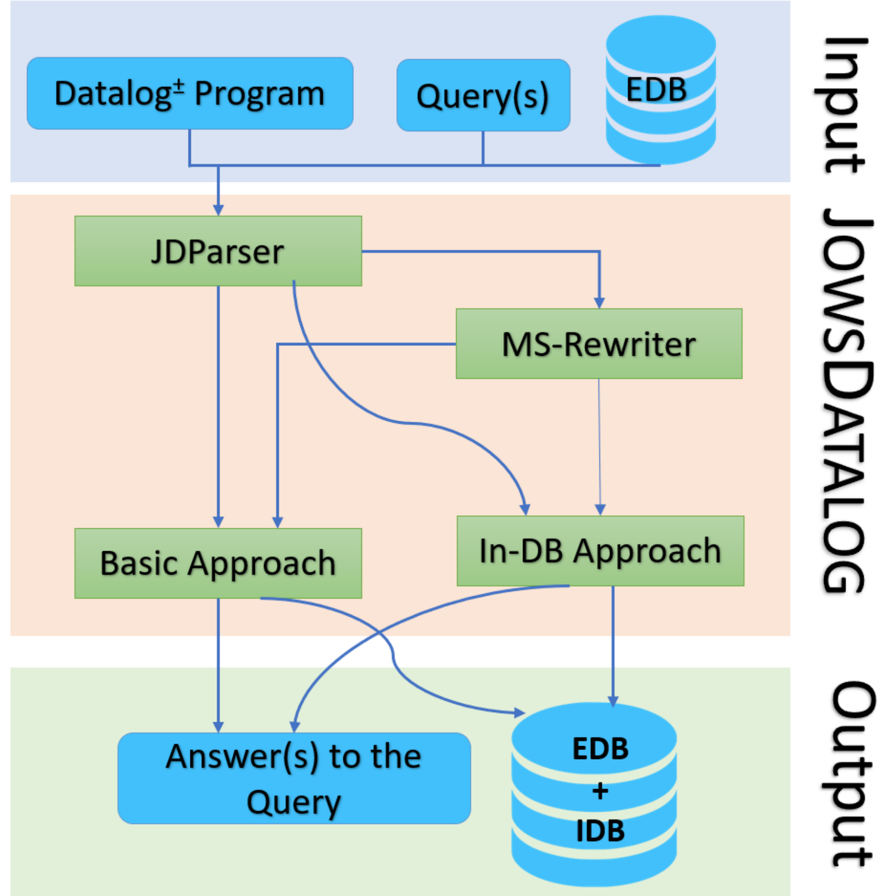


Figure 4.1: JOWSDATALOG workflow

We modified the syntax of Datalog$^{\pm}$ programs to simplify the process of parsing, as follows:

- :- is used instead of $\leftarrow$.
- $\exists$ symbols are eliminated from each rule, and left implicit.
- Queries look like: $ans_{\mathcal{Q}}(\bar{x})$ ?- $P_1(\bar{x}_1), ..., P_n(\bar{x}_n)$ (?- instead of $\leftarrow$ as in (2.5)).
- Use $Z_i$ instead of $\zeta_i$ as *null* values.
- Constants must start with ‘ and ends with ’.

These changes are demonstrated in Example 4.1.

**Example 4.1.** (ex. 2.11 cont.) The acceptable syntax in JOWSDATALOG is as follows:

$$R(\text{'}a\text{'}, \text{'}b\text{'}).$$

$$S(\text{'}a\text{'}, \text{'}c\text{'}).$$

$$M(\text{'}b\text{'}).$$

$$T(\text{'}a\text{'}, \text{'}b\text{'}).$$

$$\sigma_1: \quad R(Y, Z) \text{ :- } T(X, Y).$$

$$\sigma_2: \quad T(Y, Z) \text{ :- } R(X, Y),\ S(X, W),\ M(Y).$$

$$\sigma_3: \quad P(X, Y, Z) \text{ :- } T(X, Y).$$

$$Q(X) \text{ ?- } P(X, Y, Z).$$

Here, $Z$ in $\sigma_1,\ \sigma_2,\ \sigma_3$, and $Y,\ Z$ in the query are implicitly existentially quantified. ■

We consider a Java class for representing each element of the Datalog$^+$ program. For example, we allocate a class for *predicate*, a class for *rules*, a class for *atoms*, ... We use hash-set data structure for optimizing the space and speed of the tool. *JDParser* reads the input file, builds required objects, and fills up the following hash-set: *db* stores the facts, *tgds* keeps the *tgds*, and *queries* hoards the query(s). This is illustrated in Example 4.2.

Also, a set of positions $\mathcal{S}^{\exists}$ for program $\mathcal{P}$ is necessary for executing $\mathsf{SChQA}^{\mathcal{S}}$. JowsDatalog, computes this set from the program, stores it in a hash-set *positions*, and passes it to the next module. This is illustrated in Example 4.2.

**Example 4.2.** (ex. 4.1 cont.) The values for *db*, *tgds*, *queries*, and *positions* are as follows:

$$\begin{aligned}
db &= \{T(\text{`}a\text{'},\text{`}b\text{'}), M(\text{`}b\text{'}), S(\text{`}a\text{'},\text{`}c\text{'}), R(\text{`}a\text{'},\text{`}b\text{'})\}\\
tgds &= \{R(Y,Z) :\text{-}\ T(X,Y).,\\
&\quad T(Y,Z) :\text{-}\ R(X,\ Y),\ S(X,\ W),\ M(Y).,\\
&\quad P(X,\ Y,\ Z) :\text{-}\ T(X,\ Y).\}\\
queries &= \{Q(X)\ ?\text{-}\ P(X,Y,Z)\}\\
positions &= \{S[1], S[2], M[1], P[1], P[2], P[3]\}
\end{aligned}$$

■

## 4.3 Basic Approach

This approach provides a plain implementation of $\mathsf{SChQA}^{\mathcal{S}}$. It uses the main memory, and is developed in Java. The input of this module is provided by *JDParser*. It requires two components: *Evaluator* and *Fact-Generator*.

### 4.3.1 The Evaluator

The core of $\mathsf{SChQA}^{\mathcal{S}}$ is finding applicable pairs, which can be done by evaluating the body of a rule, and trying to match the variables with facts. We design a component called *Evaluator*, in which the input to *Evaluator* is the body of a rule, *conjunct*, and *db*. The output is a set of applicable pairs of rule-assignment, $applicablePairs$.

The *Evaluator* initializes a set for storing assignments, *assignments*, and executes a loop over all the *atoms* in the *conjunct*. For each *atom*, it tries to map the body variables to a database fact by doing a loop over all facts in *db*. It stores the results of the mapping for each *atom* in a temporary hash-set of assignments, *partialAssignment*. Then, it tries to merge these two sets, *partialAssignment and assignments*, and stores the output of the merge in *assignments*. An applicable pair consists of a rule and an assignment for body variables. The method may

find one or more sets of assignments that we have already processed during the chase. For tracking applicable pairs that have been processed and added to the database, we define a set called *applied* which contains processed applicable pairs.

The *Evaluator* goes through each assignment in *assignments* and build applicable pairs of *assignment* and the rule. If *applied* does not contain the applicable pair, the *Evaluator* will add it to *applicablePairs*.

**Example 4.3.** Consider the body of a rule is

$$conjunct = R(X, Y), S(X, W), M(Y),$$

and facts are: $db = \{R(\text{'}a\text{'}, \text{'}b\text{'}), R(\text{'}b\text{'}, \text{'}Z_1\text{'}), S(\text{'}a\text{'}, \text{'}c\text{'}), M(\text{'}b\text{'}), T(\text{'}a\text{'}, \text{'}b\text{'})\}$.

The *Evaluator* reads the first atom, $R(X, Y)$, it goes through all of the facts, and adds $\{X \mapsto a, Y \mapsto b\}$ and $\{X \mapsto b, Y \mapsto Z_1\}$ to the *partialAssignment*.

As *assignments* does not have any contradiction with *partialAssignment*, then:

$$assignments = \{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto b, Y \mapsto Z_1\}\}$$

.

The second atom is $S(X, W)$. The *partialAssignment* is $\{\{X \mapsto a, W \mapsto c\}\}$. At the merge stage, $\{X \mapsto a, W \mapsto c\}$ has contradiction with $\{X \mapsto b, Y \mapsto Z_1\}$, but it can be merged with $\{X \mapsto a, Y \mapsto b\}$. Therefore, the result is:

$$assignments = \{\{X \mapsto a, Y \mapsto b, W \mapsto c\}\}$$

The third atom is $M(Y)$. The *partialAssignment*= $\{\{Y \mapsto b\}\}$ which can be merged with assignments. The result is:

$$assignments = \{\{X \mapsto a, Y \mapsto b, W \mapsto c\}\},$$

and the applicable pair is $\{\sigma, \{X \mapsto a, Y \mapsto b, W \mapsto c\}\}$ which is added to

$applicablePairs$. ■

### 4.3.2 The Fact-Generator

The other component of this approach is *Fact-Generator*, in which the input is $applicablePairs$, and the *head* of a *rule*, *head*. The output is a set of new fact(s), which might be added to *db*. This component is illustrated in Figure 4.2, and is explained through Example 4.4.

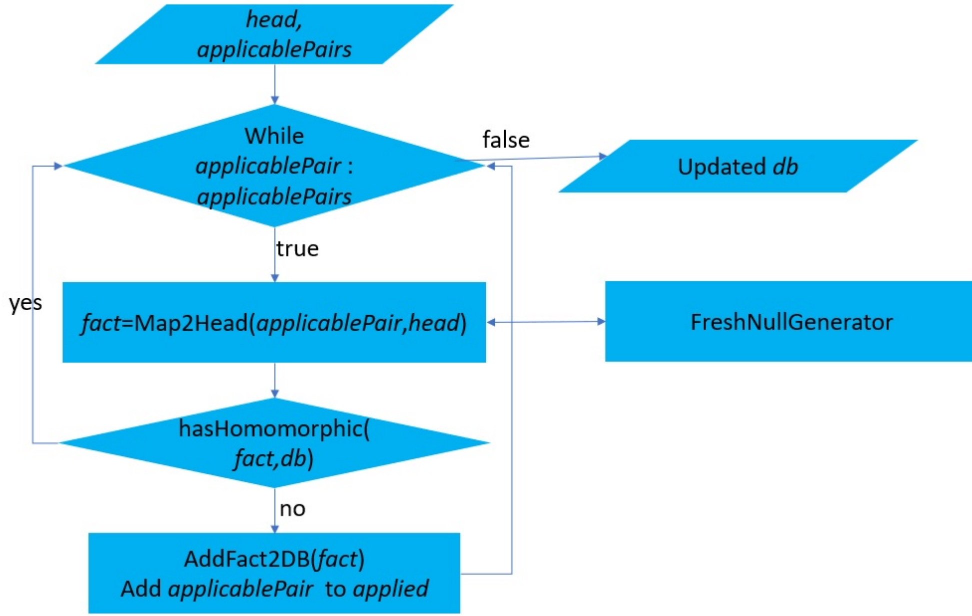


Figure 4.2: The flowchart for *Fact-Generator*

**Example 4.4.** Consider the inputs:

$$applicablePairs = \{\sigma, \{X \mapsto a, Y \mapsto b, W \mapsto c\}\}, \qquad head = T(Y, Z)$$

$ApplicablePairs$ has only one element, so the component goes through the loop once. Therefore,

$$assignment = \{X \mapsto a, Y \mapsto b, W \mapsto c\}$$

*Map2Head* tries to find the mapping for *head*'s variable based on the $assignment$. The partial mapping is $\{Y \mapsto b\}$. Then, *Map2Head* calls *FreshNullGenerator* for

every variable that has not been mapped to a *constant* or *null* yet. *FreshNullGenerator* generates a new *null* symbol, $Z_i$, and returns it to *Map2Head*, $\{Z \mapsto Z_2\}$.

Now, *HasHomomorphic* compares $fact = T(b, Z_2)$ with all other facts inside *db*, and if there is no homomorphism for it inside *db*, it will add the fact to *db*.

■

*Basic* stores *nulls* that are generated in *FreshNullGenerator*, inside a hash-set called *nonFrozenNulls*. Let's say in Example 4.4, $Z_2$ is added to *nonFrozenNulls*. This helps us to track the fresh *nulls*. If this set contains a specific *null* symbol, that symbol would be a fresh *null*. Also, *freezing nulls*, is done by removing all of elements from *nonFrozenNulls*. The component that does this process, called $freezeNulls$. We show *frozen nulls* as $'Z_i'$.

### 4.3.3 Basic Approach Workflow

The input, $queries$, *tgds*, *db*, and $positions$, are given by *JDParser* to *Basic*. Then, *Basic* does a loop over all *tgds* and finds applicable pairs from the *Evaluator*, and sends them to the *Fact-Generator*. This loop continues until there are no more applicable pairs of *rule-assignment*. After that, *Basic* applies $freezeNulls$, and repeats these steps $M_Q$ times, where $M_Q$ is the maximum number of ∃-variables in $queries$. Example 4.5 helps us to explain this.

**Example 4.5.** (ex. 4.2 cont.) *Basic* reads $\sigma_1$ where the *conjunct* is $T(X,Y)$ and *head* is $R(Y,Z)$. Then, it calls the *Evaluator* with *conjunct* and *db* as the input. The output, $assignments$ is $\{\{X \mapsto a, Y \mapsto b\}\}$. For each *applicablePair* in *applicablePairs*, *Fact-Generator* is called and a new fact, $R(\text{‘}b\text{’}, Z_1)$ is added to *db*. The next *tgd*, $\sigma_2$, gives us a new fact, $T(\text{‘}b\text{’}, Z_2)$. $\sigma_3$ adds $P(\text{‘}a\text{’}, \text{‘}b\text{’}, Z_3)$ and $P(\text{‘}b\text{’}, Z_2, Z_4)$ to *db*.

Those *applicablePairs* that create a new fact in *db*, are added to *applied*, a set for keeping the track on these kind of applicable pairs. This set helps us to

process each applicable pair once. The algorithm goes through the *tgds* again; *assignment* is $\{\{X \mapsto a, Y \mapsto b\}, \{X \mapsto b, Y \mapsto Z_2\}\}$ for $\sigma_1$. Then, *Fact-Generator* tries $\{X \mapsto a, Y \mapsto b\}$ first; this gives us $R(Z_2, Z_5)$. *HasHomomorphic* finds out that this fact has at least one homomorphism inside *db* and it won't add it to *db*.

There are no more applicable pairs and *Basic* freezes all *nulls*. Since the query has two ∃-variables, this process is repeated two more times. During the second resumption, $R(Z_2, Z_5)$ does not have any homomorphism to *db* and this tuple is added to *db*.

The third resumption does not change anything, and the output instance is:

$$\begin{aligned} db = \{&T(`a', `b'), M(`b'), S(`a', `c'), R(`a', `b'),\\ &R(`b', `Z_1'), R(`Z_2', `Z_5'), T(`b', `Z_2'),\\ &P(`a', `b', `Z_3'), P(`b', `Z_2', `Z_4')\} \end{aligned}$$

And the answer to the query is: $\{`a', `b'\}$ ■

## 4.4 In-DB Approach

The goal of this approach is to make use of an RDBMS like PostgreSQL. *In-DB* creates tables for every predicate in the program, and stores both the EDB and facts that are generated during the chase inside the database. *Tgds*' bodies, in particular, their body joins, become queries to the database. We add an attribute, $ITR$, to each table to make sure that JowsDatalog processes a rule-assignment only once. JowsDatalog processes the *tgds* in parallel during the chase.

$ITR$ is helpful for mapping the body of a rule to the database. The chase does a loop over the *tgds* until there is no more applicable pairs of rule-assignments. We add a counter, *itr*, to this loop, and whenever the component finishes a loop over the *tgds*, we increase *itr*. When we insert a new fact into the database, the

value for the *ITR* attribute in the table will be *itr + 1*. If we want to map the body of a rule to the database, we only consider those tuples that are added to the database during the latest iteration. This can be done by adding a condition to the select query statement.

In this section, we explain how *In-DB* builds the database. After that, we go through two components of *In-DB*: *Evaluator* and *Fact-Generator*. Finally, we introduce *In-DB*'s algorithm and explain it through an example.

### 4.4.1 Building the Database

*In-DB* gets *db*, *tgds*, *positions*, and *queries* from *JDParser*. In addition, it also expects a database connection information. *In-DB* does two things at this stage: (a) it creates a table for each predicate in the program if they do not have any tables in the database. (b) It inserts the facts in *db* into each table. This process is summarized in Figure 4.3 and shown in Example 4.6.

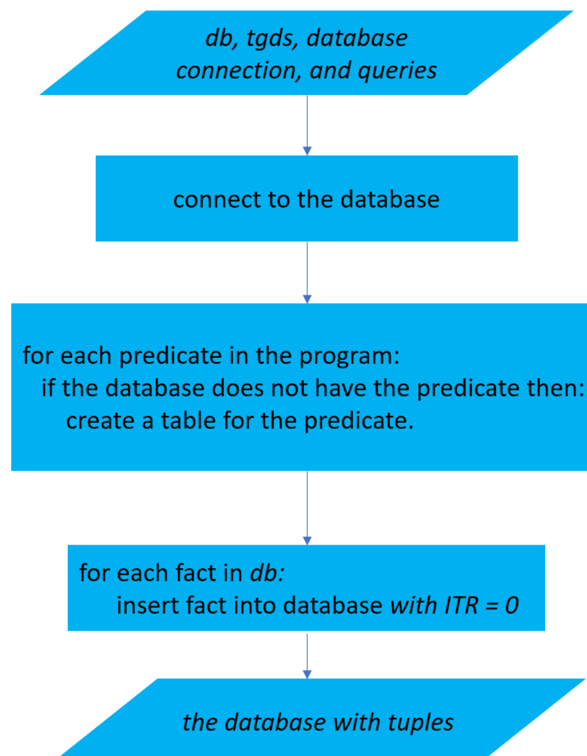


Figure 4.3: The flowchart for storing the database

**Example 4.6.** (ex. 4.2 cont.) The database does not have predicate $R$. *In-DB* executes the following query:

```
CREATE TABLE IF NOT EXISTS R(
  X0 character varying,
  X1 character varying,
  ITR int4);
```

Here, $X0$ and $X1$ are for storing the arguments in $R$ (positions R[1] and R[2]). JowsDatalog assumes that the data type is a string of character for predicates' arguments. $ITR$ assists JowsDatalog in the execution of the chase, as it will be shown in Example 4.9.

Also, *S, M, T* and *P* are added to the database. JowsDatalog executes:

```
CREATE TABLE IF NOT EXISTS S(
  X0 character varying,
  X1 character varying,
  ITR int4);


CREATE TABLE IF NOT EXISTS M(
  X0 character varying,
  ITR int4);


CREATE TABLE IF NOT EXISTS T(
  X0 character varying,
  X1 character varying,
  ITR int4);


CREATE TABLE IF NOT EXISTS P(
  X0 character varying,
  X1 character varying,
```

```
X2 character varying,
ITR int4);
```

After *In-DB* created these tables, it generates insertion queries for the tuples in *db*.

```
1. INSERT into T(X0, X1, ITR)
2.  SELECT * from (SELECT 'a','b', 0) as tmp where not exists
3.      (SELECT * from T where X0='a' and X1='b') limit 1;
```

The first line in this query indicates that we want to insert a new tuple in the table $T$. The second line determines the value for the new tuple. The "where not exists" clause insures that $T$ does not have this tuple. The select query statement in the third line provides the required data for "where not exists" clause. The "limit 1" clause dictates that existence of at least one tuple similar to the one we want to insert, is sufficient for ignoring the insertion. The rest of insertion queries are as follows:

```
INSERT into M(X0,ITR)
 SELECT * from (SELECT 'b', 0) as tmp where not exists
  (SELECT * from M where X0='b') limit 1;
INSERT into S(X0,X1,ITR)
 SELECT * from (SELECT 'a','c', 0) as tmp where not exists
  (SELECT * from S where X0='a' and X1='c') limit 1;
INSERT into R(X0,X1,ITR)
 SELECT * from (SELECT 'a','b', 0) as tmp where not exists
  (SELECT * from R where X0='a' and X1='b') limit 1;
```

■

### 4.4.2 In-DB's Evaluator

This component maps a *tgd*'s body to the database. The inputs are a *tgd*, a database connection, and *itr* (the loop counter). It executes the body joins of the *tgd* inside the database, and finds applicable pairs. For narrowing down the result set, *Evaluator* adds a condition(s) to the SQL query. This condition indicates that the value for $ITR$ must be equal to *itr* in at least one tuple participating in the join. In other words, it guarantees that at least a new tuple is involved in the join. It stores the answers into a set, $qResSet$. This process is illustrated in Example 4.7. The *In-DB* version of *Evaluator* is shown in Figure 4.4.

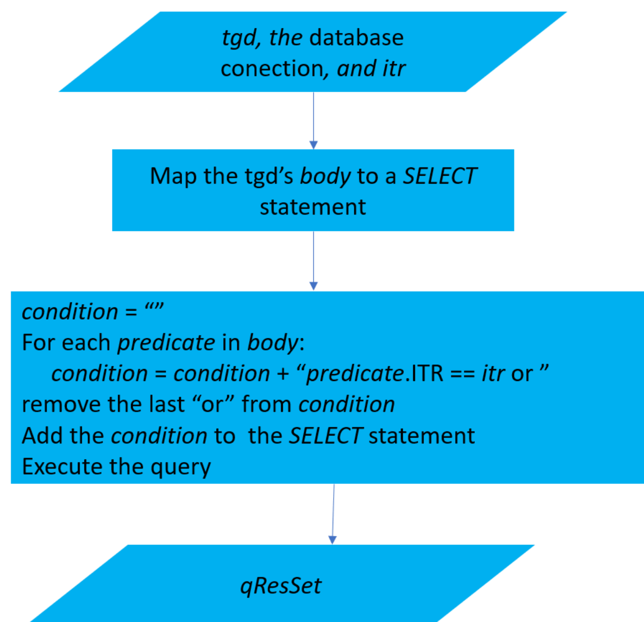


Figure 4.4: The new version of *Evaluator*

**Example 4.7.** (the *In-DB* version of Example 4.3) Consider *itr* is 0, the *tgd* is $T(y, z) \leftarrow R(x, y), S(x, w), M(y)$ , and the facts are:

$$db = \{R(\text{'}a\text{'}, \text{'}b\text{'}, 0), S(\text{'}a\text{'}, \text{'}c\text{'}, 0), M(\text{'}b\text{'}, 0), T(\text{'}a\text{'}, \text{'}b\text{'}, 0)\}$$

The select query statement corresponding to *tgd*'s body:

```
SELECT r0.X0, r0.X1, s1.X1 from
```

```
R as r0, S as s1, M as m2 where
s1.X0 = r0.X0 and m2.X0 = r0.X1 and
((r0.ITR <= 0 and r0.ITR >= 0) or
(s1.ITR <= 0 and s1.ITR >= 0) or
(m2.ITR <= 0 and m2.ITR >= 0))
```

The query joins tables $R$, $S$ and $M$ based on positions ($R$[1],$S$[1]) and ($R$[2],$M$[1]). It also contains the condition for $ITR$. The result of this query is $\{X = \text{`}a\text{'}, Y = \text{`}b\text{'}, W = \text{`}c\text{'}\}$, which is added to *qResSet*, a set for keeping the result of *Evaluator*.

■

### 4.4.3 In-DB's Fact-Generator

The logic of *In-DB*'s *Fact-Generator* is almost the same as that of *Basic*'s *Fact-Generator* (Section 4.3.2). There are two differences between these two:

- The homomorphism is checked by executing a select query.
- The new fact is added to the database via an insertion query.

This component gets a *tgds*'s *head*, a *qResSet*, and a database connection. Then, it generates a new fact for each applicable pair in the *qResSet*. If it needs a fresh null, it gets one from $FreshNullGenerator$. After that, it checks if this fact has homomorphism to the database by means of an SQL query. If there is no homomorphism for this fact, it will be added to the database with $ITR = itr + 1$. Also, if the fact has a fresh null, it will be added to *nullSet*, a hash-set for storing the fresh null labels. This process is shown in Figure 4.5, and illustrated through Example 4.8.

**Example 4.8.** (ex. 4.7 cont.) The *head* is $T(y, z)$ where $z$ is an existential variable. So, $Z_2$ is generated by $FreshNullGenerator$. Then, the homomorphism is checked by evaluating the results of the following query:

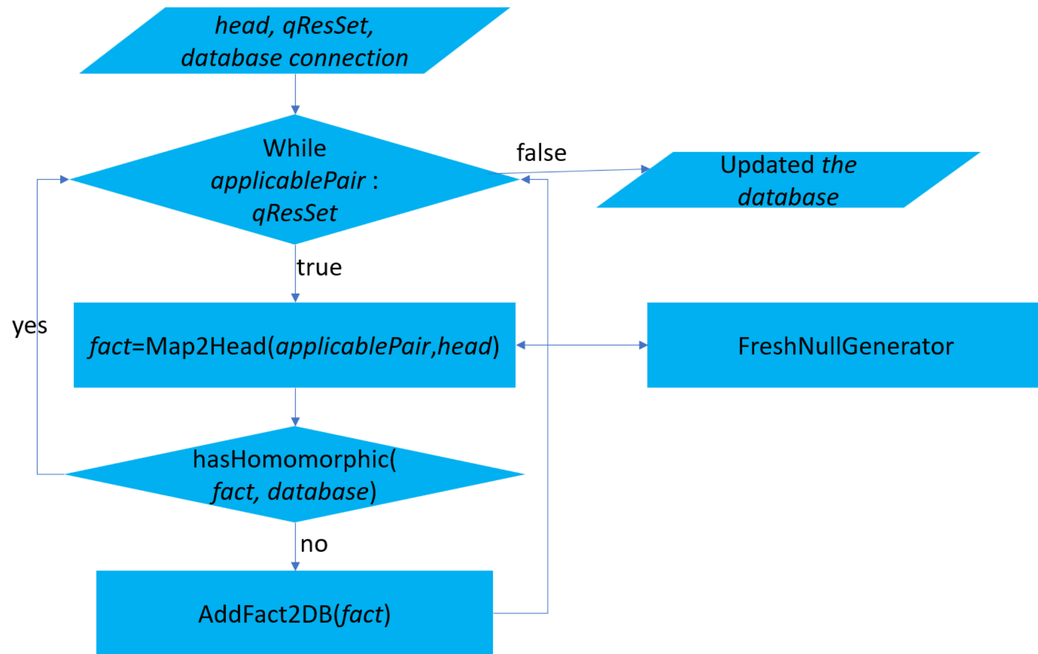


Figure 4.5: The new version of the *Fact-Generator*

```
SELECT * from T where X0 = 'b'
```

Notice that as $Z_2$ is fresh, it is not in the condition. The answer set for this query is empty. Therefore, the new tuple does not have a homomorphism to a tuple in the table $T$. JowsDatalog generates and executes the following insertion query statement:

```
INSERT into public.T(X0,X1,ITR)
  SELECT * from (SELECT 'b','Z_2', 1) as tmp
  where not exists
  (SELECT * from public.T where
  X0='b' and X1='Z_2') limit 1;
```

Note that the value for $ITR$ is set to $itr + 1$. After that, JowsDatalog adds $Z_2$ to the $nullSet$. ■

### 4.4.4 In-DB Algorithm

*In-DB* starts executing the $\mathsf{SChQA}^{\mathcal{S}}$, and computing the chase. It initializes

$nullSet$, a hash-set for storing fresh null labels, $itr = 0$, a variable for keeping the track of the iteration, $firstItr = itr$, a memory variable which helps us during the resumption stage, and $newTuple$, a flag for finding out if the database is updated during the last iteration.

Then, the module does a loop over the *tgds*, applies them by using *Evaluator* and *Fact-Generator*, and increases $itr$ by one at the end of each iteration. JowsDatalog processes the *tgds* in parallel during each iteration. This loop terminates when no more tuples have been inserted into the database during the last iteration. Then, it performs the *resumption*.

If *In-DB* does the *resumption* as stated in $\mathsf{SChQA}^{\mathcal{S}}$, the chase will terminate with an incomplete instance. The reason is that at least one new tuple must participate in the body joins. To solve this issue, we modify *resumption* as follows: (a) we freeze the nulls in the instance by removing them from $nullSet$. (b) *In-DB* applies the *tgds* once, in which the lower bound for $ITR$ is the value of $firstItr$ and the upper bound is $itr$. If the mapping generates new tuples, $ITR$ will be $itr + 1$. (c) *In-DB* updates the value for $itr$ to $itr + 1$, and $firstItr$ to $itr$.

The answers to the query is obtained by using a select statement which maps the query's body to the database. JowsDatalog implements Algorithm 4.1. The flowchart of the algorithm is shown in Figure 4.6. The algorithm is illustrated through Example 4.9.

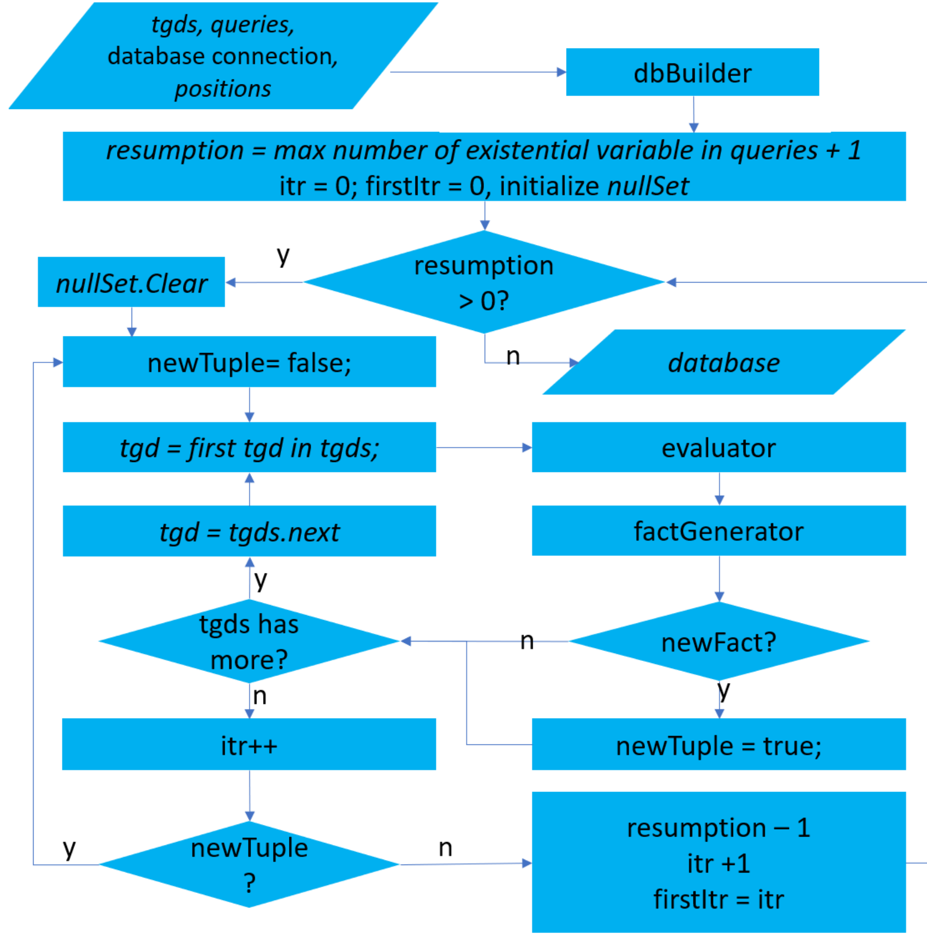


Figure 4.6: The flowchart of *In-DB*

---

**Algorithm 4.1:** The modified version of $\mathsf{SChQA}^{\mathcal{S}}$

---

**Data:** *tgds*, *positions*, *query* and the database connection string

**Result:** A materialized chase instance

1 $itr = 0$; $newTuple =$ false; $firstItr = itr$; $nullSet = \{\}$;

2 $resumptionCounter =$ The number of $\exists$-variables in the $query$;

3 **while** $resumptionCounter \geq 0$ **do**

4     **do**

5         $newTuple =$ false;

6         **foreach** tgd*:* tgds **do**

7             $qResSet =$ Map the *tgd*'s body to a select query statement and add a condition of *table.ITR* == *itr*;

8             Initial set $applicablePairs$;

9             **foreach** $ans$ : $qResSet$ **do**

10                 $applicablePair =$ map ans to the *tgd*'s head and produce a

**Example 4.9.** (ex. 4.6 cont.) The database is:

| R | X0 | X1 | ITR |
|---|---|---|---|
| | a | b | 0 |

| S | X0 | X1 | ITR |
|---|---|---|---|
| | a | c | 0 |

| T | X0 | X1 | ITR |
|---|---|---|---|
| | a | b | 0 |

| M | X0 | ITR |
|---|---|---|
| | b | 0 |

| P | X0 | X1 | X2 | ITR |
|---|---|---|---|---|

*In-DB* initializes:

$$itr = 0,\ newTuple = false,\ firstItr = 0, \text{ and } nullSet = \{\}$$

Then, it processes each *tgd* at the same time (line 4 to 16 Algorithm 4.1). The first *tgd* is $R(Y, Z) \leftarrow T(X, Y)$. *Evaluator* executes the following query:

```
SELECT t0.X0,t0.X1 from T as t0
  where  (t0.ITR <= 0 and t0.ITR >= 0)
```

This query gets the tuples with their ITR equals to 0 in relation $T$. The *qResSet* is $\{x = \text{‘}a\text{’}, y = \text{‘}b\text{’}\}$.

This set is passed to *Fact-Generator*. As the head predicate has an existential variable, it produces a new null value, $Z_1$. So, $R(\text{‘}b\text{’}, Z_1)$ may be added to the table $R$, if this table does not have an homomorphism with this tuple. It is necessary to select all the records from the head predicate, $R$, for checking the homomorphism. Constants in the new tuple are added to the select query statement as the conditions to narrow down the result set. So, *Fact-Generator* executes the following query:

```
SELECT X0,X1 from R where X0 = 'b'
```

As the answer set for this query is empty, the new tuple does not have a homomorphism with a tuple in table $R$. *Fact-Generator* generates and executes the following insertion query statement:

```
INSERT into public.R(X0,X1,ITR)
  SELECT * from (SELECT 'b','Z_1', 1) as tmp
    where not exists
      (SELECT * from public.R where
       X0='b' and X1='Z_1') limit 1;
```

As shown in the query, the value for *ITR* is 1 (*itr* + 1). Also, we add $Z_1$ to *nullSet*. Since we added a new tuple to the database, the value for *newTuple* is updated to *true*.

*In-DB* is also processing another *tgd*, $T(y, z) \leftarrow R(x, y), S(x, w), M(y)$, at the same time. The select query statement for *tgd*'s body:

```
SELECT r0.X0, r0.X1, s1.X1 from
  R as r0, S as s1, M as m2 where
  s1.X0 = r0.X0 and m2.X0 = r0.X1 and
   ((r0.ITR <= 0 and r0.ITR >= 0) or
    (s1.ITR <= 0 and s1.ITR >= 0) or
    (m2.ITR <= 0 and m2.ITR >= 0))
```

The result of this query is $\{x = \text{`}a\text{'}, y = \text{`}b\text{'}, w = \text{`}c\text{'}\}$. This may give us $T(\text{`}b\text{'}, Z_2)$ ($Z_2$ is a new null value). So, *Fact-Generator* checks for homomorphism:

```
SELECT X0,X1 from T where X0 = 'b'
```

No answer. *Fact-Generator* adds $Z_2$ to *nullSet*, and executes the following insertion query statement ($ITR = itr + 1 = 1$):

```
INSERT into public.T(X0,X1,ITR)
  SELECT * from (SELECT 'b','Z_2', 1) as tmp
   where not exists
    (SELECT * from public.T where
     X0='b' and X1='Z_2') limit 1;
```

Another *tgd* is: $P(x, y, z) \leftarrow T(x, y)$. The select query statement for its body:

```
SELECT t0.X0,t0.X1 from T as t0 where
  (t0.ITR <= 1 and t0.ITR >= 0)
```

The result of this query is $\{x = \text{'}a\text{'},\ y = \text{'}b\text{'}\}$. This may give us $P(\text{'}a\text{'}, \text{'}b\text{'}, Z_3)$ ($Z_3$ is a fresh null value). *Fact-Generator* checks for homomorphism:

```
SELECT X0,X1,X2 from P where X0 = 'a' and X1 = 'b'
```

No Answer. *Fact-Generator* adds $Z_3$ to *nullSet*, and executes the following insertion query statement ($ITR = itr + 1 = 1$):

```
INSERT into P(X0,X1,X2,ITR)
  SELECT * from (SELECT 'a','b','Z_3', 1) as tmp
    where not exists (SELECT * from P
     where X0='a' and X1='b' and X2='Z_3') limit 1;
```

*In-DB* has processed all *tgds*. The for-loop in line 6 to 14 of Algorithm 4.1 is finished. Line 15 increases *itr* by one ($itr = 1$). Line 16 is true and the algorithm goes to line 5 ($newTuple = false$). The database is:

| R | X0 | X1 | ITR |
|---|---|---|---|
| | a | b | 0 |
| | b | $Z_1$ | 1 |

| S | X0 | X1 | ITR |
|---|---|---|---|
| | a | c | 0 |

| T | X0 | X1 | ITR |
|---|---|---|---|
| | a | b | 0 |
| | b | $Z_2$ | 1 |

| M | X0 | ITR |
|---|---|---|
| | b | 0 |

| P | X0 | X1 | X2 | ITR |
|---|---|---|---|---|
| | a | b | $Z_3$ | 1 |

*In-DB* starts the loop over the *tgds* again. One of the *tgds* is $R(Y, Z) \leftarrow T(X, Y)$ and *Evaluator* executes:

```
SELECT t0.X0,t0.X1 from T as t0
  where  (t0.ITR <= 1 and t0.ITR >= 1)
```

The result set is $\{x = \text{`}b\text{'},\ y = Z_2\}$ that gives us $R(Z_2, Z_4)$. *Fact-Generator* checks for homomorphism. Note that the null symbols in this fact are fresh. Therefore, we can't add them as conditions to the query:

```
SELECT X0, X1 from R
```

The result set is $\{R(\text{`}a\text{'}, \text{`}b\text{'}), R(\text{`}b\text{'}, Z_1)\}$. $R(\text{`}a\text{'}, \text{`}b\text{'})$ is a homomorphism to $R(Z_2, Z_4)$, and *Fact-Generator* does not add this tuple to $R$.

The other *tgd* is $T(y, z) \leftarrow R(x, y), S(x, w), M(y)$, and *Evaluator* executes:

```
SELECT r0.X0, r0.X1, s1.X1 from
  R as r0, S as s1, M as m2 where
    s1.X0 = r0.X0 and m2.X0 = r0.X1 and
     ((r0.ITR <= 1 and r0.ITR >= 1) or
      (s1.ITR <= 1 and s1.ITR >= 1) or
      (m2.ITR <= 1 and m2.ITR >= 1))
```

No answer. At the same time, $P(x, y, z) \leftarrow T(x, y)$ is being processed. *Evaluator* executes:

```
SELECT t0.X0, t0.X1 from T as t0 where
  ((t0.ITR <= 1 and t0.ITR >= 1))
```

The result set is $\{x = \text{`}b\text{'},\ y = Z_2\}$. Then, *Fact-Generator* generates $P(\text{`}b\text{'}, Z_2, Z_5)$, and checks for homomorphism:

```
SELECT X0, X1, X2 from P where X0 = 'b'
```

This query does not have any results, and this tuple is inserted into $P$:

```
INSERT into P(X0, X1, X2, ITR)
  SELECT * from (SELECT 'b', 'Z_2', 'Z_5', 2) as tmp
    where not exists
     (SELECT * from P
        where X0='b' and X1='Z_2' and X2='Z_5') limit 1;
```

We are adding new tuple, then: *newTuple = true*. *In-DB* has processed all *tgds* (line 6 to 14 of Algorithm 4.1 is done). Line 15 increases *itr* by one ($itr = 2$). Line 16 is true and the algorithm goes to line 5 ($newTuple = false$). The database is:

| R | X0 | X1 | ITR |
|---|---|---|---|
| | a | b | 0 |
| | b | $Z_1$ | 1 |

| S | X0 | X1 | ITR |
|---|---|---|---|
| | a | c | 0 |

| T | X0 | X1 | ITR |
|---|---|---|---|
| | a | b | 0 |
| | b | $Z_2$ | 1 |

| M | X0 | ITR |
|---|---|---|
| | b | 0 |

| P | X0 | X1 | X2 | ITR |
|---|---|---|---|---|
| | a | b | $Z_3$ | 1 |
| | b | $Z_2$ | $Z_5$ | 2 |

The algorithm goes back to line 6 and the *tgd* is: $R(Y, Z) \leftarrow T(X, Y)$. The corresponding query is:

```
SELECT t0.X0, t0.X1 from T as t0
  where  ((t0.ITR <= 2 and t0.ITR >= 2))
```

No result. The next *tgd* is $T(y, z) \leftarrow R(x, y), S(x, w), M(y)$ with the mapping:

```
SELECT r0.X0, r0.X1, s1.X1 from
  R as r0 ,S as s1 ,M as m2 where
   s1.X0 = r0.X0 and m2.X0 = r0.X1 and
   ((r0.ITR <= 2 and r0.ITR >= 2) or
    (s1.ITR <= 2 and s1.ITR >= 2) or
    (m2.ITR <= 2 and m2.ITR >= 2))
```

No result. The next one is $P(x, y, z) \leftarrow T(x, y)$ and the query is:

```
SELECT t0.X0, t0.X1 from T as t0 where
  ((t0.ITR <= 2 and t0.ITR >= 2))
```

No result. Line 15 increase *itr* by one ($itr = 3$). The condition in line 16 of Algorithm 4.1 is false. The algorithm proceeds to line 17 which is the resumption.

The first step is clearing *nullSet*. Consider if we do the *resumption* stage as it is stated in Algorithm 2.1. Then, we apply the *tgds* until we don't get any new tuples. This can be done by applying Algorithm 4.1, line 4 to 16.

*In-DB* starts applying the *tgds*. Basically, the mappings for each *tgd* needs at least one tuple in which $ITR$ is equal to *itr* ($itr = 3$). We don't have such a tuple in the database. This causes the termination of *In-DB* with an incomplete instance. Notice that the instance used for QA in Example 4.5 contains more tuples.

To overcome this problem, we perform the line 17 of Algorithm 4.1. This line adds another step to *resumption*, and states that we apply line 6 to 13 one time with one modification: $ITR$'s value in each table must be between $firstItr$ and $itr$.

Therefore, we go back to line 6. The new mapping for $R(Y, Z) \leftarrow T(X, Y)$ is:

```
SELECT t0.X0, t0.X1 from T as t0
where  (t0.ITR <= 3 and t0.ITR >= 0)
```

The *Evaluator* executes this query and stores the result set in *qResSet*:

$$qResSet = \{\{x = \text{`}a\text{'},\ y = \text{`}b\text{'}\},\ \{x = \text{`}b\text{'},\ y = \text{`}Z_2\text{'}\}\}$$

The new potential tuples are $R(\text{`}b\text{'}, Z_6)$, $R(\text{`}Z_2\text{'}, Z_4)$ where $Z_4$ and $Z_6$ are new null symbols. *Fact-Generator* finds a homomorphism for $R(\text{`}b\text{'}, Z_6)$. But, $R(\text{`}Z_2\text{'}, Z_4)$ does not have any homomorphism ('$Z_2$' is treated as a constant). Therefore, $Z_4$ is added to *nullSet* and the tuple is inserted into the database ($ITR = itr + 1 = 4$):

```
INSERT into public.R(X0,X1,ITR)
  SELECT * from (SELECT 'Z_2','Z_4', 4) as tmp
   where not exists
    (SELECT * from public.R where
```

```
    X0='Z_2' and X1='Z_4') limit 1;
```

*In-DB* is also processing $T(y,z) \leftarrow R(x,y), S(x,w), M(y)$. at the same time; the mapping is:

```
SELECT r0.X0, r0.X1, s1.X1 from
 R as r0 ,S as s1 ,M as m2 where
   s1.X0 = r0.X0 and m2.X0 = r0.X1 and
   ((r0.ITR <= 3 and r0.ITR >= 0) or
    (s1.ITR <= 3 and s1.ITR >= 0) or
    (m2.ITR <= 3 and m2.ITR >= 0))
```

The result of this query is: $\{x = \text{'}a\text{'},\ y = \text{'}b\text{'},\ w = \text{'}c\text{'}\}$. This may give us $T(\text{'}b\text{'}, Z_6)$ ($Z_6$ is a new null value). This tuple has a homomorphism in the database.

Another *tgd* is $P(x,y,z) \leftarrow T(x,y)$ with mapping:

```
SELECT t0.X0, t0.X1 from T as t0 where
   (t0.ITR <= 3 and t0.ITR >= 0)
```

The results of this query are: $x = \text{'}a\text{'},\ y = \text{'}b\text{'}$ and $x = \text{'}b\text{'},\ y = \text{'}Z_2\text{'}$. This may give us $P(\text{'}a\text{'},\text{'}b\text{'},\ Z_6)$ and $P(\text{'}b\text{'},\text{'}Z_2\text{'},\ Z_7)$. *Fact-Generator* finds a homomorphism for each of them. Therefore, we won't add them to the database.

*In-DB* has processed all the *tgds* and the *resumption* is finished. It goes to line 4 and apply these process 2 more times. But, this does not update the database.

The materialized instance for this Datalog$^{\pm}$ program is:

| R | X0 | X1 | ITR |
|---|---|---|---|
| | a | b | 0 |
| | b | $Z_1$ | 1 |
| | $Z_2$ | $Z_4$ | 4 |

| S | X0 | X1 | ITR |
|---|---|---|---|
| | a | c | 0 |

| T | X0 | X1 | ITR |
|---|---|---|---|
| | a | b | 0 |
| | b | $Z_2$ | 1 |

| M | X0 | ITR |
|---|---|---|
| | b | 0 |

| P | X0 | X1 | X2 | ITR |
|---|---|---|---|---|
| | a | b | $Z_3$ | 1 |
| | b | $Z_2$ | $Z_5$ | 2 |

Finally, the answer to the query is obtained by executing the following query:

```
SELECT p0.X0, p0.X1, p0.X2 from P as p0
```

Answers to the query are = {‘a’, ‘b’}. ■

## 4.5 MS-Rewriter

MagicD$^+$ [Milani and Bertossi, 2016] (cf. Section 2.6) is implemented through this module. The module gets the program $\mathcal{P}$ and the query, and applies MagicD$^+$ to the combination. The output of this module is passed to either *In-DB* or *Basic*. This process is illustrated through Example 4.10 and 4.11.

**Example 4.10.** (ex. 4.1 cont.) *MS-Rewriter* gets the program from *JDParser*, and applies MagicD$^+$ steps which we explained in Section 2.6. The result is the program $\mathcal{P}'$ as follows:

$$
\begin{aligned}
& R(\text{'}a\text{'}, \text{'}b\text{'}). \\
& S(\text{'}a\text{'}, \text{'}c\text{'}). \\
& M(\text{'}b\text{'}). \\
& T(\text{'}a\text{'}, \text{'}b\text{'}). \\
\sigma'_1: \quad & R(Y, Z) \text{ :- } T(X, Y). \\
\sigma'_2: \quad & T(Y, Z) \text{ :- } R(X, Y), S(X, W), M(Y). \\
\sigma'_3: \quad & P^{fff}(X, Y, Z) \text{ :- } T(X, Y). \\
& Q'(X) \text{ ?- } P^{fff}(X, Y, Z).
\end{aligned}
$$

As the query does not have any constants, and predicate $P$ involves in all rules in $\mathcal{P}$, $\mathcal{P}'$ is the same as $\mathcal{P}$. This program can be passed to either *Basic* or *In-DB*. The chase and QA would be the same as the one in Example 4.5 or Example 4.9 resp.

■

**Example 4.11.** Assume that the program and the EDB are the same as in Example 4.1. The new query is $\mathcal{Q}(X) : \exists Y\ S(X, Y)$.

*MS-Rewriter* gets the program from *JDParser*, and applies MagicD$^+$ on it. The adorned query is $S^{ff}(X, Y)$. There are not any *tgds* with predicate $S$ in their head in the program. Therefore, the new program does not have any *tgds* and keeps only the EDB.

If we pass the output of *MS-Rewriter* to *In-DB* (or *Basic*), the chase instance is the same as the EDB. Actually, these modules do not do anything during the chase as the program does not have any *tgds*. The answer to the query is $\{\text{'}a\text{'}\}$. ■

The effectiveness of the *MS-Rewriter* depends on the combination of the query and the program. Three different effects are expected based on the combinations. For one combination, we illustrated through Example 4.11 that *MS-Rewriter* is able to completely simplify the program. In another combination, it partially

reduces the number of *tgds* (Example 1.5). With a different combination of a query and a program, *MS-Rewriter* generates a program which was the same as the input program (i.e. Example 4.10).

## 4.6 N-adaptive $\mathsf{SChQA}^{\mathcal{S}}$

As it is shown in Example 1.5, the chase is optimized on the basis of the query, since the program has now only one rule. The limitation of $\mathsf{MagicD}^{+}$ is that this algorithm generates a unique program which is specifically designed to answer the given query. If we want to answer many queries in an application, we have to execute the chase (in general, not specifically $\mathsf{SChQA}^{\mathcal{S}}$) every time for each query. The chase is a time-consuming process. However, if we use a chase-based algorithm that is independent from the query, we can run the chase algorithm once and provide the answers to different queries by using the materialized instance $I$ produced by the chase.

$\mathsf{SChQA}^{\mathcal{S}}$ creates a chase instance that is dependent on the query only in terms of the number of ∃-variables in the query. If we give an upper bound $N$ for this value, then the result of $\mathsf{SChQA}^{\mathcal{S}}$, $I$, could be used for answering to queries with at most $N$ ∃-variables. We can have an adaptive value for $N$ during the lifetime of the application. We initialize $N$ with the total number of terms in all predicates in the given program, $\mathcal{P}$. Consider the scenario in which the given query has $M$ more ∃-variables. Then, we execute $\mathsf{SChQA}^{\mathcal{S}}$ with $M$ as the number of ∃-variables and $I$ as the EDB. The new $N$ would be $N = N + M$ and $N$ adapts itself based on the situation. We call this method *N-adaptive* $\mathsf{SChQA}^{\mathcal{S}}$.

**Example 4.12.** Assume that the program and the EDB are the same as in Ex-

ample 4.1. Consider the following queries:

$$\mathcal{Q}_1(X): \ \exists Y, Z \ P(X,Y,Z).$$

$$\mathcal{Q}_2(X): \ \exists Y \ S(X,Y).$$

$$\mathcal{Q}_3(X): \ \exists Y \ R(X,Y), M(Y).$$

We employ the use of *N-adaptive* $\mathsf{SChQA}^{\mathcal{S}}$. In Example 4.9, we had a query with two $\exists$-variables (the same as $\mathcal{Q}_1$). The materialized chase instance $I$ generated in this example, is good for answering queries with up to two $\exists$-variables. Therefore, this instance can provide the answers to $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$.

Consider a query $\mathcal{Q}_4 : \ \exists X, Y, Z \ R(X,Y), S(X,Z)$. This query has three $\exists$-variables and it cannot be answered by using $I$. Therefore, we need to execute $\mathsf{SChQA}^{\mathcal{S}}$ with one as the number of $\exists$-variables and $I$ as the EDB. We input these values into either *In-DB* or *Basic*, and get the new materialized chase instance $I'$. Then, we can provide the answers for $\mathcal{Q}_4$ using $I'$.

*In-DB* cannot use zero as the initial value for *itr* for the purpose of efficiency. In this way it has to generate $I$ first, and then proceed the chase, resulting $I'$. To overcome this problem *In-DB* saves the last value for *itr*, when it was building $I$. If the instance wants to adapt itself with the new query, *In-DB* uses the saved value as the initial value for *itr* at the beginning of the chase.

■

# Chapter 5

# Experimental Evaluation

In this chapter, we discuss the performance evaluation of JowsDatalog[1]. Our experiments were conducted on a Linux machine with 12GB RAM and Intel core i7 CPU. We evaluate the performance of *In-DB* with and without *MS-Rewriter*[2] with the following experiments: (a) *JWS test*, and (b) *comparison test*. We explain these experiments in Section 5.2 and Section 5.3. The dataset and programs we used in these experiments are explained in Section 5.1.

## 5.1 Dataset and Programs

We conducted our experiments using the doctors dataset from the schema mapping literature [Geerts et al., 2014a]. This dataset is inspired from real databases of medical data. Two instances of the dataset are generated by ChaseBench [Benedikt et al., 2017] with 10K and 100K facts. The *tgds* that are defined on top of this dataset are categorized as a *WA* program. This *WA* program has five rules and nine CQ with free variables that are available in Appendix A. This program is designed to do a data exchange task in which we have four source predicates and three target predicates.

[1]We provide a prototype of JowsDatalog which is available at: `https://github.com/pouryas7/jowsdatalog`

[2]We could not provide a stable version of *Basic* that provides a complete materialized instance for a given program.

To the best of our knowledge, there are not any benchmark programs for *JWS* Datalog$^{\pm}$. Therefore, we provide a *JWS* program over the DOCTORS dataset. This program contains all of the *tgds* in the *WA Program* (cf. Appendix A), and five new *tgds*, giving us a *JWS* Datalog$^{\pm}$ program. Then, we define four new representative queries with free variables in order to illustrate the performance of *MS-Rewriter*[3]. The program and its queries are available in Appendix B. This program is not designed to perform any specific task. When we were designing this program, we considered two things: (a) the involvement of all the records in the dataset, and (b) designing a *JWS* program.

## 5.2 JWS Test

In this test, we analyze the execution time of the chase and QA over the the DOCTORS dataset and the *JWS* program (cf. Appendix B), when we use *In-DB* with and without *MS-Rewriter*. This test shows that JOWSDATALOG is able to handle *JWS* Datalog$^{\pm}$ programs. To the best of our knowledge, JOWSDATALOG is the only OBDA tool that can handle these kind of programs.

While executing *In-DB* without *MS-Rewriter*, we used the *N-adaptive* method (cf. Section 4.6). To be more specific, $N$ is initialized with five as an upper bound for the number of $\exists$-variables, and *In-DB* provides a materialized chase instance that can be used for answering to the queries with up to five $\exists$-variables. We run these two methods over the 10K and 100K instances, and their execution times are shown in Table 5.1 and 5.2

[3]Scenarios are available in: `https://github.com/pouryas7/jowsdatalog`

| **Dataset size** | 10K | 100K |
|---|---|---|
| **Chase time(sec)** | 36.4 | 1274 |
| **QA time for $\mathcal{Q}_1$ (sec)** | 0.032 | 0.19 |
| **QA time for $\mathcal{Q}_2$ (sec)** | 0.011 | 0.12 |
| **QA time for $\mathcal{Q}_3$ (sec)** | 0.005 | 0.02 |
| **QA time for $\mathcal{Q}_4$ (sec)** | 0.024 | 0.13 |
| **Total QA time (sec)** | 0.072 | 0.46 |
| **Total QA and chase time (sec)** | 36.472 | 1274.46 |

Table 5.1: The results of *In-DB* without *MS-Rewriter* (using 5-*adaptive* method); the *JWS* program and $\mathcal{Q}_1$,...,$\mathcal{Q}_4$ are available in Appendix B.

| **Dataset size** | 10K | 100K |
|---|---|---|
| **QA time for $\mathcal{Q}_1$ (sec)** | 0.08 | 0.406 |
| **QA time for $\mathcal{Q}_2$ (sec)** | 14.882 | 88.448 |
| **QA time for $\mathcal{Q}_3$ (sec)** | 4.348 | 4.146 |
| **QA time for $\mathcal{Q}_4$ (sec)** | 37.775 | 1270.4 |
| **Total QA time (sec)** | 57.085 | 1363.4 |

Table 5.2: The results of *In-DB* with *MS-Rewriter*; the *JWS* program and $\mathcal{Q}_1$,...,$\mathcal{Q}_4$ are available in Appendix B. Note that as we are using *MS-Rewriter*, we cannot use the *N-adaptive* method.

### 5.2.1 In-DB with MS-Rewriter vs. In-DB without MS-Rewriter

This experiment shows that using *In-DB* with *MS-Rewriter* speeds up the QA for some queries. However, this depends on the query. For example, *MS-Rewriter* enhanced the QA time for $\mathcal{Q}_1$ in Table 5.2. On the other hand, the QA time for $\mathcal{Q}_4$ in Table 5.2 was almost the same as the chase time in addition to QA time for the same query, when we used *In-DB* without *MS-Rewriter* (Table 5.1). The

reason is that this query depends on all *tgds* in the *WA* program. The program generated by *MS-Rewriter* is sometimes the same as the one given to this module as the input, because the query has no parameters, but only free variables, and all the rules are relevant to the query (i.g. Example 4.10). If we are able to determine the upper bound for the ∃-variables, *In-DB* without *MS-Rewriter* will perform better and provide answers in real time, because the same chase instance can be used to answer several different queries.

If the EDB updates constantly, the *MS-Rewriter* may become more suitable for these kind of EDBs. Consider for example, a website that builds an EDB from its log files and a Datalog$^{\pm}$ program is designed on top of the EDB for detecting the attacks on the website. Here, the EDB changes all the time, and *In-DB* without *MS-Rewriter* is not a good choice.

In summary, consider we are dealing with an application where we can determine the upper bound for the ∃-variables, and the EDB does not update constantly. Then, using *In-DB* without *MS-Rewriter* is a wise choice. On the other hand, if we do not have an estimation for the upper bound for the ∃-variables or a stable EDB, an EDB without changes, the *MS-Rewriter* is handy.

## 5.3 Comparison Test

We compare the performance of JowsDatalog with several state of the art systems, in particular ChaseFun [Bonifati et al., 2016], DEMo [Pichler and Savenkov, 2009], Llunatic [Geerts et al., 2013], and PDQ [Benedikt et al., 2015]. The aforementioned tools are relevant because, like JowsDatalog, their chase procedure is integrated with an RDBMS, and they support *WA* programs as well. On the other hand, they cannot deal with *JWS* Datalog$^{\pm}$ programs and we could not evaluate them with the *JWS* program in Appendix B. Therefore, we use the *WA* program in Appendix A for the comparison test.

The results of this test for JowsDatalog are available in Table 5.3. The

results of this test for ChaseFun [Bonifati et al., 2016], DEMo [Pichler and Savenkov, 2009], Llunatic [Geerts et al., 2013], and PDQ [Benedikt et al., 2015] are shown in Table 3.2. The test environment in ChaseBench is a server with 6 physical Xeon v3 cores running at 1.9GHz, 16 GB of RAM, and a 512 GB SSD, running Ubuntu v16. The comparison between the performance of JowsDatalog and other tools cannot be taken too seriously in the sense that the different systems were run on different machines, and only with *WA* programs. Furthermore, all the other systems cannot handle *JWS* programs, which go far beyond *WA*.

| **Dataset size** | **10K** | **100K** |
|---|---|---|
| **Chase time using *In-DB* without *MS-Rewriter* (sec)** | 40.72 | 1445 |
| **Total QA time using *In-DB* without *MS-Rewriter* (sec)** | 0.846 | 9.4 |
| **Average QA time using *In-DB* with *MS-Rewriter* (sec)** | 33.95 | 1255.33 |

Table 5.3: The results of running the *WA* program in Appendix A using *In-DB* with and without *MS-Rewriter*; we used *10-adaptive* method for *In-DB* without *MS-Rewriter*. As the QA time for each query were similar to each other while using *In-DB* with *MS-Rewriter*, we reported the average QA time.

### 5.3.1 Discussion

JowsDatalog did not perform well in comparison with recent state of the art, Llunatic, PDQ, and ChaseFun. Also, it is good to remind the reader that our test environment is not the same as the one in ChaseBench. They ran the test on a machine using more powerful hardware.

The other reason is that JowsDatalog does a full homomorphism check. This tool is designed to deal with more complex Datalog$^{\pm}$ programs. In this tool several queries must be executed for the mappings, the homomorphism check, and

inserting new facts into the database.

The QA time when we use *MS-Rewriter* is about the same time that we obtained running *In-DB* without this module. The reason relies on the nature of the queries. All of the *tgds* are necessary for building a materialized instance to answer these queries. Here, the same situation occurred in Example 4.10, in which the QA is not optimized by using *MS-Rewriter*.

# Chapter 6

# Conclusions and Future Work

In this work, we introduced JowsDatalog, an OBDA tool that works with Datalog$^{\pm}$ and guarantees tractable QA for *JWS* Datalog$^{\pm}$ programs. To the best of our knowledge, JowsDatalog is the only tool that works with *JWS* Datalog$^{\pm}$ programs and its subclasses. JowsDatalog uses parallel processing and an RDBMS for facilitating the chase execution. We also investigated the trade-off between using $\mathsf{SChQA}^{\mathcal{S}}$ with $\mathsf{MagicD}^{+}$ and $\mathsf{SChQA}^{\mathcal{S}}$ without $\mathsf{MagicD}^{+}$.

In relation to future work, we improve the performance of the tool by modifying the chase in a way that it runs on multiple machines at the same time, in a distributed manner. We optimize the SQL queries that we use for homomorphism check and insertion by merging them into one query. Also, We expand the tool so that it can work with other data types rather than strings. It would be desirable to add support for *equality-generating dependency* and *negative constraints* to JowsDatalog.

# References

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc.

Alviano, M., Leone, N., Manna, M., Terracina, G., and Veltri, P. (2012). Magic-sets for datalog with existential quantifiers. In *International Datalog 2.0 Workshop*, pages 31–43. Springer.

Arenas, M., Bertossi, L., and Chomicki, J. (1999). Consistent query answers in inconsistent databases. In *PODS*, volume 99, pages 68–79. Citeseer.

Arenas, M., Gottlob, G., and Pieris, A. (2014). Expressive languages for querying the semantic web. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 14–26. ACM.

Artale, A., Calvanese, D., Kontchakov, R., and Zakharyaschev, M. (2009). The dl-lite family and relations. *Journal of artificial intelligence research*, 36:1–69.

Baget, J.-F., Leclère, M., Mugnier, M.-L., and Salvat, E. (2009). Extending decidable cases for rules with existential variables. In *Twenty-First International Joint Conference on Artificial Intelligence*.

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

Baget, J.-F., Mugnier, M.-L., Rudolph, S., and Thomazo, M. (2011b). Walking the complexity lines for generalized guarded existential rules. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

Beeri, C. and Ramakrishnan, R. (1991). On the power of magic. *The journal of logic programming*, 10(3-4):255–299.

Beeri, C. and Vardi, M. Y. (1981). The implication problem for data dependencies. In *International Colloquium on Automata, Languages, and Programming*, pages 73–85. Springer.

Benedikt, M., Konstantinidis, G., Mecca, G., Motik, B., Papotti, P., Santoro, D., and Tsamoura, E. (2017). Benchmarking the chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 37–52. ACM.

Benedikt, M., Leblay, J., and Tsamoura, E. (2015). Querying with access patterns and integrity constraints. *Proceedings of the VLDB Endowment*, 8(6):690–701.

Berners-Lee, T., Hendler, J., Lassila, O., et al. (2001). The semantic web. *Scientific american*, 284(5):28–37.

Bertossi, L. (2011). Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5):1–121.

Bertossi, L. and Milani, M. (2018). Ontological multidimensional data models and contextual data quality. *Journal of Data and Information Quality (JDIQ)*, 9(3):14.

Bonifati, A., Ileana, I., and Linardi, M. (2016). Functional dependencies unleashed for scalable data exchange. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, page 2. ACM.

Borgida, A. and Mylopoulos, J. (2004). Data semantics revisited. In *International Workshop on Semantic Web and Databases*, pages 9–26. Springer.

Calì, A., Gottlob, G., and Kifer, M. (2013). Taming the infinite chase: Query answering under expressive relational constraints. *Journal of Artificial Intelligence Research*, 48:115–174.

Calì, A., Gottlob, G., and Lukasiewicz, T. (2009). Datalog±: a unified approach to ontologies and integrity constraints. In *Proceedings of the 12th International*

*Conference on Database Theory*, pages 14–30. ACM.

Calì, A., Gottlob, G., and Lukasiewicz, T. (2012a). A general datalog-based framework for tractable query answering over ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 14:57–83.

Calì, A., Gottlob, G., Lukasiewicz, T., and Pieris, A. (2011). A logical toolbox for ontological reasoning. *ACM Sigmod Record*, 40(3):5–14.

Calì, A., Gottlob, G., Orsi, G., and Pieris, A. (2012b). Querying uml class diagrams. In *International Conference on Foundations of Software Science and Computational Structures*, pages 1–25. Springer.

Calì, A., Gottlob, G., and Pieris, A. (2010). Advanced processing for ontological queries. *Proceedings of the VLDB Endowment*, 3(1-2):554–565.

Calí, A., Gottlob, G., and Pieris, A. (2012). Ontological query answering under expressive entity–relationship schemata. *Information Systems*, 37(4):320–335.

Cali, A., Gottlob, G., and Pieris, A. (2012). Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence*, 193:87–128.

Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic programming and databases*. Springer Science & Business Media.

Chandrasekaran, B., Josephson, J. R., and Benjamins, V. R. (1999). What are ontologies, and why do we need them? *IEEE Intelligent systems*, 14(1):20–26.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

Deutsch, A., Nash, A., and Remmel, J. (2008). The chase revisited. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 149–158. ACM.

Fagin, R., Kolaitis, P. G., Miller, R. J., and Popa, L. (2005). Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124.

Geerts, F., Mecca, G., Papotti, P., and Santoro, D. (2013). The llunatic data-cleaning framework. *Proceedings of the VLDB Endowment*, 6(9):625–636.

Geerts, F., Mecca, G., Papotti, P., and Santoro, D. (2014a). Mapping and cleaning. In *2014 IEEE 30th International Conference on Data Engineering*, pages 232–243. IEEE.

Geerts, F., Mecca, G., Papotti, P., and Santoro, D. (2014b). That's all folks!: llunatic goes open source. *Proceedings of the VLDB Endowment*, 7(13):1565–1568.

Gottlob, G., Orsi, G., and Pieris, A. (2014). Query rewriting and optimization for ontological databases. *ACM Trans. Database Syst.*, 39(3):25:1–25:46.

Harel, D. and Rumpe, B. (2004). Meaningful modeling: what's the semantics of" semantics"? *Computer*, 37(10):64–72.

Hitzler, P., Krotzsch, M., and Rudolph, S. (2009). *Foundations of semantic web technologies*. Chapman and Hall/CRC.

Johnson, D. S. and Klug, A. (1984). Testing containment of conjunctive queries under functional and inclusion dependencies. *Journal of Computer and system Sciences*, 28(1):167–189.

Klein, G., Rowe, S., and Décamps, R. (2001). Jflex-the fast scanner generator for java. *URL: http://www. jflex. de.*

Kolaitis, P. G., Panttaja, J., and Tan, W.-C. (2006). The complexity of data exchange. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 30–39, New York, NY, USA. ACM.

Krötzsch, M. and Rudolph, S. (2011). Extending decidable existential rules by joining acyclicity and guardedness. In *Twenty-Second International Joint Conference on Artificial Intelligence*.

Lenzerini, M. (2011). Ontology-based data management. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 5–6. ACM.

Leone, N., Manna, M., Terracina, G., and Veltri, P. (2012). Efficiently computable datalog$^{\exists}$ programs. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning.*

Maedche, A., Motik, B., Stojanovic, L., Studer, R., and Volz, R. (2003). Ontologies for enterprise knowledge management. *IEEE Intelligent Systems*, 18(2):26–33.

Maier, D., Mendelzon, A. O., and Sagiv, Y. (1979). Testing implications of data dependencies. *ACM Transactions on Database Systems (TODS)*, 4(4):455–469.

Marnette, B. (2009). Generalized schema-mappings: from termination to tractability. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 13–22. ACM.

Meier, M., Schmidt, M., and Lausen, G. (2009). On chase termination beyond stratification. *Proceedings of the VLDB Endowment*, 2(1):970–981.

Milani, M. and Bertossi, L. (2016). Extending weakly-sticky datalog$^{\pm}$: Query-answering tractability and optimizations. In *International Conference on Web Reasoning and Rule Systems*, pages 128–143. Springer.

Milani, M., Bertossi, L., and Ariyan, S. (2014). Extending contexts with ontologies for multidimensional data quality assessment. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 242–247. IEEE.

Pichler, R. and Savenkov, V. (2009). data exchange modeling tool. *Proceedings of the VLDB Endowment*, 2(2):1606–1609.

Shadbolt, N., Berners-Lee, T., and Hall, W. (2006). The semantic web revisited. *IEEE intelligent systems*, 21(3):96–101.

# Appendix A

# A Weakly-Acyclic Program and its Queries

## 1. Program

The following program is extracted from ChaseBench [Benedikt et al., 2017] literature. For the ease of representation, we changed the variables' names.

$$\exists z_1\, prescription(x, y, w, z_1) \leftarrow treatment(x, y, l, w, m), physician(w, n, o, k).$$

$$\exists z_2\, doctor(x, y, w, m, z_2) \leftarrow treatment(o, s, n, x, v), physician(x, y, w, l).$$

$$\exists z_3\, prescription(x, y, w, z_3) \leftarrow medprescription(x, y, w, l, m, n).$$

$$\exists z_4, z_5\, doctor(x, y, w, z_4, z_5) \leftarrow medprescription(l, m, x, y, w, n).$$

$$targethospital(x, y, w, m, n) \leftarrow hospital(x, y, w, m, n).$$

## 2. Queries over the 10K Instance

$\mathcal{Q}_1(x): \ \exists z_1, ..., z_{10}\ doctor(z_1, z_2, x, z_3, z_4),$

$targethospital(z_2, z_5, z_6, z_7, z_8), prescription(z_9, z_{10}, z_1, z_4).$

$\mathcal{Q}_2(x, y, w): \ \exists z_1, ..., z_9 targethospital(z_1, z_2, w, z_3, z_4),$

$doctor(z_5, x, z_2, z_6, z_7), prescription(z_8, y, z_5, z_9).$

$\mathcal{Q}_3(x, y): \ \exists z_1, ..., z_{10}\ doctor(z_1, z_2, z_3, z_4, z_5),$

$targethospital(y, z_6, z_7, z_1, z_8), prescription(x, z_9, z_1, z_{10}).$

$\mathcal{Q}_4(x, y): \ \exists z_1, ..., z_{10}\ prescription(x, z_1, z_2, z_3),$

$targethospital(z_4, y, z_5, z_2, z_6), doctor(z_7, z_8, y, z_9, z_{10}).$

$\mathcal{Q}_5(x, y, w): \ \exists z_1, ..., z_9\ prescription(z_1, z_2, z_3, z_4),$

$targethospital(y, z_5, z_6, z_3, z_7), doctor(w, z_8, x, z_6, z_9).$

$\mathcal{Q}_6(x, y): \ \exists z_1, ..., z_{10}\ doctor(z_1, z_2, z_3, z_4, z_5),$

$targethospital(z_2, z_6, z_7, z_8, z_9), prescription(x, y, z_1, z_{10}).$

$\mathcal{Q}_7(x, y): \ \exists z_1, ..., z_{10}\ prescription(z_1, z_2, y, z_3),$

$doctor(y, x, z_4, z_5, z_6), targethospital(x, z_7, z_8, z_9, z_{10}).$

$\mathcal{Q}_8(x, y, w, l, m): \ \exists z_1, ..., z_6\ doctor(z_1, l, w, y, z_2),$

$targethospital(m, w, \text{`}HH65795\text{'}, z_3, z_4), prescription(x, z_5, z_3, z_6).$

$\mathcal{Q}_9(x, y, w, l, m): \ \exists z_1, ..., z_6\ targethospital(z_1, w, \text{`}HH30727\text{'}, x, z_2),$

$doctor(x, z_3, z_4, m, z_5), prescription(y, l, x, z_6).$

## 3. Queries over the 100K Instance

$\mathcal{Q}_1(x):\ \exists z_1,...,z_{10}\ doctor(z_1,z_2,x,z_3,z_4),$

$targethospital(z_2,z_5,z_6,z_7,z_8), prescription(z_9,z_{10},z_1,z_4).$

$\mathcal{Q}_2(x,y,w):\ \exists z_1,...,z_9 targethospital(z_1,z_2,w,z_3,z_4),$

$doctor(z_5,x,z_2,z_6,z_7), prescription(z_8,y,z_5,z_9).$

$\mathcal{Q}_3(x,y):\ \exists z_1,...,z_{10}\ doctor(z_1,z_2,z_3,z_4,z_5),$

$targethospital(y,z_6,z_7,z_1,z_8), prescription(x,z_9,z_1,z_{10}).$

$\mathcal{Q}_4(x,y):\ \exists z_1,...,z_{10}\ prescription(x,z_1,z_2,z_3),$

$targethospital(z_4,y,z_5,z_2,z_6), doctor(z_7,z_8,y,z_9,z_{10}).$

$\mathcal{Q}_5(x,y,w):\ \exists z_1,...,z_9\ prescription(z_1,z_2,z_3,z_4),$

$targethospital(y,z_5,z_6,z_3,z_7), doctor(w,z_8,x,z_6,z_9).$

$\mathcal{Q}_6(x,y):\ \exists z_1,...,z_{10}\ doctor(z_1,z_2,z_3,z_4,z_5),$

$targethospital(z_2,z_6,z_7,z_8,z_9), prescription(x,y,z_1,z_{10}).$

$\mathcal{Q}_7(x,y):\ \exists z_1,...,z_{10}\ prescription(z_1,z_2,y,z_3),$

$doctor(y,x,z_4,z_5,z_6), targethospital(x,z_7,z_8,z_9,z_{10}).$

$\mathcal{Q}_8(x,y,w,l,m):\ \exists z_1,...,z_6\ doctor(z_1,l,w,y,z_2),$

$targethospital(m,w,\text{`}MPTR76804\text{'},z_3,z_4), prescription(x,z_5,z_3,z_6).$

$\mathcal{Q}_9(x,y,w,l,m):\ \exists z_1,...,z_6\ targethospital(z_1,w,\text{`}MMST63178\text{'},x,z_2),$

$doctor(x,z_3,z_4,m,z_5), prescription(y,l,x,z_6).$

# Appendix B

# A Jointly-Weakly-Sticky Program and its Queries

## 1. Program

$$
\begin{aligned}
\exists z_1\ Prescription(x,y,w,z_1) &\leftarrow Treatment(x,y,l,w,m), physician(w,n,o,k).\\
\exists z_2\ Doctor(x,y,w,m,z_2) &\leftarrow Treatment(o,s,n,x,v), physician(x,y,w,l).\\
\exists z_3\ Prescription(x,y,w,z_3) &\leftarrow MedPrescription(x,y,w,l,m,n).\\
\exists z_4, z_5\ Doctor(x,y,w,z_4,z_5) &\leftarrow MedPrescription(l,m,x,y,w,n).\\
TargetHospital(x,y,w,m,n) &\leftarrow Hospital(x,y,w,m,n).\\
R(x,y) &\leftarrow TargetHospital(y,x,w,m,n).\\
R(x,y) &\leftarrow TargetHospital(x,w,y,m,n).\\
U(x) &\leftarrow Doctor(y,x,y,w,m).\\
P(x,w) &\leftarrow R(x,y), R(y,w).\\
\exists z_6 R(y,z_6) &\leftarrow U(y), R(x,y).
\end{aligned}
$$

## 2. Queries

$$\mathcal{Q}_1(x,y):\ \exists z_1,z_2,z_3\ Treatment(x,y,z_1,z_2,z_3).$$

$$\mathcal{Q}_2(x,y):\ \exists z_1,z_2,z_3\ Doctor(z_1,x,y,z_2,z_3).$$

$$\mathcal{Q}_3:\ \exists z_1\ P(\text{`}Er2508\text{'},z_1).$$

$$\mathcal{Q}_4(y):\ \exists z_1,z_2\ R(z_1,y),R(y,z_2),U(y).$$