MULTIDIMENSIONAL ONTOLOGIES FOR CONTEXTUAL QUALITY DATA SPECIFICATION AND EXTRACTION

by

Mostafa Milani

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY IN CONPUTER SCIENCE

at

CARLETON UNIVERSITY

Ottawa, Ontario
January 2017,

Table of Contents

List of Tables

List of Figures

Abstract

Data quality assessment and data cleaning are context-dependent activities. Starting from this observation, in previous work a context model for the assessment of the quality of a database was proposed. A context takes the form of a possibly virtual database or a data integration system into which the database under assessment is mapped, for additional analysis, processing, and quality data extraction. In this work, we extend contexts with dimensions, and by doing so, multidimensional data quality assessment becomes possible. At the core of multidimensional contexts we introduce ontologies with provably good properties in terms of *query answering (QA)*. We use the ontologies to represent dimension hierarchies, dimensional constraints, dimensional rules, and specifying quality data. Query answering relies on and triggers dimensional navigation, and becomes an important tool for the extraction of quality data.

We introduce and investigate an *ontological-multidimensional (OMD) data model* for which the aforementioned multidimensional ontology is a particular case. The OMD model extends the traditional multidimensional data model, embedding it into a Datalog$^{\pm}$ ontology. The ontology allows for the introduction of generalized fact-tables, called *categorical relations*, which may be incomplete and associated to categories at arbitrary levels of the dimensions. The *dimensional rules* in the ontology are represented as Datalog$^{\pm}$ rules, and they enable *dimensional navigation* while propagating data between different dimension levels, for data completion where data is missing. The *dimensional constraints* are semantic conditions that have to be satisfied and are represented as Datalog$^{\pm}$ constraints. It turns out that the ontologies created according to the OMD model correspond to *weakly-sticky (WS)* programs, for which tractability of conjunctive QA is guaranteed. We analyse the representational and computational properties of the OMD model, we investigate QA and optimization for the *WS* programs which was only partly studied in the literature.

Abbreviations

BCQ boolean conjunctive query. 16, 23

CQ conjunctive query. 16, 22

CWA closed world assumption. 23

DL description logic. 55, 64

EDG existential dependency graph. 33

egd equality-generating dependency. 14, 23

FD functional dependency. 23

FO first-order. 22

GSCh generalized-stickiness of the chase. 103

HM Hurtado-Mendelzon. 4, 50

IC integrity constraint. 23

JA joint-acyclic. 33

MD multidimensional. 6

NC negative constraint. 14, 23

OBDA ontology-based data access. 6, 64

OMD ontological-multidimensional. 6, 70

OWA open world assumption. 25

SCh sticky-chase. 35

tgd tuple-generating dependency. 14, 23

WA weakly-acyclic. 15, 32

WS weakly-sticky. 40

Notations

D Database (instance). 3, 22

$\mathfrak{C}$ Context. 3, 55

$\mathcal{R}$ Database schema. 10, 22

$\mathcal{M}$ Multidimensional ontology. 10, 71

I Instance. 10, 22

$\mathcal{C}$ Constants. 22

$\mathcal{N}$ Nulls. 22

$\mathcal{V}$ Variables. 22

$\mathcal{Q}(I)$ The set of answers to the query $\mathcal{Q}$ over the instance I. 23

$head(\sigma)$ The head atom of the tgd σ. 23

$body(\sigma)$ The set of body atoms of σ. 23

$\perp$ The boolean constant symbol that is always false. 23

Π Program. 24

Π^R Program rules. 24

$\exists$-variable Existentially quantified variable. 24

$Mod(\Pi)$ Models of the program Π. 25

$ans(\mathcal{Q}, \Pi)$ The set of answers to the query $\mathcal{Q}$ over the program Π. 25

$I \xrightarrow{\sigma,\theta} I'$ Tgd chase step. 26

$level(A)$ The level of the atom a in the chase. 27

$chase(\Pi)$ The chase instance of the program Π. 28

$chase^k(\Pi)$ The set of atoms in the chase instance with level up to k. 28

$chase^{[k]}(\Pi)$ The chase instance after k tgd chase steps. 28

$\xrightarrow{\Pi}^*$ The derivation relation of the program Π. 29

$\forall$-variable Universally quantified variable. 31

$rank(p)$ The rank of the position p. 32

$\pi_F(\Pi)$ The set of finite-rank positions of the program Π. 32

B_x The set of positions where x appears in the body of a rule. 33

H_x The set of positions where x appears in the head of a rule. 33

T_z The set of target positions of the $\exists$-variable z. 33

Acknowledgements

I would like to express my sincerest appreciation to Professor Dr. Leopoldo Bertossi, whose thoughtful consideration, guidance, and constructive criticism has been invaluable. Without his support and inspiration during the most critical period of my PhD journey, I would not have been able to accomplish this study.

My sincere thanks also go to Professor Dr. Andrea Cali for his help and support on this thesis. I also would like to thank my examiners Professor Dr. Kevin Cheung, Professor Dr. Amy Felty, Professor Dr. Yuhong Guo, Professor Dr. Letizia Tanca for the fruitful conversation we had during the examination, and also for their constructive comments and suggestions.

I have been very fortune to have the constant support of my family; special thanks to my parents for all of the sacrifices that they've made on my behalf. I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was funded by an RA through the NSERC Strategic Network on Business Intelligence (BIN).

Chapter 1

Introduction

The notion of data quality has different definitions in various areas of computer science. In knowledge representation and also data management, data quality refers to the degree to which the data fits or fulfills a form of usage [Batini & Scannapieco, 2006; Herzog et al., 2009]. Problems related to the quality of data in data management systems have been increasingly evident for organizations, companies and businesses. Specifically, decision making based on data of poor quality costs them hugely everyday. Those organizations, companies and businesses that invest in managing and improving data quality experience tangible and intangible benefits [Batini & Scannapieco, 2006; Eckerson, 2002; Redman, 1998].

Data quality in data management has several dimensions (also called data quality attributes, or aspects), most importantly, among other dimensions [Batini & Scannapieco, 2006]: (1) Consistency refers to the validity and integrity of data representing real-world entities typically identified as satisfaction of integrity constraints, (2) Currency (timeliness) aims to identify the current values of entities represented by tuples in a (possibly stale) database, and to answer queries with the current values, (3) Accuracy refers to the closeness of values in a database to the true values for the entities that the data in the database represents, and (4) Completeness is characterized in terms of the presence/absence of values.

1.1 Context and Data Quality

Independently from the quality dimension we may consider, *data quality assessment and data cleaning are context-dependent activities.* This is our starting point, and the one leading our research. In more concrete terms, the quality of data has to be assessed with some form of contextual knowledge; and whatever we do with the data in the direction of data cleaning also depends on contextual knowledge. For example, contextual knowledge can tell us if the data we have is incomplete or inconsistent. In the latter case, the context knowledge is provided by explicit semantic constraints.

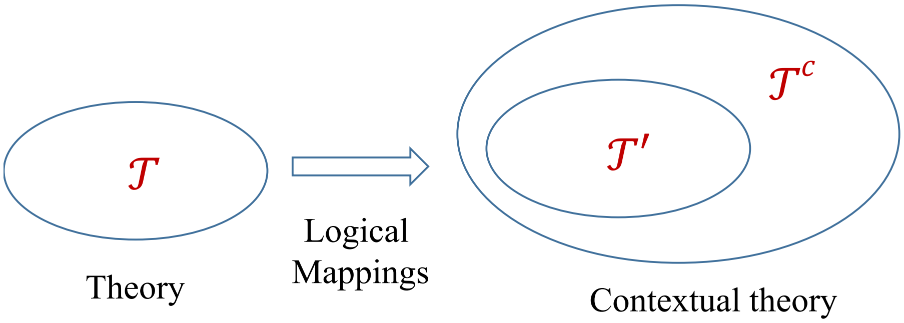


Figure 1.1: Embedding into a contextual theory

In order to address contextual data quality issues, we need a formal model of context. In very general terms, the big picture is as in Figure 1.1. A database can be seen as a logical theory, $\mathcal{T}$, and a context for it, as another logical theory, $\mathcal{T}^c$, into which $\mathcal{T}$ is mapped by means of a set of *logical mappings*. This embedding of $\mathcal{T}$ into $\mathcal{T}'$ could be seen as an *interpretation* of $\mathcal{T}$ in $\mathcal{T}^c$.[1] The additional knowledge in $\mathcal{T}^c$ may be used as extra knowledge about $\mathcal{T}$, as a logical extension of $\mathcal{T}$. For example, $\mathcal{T}^c$ can provide additional knowledge about predicates in $\mathcal{T}$, such as additional semantic constraints on elements of $\mathcal{T}$ (or their images in $\mathcal{T}^c$) or extensions of their definitions.

[1] Interpretations between logical theories have been investigated in mathematical logic [Enderton, 2001, sec. 2.7] and used, e.g. to obtain (un)decidability results [Rabin, 1965].

In this way, $\mathcal{T}^c$ conveys more semantics or meaning about $\mathcal{T}$, contributing to *making more sense* of $\mathcal{T}$'s elements. $\mathcal{T}^c$ may also contain additional knowledge, e.g. data and logical rules, that can be used for further processing or using knowledge in $\mathcal{T}$. The embedding of $\mathcal{T}$ into $\mathcal{T}^c$ can be achieved via predicates in common or, more complex logical formulas.

In this work, building upon and considerably extending the framework in [Bertossi et al., 2011a, 2016], context-based data quality assessment, data quality extraction and data cleaning on a relational database D are approached by creating a context model where D is the theory $\mathcal{T}$ above (it could be expressed a logical theory [Reiter, 1984]), the theory $\mathcal{T}^c$ is a (logical) ontology $\mathfrak{C}$; and, considering that we are using theories around data, the mappings can be logical mappings as used in virtual data integration [Lenzerini, 2002] or data exchange [Barcelo, 2009]. In this work, the mappings turn out to be quite simple: The ontology contains, among other predicates, *nicknames* for the predicates in D (i.e. copies of them), so that each predicate R in D is directly mapped to its copy R' in $\mathfrak{C}$.

Once the data in D is mapped into $\mathfrak{C}$, the extra elements in it can be used to define alternative versions of D, in our case, *clean or quality versions*, D^q, of D in terms of data quality. The data quality criteria are imposed within $\mathfrak{C}$. This may determine a class of possible quality versions of D, virtual or material. The existence of several quality versions reflects the uncertainty that emerges from not having in D fully quality data.

The whole class, $\mathcal{D}^q$, of quality versions of D determines or characterizes the quality data in D, through what is *certain* with respect to $\mathcal{D}^q$. One way to go in this direction consists in keeping only the data that are found in the intersection of all the instances in $\mathcal{D}^q$. A more relaxed alternative consists in considering as quality

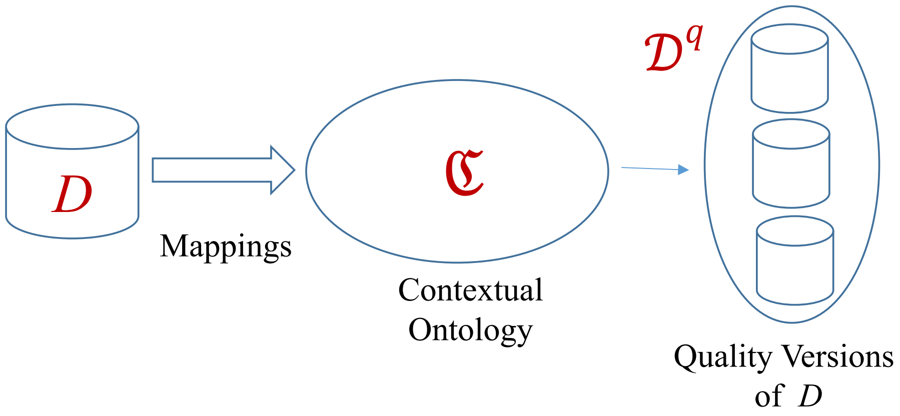


Figure 1.2: Contextual ontology and quality versions

data those that are obtained as *certain answers* to queries posed to D, but answered through $\mathcal{D}^q$: The query is posed to each of the instances in $\mathcal{D}^q$ (which essentially have the same schema as D), but only those answers that are shared by those instances are considered to be certain [Imielinski & Lipski, 1984].

The main question is about the kind of contextual ontologies that are appropriate for our tasks. There are several basic conditions to satisfy. First of all, $\mathfrak{C}$ has to be written in a logical language. As a theory it has to be expressive enough, but not too much so that computational problems, such as (quality) data extraction via queries becomes intractable, if not impossible. It also has to combine well with relational data. And, as we emphasize and exploit in our work, it has to allow for the representation and use of *dimensions of data*, as found in multidimensional databases and data warehouses [Jensen et al., 2010]. Dimensions are almost essential elements of contexts, in general, and crucial if we want to analyze data from different perspectives or points of view.

The language of choice for the contextual ontologies will be Datalog$^{\pm}$ [Calì et al., 2010b]. As an extension of Datalog, a declarative query language for relational databases [Ceri et al., 1990], it provides perfect extensions of relational data by means of expressive rules and constraints. Certain classes of Datalog$^\pm$programs have non-trivial expressive power and good computational properties at the same time. One of those good classes is that of *weakly-sticky* Datalog$^\pm$ [Calì et al., 2012c]. Programs in that class allow us to represent a logic-based extension of the *Hurtado-Mendelzon (HM) multidimensional data model* [Hurtado & Mendelzon, 2002; Hurtado et al., 2005], which allows us to bring data dimensions into contexts.

The main components of an HM model are *dimensions* and *fact-tables*. A dimension is represented by a dimension schema, i.e. a hierarchy (more generally, a lattice) of category names, plus a dimension instance that assigns (data) members to the categories.

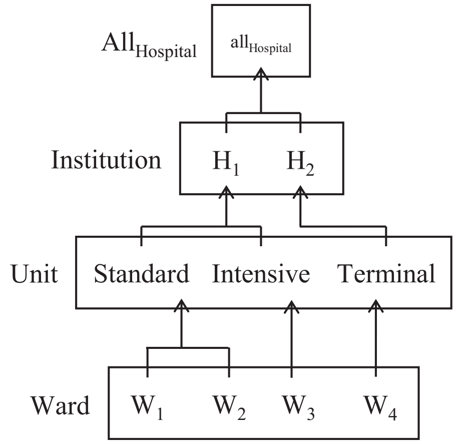


Figure 1.3: The Hospital dimension.

Example 1.1.1 Figure 1.3 shows the Hospital dimension with a hierarchy of category names (e.g. *Unit*) and a parallel hierarchy of data elements in the categories (e.g. *Standard*). The bottom category of Hospital is *Ward*, with four data elements. In every dimension there is always a single top category, *All*, with a single element, *all*. □

The HM model has some limitations when we want to go beyond the usual applications to DWHs and OLAP, in particular towards context modeling. In the HM model, we find the assumption that data is complete, which may not make sense in some applications. Furthermore, relational tables are linked to the dimensions as fact-tables, or possibly, as tables representing materialized aggregate data at higher dimension levels. In some applications we may find convenient to have tables directly and initially linked to arbitrary dimension levels, and not only in relation to numerical data. The HM model considers some semantic conditions, such as *homogeneity* and *strictness* [Hurtado et al., 2005] that restrict the hierarchy structure, but do not say much about data-value dependencies between different categories. Another important limitation of the HM model, at least for the applications we have in mind, is the lack of logical integration and simultaneous representation of the metadata (the schema) and the actual dimension and table contents (the instances).

We overcome these limitations through the use of *multidimensional (MD) ontologies*, with a logical layer containing formulas representing metadata (or a multidimensional conceptual model); and a data layer representing different kinds of relations, and at different levels of the hierarchies. Expressive semantic constraints are included as logical formulas in the ontology. This creates a scenario that is similar to that of ontology-based data access (OBDA) [Poggi et al., 2008]. The MD ontologies that we propose and investigate in this work can be used for data modeling, reasoning, and QA. They are the basis for our proposed *ontological-multidimensional (OMD) data model*.

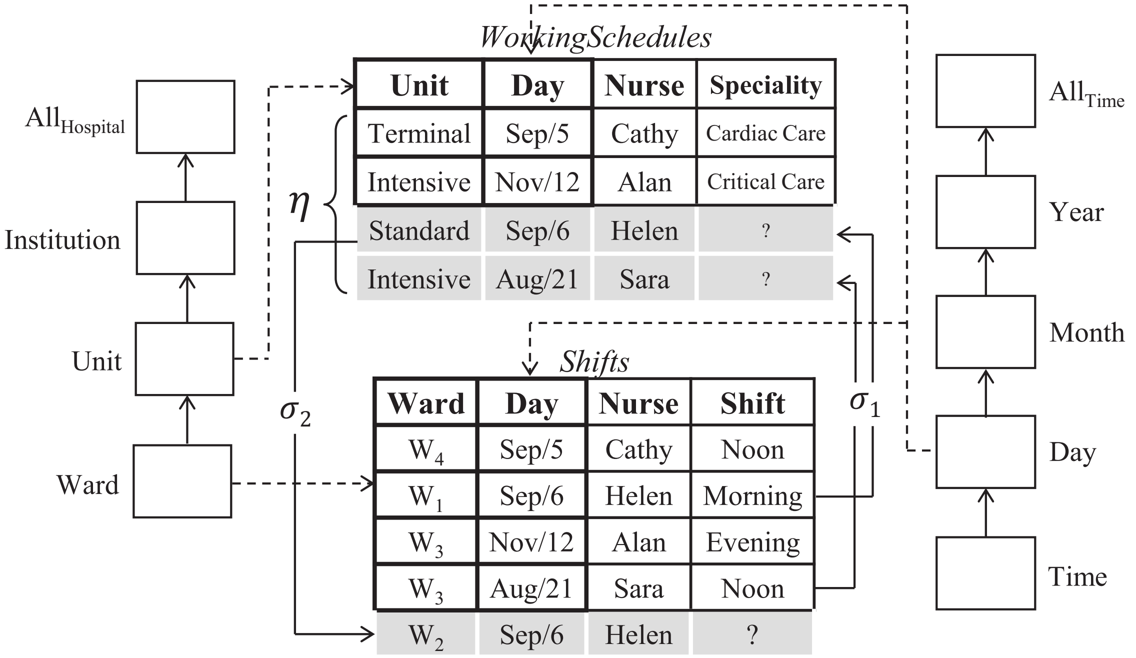
Now we give a few more introductory details about the general ingredients of OMD models. They can be used to produce particular ontological models depending on the application domain. OMD models allow for the introduction of *categorical relations* that are associated to categories in different dimensions, at arbitrary levels of their hierarchies. However, a categorical relation may be linked to a single dimension.

Our categorical relations may be incomplete [Abiteboul et al., 1995; Imielinski & Lipski, 1984]. Intuitively, data will be completed by data propagation from other categorical relations through navigation along the dimension hierarchies. For this there are data-creating *rules*, and also *constraints* that regulate data propagation. Hence, OMD models include *dimensional rules* and *dimensional constraints*. The former are intended to be used for data completion, to generate data through their enforcement via *dimensional navigation*. The latter can be seen as *dimensional integrity constraints* on categorical relations. They are typically *denial constraints* that forbid certain (positive) combinations of values, in particular, joins.

Example 1.1.2 An OMD model is shown in Figure 1.4. It has two dimensions, Hospital and Time. Each of them has a unary relation for each of its categories, e.g. *Unit* for the second category from the bottom of the Hospital dimension. Dimensions also have a binary relation for each child-parent pair of categories, e.g. *WardUnit*, representing the data associations between the "child category" *Ward* and its "parent category" *Unit*. For example, according to Figure 1.3, $(\mathsf{W}_1, \mathsf{standard}) \in \mathit{WardUnit}$. Similarly, *DayMonth* is a child-parent binary relation for the Time dimension.

In addition to all these purely "dimensional" data, we find in the middle of Figure 1.3, two relational tables with data (the non-shaded tuples in them), *WorkingSchedules* and *Shifts*. They are categorical relations that store schedules of nurses in units, and shifts of nurses in wards, respectively. Attribute *Unit* in the categorical relation *WorkingSchedules* takes values from this *Unit* category, which makes the former a *categorical attribute*. Similarly, the *Day* attribute in this relation is categorical, but *Nurse* and *Speciality* are not. *We make a difference between the two kinds of attributes by using a semi-colon to separate them*, as in *WorkingSchedules(Unit,Day;Nurse,Speciality)*.

The model is also endowed with the two *dimensional rules*, σ_1 and σ_2, in (1.1) and (1.2), resp., and a *dimensional constraint*, η, in (1.3), which contains two constants.



WorkingSchedules

Unit	Day	Nurse	Speciality
Terminal	Sep/5	Cathy	Cardiac Care
Intensive	Nov/12	Alan	Critical Care
Standard	Sep/6	Helen	?
Intensive	Aug/21	Sara	?

Shifts

Ward	Day	Nurse	Shift
W_4	Sep/5	Cathy	Noon
W_1	Sep/6	Helen	Morning
W_3	Nov/12	Alan	Evening
W_3	Aug/21	Sara	Noon
W_2	Sep/6	Helen	?

$$\sigma_1\colon\ \mathit{Shifts}(w,d;n,s),\ \mathit{WardUnit}(w,u)\ \rightarrow\ \exists t\ \mathit{WorkingSchedules}(u,d;n,t). \quad (1.1)$$

$$\sigma_2\colon\ \mathit{WorkingSchedules}(u,d;n,t),\ \mathit{WardUnit}(w,u)\ \rightarrow\ \exists s\ \mathit{Shifts}(w,d;n,s). \quad (1.2)$$

$$\eta\colon\ \mathit{WorkingSchedules}(\mathsf{intensive},d;n,s),\ \mathit{DayMonth}(d,\mathsf{jan})\ \rightarrow\ \bot. \quad (1.3)$$

Figure 1.4: An OMD model with categorical relations, dimensional rules, and constraints

Their role and use will be described throughout the example.

Now, a query to *WorkingSchedules* asks about unit/day schedules for *Helen*. The extensional data for *WorkingSchedules* (again, the non-shaded tuples) do not show any entry for *Helen*, which could be due to the incompleteness of the given table. However, it could be the case that missing data could be obtained through the logical relationships between *WorkingSchedules* and the dimensional relation *Shifts*.

Actually, the dimensional rule σ_1 in (1.1) tells us: *"If a nurse has shifts in a ward on a specific day, he/she has a working schedule in the unit of that ward on the same day"*. (It is shown as a labeled arrow on the right-hand-side of the central tables in Figure 1.4.)

Since the contents of relation *WardUnit* is given by the Hospital dimension (as in Figure 1.3), rule σ_1 enables upward navigation and data movement from the *Ward* to the *Unit* category levels, completing data in *WorkingSchedules* through the use of data in *Shifts*. This is possible due to the join in its body that involves the child-parent relation *WardUnit*. Since $(\mathsf{W}_1, \mathsf{standard}) \in \mathit{WardUnit}$, the second tuple in *Shifts* implies that *Helen* works in the *Standard* unit on *Sep/6*. On this basis, the third, shaded tuple in *WorkingSchedules* is created. Notice that we may not know some attribute values, as in this case, for attribute *Speciality*, whose variable is existentially quantified in (1.1).

Elaborating on this example, let us now consider the dimension constraint η, imposed on dimension Time and the relation *WorkingSchedules* linked to its *Day* category. It tells us (possibly because the *Intensive* care unit was closed during January) that: *"No personnel was working in the Intensive care unit during January"*. (Shown on the left-hand-side of Figure 1.4.)

For the ontology to be consistent with respect to the *dimensional constraint*, η, the constraint is expected to be satisfied by the combination of the extensional data for *WorkingSchedules* (non-shaded tuples in Figure 1.4) and the intensional data, i.e. tuples generated by σ_1 (shown in Figure 1.4 as shaded tuples). In this example, η is satisfied. It involves the Hospital and Time dimensions. More specifically, checking η requires upward navigation through the Time dimension. This is because *January*, appearing in η, belongs to the *Month* category, and *WorkingSchedules* is linked to the *Day* category. Also, the Hospital dimension implicitly affects η, because it has to be satisfied by the intentional data generated by σ_1 through upward navigation from *Ward* to *Unit*. □

We have shown that upward dimensional navigation may be used for QA. This is enabled by the "upward" rule (1.1), that propagates data from *Shifts* at the lower level of the *Ward* category to *WorkingSchedules*, which is at the higher level of the *Unit* category. Downward navigation is also supported by the OMD model, and can be used for QA and data propagation (or generation) at lower levels in a dimensional hierarchy. This is shown in the next example, through the use of the "downward" rule (1.2), which allows for the propagation of data from *WorkingSchedules* (at the level of the *Unit* category) to *Shifts* (at lower level of the *Ward* category).

Example 1.1.3 (ex. 1.1.2 cont.) Now, a query in terms of the *Shifts* predicate asks for the wards where *Helen* was working on *Sep/6*. The only answer directly provided by the initial extensional data in *Shifts* is W_1 (the second tuple in *Shifts*). However, rule σ_2 in (1.2), which expresses an institutional guideline stating that *"If a nurse works in a unit on a specific day, he/she has shifts in every ward of that unit on the same day"*, can be used to obtain additional answers.

Actually, applying this rule with the third tuple in *WorkingSchedules* and the pairs $(\mathsf{W}_1, \mathsf{standard})$, $(\mathsf{W}_2, \mathsf{standard})$ in *WardUnit*, we obtain that *Helen* has shifts in both W_1 and W_2 on *Sep/6* (cf. the fifth, shaded tuple in *Shifts* in Figure 1.4, again showing an unknown value due to the existential quantifier in σ_2). The new answer, W_2, has been obtained by downward navigation from the *Standard* unit to its wards.

□

1.2 The OMD Model and Data Quality

The ontologies in the OMD model can be used to support the specification and extraction of quality data, as shown in Figure 1.5. In this framework, D is a database

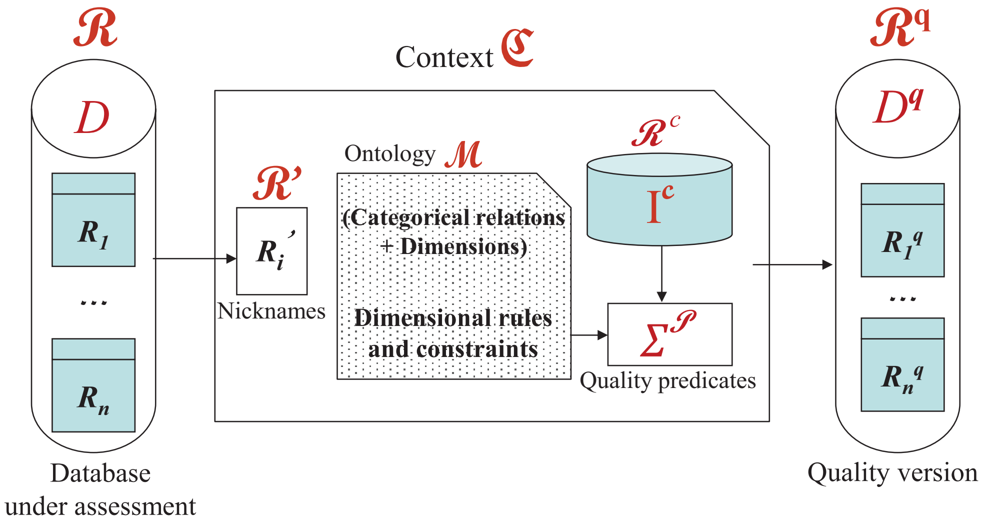


Figure 1.5: A multidimensional context

instance for a relational schema $\mathcal{R}= \{R_1, ..., R_n\}$ that is under quality data specification and extraction. Context $\mathfrak{C}$ contains: (a) $\mathcal{R}' = \{R_1', ..., R_n'\}$, a copy of schema $\mathcal{R}$ with nicknames (the primed versions) for the elements of the latter. They are used to bring data from D into $\mathfrak{C}$. (b) A core MD ontology $\mathcal{M}$ (as described in Section 4.1) that is used for dimensional data manipulation. (c) Application-dependent *quality predicates* $\mathcal{P}$ (defined by $\Sigma^{\mathcal{P}}$, cf. Figure 1.5) capturing data quality concerns. (d) A contextual relational schema $\mathcal{R}^c$, with an instance I^c, which contains materialized data at the contextual level context.

The quality predicates are defined by non-recursive Datalog rules in terms of the categorical predicates in $\mathcal{M}$, predicates in $\mathcal{R}^c$ and possibly built-in predicates.[2] The quality predicates in $\mathcal{P}$ are used to define quality versions, $R_1^q, ..., R_n^q$, for the corresponding relations in D, under quality assessment (cf. Figure 1.5, right). The quality predicate definitions may be based on *data quality guidelines* that are captured as rules or semantic constraints, both of which may refer to categorical attributes of

[2] We could use more expressive languages, but Datalog seems to be good enough for this task.

predicates in a core multidimensional ontology $\mathcal{M}$, without being part of $\mathcal{M}$. Rather, this "quality part" of the context comes on top of $\mathcal{M}$. We establish in Section 5.1 that under reasonable conditions on these extra definitions, the resulting extension of $\mathcal{M}$ still retains the good computational properties of the core ontology of the OMD model.

The process of data quality assessment and extraction is guided by query-answering, in the sense that the queries are rewritten using the quality predicates and the quality versions of the predicates under assessment. The rewritten queries are answered on the core OMD model, the nickname relations $R'_1, ..., R'_n$ and the relations in I^c.

Example 1.2.1 (ex. 1.1.3 cont.) The relational table *Temperatures* (Table 1.1) shows body temperatures of patients in an institution. A doctor wants to know *"The body temperatures of Tom Waits for August 21 taken around noon with a thermometer of brand B_1"* (as he expected). Possibly a nurse, unaware of this requirement, used a thermometer of brand B_2, storing the data in *Temperatures*. In this case, not all the temperature measurements in the table are up to the expected quality. However, table *Temperatures* alone does not discriminate between intended values (those taken with brand B_1) and the others.

For assessing the quality of the data in *Temperatures* according to the doctor's quality requirement, extra contextual information about the thermometers in use may help. In this case, the contextual information is in form of a *guideline* prescribing that: *"Nurses in*

Table 1.1: *Temperatures*

	Time	**Patient**	**Value**	**Nurse**
1	Sep/1-12:10	Tom Waits	38.2	Anna
2	Sep/6-11:50	Tom Waits	37.1	Helen
3	Nov/12-12:15	Tom Waits	37.7	Alan
4	Aug/21-12:00	Tom Waits	37.0	Sara
5	Sep/5-11:05	Lou Reed	37.5	Helen
6	Aug/21-12:15	Lou Reed	38.0	Sara

intensive care unit use thermometers of Brand B_1". It can be used for data quality assessment when combined with categorical table *WorkingSchedules*, for which the complete data is obtained in the MD ontology as explained in Example 1.1.2.

Table 1.2: *Temperatures*^{q}

	Time	Patient	Value	Nurse
1	Nov/12-12:15	Tom Waits	37.7	Alan
2	Aug/21-12:00	Tom Waits	37.0	Sara
3	Aug/21-12:15	Lou Reed	38.0	Sara

According to the guideline, it is now possible to conclude that the measurements taken by *Alan* and *Sara* were taken with the expected thermometer: these nurses were working in the intensive care unit (cf. *WorkingSchedules* in Figure 1.4), where according to the guideline they used thermometers of brand B_1. These "quality data" appear in relation *Temperatures*^{q} (Table 1.2), and the second tuple provides the answer to the doctor's query.

Notice that the second and the third tuples in *Temperatures*^{q} are obtained using the fact that *Sara* was in the intensive care unit on *Aug/21* (the last gray tuple in *WorkingSchedules*), which is generated by data completion and upward navigation from *Shifts* in the MD ontology.

In more formal terms, *Temperatures*^{q} is defined by the non-recursive Datalog rule:

$$\mathit{Temperatures}'(t,p,v,n),\ \mathit{TakenWithTherm}(t,n,\mathsf{b1}) \rightarrow \mathit{Temperatures}^q(t,p,v,n), \quad (1.4)$$

where *Temperatures*$'$ is the nickname in the context for *Temperatures*, and *TakenWithTherm* is a quality predicate that is defined by the Datalog rule:

$$\mathit{WorkingTimes}(\mathsf{intensive},t;n,y)\ \rightarrow\ \mathit{TakenWithTherm}(t,n,\mathsf{b1}). \quad (1.5)$$

Here, *WorkingTimes* is a categorical relation that contains the schedules in *WorkingSchedules* at the *Time* level rather than the *Day* level.

The doctor's query:

$$\mathcal{Q}(v) : \exists n\ \exists t\ (\mathit{Temperatures}(t,\mathsf{tom\ waits},v,n)\ \wedge\ \mathsf{aug\text{-}21/11\!:\!45} \leq t \leq \mathsf{aug\text{-}21/12\!:\!15})$$

is answered by: (a) replacing *Temperatures* with its quality version *Temperatures*^{q}, (b) unfolding the definitions of *Temperatures*^{q} and then *TakenWithTherm* using (1.4) and (1.5) resp., and (c) answering the obtained query over $\mathcal{M}$ and nickname relations with the data of D. □

1.3 Datalog$^\pm$ Representation of Multidimensional Ontologies

An OMD model has a semantics that is determined by its rules, constraints, and underlying extensional data. More precisely, dimensional rules in an OMD model act as *tuple-generating dependencies (tgds)* [Abiteboul et al., 1995], and are represented by *existential rules*, such as (1.1) and (1.2), with the syntax and semantics of Datalog$^\pm$[Calì et al., 2009, 2010b, 2011]. Datalog$^\pm$ is an extension of classical Datalog (hence the +) that allows existential rule heads, and constraints. About the constraints, Datalog$^\pm$, and also the OMD model, supports *equality-generating dependencies (egds)*, i.e. rules with just an equality atom in the head [Abiteboul et al., 1995], and *negative constraints (NCs)*, such as that in (1.3).

The enforcement of tgds creates an iterative data propagation and generation process that starts from the extensional data, the so-called *chase procedure* [Aho et al., 1979; Maier et al., 1979]. This chase determines a possibly infinite instance for the ontological schema, which is also called *the chase*. In the chase, the initially possibly incomplete predicate extensions are completed with data. This *chase-based semantics* naturally extends the forward data-propagation process underlying Datalog [Ceri et al., 1990], through *value invention* corresponding to existential quantifiers.

From the conceptual point of view, queries are posed to and answered from the chase. However, doing QA on a completely materialized chase is inefficient if not impossible. For this reason, and depending on the kind of Datalog$^\pm$ontology at hand, the data generation process can be triggered but also restricted by QA; or not done at all, but replaced by query rewriting, according to which the original query is replaced by one that can be posed and answered from the initial extensional data [Calì et al., 2012c; Gottlob et al., 2014, 2015].

Actually, Datalog$^\pm$ imposes some syntactic restrictions on the sets of rules, to guarantee good computational behavior when it comes to QA (hence the $-$). There are different classes of restrictions, which give rise to a family of Datalog$^\pm$ languages, with differences in expressive power and computational properties. Datalog$^\pm$ has been used to represent different kinds of ontologies, enabling OBDA [Calì et al., 2012a,b], and representation and querying of semantic web data [Arenas et al., 2014].

Our OMD models become Datalog$^\pm$ ontologies that extend HM models. In them, dimensional rules are represented as tgds; and dimensional constraints as egds or NCs. We impose two natural conditions on dimensional rules in OMD models: (a) Data generation along dimensional navigation is enabled by tgds whose body joins (i.e. in the antecedent) are on *categorical attributes* (and then, take values from dimension categories); and (b) No values are invented for categorical attributes. For example, in (1.1) the join is on the categorical attribute *Ward*; and the invented value (the existential quantifier) if for the non-categorical attribute *Speciality*. (We discuss in Section 4.3.1 the case where these assumptions do not hold or are relaxed.)

These conditions on OMD models make the corresponding Datalog$^\pm$ ontologies belong to one of the previously identified and investigated syntactic classes of Datalog$^\pm$ programs, that of *weakly-sticky (WS)* programs [Calì et al., 2012c]. This allows us to apply some established results in relation to QA. For example, it is known that conjunctive QA can be done in polynomial-time (in data complexity) [Calì et al., 2012c].

The class of *WS* programs is a generalization of *sticky* Datalog$^\pm$. The latter is a syntactic class of programs characterized by restrictions on variables participating in a body join. *WS* Datalog$^\pm$ extends sticky Datalog$^\pm$ by also capturing the well-known class of *weakly-acyclic (WA) programs* [Fagin et al., 2005], which is defined in terms of the syntactic notions of *finite-* and *infinite-rank* positions. Actually, *WS* Datalog$^\pm$ is defined through restrictions on join variables occurring in infinite-rank positions.

At the end of Example 1.1.2, we saw the dimensional negative constraint η satisfied after the dimensional tgds are enforced. Actually, evaluating satisfaction of dimensional NCs on the chase is not different from *boolean conjunctive query (BCQ)* answering on the MD ontology.[3] If a query obtained from the body of the dimensional NCs is positively answered, the ontology is inconsistent. In Example 1.1.2, the BCQ obtained from the body of η is:

$$\mathcal{Q}_\eta : \exists d\, \exists n\, \exists s\, (\mathit{WorkingSchedules}(\mathsf{intensive}, d; n, s)\ \wedge\ \mathit{DayMonth}(d, \mathsf{jan})), \tag{1.6}$$

and η holds since the answer to $\mathcal{Q}_\eta$ is *false*. Query answering is trivial on an inconsistent ontology since every query is entailed. Otherwise, every CQ can be answered by ignoring the dimensional NCs.

While checking NCs is done effortlessly, the possible interaction between egds and tgds (cf. Example 1.3.1) can lead to undecidability of QA [Chandra & Vardi, 1985].

Example 1.3.1 (ex. 1.1.2 cont.) Consider a dimensional egd constraint ϵ, stating that nurses working in the same institution are of the same speciality:[4]

$$\begin{aligned}[\,&\mathit{WorkingSchedules}(u, t; n, s),\, \mathit{WorkingSchedules}(u', t'; n', s'),\\ &\qquad \mathit{UnitInstitution}(u, i),\, \mathit{UnitInstitution}(u', i)]\ \rightarrow\ s = s',\end{aligned}$$

[3] A *conjunctive query (CQ)* with no free variable and its answer as either *true* or *false*.

[4] The square brackets in the rule show the beginning and the end of the rule's body.

and a tgd σ_3:

$$WorkingSchedules(u,t;n,\mathsf{critical\text{-}care}),\, TimeDay(t,d) \ \rightarrow\ CriticalCareNurses(d;n).$$

According to the last three tuples of *WorkingSchedules* in Figure 1.4, *Alan*, *Helen* and *Sara* have schedules in H_1 since *intensive* and *Standard* are units in H_1. *Alan* in the third tuple is a critical-care nurse. Therefore to enforce ϵ, *Alan* and *Helen* have to be critical-care nurses, i.e. the unknown values in the last two tuples in *WorkingSchedules* have to change to critical-care. Now, this makes σ_3 applicable which in turn adds $(\mathsf{sep\text{-}6},\mathsf{helen})$ and $(\mathsf{aug\text{-}21},\mathsf{sara})$ to *CriticalCareNurses*. This shows an interaction between tgds and egds. More precisely, the application of the tgd σ_1 activates the enforcement of egd ϵ, which triggers tgd σ_3. □

Separability [Calì et al., 2012a] is a semantic condition for tgds and egds that guarantees there is no harmful interaction. Intuitively, it means either (i) tgds and egds do not interact, or (ii) the interaction does change answers to queries.

Example 1.3.2 (ex. 1.3.1 cont.) The set of dependencies formed by σ_1, ϵ and σ_3 is not separable, due to the interaction between ϵ and σ_3. This interaction also changes query answers. Specifically, the CQ, $\mathcal{Q}(n)\,:\,\exists d\ CriticalCareNurses(d;n)$, answers $\{\mathsf{Alan},\mathsf{Sara}\}$, where *Sara* is obtained from the interaction between ϵ and σ_3. □

For separable tgds and egds checking satisfaction of egds can be postponed to after the application of the tgds, so is that of NCs [Calì et al., 2012a]. In our work, we identify a syntactic condition on the dimensional egds that guarantees the separability of the combination of dimensional tgds and egds.

1.4 Query Answering under Weakly-Sticky Datalog$^\pm$

We study CQ answering over Datalog$^\pm$ programs, containing only tgds (specially sticky and *WS* tgds), since it becomes crucial for quality QA. There are two general approaches for QA over a Datalog$^\pm$ program [Calì et al., 2013, 2010a; Gottlob et al., 2014]:

(i) Bottom-up chasing or expansion of the program extensional data through the program rules, to obtain an instance satisfying the tgds that is used for QA.

(ii) Query rewriting according to the rules into another query (possibly in another language), so that the correct answers can be obtained by evaluating the new query directly on the initial extensional data.

QA over sticky Datalog$^\pm$ can be done by query rewriting [Calì et al., 2010a], which is proved impossible for *WS* programs [Calì et al., 2009]. A non-deterministic QA algorithm for *WS* Datalog$^\pm$ is presented in [Calì et al., 2012c], to obtain polynomial-time complexity upper bound rather than provide a practical algorithm.

In order to attack practical QA under *WS* programs, we set ourselves the following motivations, goals, and results (among others):

(A) Provide a practical bottom-up QA algorithm for *WS* Datalog$^\pm$.

(B) Apply a *magic-sets* rewriting optimization technique to the bottom-up algorithm in (A) to make it more query sensitive, and therefore more efficient.

(C) Present a *hybrid* QA algorithm that combines the algorithm in (A) and a form of query rewriting.

For (B), we use a magic-sets technique for existential rules introduced in [Alviano et al., 2012] that extends classical magic-sets for Datalog [Ceri et al., 1990]. Unfortunately, the class of *WS* Datalog$^{\pm}$ programs is provably not closed under this rewriting, meaning that the result of applying the rewriting to a *WS* program may not be *WS* anymore. This led us to search for a more general class of programs that: (i) is closed under the magic-sets rewriting, (ii) extends *WS* Datalog$^{\pm}$, (iii) still has tractable QA, and (iv) allows the application of the proposed bottom-up QA in (A).

More specifically, we propose the class of *joint weakly sticky (JWS)* programs. It extends both sticky and *WS* Datalog$^{\pm}$ using the notions of *existential dependency graph* and *joint acyclicity* [Krötzsch & Rudolph, 2011]. This new syntactic class of programs satisfies the desiderata above.

About (A), we provide a polynomial-time, chase-based, bottom-up QA algorithm, that can be applied to a range of program classes that extend sticky Datalog$^{\pm}$, in particular *JWS* and *WS*.

In relation to (C), we propose a *hybrid* algorithm between the bottom-up algorithm mentioned above and rewriting. It transforms a *WS* program using its extensional data into a sticky program, for which known query rewriting algorithms [Calì et al., 2010a; Gottlob et al., 2014] can be applied. This is done by partial grounding of the program rules, i.e. replacing variables that break the syntactic property of sticky Datalog$^{\pm}$ with selected constants from the program extensional data. Grounding a program is replacing every variable in its rules with data values, considering all possible conditions, obtaining basically a propositional program. Our grounding is only partial since it replaces only some of the variables.

1.5 Outline and Contributions

Summarizing, in this thesis we make the following contributions:

1. We present MD ontologies and the OMD model that extend the HM model with: (a) categorical relations as generalized facts-tables, (b) dimensional rules as tgds to specify data generation in categorical relations; and (c) dimensional constraints as egds and NCs that restrict the data generation process, by preventing some combinations of values in the relations.

2. We establish that the MD ontologies belong to the class of *WS* Datalog$^\pm$ programs, which enjoys tractability of QA. As a consequence, QA can be done in polynomial time in data.

3. We analyze the effect of dimensional constraints on QA, specifically the *separability condition* between dimensional rules (tgds) and dimensional constraints (egds). We show that by making variables in equalities appear as categorical attributes, separability holds.

4. We present two QA algorithms; a bottom-up chase-based algorithm, and a hybrid algorithm as a combination of grounding and rewriting.

5. We integrate the first algorithm with magic-sets rewriting technique for further optimization.

6. We introduce the class of *JWS* programs that extends sticky and *WS* Datalog$^\pm$ and we show that the bottom-up algorithm and its magic-sets optimization are applicable for *JWS* programs.

7. We propose a general approach for contextual quality data specification and extraction that is based on MD ontologies, emphasizing the dimensional navigation process that is triggered by queries about quality data. We illustrate the application of this approach by means of an extended example.

8. We capture semantic constraints on dimensions in the HM model, namely strictness and homogeneity [Hurtado & Mendelzon, 2002], as dimensional rules and dimensional constraints in the MD ontology.

9. We show the connection of the OMD model with some other similar hierarchical models. Particularly, we explain how the OMD model can fully capture the extended relational algebra proposed in [Martinenghi & Torlone, 2009, 2010, 2014].

Chapter 2

Background

2.1 Relational Databases

We start with a relational schema $\mathcal{R}$ containing two disjoint data domains: $\mathcal{C}$, a possibly infinite domain of *constants*, and $\mathcal{N}$, of infinitely many *labeled nulls.* It also contains predicates of fixed finite arities. We use capital letters, e.g. P, R, S, and T, possibly with sub-indices, for database predicates; and small letters, e.g. x, y, and z, denote variables. 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. With $\mathcal{R}$, $\mathcal{C}$, $\mathcal{N}$ we can build a language $\mathfrak{L}$ of first-order (FO) predicate logic, that has $\mathcal{V}$ as its infinite set of *variables.* We denote with $\bar{x}$, etc., finite sequences of variables. A *term* of the language is a constant, a labelled null, or a variable. An *atom* is of the form $P(t_1, \ldots, t_n)$, with $P \in \mathcal{R}$, n-ary, and $t_1, \ldots, t_n$ terms. An atom is *ground* if it contains no variables. An *instance* I for schema $\mathcal{R}$ is a possibly infinite set of ground atoms. A *database instance* is a finite instance that contains no labelled nulls. The *active domain* of a database instance D, denoted $Adom(D)$, is the set of constants that appear in D. Instances can be used as interpretation structures for the FO language $\mathfrak{L}$. Accordingly, we can use the notion of formula satisfaction of FO predicate logic.

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 $\bar{x} := \bigcup \bar{x}_i \smallsetminus \bar{y}$ as a list of m variables. For an instance I, $\bar{t} \in (\mathcal{C} \cup \mathcal{N})^m$ is an *answer* to $\mathcal{Q}$ if $I \models \mathcal{Q}[\bar{t}]$, meaning that I makes $Q[\bar{t}]$ true, where $\mathcal{Q}[\bar{t}]$ is $\mathcal{Q}$ with the variables in $\bar{x}$ replaced with 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, $\mathcal{Q}(I) := \{yes\}$. Otherwise, $\mathcal{Q}(I) = \emptyset$.

A *tuple-generating dependency (tgd)*, also called *existential rule* or simply a *rule*, is a sentence, σ, of $\mathfrak{L}$ of the form:

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

with $\bar{x}_i$ indicating the variables appearing in P_i (possibly among with elements from $\mathcal{C}$), and an implicit universal quantification over all variables in $\bar{x}_1, \ldots, \bar{x}_n, \bar{x}$, and $\bar{x} \subseteq \bigcup_i \bar{x}_i$, and the dots and the commas 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 *head*$(\sigma)$ and *body*(σ) we denote the atom in the consequent and the set of atoms in the antecedent of σ, respectively.

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

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

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

where $x, x' \in \bigcup_i \bar{x}_i$, and $\bot$ is a symbol that denotes the Boolean constant that is always false. The notion of satisfaction of program rules and program constraints by an instance I is defined as in FO logic.

In relational databases, the above rules and constraints are called *dependencies*, and are considered to be general forms for *integrity constraints (ICs)* [Abiteboul et al., 1995]. In particular, tgds generalize *inclusion dependencies (IDs), a.k.a. referential* *constraints*, and egds subsume *key constraints* and *functional dependencies (FDs)*. Relational databases make the complete data assumption (*closed world assumption (CWA)*) [Abiteboul et al., 1995] and as a result the application of these dependencies amounts to checking them over database instances.

A *functional dependency (FD)* $R : \bar{A} \rightarrow \bar{B}$, where $\bar{A}$ and $\bar{B}$ are sets of positions of the predicate R, is satisfied if for every pair of tuples $\bar{t}$ and $\bar{t}'$ in the extension of R, $\bar{t}[\bar{A}] = \bar{t}'[\bar{A}]$ implies that $\bar{t}[\bar{B}] = \bar{t}'[\bar{B}]$ holds.[1] An *inclusion dependency (ID)* $P[i] \subseteq R[j]$ is satisfied, if for every tuple $\bar{t}$ in the extension of P there is a tuple $\bar{t}'$ in the extension of R, such that $t_i = t'_j$ [Abiteboul et al., 1995].

Datalog is a declarative query language for relational databases that is based on the logic programming paradigm. Datalog allows to define recursive views, which goes beyond the traditional relational query languages, i.e. *relational calculus (RC)* and *relational algebra (RA)* [Abiteboul et al., 1995; Ceri et al., 1990]. A Datalog program Π of schema $\mathcal{R}$ is a set Π^R of function-free horn clauses of FO logic, i.e. tgds as in (2.2), but without $\exists$-variables, plus a database D. The predicates in $\mathcal{R}$ are either *extensional*, i.e. they do not appear in rule heads and have complete data in D, or *intentional*, and are defined by the rules, without an extension in D.

The semantics of a Datalog program is given by a fixed-point semantics [Abiteboul et al., 1995]. According to this semantics, the extensions of the intentional predicates are obtained by, starting from the extensional database, iteratively enforcing the rules and creating tuples for the intentional predicates. This coincides with the model-theoretic semantics [Abiteboul et al., 1995], a.k.a. minimal-model semantics for Datalog, determined by a minimal model for the database and the rules (it always exists and is unique).

[1] $\bar{t}[\bar{A}]$ are the values of $\bar{t}$ in the positions of $\bar{A}$.

Example 2.1.1 A Datalog program Π containing the rules:

$$\begin{aligned} P(x,y) &\rightarrow R(x,y), \\ P(x,y), R(y,z) &\rightarrow R(x,z), \end{aligned}$$

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

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

$$P_1(\bar{x}_1), ..., P_n(\bar{x}_n) \rightarrow \mathit{ans}_{\mathcal{Q}}(\bar{x}), \tag{2.5}$$

where $\mathit{ans}_{\mathcal{Q}}(\cdot) \notin \mathcal{R}$ is an auxiliary predicate. The query answers form the extension of the answer-collecting predicate $\mathit{ans}_{\mathcal{Q}}(\cdot)$. When $\mathcal{Q}$ is a BCQ, $\mathit{ans}_{\mathcal{Q}}$ is a propositional atom; and if $\mathcal{Q}$ is true in I, then generating the atom $\mathit{ans}_{\mathcal{Q}}$ can be reinterpreted as the query answer (being *Yes*).

A Datalog$^{\pm}$ program $\Pi = \Pi^R \cup \Pi^C \cup D$ is, in general, formulated by a set of rules Π^R of the form (2.2), a (possibly empty) set of constraints Π^C as in (2.3) and (2.4), and a database D that provides extensional data for the programs.[2] The semantics of tgds, egds, and NCs in a Datalog$^{\pm}$ program is notably different from their semantics in relational databases. With Datalog$^{\pm}$, we make the *open world assumption (OWA)*, which allows incomplete data for all program predicates, and tgds are used to complete the data through data generation, and egds and NCs to restrict this process.

The set of models of Π, denoted by $\mathit{Mod}(\Pi)$, contains all instances I, such that $I \supseteq D$ and $I \models \Pi^R \cup \Pi^C$. Given a CQ $\mathcal{Q}$, the set of answers to $\mathcal{Q}$ from Π is defined by $\mathit{ans}(\mathcal{Q},\Pi) := \bigcap_{I \in \mathit{Mod}(\Pi)} \mathcal{Q}(I)$, a *certain answer* semantics.

[2] For simplicity of notation, when a program Π has only rules (without constraints, i.e. $\Pi^C = \emptyset$), we use Π to refer to the program (i.e. set of rules plus extensional data) and also its set of rules.

A *homomorphism* is a structure-preserving mapping, $h\colon \mathcal{C} \cup \mathcal{N} \rightarrow \mathcal{C} \cup \mathcal{N}$, between two instances I and I' over the same schema $\mathcal{R}$ such that: (a) $t \in \mathcal{C}$ implies $h(t) = t$, and (b) for every ground atom $P(\bar{t})$: if $P(\bar{t}) \in I$, then $P(h(\bar{t})) \in I'$. An *isomorphism* is a bijective homomorphism. We will use the notions of homomorphism and isomorphism in Chapter 7.

2.2 The Chase Procedure

The *chase* procedure [Aho et al., 1979; Beeri & Vardi, 1984] is a fundamental algorithm used for various database problems, including implication of database dependencies, query containment, CQ answering under dependencies, and data exchange [Beeri & Vardi, 1984; Calì et al., 2003; Fagin et al., 2005; Johnson & Klug, 1984; Maier et al., 1979]. The idea is that, given a set of dependencies over a database schema and an instance as input, the chase enforces the dependencies by adding new tuples into the instance, so that the result satisfies the dependencies.

Here, we review the tgd-based chase procedure that is used with Datalog$^+$ programs, i.e. programs without constraints. In Section 2.4, we discuss adding program constraints.

The chase procedure on a Datalog$^+$ program Π, i.e. a Datalog$^\pm$ program with set of rules Π^R and database D (without program constraints, $\Pi^C = \emptyset$), starts from the extensional database D, and iteratively applies the tgds in Π^R through some tgd-based chase steps.

Definition 2.2.1 (tgd-chase step) Consider a Datalog$^+$ program Π of schema $\mathcal{R}$, an instance I over the same schema $\mathcal{R}$. A tgd rule $\sigma \in \Pi$ and an assignment θ are *applicable* if θ maps the body of σ into I.[3]

[3] Sometimes we say the pair (σ, θ) is applicable.

A *chase step* applies on instance I the applicable pair (σ, θ) and results in instance $I' = I \cup \{\theta'(head(\sigma))\}$, where θ' is an extension of θ that maps the $\exists$-variables of σ into different fresh nulls (i.e. not appearing in I) in $\mathcal{N}$. This is denoted by $I \xrightarrow{\sigma,\theta} I'$.□

The chase step in Definition 2.2.1 is called *oblivious* [Calì et al., 2013], as it applies a rule when its body can be mapped to an instance, ignoring whether the rule is satisfied.

Remark 2.2.1 In a sequence of chase steps, denoted by $I_0 \xrightarrow{\sigma_1,\theta_1} I_1 \xrightarrow{\sigma_2,\theta_2} I_2 \ldots$, each applicable rule/assignment pair is applied only once. The sequence terminates if every applicable pair has been applied.

The instances in a sequence are monotonically increasing, but not necessarily strictly increasing, because a chase step can generate an atom that is already in the current instance. Depending on the program and its extensional database, the instances in a chase sequence may be properly extended indefinitely. □

Different orders of chase steps may result in different sequences. The chase procedure uses the notion of the level of atoms to define a *"canonical"* sequence of chase steps [Calì et al., 2013].

Definition 2.2.2 Let $I_0 \xrightarrow{\sigma_1,\theta_1} I_1 \ldots \xrightarrow{\sigma_k,\theta_k} I_k$ be a sequence of tgd-chase steps of a program Π, with $0 < k, I_0 := D$. The *level* of an atom $A \in I_k$, denoted $level(A)$, is: (a) 0 if A is in I_0, and (b) the maximum level of the atoms in $\theta_i(body(\sigma_i))$ plus one when $A \in (I_i \setminus I_{i-1})$, and $I_{i-1} \xrightarrow{\sigma_i,\theta_i} I_i$ is a chase step with $0 < i \leq k$.

The level of an applicable rule/assignment pair, (σ, θ), in I_k is the maximum level of the atoms in $\theta(body(\sigma))$. □

The chase applies the applicable rule/assignment pairs in a deterministic manner. That is if there are several applicable pairs after the k-th chase step, the chase procedure chooses the pair with the minimum level in the sequence of tgd-chase steps so far. If there are still several pairs with the minimum level, the chase applies the one with lexicographically smaller body image,[4] where the body image of (σ, θ) is the sequence of atoms obtained from applying θ on the body of σ.

Example 2.2.1 Consider a program Π with extensional database $D = \{R(a, b)\}$ and set of rules:

$$\sigma: \qquad R(x, y) \rightarrow \exists z\ R(y, z).$$

$$\sigma': \qquad R(x, y), R(y, z) \rightarrow S(x, y, z).$$

With the instance $I_0 := D$, (σ, θ_1), with $\theta_1 :\ x \mapsto a, y \mapsto b$, is applicable: $\theta_1(\mathit{body}(\sigma)) = \{R(a, b)\} \subseteq I_0$. The chase inserts a new tuple $R(b, \zeta_1)$ into I_0 (ζ_1 is a fresh null, i.e. not in I_0), resulting in instance I_1. The level of the new atom, $R(b, \zeta_1)$, is 1.

Now, (σ', θ_2), with $\theta_2\colon\ x \mapsto a, y \mapsto b, z \mapsto \zeta_1$, is applicable, because $\theta_2(\mathit{body}(\sigma')) = \{R(a, b), R(b, \zeta_1)\} \subseteq I_1$. The pair (σ, θ_3), with $\theta_3\colon\ x \mapsto b, y \mapsto \zeta_1$, is also applicable since $\theta_3(\mathit{body}(\sigma)) = \{R(b, \zeta_1)\} \subseteq I_1$. The levels of both pairs are 1 as the maximum levels of the atoms in their bodies are 1. The procedure applies (σ', θ_2) since $R(a, b), R(b, \zeta_1)$ is lexicographically smaller than $R(b, \zeta_1)$. $R(a, b), R(b, \zeta_1)$ is the body image of (σ', θ_2) while, $R(b, \zeta_1)$ is the body image of (σ, θ_3). The chase adds $S(a, b, \zeta_1)$ into I_1, resulting in I_2. □

The result of the chase procedure is an instance called "*the chase*", denoted by $\mathit{chase}(\Pi)$ or $\mathit{chase}(D, \Pi^R)$. If the chase does not terminate, the chase is an infinite

[4] This lexicographical order is based on a pre-established order between constants, nulls and predicate names.

instance: $\mathit{chase}(\Pi) := \bigcup_{i=0}^{\infty}(I_i)$, with $I_0 := D$, and, I_i is the result of the i-th chase step for $i > 0$. If the chase stops after m steps, $\mathit{chase}(\Pi) := \bigcup_{i=0}^{m}(I_i)$. The chase instance containing atoms up to level $k \geq 0$ is denoted by $\mathit{chase}^k(\Pi)$, while $\mathit{chase}^{[k]}(\Pi)$ is the instance constructed after $k \geq 0$ chase steps.

Example 2.2.2 (ex. 2.2.1 cont.) The chase continues, without stopping, creating an infinite instance:

$$\mathit{chase}(\Pi) = \{R(a,b), R(b,\zeta_1), S(a,b,\zeta_1), R(\zeta_1,\zeta_2), R(\zeta_2,\zeta_3), S(b,\zeta_1,\zeta_2), \ldots\}.$$

According to the chase procedure of Π:

$$\mathit{chase}^{[0]}(\Pi) = \mathit{chase}^0(\Pi) = D = \{R(a,b)\}.$$

$$\mathit{chase}^{[1]}(\Pi) = \mathit{chase}^1(\Pi) = \{R(a,b), R(b,\zeta_1)\}.$$

$$\mathit{chase}^{[2]}(\Pi) = \{R(a,b), R(b,\zeta_1), S(a,b,\zeta_1)\}.$$

$$\mathit{chase}^2(\Pi) = \{R(a,b), R(b,\zeta_1), R(\zeta_1,\zeta_2), S(a,b,\zeta_1)\}.$$

□

In Section 2.3.4, we use a *derivation relation* for a program Π, a binary relation between atoms in the chase of Π. Intuitively, if atoms A and B are in the derivation relation of Π, then, B is either directly obtained from A in a chase step, or indirectly derived by a sequence of chase steps, while A appears in the body image of some of the applied rule/assignment pairs in the chase steps.

Definition 2.2.3 (derivation relation) Consider a program Π and the k-th chase step, $I_k \xrightarrow{\sigma_k,\theta_k} I_{k+1}$, with $k \geq 1$, in the chase of Π. Let $C_k = \theta(\mathit{body}(\sigma_k)) \times (I_{k+1} \setminus I_k)$. Then, $\xrightarrow{\Pi}$ is defined as $\bigcup_{i=1}^{\infty} C_i$ if the chase does not stop, and $\bigcup_{i=1}^{m} C_i$ if it stops after m steps. The *derivation relation* of Π, denoted by $\xrightarrow{\Pi}^{*}$, is the transitive closure of $\xrightarrow{\Pi}$. □

In Definition 2.2.3, each C_k is a binary relation between atoms in the chase of Π.

Example 2.2.3 (ex. 2.2.1 cont.) According to the chase of Π:

$$\begin{aligned}
C_1 &= \{\langle R(a,b), R(b,\zeta_1)\rangle\}.\\
C_2 &= \{\langle R(b,\zeta_1), R(\zeta_1,\zeta_2)\rangle\}.\\
C_3 &= \{\langle R(a,b), S(a,b,\zeta_1)\rangle, \langle R(b,\zeta_1), S(a,b,\zeta_1)\rangle\}.\\
&\quad \ldots \text{(non-terminating chase)}\\
\xrightarrow{\Pi} &= C_1 \cup C_2 \cup C_3 \cup ...\\
&= \{\langle R(a,b), R(b,\zeta_1)\rangle, \langle R(a,b), S(a,b,\zeta_1)\rangle,\\
&\quad \langle R(b,\zeta_1), S(a,b,\zeta_1)\rangle, \langle R(b,\zeta_1), R(\zeta_1,\zeta_2)\rangle, ...\}.\\
\xrightarrow{\Pi}^{*} &= \{\langle R(a,b), R(b,\zeta_1)\rangle, \langle R(a,b), S(a,b,\zeta_1)\rangle,\\
&\quad \langle R(b,b), S(a,b,\zeta_1)\rangle, \langle R(b,b), R(\zeta_1,\zeta_2)\rangle, ...\}.
\end{aligned}$$

□

Given a program Π, its chase (instance) is a *universal model* [Fagin et al., 2005], i.e. a representative of all models in $\mathit{Mod}(\Pi)$, in the sense that, for every model in $\mathit{Mod}(\Pi)$, there is a homomorphism that maps the universal model to that model. For this reason, as it is shown in [Fagin et al., 2005, Proposition 2.6], the (certain) answers to a CQ $\mathcal{Q}$ under Π, i.e. those in $\mathit{ans}(\mathcal{Q},\Pi)$, can be computed by evaluating $\mathcal{Q}$ over the chase instance (and discarding the answers containing nulls).

There are various chase procedures [Calì et al., 2013; Deutsch et al., 2008; Fagin et al., 2005; Marnette, 2009] that compute universal models. They differ in: (a) The definition of applicable rule/assignment pairs in chase steps. For example, in a *restricted* chase step [Calì et al., 2013] a tgd is applicable only if it is not satisfied. (b) The order of their chase step applications. For example, *core chase* [Deutsch et al., 2008] applies all applicable pairs simultaneously. In this thesis, we use the *oblivious chase* [Calì et al., 2013], that uses an oblivious chase step plus the restriction on repetition of rule applications in Remark 2.2.1, since it simplifies our algorithms and proofs.

2.3 Programs Classes and Datalog$^{\pm}$

CQ answering over Datalog^{+} programs with arbitrary sets of tgds is in general undecidable [Beeri & Vardi, 1981], and it becomes decidable for those programs with a terminating chase. However, it is in general undecidable if the chase terminates, even for a fixed instance [Beeri & Vardi, 1981; Deutsch et al., 2008]. Several sufficient conditions, syntactic [Deutsch et al., 2008; Fagin et al., 2005; Krötzsch & 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 & Rudolph, 2011] are two kinds of syntactic conditions that use a static analysis of a dependency graph for the predicate positions in the program.

A non-terminating chase does not imply that CQ answers are uncomputable. 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, 2010a, 2011, 2012a], *shy Datalog*$^{\exists}$ [Leone et al., 2012], and *finite expansion sets (fes)*, *finite unification sets (fus)*, *bounded-treewidth sets (bts)* [Baget et al., 2009, 2011a,b]. Each program class defines conditions on the program rules that lead to good computational properties for QA (Figure 6.2 in Section 6.3 shows the generalization relation among the program classes in the thesis). In the following, we focus on sticky and weakly-sticky Datalog$^{\pm}$ programs because of their relevance to MD ontologies.

2.3.1 Weakly-acyclic programs

Weakly-acyclic programs are defined using dependency graphs. The *dependency graph (DG)* of a program Π with schema $\mathcal{R}$ (cf. Figure 2.1) is a directed graph whose vertices are the positions of $\mathcal{R}$. The edges are defined as follows: for every $\sigma \in \Pi$, and every universally quantified variable ($\forall$-variable)[5] 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. $\pi_F(\Pi)$ denotes the set of finite-rank positions in Π. A program is *Weakly-acyclic (WA)* if all of the positions have finite-rank [Fagin et al., 2005].

Example 2.3.1 Let Π be a program with rules:

$$\begin{aligned} U(x) &\rightarrow \exists y\ R(x,y), \\ R(x,y) &\rightarrow P(y,x). \end{aligned}$$

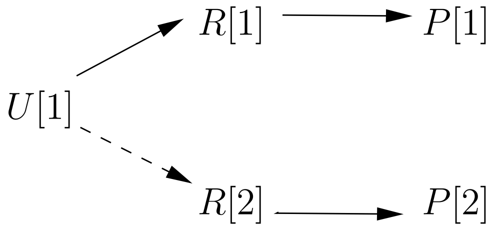


Figure 2.1: Dependency graph

According to the DG of Π, shown in Figure 2.1, the ranks of $U[1]$, $R[1]$, and $P[1]$ are 0, and the ranks of $R[2]$ and $P[2]$ are 1. Π is *WA* since all positions have finite-rank. □

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

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

polynomial time w.r.t. the size of the data [Fagin et al., 2005]. The same problem is 2EXPTIME-complete in combined complexity, i.e. w.r.t. the size of both program rules and the data [Kolaitis et al., 2006].

2.3.2 Jointly-acyclic programs

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

Let Π be a program with standardized apart rules, i.e. no variable appears in more than one rule. For a variable x in rule σ, let B_x and H_x be the sets of positions where x occurs in the body, resp. in the head, of σ. 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 null values invented for the $\exists$-variable z may appear during the chase.

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

Example 2.3.2 Consider a program Π with the following rules:

$$P(x_1, y_1) \rightarrow \exists z_1 \; R(y_1, z_1). \qquad (2.6)$$

$$R(x_2, y_2), U(x_2), U(y_2) \rightarrow \exists z_2 \; P(y_2, z_2). \qquad (2.7)$$

$$P(x_3, y_3) \rightarrow \exists z_3 \; S(x_3, y_3, z_3). \qquad (2.8)$$

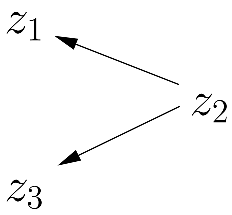


Figure 2.2: The EDG of Π.

$B_{y_1} = \{P[2]\}$ and $H_{y_1} = \{R[1]\}$ are the sets of positions where the variable y_1 appears in the body and, resp. the head of rule (2.6). 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 the sets of target positions of z_1 and resp. z_2.

In the EDG of Π in Figure 2.2 there is an edge from z_2 to z_1 since for the body variable y_1 in rule (2.6), 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 (2.8), 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 (2.7), $B_{x_2} \not\subseteq T_{z_1}$ and $B_{y_2} \not\subseteq T_{z_1}$. For a similar reason, there is no self-loop for z_2. The graph is acyclic, and Π is *JA*. □

JA programs have polynomial size (finite) chase w.r.t. 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 & Rudolph, 2011].

The program classes introduced in this section so far have a finite chase. Now, we review program classes for which the chase may be infinite, but still enjoy good properties w.r.t. QA.

2.3.3 Stickiness of the chase

The syntactic classes of sticky and weakly-sticky programs (cf. Section 2.3.4 and Section 2.3.5) are defined on the basis of the notion of the *"stickiness property of the chase"* (*sch-property*) [Calì et al., 2012c]. The latter is a "semantic" property of Datalog$^+$ programs in relation to the way the program's chase behaves with the extensional data. Informally, a program has this property if, due to the application of a rule σ, when a value replaces a repeated variable in a rule-body, then that value also appears in all the head atoms introduced through the iterative enforcement of applicable rules that starts with σ's application. In short, the value is propagated through all possible subsequent chase steps.

Definition 2.3.1 (Stickiness of the chase) [Calì et al., 2012c] A Datalog$^+$ program Π (including extensional data) has the *stickiness property* of the chase (in short the *sch-property*), if and only if for every chase step $I_i \xrightarrow{\sigma_i,\theta_i} I_i \cup \{A_i\}$ during the chase of Π, the following holds: If a variable x appears more than once in $body(\sigma_i)$, $\theta_i(x)$ occurs in A_i and every atom B for which, $A_i \xrightarrow{\Pi}^{*} B$. *Sticky-chase (SCh)* is the class of programs with the *sch-property*. □

The stickiness of the chase of a program is a semantic notion, relative to the program's extensional data.

Example 2.3.3 Consider Π_1 with $D_1 = \{R(a,b), R(b,c)\}$, and the following rules:

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

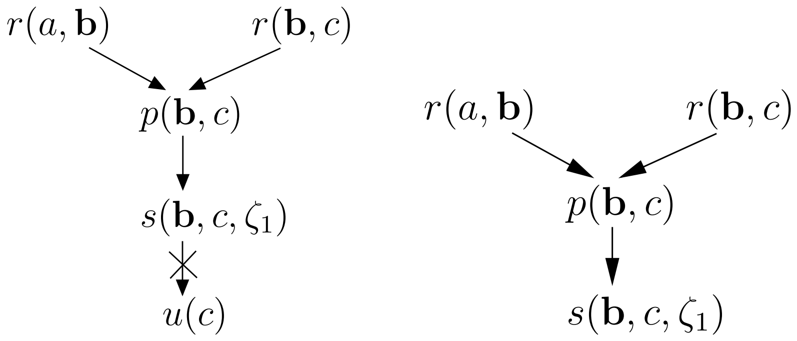


Figure 2.3: The *sch-property*.

Π_1 does not have the *sch-property*, as the chase in Figure 2.3 (left-hand side) shows: value b is not propagated all the way down to $U(c)$. However, a program Π_2 with the same database $D_2 = D_1$ and the rules in Π_1 without its third rule, has the *sch-property*, as shown in Figure 2.3 (right-hand side). □

Remark 2.3.1 The *sch-property* for a program guarantees PTIME data complexity of CQ answering: a CQ can be answered on an initial fragment of the chase of polynomial size in the size of the program's data (cf. the QA algorithm in Chapter 7 and Theorem 7.2.1). In particular, the values that are propagated during the chase, either:

(a) Replace a join variable in a tgd-chase step and continue to appear in the head atoms in the subsequent tgd-chase steps, or

(b) Do not replace any join variables in the tgd-chase steps.

The values in (b) can be interchangeably used during the chase. So, they can be considered as placeholders. For example, atoms $U(\zeta_1)$ and $U(\zeta_2)$ can be interchangeably used during the chase, if ζ_1 and ζ_2 never replace a repeated variable in any chase step (more details are in the proof of Theorem 7.2.1). The number of values in (a) is a polynomial function of the number of constants in the program's extensional database [Fagin et al., 2005] (cf. Example 2.3.4). Therefore, the values in (a) and (b) generate polynomially many atoms in the fragment of the chase for answering a CQ at hand. In Chapter 7, we propose a QA algorithm for *SCh* (and some generalizations) that generates this fragment of the chase. □

Example 2.3.4 Consider a program Π with $D = \{U(a), U(b)\}$, a BCQ $\mathcal{Q} : \exists x\ \exists y\ (U(x) \wedge P(x,y) \wedge V(y))$, and the set of rules:

$$U(x) \rightarrow \exists y\ P(x,y). \tag{2.9}$$

$$P(x,y) \rightarrow V(y). \tag{2.10}$$

$$U(x), P(x,y), V(y) \rightarrow \mathit{ans}_{\mathcal{Q}}(x,y). \tag{2.11}$$

Rule (2.11) defines the query answer collection predicate $\mathit{ans}_{\mathcal{Q}}$. The chase of Π terminates and results in:

$$\mathit{chase}(\Pi) = \{U(a), U(b), P(a,\zeta_1), P(b,\zeta_1), R(\zeta_1,\zeta_3), R(\zeta_2,\zeta_4), \mathit{ans}_{\mathcal{Q}}(a,\zeta_1), \mathit{ans}_{\mathcal{Q}}(b,\zeta_2)\}.$$

The values a, b, ζ_1 and ζ_2 are examples of values in (a), in Remark 2.3.1, as they replace join variables in tgd-chase steps with (2.11). They continue to appear in the next atoms, in this case, $\mathit{ans}_{\mathcal{Q}}(a,\zeta_1)$ and $\mathit{ans}_{\mathcal{Q}}(b,\zeta_2)$. The number of these values in limited by the arity of the predicates (no more than two in this case as $\mathit{ans}_{\mathcal{Q}}$ can hold at most two values). This shows the number of such values polynomially depends on the number of constants in the extensional database. □

SCh is the semantic class of Datalog$^\pm$ programs with the *sch-property*. Sticky Datalog$^\pm$ (cf. Section 2.3.4) is a syntactic class of programs that enjoy the sch-property, for *every* extensional database [Calì et al., 2012c]. The *SCh* class can be extended to several larger, semantic, program classes that enjoy a relaxed form of the *sch-property* (cf. Section 6.1).

2.3.4 Sticky programs

The class of sticky programs is characterized through a body variable *marking procedure* whose input is the set of rules in the program Π (the extensional data do not participate in it). The procedure has two steps:

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

(b) *Propagation step*: For each $\sigma \in \Pi$, if a marked variable in $\mathit{body}(\sigma)$ appears at position p, then for every $\sigma' \in \Pi$ (including σ), mark each occurrence of the variables in $\mathit{body}(\sigma')$ that appear in $\mathit{head}(\sigma')$ in the same position p.

Π is *sticky* when, after applying the marking procedure, there is no rule with a marked variable appearing more than once in its body. Notice that a variable never appears both marked and unmarked in a same body.

Example 2.3.5 The original set of three rules is shown on the left-hand side below. The second rule already shows marked variables (with a hat) after the preliminary step. The set of rules on the right-hand side are the result of whole marking procedure.

$$\begin{array}{rclcrcl}
R(x,y), P(x,z) & \rightarrow & S(x,y,z). & \qquad & R(\hat{x},y), P(\hat{x},\hat{z}) & \rightarrow & S(x,y,z). \\
S(\hat{x},y,\hat{z}) & \rightarrow & U(y). & & S(\hat{x},y,\hat{z}) & \rightarrow & U(y). \\
U(x) & \rightarrow & \exists y\, R(y,x). & & U(x) & \rightarrow & \exists y\, R(y,x).
\end{array}$$

Variables x and z in the first rule-body end up marked after the propagation step: they appear in the same rule's head, in positions where marked variables appear in the second rule ($S[1]$ and $S[3]$). Accordingly, the set of rules is *not* sticky: x in the first rule's body is marked and occurs twice in it (in $R[1]$ and $P[1]$). □

Sticky Datalog$^{\pm}$ is properly included in *SCh*. That is programs with sch-property may not be syntactically sticky.

Example 2.3.6 Let Π be a Datalog^{+} program with extensional data $D = \{R(a,b)\}$ and the tgd rule, $R(x,y), R(y,z) \ \rightarrow \ R(x,z)$. Π is *not* sticky as y is marked and appears twice in the body of the rule. The chase of Π does not apply the rule. So, the program trivially has the *sch-property*. □

With sticky programs, QA can be done in PTIME in data complexity and EXPTIME-complete in combined complexity [Calì et al., 2012c]. In fact, CQ answering over sticky programs is *first-order rewritable* [Calì et al., 2010a; Gottlob et al., 2011].

Definition 2.3.2 (first-order rewritability) [Calvanese et al., 2007][6] CQ answering over a Datalog^{+} program Π with extensional database D is *first-order (FO) rewritable* if, for every CQ $\mathcal{Q}$, a FO query $\mathcal{Q}'$ can be constructed such that, $\mathit{ans}(\mathcal{Q},\Pi) = \mathcal{Q}'(D)$. □

Example 2.3.7 Let Π be a program with a database $D = \{P(a,b), U(b)\}$ and the rule:

$$P(x,y), U(y) \ \rightarrow \ \exists z \ S(x,y,z).$$

Π is sticky since it does not have marked variables. Being sticky, it is also FO rewritable. For the CQ $\mathcal{Q}(x) : \exists y \ \exists z \ S(x,y,z)$, $\mathcal{Q}(D) = \emptyset$, since S has no extension in D. However, $\mathit{ans}(\mathcal{Q},\Pi) = \{a\}$ because S has intensional data obtained through the

[6] First-order rewritability for *ontology-based query answering* (cf. Section 3.4 for details) is first introduced in [Calvanese et al., 2007].

rule. $\mathcal{Q}$ can be rewritten into the FO query $\mathcal{Q}'(x) \colon \exists y \exists z\ S(x,y,z) \vee \exists t\ (P(x,t) \wedge U(t))$, for which, $ans(\mathcal{Q},\Pi) = \mathcal{Q}'(D)$. Here, $\mathcal{Q}'$ is obtained by relaxing $\mathcal{Q}$ and adding $\exists t\ (P(x,t) \wedge U(t))$ that is extracted from the rule body. □

FO rewritability is a desirable property as it is well known that the evaluation of FO queries is in the highly tractable class AC_0 (in data complexity) [Vardi, 1995].

2.3.5 Weakly-sticky programs

Weakly-sticky programs 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 Section 2.3.1 and, resp., Section 2.3.4: A set of rules Π is *weakly-sticky (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 some positions in $\pi_F(\Pi)$.

Example 2.3.8 Consider Π with the set of rules:

$$R(x,y) \ \rightarrow \exists z\ R(y,z).$$

$$R(x,y), U(y), R(y,z) \ \rightarrow R(x,z).$$

According to the graph of Π, $\pi_F(\Pi) = \{U[1]\}$, and $\pi_\infty(\Pi) = \{R[1], R[2]\}$. After applying the marking procedure, every body variable in Π is marked. Π is *WS* since the only repeated marked variable is y, in the second rule, and it appears in $U[1] \in \pi_F(\Pi)$.

Now, let Π' be the program with the first rule of Π and the second rule as follows:

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

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

Intuitively, *WS* generalizes the syntactic stickiness condition by prohibiting repeated marked variables appearing only in infinite-rank positions. The *WS* condition guarantees tractability of CQ answering, because a CQ can be answered on an initial fragment of the chase whose size is polynomial in the size of the extensional database (it also depends on the query). In fact, the data complexity and combined complexity of CQ answering over *WS* programs are PTIME-complete and 2EXPTIME-complete, respectively [Calì et al., 2012a].

The polynomial data complexity of QA under a *WS* program relies on the following facts about the values in the fragment of its chase just mentioned: (a) Polynomially many values in the size of the program's data may appear in the finite-rank positions [Fagin et al., 2005], (b) The values in the infinite-rank positions enjoy the stickiness property. As explained in Section 2.3.3, there are polynomially many values of this kind. Consequently, the values in (a) and (b) can generate polynomially many atoms in the fragment of the chase of a *WS* program that is necessary for answering a CQ.

This argument about PTIME data complexity of QA under *WS* programs can also be applied to more general, syntactic and semantic classes of programs that are characterized through the use of the stickiness condition on positions where infinitely many values may appear during the chase. (*WS* programs are a special case, where those positions have infinite-rank; and stickiness is enforced by the syntactic variable-marking mechanism.) Actually, we can make the general claim that the combination of finitely many values in finite positions plus chase-stickiness on infinite positions makes QA decidable (cf. Theorem 7.2.1 and the QA algorithm in Chapter 7 for classes of programs that generalize *WS*).

Table 2.1 is a summary of complexity of BCQ answering under programs that have been reviewed in this section.

	Data complexity	**Combined complexity**
WA	PTIME-complete	2EXPTIME-complete
JA	PTIME-complete	2EXPTIME-complete
sticky	in AC_0	EXPTIME-complete
WS	PTIME-complete	2EXPTIME-complete

Table 2.1: Complexity of BCQ answering under programs in Section 2.3

2.4 Program Constraints

So far in this chapter we have considered programs without constraints, i.e. only programs with extensional databases and rules of the form (2.2). In this section, we extend Datalog$^+$ with NCs and egds of the forms (2.4) and (2.3), resp. These ICs are called *program constraints* in the context of Datalog$^+$ programs.

2.4.1 Negative constraints

We recall the syntax and the semantics of NCs introduced in Section 2.1. A NC is of the form (2.4), $\eta: P_1(\bar{x}_1), \dots, P_n(\bar{x}_n) \rightarrow \bot$; and η holds in an instance I if there is no assignment θ that maps $P_1(\bar{x}_1), \dots, P_n(\bar{x}_n)$ into I. This can be checked by evaluating a BCQ associated to η, $\mathcal{Q}_\eta: \exists x_1 \dots \exists x_n\ (P_1(\bar{x}_1) \wedge \dots \wedge P_n(\bar{x}_n))$ in I. The NC η holds in I if and only if $\mathcal{Q}_\eta$ is false in I.

For a program Π, adding NCs may change CQ answers under the program, by eliminating some models from $\mathit{Mod}(\Pi)$.

Example 2.4.1 Consider a program Π with the database $D = \{U(a)\}$ and the tgd, $\sigma : U(x) \ \rightarrow \ \exists y\ R(x,y)$. $I_1 = \{U(a), R(a,a)\}$ and $I_2 = \{U(a), R(a,b)\}$ are two models in $\mathit{Mod}(\Pi)$. The BCQ $\mathcal{Q} : \exists x\ R(x,x)$ is false in Π since $I_2 \not\models \mathcal{Q}$ (according to the certain answers semantics, cf. Section 2).

Let Π' be a program as Π, but with the additional NC, $\eta : U(x) \ \rightarrow \ \bot$. The NC eliminates I_1 and I_2 from $\mathit{Mod}(\Pi')$, since $\mathcal{Q}_\eta : \exists x\ U(x)$ is true in both instances. In fact, Π' does not have any model, $\mathit{Mod}(\Pi') = \emptyset$, and $\mathcal{Q}$ is trivially true under Π'. This shows that adding NCs may change CQ answers.

Adding NCs to a program Π does not necessarily lead to different query answers, even if the NCs eliminate certain instances from $\mathit{Mod}(\Pi)$. In fact, consider now the program Π'' as Π with the additional NC, $\eta' : R(x,x) \ \rightarrow \ \bot$. Now, $I_2 \in \mathit{Mod}(\Pi'')$ and $I_1 \notin \mathit{Mod}(\Pi'')$ since $\mathcal{Q}_{\eta'} : \exists x\ R(x,x)$ is false in I_2 and true in I_1. $\mathcal{Q}$ is false under Π'' because $I_2 \not\models \mathcal{Q}$. Here, adding η' to Π eliminated I_1 from $\mathit{Mod}(\Pi'')$ but Π'' still answers $\mathcal{Q}$ the same as Π. □

Example 2.4.1 confirms that NCs have to be considered for CQ answering when programs contain them.

According to [Calì et al., 2009, Theorem 11], CQ answering under a program Π with rules Π^R and NCs Π^C can be reduced to CQ answering under a program, Π', with only the rules Π^R (and without the NCs). This can be done by:

(a) Checking if the NCs in Π^C are satisfied by Π'. More precisely, for $\eta \in \Pi^C$, we evaluate the BCQ $\mathcal{Q}_\eta$ over Π'. If at least one of such queries answers positively, Π is inconsistent, and thus QA is trivial since every query is entailed.

(b) If the NCs are not satisfied by Π', for every BCQ $\mathcal{Q}$, $\Pi \models \mathcal{Q}$ if and only if $\Pi' \models \mathcal{Q}$, i.e. we can answer queries over Π', ignoring the NCs.

Notice that in (a), if Π' (the program without egds) is *WS*, for which the chase might not terminate, we can answer $\mathcal{Q}_\eta$ on the limited portion of the chase of Π', as explained in Section 2.3.5.

Example 2.4.2 (ex. 2.4.1 cont.) For Π' and as in (a) above, we evaluate $\mathcal{Q}_\eta$ under Π (Π' without η). The answer is true, which means η does not hold, and Π' is inconsistent. Every CQ is trivially true under Π'.

For Π'', we first evaluate $\mathcal{Q}_{\eta'}$ under Π and since it is false, η' is satisfied by Π''. So, we ignore the constraint: $\Pi'' \not\models \mathcal{Q}$ because $\Pi \not\models \mathcal{Q}$, with $\mathcal{Q} : \exists x \ R(x, x)$. □

We can see that answering BCQs under Datalog$^\pm$ programs with NCs has the same data complexity of answering BCQs on Datalog$^+$ programs with tgds alone.

2.4.2 Equality-generating dependencies

Let us retake egds of the form (2.3) in Section 2.1, with their semantics defined as FO sentences. An egd ϵ holds in an instance I if and only if any assignment that maps $body(\epsilon)$ to I, maps the head variables of ϵ to the same terms. So as with NCs, adding egds to a Datalog$^+$ program Π may eliminate certain models from $Mod(\Pi)$, which in turn may change CQ answers. However, imposing egds on a Datalog$^+$ program is different from imposing NCs. This is specially due to possible interactions between the egds and tgds during the chase procedure, as we show now.

Example 2.4.3 Consider a program Π with $D = \{R(a, b)\}$ and the following rules:

$$R(x, y) \;\rightarrow\; \exists z\, \exists w\, S(y, z, w). \tag{2.12}$$

$$S(x, y, y) \;\rightarrow\; P(x, y). \tag{2.13}$$

$\mathit{chase}(\Pi) = \{R(a, b), S(b, \zeta_1, \zeta_2)\}$, with ζ_1 and ζ_2 fresh nulls. Rule (2.13) is not applied since ζ_1 and ζ_2 are not equal, as required by the body. The answer to a BCQ $\mathcal{Q}$: $\exists x\; \exists y\; P(x, y)$ is *false* under Π as $\mathit{chase}(\Pi) \not\models \mathcal{Q}$. Now consider Π' that is obtained by adding the following egd to Π:

$$S(x, y, z) \;\rightarrow\; y = z. \tag{2.14}$$

The chase of Π' first applies rule (2.12) and results in $I_1 = \{R(a, b), S(b, \zeta_1, \zeta_2)\}$. Now, there is no more tgd/assignment applicable pair. But, if we apply the egd (2.14), it equates ζ_1 and ζ_2, and results in $I_2 = \{R(a, b), S(b, \zeta_1, \zeta_1)\}$. (This kind of egd-chase step is defined in Definition 2.4.1.) Now, rule (2.13) and $\theta' : x \mapsto b, y \mapsto \zeta_1$ are applicable and they add $P(b, \zeta_1)$ to I_2, generating $I_3 = \{R(a, b), S(b, \zeta_1, \zeta_1), P(b, \zeta_1)\}$.

The procedure terminates since no more tgds or egds can be applied. The chase result, $\mathit{chase}(\Pi')$, is I_3. $\mathcal{Q}$ holds under Π': $\mathit{chase}(\Pi') \models \mathcal{Q}$. □

Example 2.4.3 shows that adding an egd to a program may change query answers. Also, the chase of a program may apply an egd between tgd-chase steps. Actually, there might be interactions between tgds and an egd, i.e. the application of a tgd activates the egd, which in turn might make some tgds applicable. This confirms that, unlike NCs, checking egds can not be postponed until all tgds have been applied: they have to be applied during the chase procedure, through *egd-chase steps.*

Definition 2.4.1 (egd chase step) [Calì et al., 2009] Let Π be a program with database D, tgds Π^R, and egds Π^C. The egd $\epsilon : P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n) \rightarrow x = x'$ in Π^C

and assignment θ are *applicable* on an instance I if $\theta(body(\epsilon)) \in I$ and $\theta(x) \neq \theta(x')$. In this case, the *effect* of the application of the pair, (ϵ, θ), an *egd-chase step*, is as follows:

(a) If $\theta(x)$ and $\theta(x')$ are two distinct constants,[7] then the result is a *hard constraint violation*, which causes the *failure of the chase*, and the halting of its computation. We say the program is *inconsistent*.

(b) Otherwise (i.e. at least one of them is a null), the result is the replacement of all occurrences of $\theta(x')$ in I by $\theta(x)$, where $\theta(x)$ precedes $\theta(x')$ in the lexicographical order.[8] □

Notice that this definition also defines the failure of the chase with egds.

The (combined) chase procedure of a program, with tgds and egds, iteratively applies both tgd and egd chase steps, as follows: (a) Apply applicable pairs of egd/assignment exhaustively, as long as they exist, and according to a pre-established order (such as tgd-chase steps). (b) Apply a tgd-chase step as described in Section 2.2. In other words, a sequence of steps in the (combined) chase procedure is formed by a sequence of tgd-chase steps, while before each tgd-chase step every possible egd-chase step is applied.

The (combined) chase *terminates* if it either fails (always due to a failed egd-step) or there are no more applicable pairs of egd/assignment or tgd/assignment. The (combined) chase failure results in an inconsistent program that answers every BCQ positively. If the (combined) chase does not fail, the result is a possibly infinite universal model, that satisfies both the tgds and egds [Calì et al., 2013].

[7] This includes the constants that the tgds may introduce.

[8] We assume a lexicographical order between constants in $\mathcal{C}$ and also between nulls in $\mathcal{N}$ (cf. Section 2.2), in which constants precede all null values.

The interaction of tgds and egds during the chase procedure (as shown in Example 2.4.3) may lead to undecidability of QA [Johnson & Klug, 1984]. In fact, this is true even in simple cases, such as combinations of *functional dependencies (FDs)* and *inclusion dependencies (IDs)* [Chandra & Vardi, 1985], or *key constraints* and IDs [Calì et al., 2013]. A *separability condition* on the combination of egds and tgds guarantees a harmless interaction, i.e. CQ answering becomes decidable [Calì et al., 2012a,c].

Definition 2.4.2 (Separability) [Calì et al., 2012a] Let Π be a program with a database D, a set of tgds Π^R, and a set of egds Π^C, and let Π' be the program with D and Π^R (without the egds). Π^R and Π^C are *separable* if either (a) the chase of Π fails, or (b) for any BCQ $\mathcal{Q}$, $\Pi \models \mathcal{Q}$ if and only if $\Pi' \models \mathcal{Q}$. □

In Example 2.4.3, the tgds and the egd are not separable as the chase does not fail, and the egd changes CQ answers (in that case, $\Pi \not\models \mathcal{Q}$ and $\Pi' \models \mathcal{Q}$).

Separability is a semantic condition, relative to the chase, and depends on a program's extensional data. It guarantees that, as for programs with tgds and NCs, CQ answering under a program Π with tgds Π^R and egds Π^C can be reduced to CQ answering under a program, Π', with only the tgds in Π^R (and without the egds) [Calì et al., 2012a]. More precisely, if separability holds,

(a) Combined chase failure can be decided by posing the BCQs obtained from the egds directly to the program without the egds [Calì et al., 2012d, Theorem 1].

More specifically, for the egds in Π^C, $\epsilon : P_1(\bar{x}_1), \ldots, P_n(\bar{x}_n) \rightarrow x = x'$, the obtained BCQs are $\mathcal{Q}_\epsilon : \exists\bar{x}_1, ..., \exists\bar{x}_n \; (P_1(\bar{x}_1) \wedge \ldots \wedge P_n(\bar{x}_n) \wedge x \neq x')$. The chase fails iff the answer is positive at least for one of them.

(b) If it does not fail, CQ answering can be done with the tgds alone [Calì et al., 2012a,d].

Notice that in (a), if Π' is *WS*, for which the chase might not terminate, we can answer $\mathcal{Q}_\epsilon$ on the limited portion of the chase of Π', explained in Section 2.3.5.

Example 2.4.4 (ex. 2.4.3 cont.) Let Π'' be Π with an additional egd:

$$\epsilon': \quad R(x,y) \;\rightarrow\; x = y. \tag{2.15}$$

The tgds and ϵ' are separable. Intuitively, this is because R in the body of ϵ' does not appear in the head of the tgds, and as a result, ϵ' can only equate values from $Adom(D)$ during the (combined) chase of Π''. Therefore, the application of ϵ either causes failure, or it does not change the chase result or CQ answers. In fact, this observation leads to a sufficient syntactic condition for separability (cf. Condition (a) Definition 2.4.3).

Since the tgds and ϵ' are separable, we can decide if the chase fails by posing the BCQ $\mathcal{Q}_{\epsilon'} : \exists x\, \exists y\; (R(x,y) \wedge x \neq y)$ to Π (the program without the egd). The answer is positive, which means the (combined) chases fails, and the program is inconsistent.□

The problem of deciding if a set of tgds and egds is separable is undecidable [Calì et al., 2010a]. For *functional dependencies (FDs)*, as opposed to general egds, a syntactic sufficient condition for separability is a *non-conflicting* combination of tgds and FDs [Calì et al., 2010a, 2012d].

Definition 2.4.3 (Non-conflicting FDs) [Calì et al., 2010a] Let Π be a program with a set of tgds, Π^R, and a set Π^C of FDs. Π^R and Π^C are *non-conflicting* if, for every pair formed by a tgd $\sigma \in \Pi^R$ and an FD ϵ of the form $R : \bar{A} \rightarrow \bar{B}$ in Π^C, at least one of the following holds: (a) $\mathit{head}(\sigma)$ is not an R-atom, (b) $U_\sigma \not\supseteq \bar{A}$, or (c) $U_\sigma = \bar{A}$ and each $\exists$-variable in σ occurs just once in the head of σ. Here, U_σ is the set of positions of $\forall$-variables in the head of σ. □

Example 2.4.5 Consider $\mathcal{R}$, a schema with a ternary predicate S and a unary predicate V, a tgd $\sigma : V(x) \ \rightarrow \ \exists y\, \exists z\ S(x,y,z)$, and the FD $\epsilon : \{S[1], S[2]\} \rightarrow \{S[3]\}$. The FD ϵ can be written as an egd: $S(x,y,z), S(x,y,z') \ \rightarrow \ z = z'$. Here, σ and ϵ are non-conflicting, because (b) holds: $U_\sigma \not\supseteq A$, with $U_\sigma = \{S[1]\}$ and $A = \{S[1], S[2]\}$.

Now, consider the tgd $\sigma' : V(x) \ \rightarrow \ \exists y\ S(x,y,y)$, and the FD $\epsilon' : \{S[1]\} \rightarrow \{S[2], S[3]\}$. They are not non-conflicting, because none of (a)-(c) holds: For (a), S appears in the head of σ and the body of ϵ. For (b) and (c), $A = U_{\sigma'} = \{S[1]\}$, but y appears twice in the head of σ'. □

Conditions (a) and (b) in Definition 2.4.3 imply separability, by ensuring that the application of a tgd can not make an egd applicable. In particular for (a), the atoms introduced by a tgd never appear in the body of an egd. For (b), these atoms do not make the egd applicable since they introduce fresh nulls in the positions in A. With respect to (c), the atoms can make the egd applicable, but applying the egd does not change CQ answers (as shown in Example 2.4.6), which still guarantees separability. Notice that the non-conflicting condition is decidable.

Example 2.4.6 Let Π be a program with $D = \{P(a,b), V(a)\}$, FD $\epsilon : \{P[1]\} \rightarrow \{P[2]\}$, and tgd $\sigma : V(x) \rightarrow \exists y\ P(x,y)$. According to (c), ϵ and σ are non-conflicting: $A = U_\sigma = \{P[1]\}$ and y appears once in the head of σ.

The chase of Π applies (σ, θ), with $\theta : x \mapsto a$, and results in $I_1 = \{P(a,b), V(a), P(a,\zeta_1)\}$. Now, ϵ is applied, which converts ζ_1 into b, and results in $I_2 = D$. This egd application does not change CQ answers since for every CQ $\mathcal{Q}$, it holds $\mathcal{Q}(I_1) = \mathcal{Q}(I_2)$. $\square$

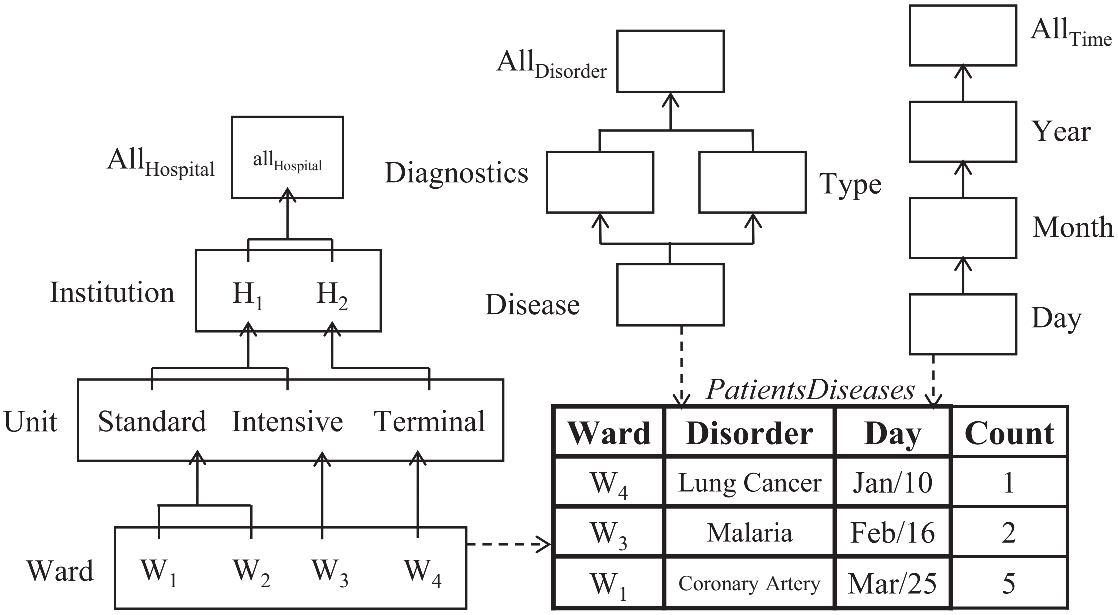
2.5 The Hurtado-Mendelzon Multidimensional Data Model

According to the *Hurtado-Mendelzon (HM) multidimensional data model* [Hurtado & Mendelzon, 2002], a *dimension schema*, $\mathscr{H} = \langle \mathscr{K}, \nearrow \rangle$, consists of a set $\mathscr{K}$ of *categories* (a.k.a. *levels*), and an irreflexive, binary relation $\nearrow$, called the *child-parent relation*, between categories (the first category is a child and the second category is a parent). $\nearrow^*$ denotes the transitive and reflexive closure of $\nearrow$. It is a partial order (a lattice) with a *top category*, *All*, which is reachable from every other category: $K \nearrow^* \mathit{All}$, for every category $K \in \mathscr{K}$. There is a unique *base category*, K^b, that has no children: for no category K, $K \nearrow K^b$ holds. There are no *"shortcuts"*, i.e. if $K \nearrow K'$, there is no category K'', distinct from K and K', with $K \nearrow^* K''$, $K'' \nearrow^* K'$.

A *dimension instance* for schema $\mathscr{H}$ is a structure $\mathscr{L} = \langle \mathcal{U}, <, m \rangle$, where $\mathcal{U}$ is a non-empty, finite set of data values called *members*, $<$ is an irreflexive binary relation between members, called the *child-parent relation* (the first member is a child and the second member is a parent),[9] and $m\colon \mathcal{U} \rightarrow \mathscr{K}$ is the total *membership function*. Relation $<$ parallels (is consistent with) relation $\nearrow$ between the categories: $e < e'$ implies $m(e) \nearrow m(e')$. The statement $m(e) = K$ is also expressed as $e \in K$. $<^*$ is the transitive and reflexive closure of $<$, and is a partial order over the members. There is a unique member *all*, the only member of *All*, which is reachable via $<^*$ from any other member: $e <^* \mathit{all}$, for every member e. A child member in $<$ has

[9] There are two child-parent relations in a dimension: $\nearrow$ is between the categories, and $<$ is between their members.

only one parent member in the same category: for members e, e_1, and e_2, if $e < e_1$, $e < e_2$ and e_1, e_2 are in the same category (i.e. $m(e_1) = m(e_2)$), then $e_1 = e_2$. $<^*$ is used to define the *roll-up relations* for any pair of distinct categories K and K', with $K \nearrow^* K'$: $L_K^{K'}(\mathscr{L}) = \{(e, e') \mid e \in K,\ e' \in K' \text{ and } e <^* e'\}$.



Ward	Disorder	Day	Count
W$_4$	Lung Cancer	Jan/10	1
W$_3$	Malaria	Feb/16	2
W$_1$	Coronary Artery	Mar/25	5

Figure 2.4: An HM model

Example 2.5.1 An HM model is shown in Figure 2.4, with three dimensions. For the $\mathsf{Hospital}$ dimension, on the left-hand-side, $\mathscr{K} = \{All_{Hospital}, Institution, Unit, Ward\}$, with top category $All_{Hospital}$ and base category $Ward$. The child-parent relation $\nearrow$ contains $(Institution, All_{Hospital})$, $(Unit, Institution)$, and $(Ward, Unit)$. The category of each member is specified by m, e.g. $m(\mathsf{H}_1) = Institution$. The child-parent relation, $<$, between the members contains $(\mathsf{W}_1, \mathsf{standard})$, $(\mathsf{W}_2, \mathsf{standard})$, $(\mathsf{W}_3, \mathsf{intensive})$, $(\mathsf{W}_4, \mathsf{terminal})$, $(\mathsf{standard}, \mathsf{H}_1)$, $(\mathsf{intensive}, \mathsf{H}_1)$, $(\mathsf{terminal}, \mathsf{H}_2)$, $(\mathsf{H}_1, \mathsf{all}_{Hospital})$, and $(\mathsf{H}_2, \mathsf{all}_{Hospital})$. Finally, $L_{Ward}^{Institution}$ is one of the roll-up relations and contains $(\mathsf{W}_1, \mathsf{H}_1)$, $(\mathsf{W}_2, \mathsf{H}_1)$, $(\mathsf{W}_3, \mathsf{H}_1)$, and $(\mathsf{W}_4, \mathsf{H}_2)$. □

A dimension can be represented in relational terms.[10] The relational dimension schema is $\mathcal{H} = \mathcal{K} \cup \mathcal{L}$, where $\mathcal{K}$ is a set of unary *category predicates*, and $\mathcal{L}$ is a set of binary *child-parent predicates*. In child-parent predicates, the first attribute is a child and the second attribute is a parent. The data domain of the schema is $\mathcal{U}$ (the set of members in the dimension). Accordingly, a dimension instance is a database instance $D^{\mathcal{H}}$ over $\mathcal{H}$, giving extensions of the predicates in $\mathcal{H}$. The extensions of the category predicates form a partition of $\mathcal{U}$.

The relational dimension schema and its relational instance are in one-to-one correspondence with the dimension schema and its dimension instance. In particular, for each category $K \in \mathscr{K}$ there is a category predicate $K(\cdot) \in \mathcal{K}$, and the extension of the predicate contains the members of the category. Also, for every pair of categories K, K' with $K \nearrow K'$, there is a corresponding child-parent predicate in $\mathcal{L}$, and its extension contains the child-parent relationships between members of K and K', according to $<$. In other words, each child-parent predicate in $\mathcal{L}$ stands for a roll-up relation between two child-parent categories.

Example 2.5.2 (ex 2.5.1 cont.) In the relational representation of $\mathsf{Hospital}$, $\mathcal{K}$ contains unary predicates $\mathit{All}_{\mathit{Hospital}}(\cdot)$, $\mathit{Institution}(\cdot)$, $\mathit{Unit}(\cdot)$, and $\mathit{Ward}(\cdot)$. The instance $D^{\mathcal{H}}$ gives to them the following extensions: $\mathit{All}_{\mathit{Hospital}} = \{\mathsf{all}_{\mathit{Hospital}}\}$, $\mathit{Institution} = \{\mathsf{H}_1, \mathsf{H}_2\}$, $\mathit{Unit} = \{\mathsf{standard}, \mathsf{intensive}, \mathsf{terminal}\}$, and $\mathit{Ward} = \{\mathsf{W}_1, \mathsf{W}_2, \mathsf{W}_3, \mathsf{W}_4\}$. $\mathcal{L}$ contains binary predicates $\mathit{InstitutionAll}_{\mathit{Hospital}}(\cdot,\cdot)$, $\mathit{UnitInstitution}(\cdot,\cdot)$, and $\mathit{WardUnit}(\cdot,\cdot)$, with the following extensions for the relational instance $D^{\mathcal{H}}$:

[10] We will use this representation in Section 4.1 to extend the HM model.

$$\begin{aligned} \mathit{InstitutionAll}_{\mathit{Hospital}} &= \{(\mathsf{H}_1, \mathsf{all}_{\mathit{Hospital}}), (\mathsf{H}_2, \mathsf{all}_{\mathit{Hospital}})\}, \\ \mathit{UnitInstitution} &= \{(\mathsf{standard}, \mathsf{H}_1), (\mathsf{intensive}, \mathsf{H}_1), (\mathsf{terminal}, \mathsf{H}_2)\}, \\ \mathit{WardUnit} &= \{(\mathsf{W}_1, \mathsf{standard}), (\mathsf{W}_2, \mathsf{standard}), (\mathsf{W}_3, \mathsf{intensive}), \\ &\qquad (\mathsf{W}_4, \mathsf{terminal})\}. \end{aligned} \qquad \square$$

In order to recover the hierarchy of a dimension in its relational representation, we have to impose some *integrity constraints (ICs)*. First, *inclusion dependencies (IDs)* associate the child-parent predicates to the category predicates. For example, the following IDs associate the first and second positions of $\mathit{WardUnit}(\cdot,\cdot)$ to $\mathit{Ward}(\cdot)$ and $\mathit{Unit}(\cdot)$, resp.: $\mathit{WardUnit}[1] \subseteq \mathit{Ward}[1]$, and $\mathit{WardUnit}[2] \subseteq \mathit{Unit}[1]$ (cf. Section 2.1 for the definition of IDs). We need key constraints for the child-parent predicates: the first attribute (child) is the *key* attribute. For example, $\mathit{WardUnit}[1]$ is the key attribute for $\mathit{WardUnit}(\cdot,\cdot)$.

We can have multiple dimensions reflected with disjoint relational dimensional schemas, one for each dimension. They can be put together into a single multidimensional schema that is the union of the individual ones. In particular, there are now top and base categories predicates in $\mathcal{K}$, for each dimension.

Assume $\mathcal{H}$ is the relational schema with multiple dimensions. A *fact-table schema* over $\mathcal{H}$ is a predicate $T(C_1, ..., C_n, M)$, where $C_1, ..., C_n$ are attributes with domain $\mathcal{U}$, and M is an attribute, called *measure*, with a numerical domain. Attribute C_i is associated with base-category predicate $K_i^b(\cdot) \in \mathcal{K}$. This is represented by an ID $T[i] \subseteq K_i^b[1]$. Additionally, $\{C_1, ..., C_n\}$ is a key for T, intuitively each point in the multidimensional space is mapped to at most one measure. A *fact-table (instance)* contains an extension of T.

Example 2.5.3 A tuple in a fact-table (cf. *PatientsDiseases* at the bottom-right in Figure 2.4) represents a numerical value, say a measurement, that is given context by the other entries in the tuple, which are members from the categories at the bottom of the dimension hierarchies. □

This multidimensional representation enables aggregation of numerical data at different levels of granularity, depending on the different levels of the categories in the dimension hierarchies. The roll-up relations can be used for this kind of aggregation.

Chapter 3

State of the Art

Our research builds upon and starts from work on *context-dependent data quality assessment* [Bertossi et al., 2011a, 2016] and *context-aware databases* [Martinenghi & Torlone, 2009, 2010, 2014]. Other closely related research, in regard to context modeling, is *context-aware data tailoring* and *context-dimension trees (CDTs)* [Bolchini et al., 2007a,b, 2009]. In relation to OBDA and ontologies, *Description logics (DLs)* [Baader et al., 2007] is a family of knowledge representation languages widely used in OBDA, similar to Datalog$^{\pm}$ that we used in our research. In this chapter, we briefly review them.

3.1 Contextual Data Quality Assessment

We first review previous work in [Bertossi et al., 2011a, 2016] on context-based data quality assessment. The starting point is that *data quality is context-dependent*. A context provides *knowledge about the way data is interrelated, produced and used*, which allows to make sense of the data. Furthermore, both the database under quality assessment and the context can be formalized as logical theories. The former is then *put in context* by mapping it into the latter, through logical mappings and possibly shared predicates.

In Figure 3.1, D is a relational database (with schema $\mathcal{R}$) under quality assessment. It can be represented as a logical theory [Reiter, 1984]. The context, $\mathfrak{C}$ in the

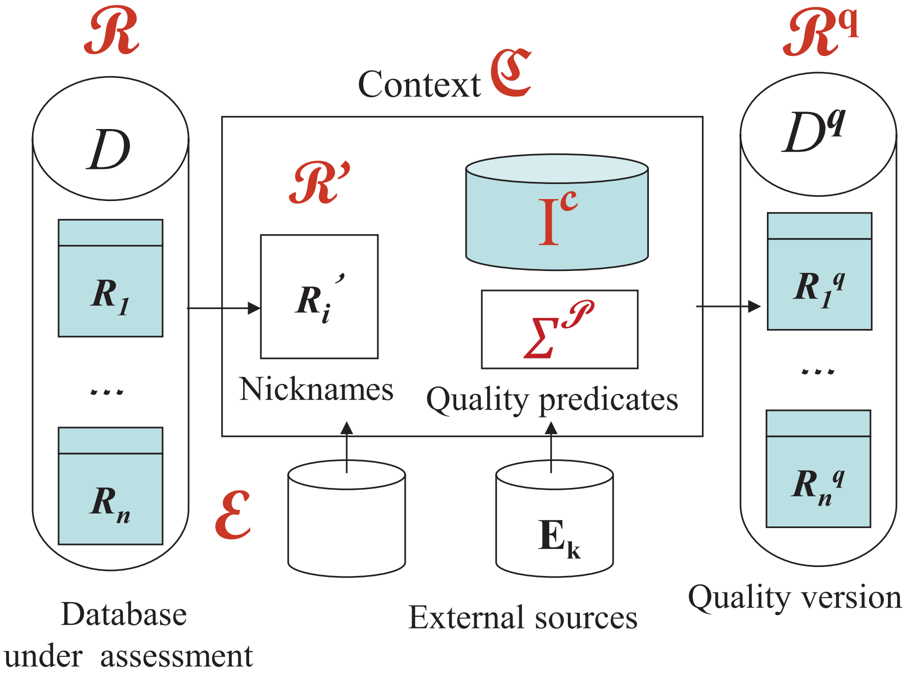


Figure 3.1: A context for data quality assessment

middle, resembles a virtual data integration system, which can also be represented as a logical theory [Lenzerini, 2002]. The context $\mathfrak{C}$ has a relational schema (or signature), in particular predicates with possibly partial extensions (incomplete relations). The mappings between $\mathfrak{C}$ and D are of the kind used in data integration or data exchange [Fagin et al., 2005], that can be expressed as logical formulas. In [Bertossi et al., 2011a, 2016], the concern is not about how such a context is created, but about how it is used for the purpose of data quality specification and extraction.

The context $\mathfrak{C}$ has nicknames (copies) R' for predicates R in $\mathcal{R}$. Nicknames are used to map (via α_i) the data in D into $\mathfrak{C}$, for further logical processing. So, a schema of $\mathfrak{C}$ can be seen as an expansion of $\mathcal{R}$ through a subschema $\mathcal{R}'$ that is a copy of $\mathcal{R}$. Some predicates in the schema of $\mathfrak{C}$ are meant to be *quality predicates* ($\mathcal{P}$ in Figure 3.1), which are used to specify single quality requirements. There may be semantic constraints on the schema of $\mathfrak{C}$, and also access (mappings) to external data sources, in $\mathcal{E}$, that could be used for data quality assessment or cleaning. The schema of $\mathfrak{C}$ also includes a contextual relational schema $\mathcal{R}^c$, with an instance I^c (in the middle of Figure 3.1), which contains materialized data at the level of context.

A clean version of D, obtained through the mapping of D and $\mathfrak{C}$, is possibly a virtual instance D^q, or a collection of thereof $\mathcal{D}^q$, for schema $\mathcal{R}^q$ (a "quality" copy of schema $\mathcal{R}$).[1] The extension of every predicate in it, say R^q, is the "quality version" of relation R in D, and is defined as a view (via the α_i^q) in terms of the nickname predicates in $\mathcal{R}'$, in $\mathcal{P}$, and other contextual predicates.

The quality of (the data in) instance D can be measured by comparing D with the instance D^q or the set, $\mathcal{D}^q$, of them. This latter set can also be used to define and possibly compute the *quality answers* to queries originally posed to D, as the *certain answers* w.r.t. $\mathcal{D}^q$ (cf. [Bertossi et al., 2011a, 2016] for more details). In any case, the main idea is that quality data can be extracted from D by querying the possibly virtual class of quality instances $\mathcal{D}^q$.

In this thesis, we extend the approach to data quality specification and extraction we just described, by adding dimensions to contexts, for multidimensional data quality specification and extraction. In this case, the context contains a generic MD ontology, the shaded $\mathcal{M}$ in Figure 3.2, a.k.a. "core ontology" (and described in Chapter 4). $\mathcal{M}$ represents multidimensional data within the context by means of categorical relations associated with dimensions (the elements in $\mathcal{M}$ in Figure 3.2). This

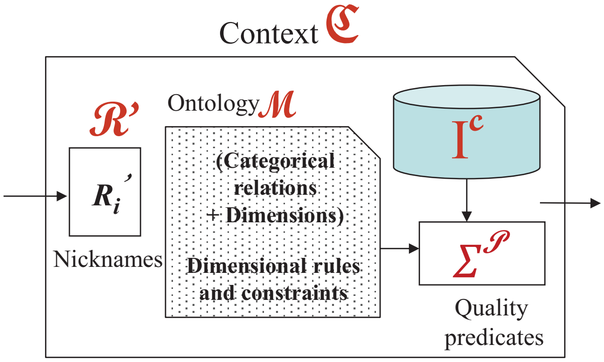


Figure 3.2: A multidimensional context

[1] Figure 3.1 shows the case when there is only one instance D^q. Figure ??, in Section 1.1, better illustrates the case when there is a collection $\mathcal{D}^q$ of instances.

ontology can be extended, within the context, with additional rules and constraints that depend on specific data quality concerns (cf. Chapter 5).

3.2 Querying Context-Aware Databases

In *the context-aware data model* [Martinenghi & Torlone, 2009], the notion of context is implicit and indirectly captured by relational attributes that take as values members of dimension categories.[2] In particular, in a relation in this model, the context of a tuple is captured by its values in dimensions, while the categories of these members specify the granularity level of the context.

Example 3.2.1 Consider relation $\mathit{Schedules}(\mathit{Nurse}, \mathit{Shift}, \mathit{Unit}, \mathit{Day})$, with the tuples $(\mathsf{cathy}, \mathsf{night}, \mathsf{terminal}, \mathsf{sep/5})$ and $(\mathsf{helen}, \mathsf{morning}, \mathsf{standard}, \mathsf{sep/6})$ in its extension. The values of *Unit* and *Day* attributes are members from *Unit* and *Day* categories in the $\mathsf{Hospital}$ and Time dimensions, resp. So, $(\mathsf{terminal}, \mathsf{sep/5})$ and $(\mathsf{standard}, \mathsf{sep/6})$ define the context of these tuples, with the granularity level specified by *Unit* and *Day* categories. □

The context-aware data model has a query language that extends the relational algebra, by introducing new operators for manipulating the granularity of *contextual attributes* (i.e. attributes with values as members of dimensions). These operators add new contextual attributes and their values to a relation. The new attributes are associated with higher or lower categories of the original contextual attributes, and they make it possible to specify contexts with coarser or finer granularities. The language inherits the standard operators from the relational algebra, i.e. projection, selection and join operators.

[2] Dimensions are defined as in the HM model.

In the following, we review the context-aware data model in detail, using our running Example 3.2.1.

Let $\mathcal{H}$ be a set of dimensions. $R^c = (C_1 : l_1, ..., C_m : l_m)$ is a *context schema*, where each C_i is an *attribute name* and each l_i is a level or category of some dimensions in $\mathcal{H}$. A *context* $\bar{c}$ over R^c is a function that maps each attribute C_i to a member of l_i. Notice that multiple attributes can share an attribute name: they represent the same attribute name at different granularity levels. For example, $C : l$ and $C : l'$ represent C in levels l and l', resp.

Example 3.2.2 (ex. 3.2.1 cont.) $\mathit{Schedules}^c = (\mathit{Loc} : \mathit{Unit}, \mathit{Date} : \mathit{Day})$ is a context schema, where $\mathit{Loc} : \mathit{Unit}$ and $\mathit{Date} : \mathit{Day}$ are attributes associated with *Unit* and *Day* attributes in Hospital and Time dimensions, resp.[3] Two possible contexts over $\mathit{Schedules}^c$ are $(\mathsf{terminal}, \mathsf{sep/5})$ and $(\mathsf{standard}, \mathsf{sep/6})$. □

As in the relational data model, $R^r = (A_1 : V_1, ..., A_k : V_k)$ is a *relation schema* (which is different from a context schema), where each A_i is a distinct attribute and each V_i is a set of values called the *domain of* A_i. A *tuple* $\bar{t}$ over a relation schema R^r is a function that associates with each A_i occurring in R^r a value taken from V_i. A *relation over a relation schema* R^r is a finite set of tuples over R^r.

$R(R^r \parallel R^c)$ is a *contextual relation (c-relation) schema*, where R^r is a *relation schema*, and R^c is a *context schema*. A *c-relation (instance)* over R is a set of tuples $\bar{t} = (\bar{r} \parallel \bar{c})$, where $\bar{r}$ is a tuple over R^r, and $\bar{c}$ is a context over R^c.

Example 3.2.3 (ex. 3.2.2 cont.) $\mathit{Schedules}(\mathit{Nurse} : \mathit{String}, \mathit{Shift} : \mathit{String} \parallel \mathit{Loc} : \mathit{Unit}, \mathit{Date} : \mathit{Day})$ is a c-relation schema, where $(\mathit{Nurse} : \mathit{String}, \mathit{Shift} : \mathit{String})$ is a relation schema and $(\mathit{Loc} : \mathit{Unit}, \mathit{Date} : \mathit{Day})$ is a context schema, separated by " $\parallel$ ". A possible

[3] *Loc* is short for *Location*.

extension of *Schedules* contains $(\mathsf{cathy}, \mathsf{night} \parallel \mathsf{terminal}, \mathsf{sep/5})$ and $(\mathsf{helen}, \mathsf{morning} \parallel \mathsf{standard}, \mathsf{sep/6})$, with $(\mathsf{terminal}, \mathsf{sep/5})$ and $(\mathsf{standard}, \mathsf{sep/6})$ as their contexts, resp.□

Context-relational algebra (CRA) is the query language in the context-aware data model that extends relational algebra by introducing two new operators, *upward extension* and *downward extension*, explained below.

Let R be a c-relation with schema $R(R^r \parallel R^c)$ and contextual attribute C in R^c associated to the level l, such that l rolls up to a level l' (cf. Section 2.5 for roll-up relationships). The *upward extension of R from the attribute $C : l$ to l'*, denoted by $\hat{\mathcal{E}}^{C:l'}_{C:l}(R)$, is the c-relation of schema $R(R^r \parallel R^c \cup \{C : l'\})$, defined as follows,

$$\hat{\mathcal{E}}^{C:l'}_{C:l}(R) = \{\bar{t}' \mid \exists\, \bar{t} \in R, \bar{t}'[R^c] = \bar{t}[R^c], \bar{t}'[R^r] = \bar{t}[R^r], \bar{t}'[C : l'] = L_l^{l'}(\bar{t}[C : l])\},$$

where $\bar{t}[C : l]$ is the value of attribute $C : l$ in $\bar{t}$, $\bar{t}[R]$ are the values of attributes of R in $\bar{t}$, and $L_l^{l'}$ is the roll-up relation between levels l and l' (cf. Section 2.5). Intuitively, $\hat{\mathcal{E}}^{C:l'}_{C:l}(R)$ has the same schema of R with additional contextual attribute $C : l'$ that represents C in level l'. Members of the new attribute are specified by roll up (using $L_l^{l'}$) from members of $C : l$ to level l'.

Example 3.2.4 (ex. 3.2.3 cont.) $\hat{\mathcal{E}}^{Loc:Inst}_{Loc:Unit}(Schedules)$ is the upward extension of *Schedules* from *Loc*: *Unit* to the level *Institution* (*Inst* in short), with schema (*Nurse*: *String*, *Shift*:*String* ‖ *Loc*:*Unit*, *Date*:*Day*, *Loc*:*Inst*), where *Loc*:*Inst* is the additional contextual attribute. There are two tuples $(\mathsf{cathy}, \mathsf{night} \parallel \mathsf{terminal}, \mathsf{sep/5}, \mathsf{H}_2)$ and $(\mathsf{helen}, \mathsf{morning} \parallel \mathsf{standard}, \mathsf{sep/6}, \mathsf{H}_1)$ in the extension of $\hat{\mathcal{E}}^{Loc:Inst}_{Loc:Unit}(Schedules)$, where $\mathsf{terminal}$ and $\mathsf{standard}$ roll up to H_2 and H_1, resp. □

Now let l'' be a level such that l *drills down* to l'', i.e. l'' rolls up to l. The *downward extension of R from the attribute $C : l$ to l''*, denoted by $\check{\mathcal{E}}^{C:l}_{C:l''}(R)$, is the c-relation with schema $R(R^r \parallel R^c \cup \{C : l''\})$, defined as follows:

$$\hat{\mathcal{E}}_{C:l}^{C:l''}(R) = \{\bar{t}'' \mid \exists\, \bar{t} \in R, \bar{t}''[R^c] = \bar{t}[R^c], \bar{t}''[R^r] = \bar{t}[R^r], \bar{t}[C:l] = L^{l}_{l''}(\bar{t}''[C:l''])\}.$$

Here, members of the new attribute $C : l''$ are specified by drill down from members of C in level l to level l''.

Example 3.2.5 (ex. 3.2.3 cont.) $\check{\mathcal{E}}_{Loc:Ward}^{Loc:Unit}(Schedules)$ is the downward extension of *Schedules* from *Loc* : *Unit* to the level *Ward*, with schema $(Nurse\!:\!String, Shift\!:\!String \parallel Loc\!:\!Unit, Date\!:\!Day, Loc\!:\!Ward)$, where *Loc*: *Ward* is the additional contextual attribute. There are *three* tuples in the extension of $\check{\mathcal{E}}_{Loc:Ward}^{Loc:Unit}(Schedules)$: $(\mathsf{cathy}, \mathsf{night} \parallel \mathsf{terminal}, \mathsf{sep/5}, \mathsf{W}_4)$, $(\mathsf{helen}, \mathsf{morning} \parallel \mathsf{standard}, \mathsf{sep/6}, \mathsf{W}_1)$, $(\mathsf{helen}, \mathsf{morning} \parallel \mathsf{standard}, \mathsf{sep/6}, \mathsf{W}_2)$. This is because $\mathsf{terminal}$ drills down to W_4 and $\mathsf{standard}$ drills down to *two* members W_1 and W_2. □

The main rationale behind the upward and downward extensions is the need to relax a query with respect to the level of detail of the relations. For example, in the *Schedule* c-relation, one might want to find schedules of a nurse in an institution, even though the schedules might be stored with a lower level (e.g., unit). Both downward and upward extensions meet needs that arise naturally in several application domains [Martinenghi & Torlone, 2010].

The combination of the standard operators of the relation algebra and the new upward and downward extensions makes new operators, e.g. *upward selection* and *downward selection*, that are explained in detail in [Martinenghi & Torlone, 2014]. Context-aware databases have applications beyond context modeling, and they are referred by the more general term of *taxonomy-based databases* [Martinenghi & Torlone, 2010, 2014].

Our extension of the HM model in Section 4.1 has similarities with c-relations and the context-aware data model that we investigate in detail in Section 4.3.6.

3.3 Context-Aware Data Tailoring for Relational Databases

Context-aware data tailoring is defined in [Bolchini et al., 2007a,b, 2009] as the activities of (a) specifying data views over a database, based on the identification of the various contexts the application user is going to experience, and (b) using these views to extract a portion of the database.

In [Bolchini et al., 2007a], context-based data tailoring is included in a design methodology for *very small databases (VSDB)* aimed at being hosted by portable devices. Here the existence of a database is assumed that has a global schema and resides on a central fixed device. VSDBs are defined as collections of materialized views over the database based on different user contexts. In this case, context is defined by a *chunk configuration* that is a set of values for certain "*ambient dimensions*", such as time, space, and situation.

In [Bolchini et al., 2009, 2007b], the ambient dimensions are extended to *context dimension trees (CDTs)* that can represent context in finer granularity. Specifically, a chunk configuration is specified by a set of values for different ambient dimensions and sub-dimensions. The process of extracting database views w.r.t. a chunk configuration is discussed with details in [Bolchini et al., 2007b] where two strategies are proposed for this purpose: (a) *configuration-based*, in which each chunk configuration is linked to a view over a relational database, (b) *value-based*, in which *partial views* are linked to values from the chunk configuration and for each chunk configuration a view is built by combining the relevant partial views.

Example 3.3.1 [Bolchini et al., 2007b] Consider the relational database of a large real estate agency that stores data related to customers, estates, sales with the following schema:

$\mathit{Customers}(\mathit{CustomerId}, \mathit{Name}, \mathit{Surname}, \mathit{Type}, \mathit{Budget}, \mathit{Address}, \mathit{City}, \mathit{PhoneNumber})$

$\mathit{Estate}(\mathit{StateId}, \mathit{OwnerId}, \mathit{Category}, \mathit{StateId}, \mathit{Area}, \mathit{City}, \mathit{Provinance}, \mathit{RoomsNumber}, \mathit{Bedrooms})$

$\mathit{Sales}(\mathit{EstateId}, \mathit{Agent}, \mathit{CustomerId}, \mathit{Date}, \mathit{AgreedPrice}, \mathit{Status})$

The objective is designing a number of views, to be made available to different possible actors of the scenario: supervisors, agents, buyers and sellers.

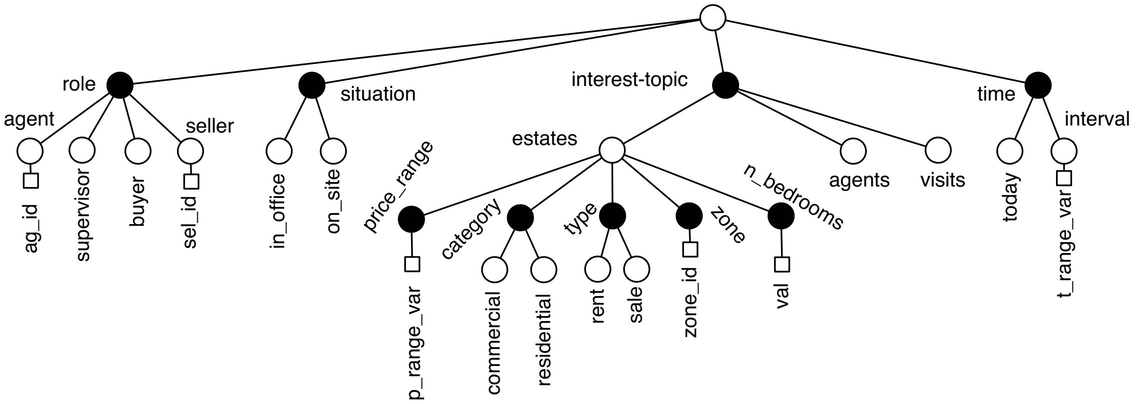


Figure 3.3: A CDT for modeling context.

Figure 3.3 shows the CDT for modeling context in this application domain. In a CDT, the first-level dark nodes, connected to the root, are ambient dimensions and the second-level white nodes connected to them are their possible values. The third-level dark nodes are sub-dimensions that can specify values for each dimension with a finer granularity.

The following chunk configuration, $\mathscr{C}$, specifies the context of an agent who is currently at the office, and needs to check the sales of residential estates located in a certain area that have been concluded today:

$$\mathscr{C} = \langle\{\mathsf{agent}(\$\mathsf{agentId})\}, \{\mathsf{in\text{-}office}\}, \{\mathsf{today}\}, \{\mathsf{residential}, \mathsf{sale}, \mathsf{zone}(\$\mathsf{zoneId})\}\rangle.$$

According to the configuration-based strategy, the chunk configuration is linked to a view defined by a relational calculus expression such as:

$$R(\mathscr{C}) = \sigma_{Category=\mathsf{residential} \wedge Area=\mathsf{\$zoneId}}(Estate) \bowtie \sigma_{Date=\mathsf{today}}(Sale) \bowtie Customer.$$

In the value-based strategy, views for chunk configurations are automatically extracted by combining partial views. Here, let us consider the following partial views associated to values of dimensions, buyer and residential:

$$R(\mathsf{buyer}) = \{\Pi_{Area,RoomsNumber,Bedrooms}(Estate)\}. \tag{3.1}$$

$$R(\mathsf{residential}) = \{\sigma_{Category=\mathsf{residential}}(Estate)\}. \tag{3.2}$$

A view for the chunk configuration $\mathscr{C}' = \langle\{\mathsf{buyer}\}, \{\mathsf{residential}\}\rangle$ is extracted by combining (3.1) and (3.2) using operators, i.e. intersection or union, over relation schemas. □

3.4 Ontology-Based Data Access

Ontology-based data access (OBDA) [Poggi et al., 2008] is an important approach to data access, according to which an ontology is used to mediate between data users and data sources. The ontology allows queries to be formulated in terms of a user-oriented conceptual model that abstracts away complex implementation-level details that are typically encountered in databases. OBDA is a virtual approach, since it provides an access layer on top of databases while leaving the data in its original stores. Thus, OBDA has the potential to improve data access with a minimal change to the existing data management infrastructure.

3.4.1 Description logics

Description logic (DL) [Baader et al., 2007; Calvanese et al., 2007; Artale et al., 2007, 2009] is a family of knowledge representation languages that are widely used in OBDA. It provides the formalism for the *Web Ontology Language (OWL)* as standardised by the *World Wide Web Consortium (W3C)*.[4] However, DLs have been used in knowledge representation long before the advent of OBDA, and the Semantic Web [Baader et al., 2007].

DLs are decidable fragments of FO logic, and as such they are equipped with a formal semantics, which gives meaning to DL ontologies. This formal semantics makes it possible to use logical reasoning, i.e. to infer additional information from the facts and axioms explicitly stated in the ontology. There is not just a single DL, but several, each with a language that balances expressivity and complexity of reasoning.

DL models relationships between individuals in a domain of interest. It is based on three elements: *concepts* representing sets of individuals, *roles* representing binary relations between the individuals, and *individual names* representing single individuals in the domain. For example, an ontology that models relationships in a hospital might use concept such as Doctor to represent the set of all doctors, a role patientOf to represent the (binary) relationship between doctors and their patients, and individual name such as joe to represent the individual Joe.

Unlike a database, a DL ontology does not fully describe *"state of the world"* or situation; rather it consists of a set of *axioms*, each of which must be true. As such, they typically capture only partial knowledge about the situation, and therefore there may be many different states of the world (i.e. models) that are consistent with the ontology.

Axioms in DL are separated into two groups: *assertional (ABox) axioms*, and

[4] https://www.w3.org/

terminological (TBox) axioms. ABox axioms capture knowledge about named individuals, i.e., $\mathsf{Doctor(julia)}$ asserts that the individual named julia is an instance of the concept Doctor. TBox axioms describe relationships between concepts. For example, the fact that all doctors are personnel of the hospital is expressed by the concept inclusion $\mathsf{Doctor} \sqsubseteq \mathsf{Personnel}$, i.e. the concept Doctor is subsumed by the concept $\mathsf{Personnel}$. Also, $\mathsf{Doctor} \sqcap \mathsf{Patient} \sqsubseteq \bot$ is an axiom that states no patient is a doctor ($\bot$ is the concept with no individual).

DLs are distinguished by the constructors they provide in their TBoxes and ABoxes [Baader et al., 2007]. There are various DLs as extensions of the $\mathcal{AL}$*-language (attributive language)*, a minimal language that is of practical interest [Schmidt-Schauß & Smolka, 1991]. For example, the DL $\mathcal{SHOIQ}$ [Horrocks et al., 2006] extends $\mathcal{AL}$ and it is one of the most expressive DLs at the basis of the ongoing standardization of OWL 2, a new version of OWL. Reasoning in $\mathcal{SHOIQ}$ is computationally expensive, and several more tractable languages have been proposed in the Semantic Web community, among them $\mathcal{EL}^{++}$, and the DL-Lite family.

The description logic $\mathcal{EL}^{++}$ is an extension of $\mathcal{EL}$ [Baader et al., 2005] both enjoying PTIME-complete reasoning. *The DL-Lite family* of DLs is proposed and investigated in [Calvanese et al., 2007], and later extended in [Artale et al., 2007, 2009]. The members of this family represent many important types of constraints, and at the same time guarantee good computational properties w.r.t. QA.

For OBDA, DL and Datalog$^\pm$ both serve as ontological languages, and have members with similar properties, in terms of expressivity and complexity of reasoning. Some of the DLs are shown to be subsumed by certain Datalog$^\pm$ members. In particular, linear Datalog$^\pm$ with NCs and non-conflicting keys can express the whole DL-Lite family of tractable description logics. *Guarded* Datalog$^\pm$ can express the tractable DL $\mathcal{EL}$ [Calì et al., 2012b]. In this research, we used Datalog$^\pm$ languages. For ontological query answering and reasoning, they turn out to be extremely versatile and expressive. Datalog$^\pm$ languages naturally extend Datalog, which is a language for querying relational databases.

3.4.2 Closed predicates

The aim of OBDA is to facilitate querying of data that is essentially incomplete. To account for the incompleteness, OBDA formalisms typically adopt the OWA, in which predicates can have any interpretation that includes the predicate's extensional data, and possibly more. In some applications, there are selected parts of the data for which the CWA is more appropriate [Lutz et al., 2013], and the standard semantics from relational databases is adapted: the interpretation of closed predicates is fixed to what is explicitly stated in the extensional data. Making the CWA for dimensional predicates in MD ontologies is a case that we investigate in Section 4.3.1 is an example.

Closed predicates have a strong effect on the complexity of QA, which becomes CoNP-hard in data already when ontologies are formulated in inexpressive DLs such as DL-Lite [Franconi et al., 2011], while CQ answering without closed predicates is in AC_0 for DL-Lite [Calvanese et al., 2007]. The combined complexity in the presence of closed predicates is analysed in [Ngo et al., 2015].

In [Lutz & Wolter, 2012], the main finding is that there are syntactic conditions for ontologies in DL-Lite and $\mathcal{EL}$ that guarantee tractability of CQ answering with closed predicates. In [Lutz et al., 2015], an even more fine-grained approach is taken for analysing the complexity of this problem.

3.4.3 Inconsistency-tolerant ontologies

In real-life applications, involving large amounts of data, it is possible that the data of an ontology is inconsistent with the ontology's rules and constraints. Since the semantics of ontologies follows the FO logic semantics, every fact is trivially entailed by an inconsistent ontology which makes QA pointless. This shows the need for developing *inconsistency-tolerant semantics* for ontological reasoning, that is QA and reasoning is done under the repairs of the inconsistent ontology, i.e. consistent ontologies that minimally depart from the inconsistent ontology.

There has been a recent and increasing focus on the development of *inconsistency-tolerant semantics* for QA purposes [Bienvenu, 2012; Bienvenu & Rosati, 2013; Bienvenu at al., 2014b; Lembo er al., 2010; Lukasiewicz et al., 2012]. The *AR semantics* for several DLs [Lembo er al., 2010] is the most widely accepted semantics for querying inconsistent ontologies, which is based on the ideas of *consistent query answering and repairs* in relational databases [Arenas et al., 1999; Bertossi, 2011b].

According to the AR semantics, an answer is considered to be valid if it can be inferred from each of the repairs of the extensional data D, i.e. the $\subseteq$-maximal consistent subsets of D that make a consistent ontology. Obtaining the set of consistent answers under the AR semantics is known to be a computationally hard problem, even for very simple languages [Lembo er al., 2010]. For this reason, several other semantics have been developed, with the aim of approximating the set of consistent answers [Bienvenu, 2012; Lembo er al., 2010; Lukasiewicz et al., 2012].

The complexity of QA under the AR semantics (and also under several other semantics) for DL ontologies is rather well-understood. The data and combined complexity were studied in [Rosati, 2011], for a wide spectrum of DLs, while [Bienvenu, 2012] identifies cases of simple ontologies (within the DL-Lite family) for which tractable data complexity can be guaranteed.

The AR semantics has also been studied when the ontology is described using Datalog$^{\pm}$ [Lukasiewicz et al., 2012, 2015], where the data complexity of the AR semantics is studied for several classes of tgds enriched with NCs. They show that the problem of BCQ answering under different inconsistency-tolerant semantics is intractable for Datalog$^{\pm}$ ontologies.

Chapter 4

Multidimensional Ontological Data Model

In this chapter, we present the OMD model as an extension of the HM model. We propose a Datalog$^\pm$ representation of the extended model as an MD ontology and we study the computational properties of such ontologies.

4.1 Extending the Hurtado-Mendelzon Data Model

A *database schema* in the *multidimensional-ontological (OMD) data model* is $\mathcal{R}^\mathcal{M} = \mathcal{H} \cup \mathcal{R}^r$, where $\mathcal{H}$ is a relational schema with multiple dimensions (with category predicates $\mathcal{K}$ and child-parent predicates $\mathcal{L}$, as defined in Section 2.5); and $\mathcal{R}^r$ is a set of *categorical predicates*. Categorical predicates replace the fact-tables in the HM model. The attributes of a categorical predicate are either *categorical* that can take values as members of dimensions, or *non-categorical* with arbitrary domains.

We show a categorical predicate as $R(C_1, \ldots, C_m; N_1, \ldots, N_n)$, where, to highlight, categorical and non-categorical attributes (C_is vs. N_js) are separated by ";". The connection between the categorical attributes and the category predicates is established through IDs. In particular, $R[i] \subseteq K_i[1]$ connects categorical attribute C_i to category predicate $K_i(\cdot)$ in $\mathcal{K}$, for $i \leq m$.

The (extensional) data $D^\mathcal{M} = D^\mathcal{H} \cup D^r$ associated to the schema $\mathcal{R}^\mathcal{M}$ has the complete extensions for dimensional predicates (category predicates and child-parent predicates) in $\mathcal{H}$ that come from the instance $D^\mathcal{H}$. The categorical relations in D^r

(with predicates in $\mathcal{R}^r$) may contain partial data, i.e. they may be incomplete.

Example 4.1.1 (ex. 1.1.2 and 2.5.2 cont.) *WorkingSchedules*(*Unit*,*Day*;*Nurse*, *Speciality*) in Figure 1.4 is a categorical relation with categorical attributes *Unit* and *Day*, connected to the Hospital and Time dimensions, resp. This is through IDs, $\mathit{WorkingSchedules}[1] \subseteq \mathit{Unit}[1]$, and $\mathit{WorkingSchedules}[2] \subseteq \mathit{Day}[1]$. *Nurse* is non-categorical, i.e. a foreign key to another relation. The dimension schema $\mathcal{H}$, with $\mathcal{K}$ and $\mathcal{L}$, is as specified in Example 2.5.2, and $\mathcal{R}^r$ contains *WorkingSchedules*. □

A *multidimensional (MD) ontology*, $\mathcal{M}$, in addition to a database $D^{\mathcal{M}}$ with schema $\mathcal{R}^{\mathcal{M}}$ (as in the OMD model), includes a set of *basic constraints* $\Omega_{\mathcal{M}}$, a set of *dimensional rules* $\Sigma_{\mathcal{M}}$, and a set of *dimensional constraints* $\kappa_{\mathcal{M}}$. These rules and constraints are defined over the same schema $\mathcal{R}^{\mathcal{M}}$.

Below, there are the basic egds and NCs in $\Omega_{\mathcal{M}}$, where (a) and (b) represent the ICs of $D^{\mathcal{H}}$, and (c) expresses the IDs that associate categorical relations to category predicates.

(a) IDs between the child-parent predicate $P \in \mathcal{L}$ and the category predicates $K, K' \in \mathcal{K}$ (i.e. $P[1] \subseteq K[1]$ and $P[2] \subseteq K'[1]$), as NCs:

$$P(e, e'), \neg K(e) \ \rightarrow\ \bot. \tag{4.1}$$

$$P(e, e'), \neg K'(e') \ \rightarrow\ \bot. \tag{4.2}$$

Notice that K and K', to which negation is applied, are closed (with complete data in $D^{\mathcal{H}}$).

(b) Key attributes of the child-parent predicates are defined by egds:

$$P(e, e_1), P(e, e_2) \ \rightarrow\ e_1 = e_2. \tag{4.3}$$

(c) IDs between categorical attributes and categories as NCs:[1] $(R \in \mathcal{R}^r,\ K \in \mathcal{K};$ $\bar{e}, \bar{a}$ are categorical, non-categorical, resp.; $e \in \bar{e})$

$$R(\bar{e};\bar{a}), \neg K(e) \ \rightarrow\ \bot. \tag{4.4}$$

Example 4.1.2 (ex. 2.5.2 and 4.1.1 cont.) For the Hospital dimension, the ID $\mathit{WardUnit}[2] \subseteq \mathit{Unit}[1]$ is expressed by constraint of the form (4.2):

$$\mathit{WardUnit}(w,u), \neg\, \mathit{Unit}(u) \ \rightarrow \bot. \tag{4.5}$$

and the key constraint of *WardUnit* is captured by a constraint of the form (4.3):

$$\mathit{WardUnit}(w,u),\ \mathit{WardUnit}(w,u') \ \rightarrow u = u'. \tag{4.6}$$

In *WorkingSchedules*, the categorical attribute *Unit* takes values from the *Unit* category. We use a constraint of the form (4.4), namely:

$$\mathit{WorkingSchedules}(u,d;n,t), \neg\, \mathit{Unit}(u) \ \rightarrow \bot. \tag{4.7}$$

□

Now, we present dimensional rules $\Sigma_{\mathcal{M}}$ and dimensional constraints $\kappa_{\mathcal{M}}$:

(a) *Dimensional constraints*, as egds or NCs: $(R_i \in \mathcal{R}^r,\ P_j \in \mathcal{L}$, and x, x' stand both for either categorical or non-categorical attributes in the body of (4.8))

$$R_1(\bar{e}_1;\bar{a}_1), ..., R_n(\bar{e}_n;\bar{a}_n), P_1(e_1,e_1'), ..., P_m(e_m,e_m') \ \rightarrow\ x = x'. \tag{4.8}$$

$$R_1(\bar{e}_1;\bar{a}_1), ..., R_n(\bar{e}_n;\bar{a}_n), P_1(e_1,e_1'), ..., P_m(e_m,e_m') \ \rightarrow\ \bot. \tag{4.9}$$

[1] As an alternative, we may use tgds between categorical attributes and categories (cf. Section 4.3.1).

(b) *Dimensional rules* as Datalog$^\pm$ tgds:

$$R_1(\bar{e}_1;\bar{a}_1), ..., R_n(\bar{e}_n;\bar{a}_n), P_1(e_1,e_1'), ..., P_m(e_m,e_m') \rightarrow \exists \bar{a}_z\ R_k(\bar{e}_k;\bar{a}_k). \quad (4.10)$$

Here, $\bar{a}_z \subseteq \bar{a}_k$, $\bar{e}_k \subseteq \bar{e}_1 \cup ... \cup \bar{e}_n \cup \{e_1, ..., e_m, e_1', ..., e_m'\}$, $\bar{a}_k \smallsetminus \bar{a}_z \subseteq \bar{a}_1 \cup ... \cup \bar{a}_n$; and repeated variables in bodies are only in positions of categorical attributes (in the categorical relations $R_i(\bar{e}_i;\bar{a}_i)$), and attributes in child-parent predicates $P_j(e_j,e_j')$.[2] Value invention is only on non-categorical attributes (we will consider relaxing this in Section 4.3.1).

Some of the lists in the bodies of (4.8)-(4.10) may be empty, i.e. $n = 0$ or $m = 0$. This allows us to represent, in addition to properly "dimensional" constraints, also classical constraints on categorical relations, e.g. keys or FDs.

Example 4.1.3 (ex. 1.1.2 and 4.1.1 cont.) The constraint η from Example 1.1.2, *"No nurse in the Intensive care unit during January"* is a dimensional (navigational) constraint of the form (4.9):

$$\mathit{WorkingSchedules}(\mathsf{intensive}, d; n, s), \mathit{DayMonth}(d, \mathsf{jan}) \ \rightarrow\ \bot. \quad (4.11)$$

An egd of the form (4.8) says that *"All thermometers in a unit are of the same type"*:

$$\mathit{Therm}(w,t;n), \mathit{Therm}(w',t';n'), \mathit{WardUnit}(w,u), \mathit{WardUnit}(w',u) \rightarrow t = t'. \quad (4.12)$$

with $\mathit{Therm}(\mathit{Ward}, \mathit{Thertype}; \mathit{Nurse})$ a categorical relation, and *Ward*, *Thertype* categorical attributes (the latter for an Instrument dimension). This egd illustrates the flexibility of our approach. Even without having a categorical relation at the *Unit*, we could still impose a condition at that level.[3]

[2] This is a natural restriction since dimensional navigation is captured by the joins (repeated variables) only between variables of these attributes.

[3] If we have that relation, then (4.12) could be replaced by a "static", non-dimensional FD.

The dimensional rules in Example 1.1.2 (σ_1 and σ_2) that generate data from *WorkingSchedules* to *Shifts*, and vice versa are of the form (4.10):

$$\sigma_1: \quad \mathit{Shifts}(w,d;n,s),\ \mathit{WardUnit}(w,u)\ \rightarrow\ \exists t\ \mathit{WorkingSchedules}(u,d;n,t).$$

$$\sigma_2: \quad \mathit{WorkingSchedules}(u,d;n,t),\ \mathit{WardUnit}(w,u)\ \rightarrow\ \exists s\ \mathit{Shifts}(w,d;n,s).$$

The $\exists$-variables t and s make up for the missing, non-categorical attributes *Speciality* and *Shift* in *WorkingSchedules* and *Shifts*. □

Remark 4.1.1 A general tgd of the form (4.10) enables *upward-* or *downward-navigation*, depending on the body joins. The direction is determined by the dimension levels of categorical attributes in the joins. For simplicity, assume that there is a single $P_j \in \mathcal{L}$ in the body (as in σ_1 and σ_2). If the join is between $R_i(\bar{e}_i;\bar{a}_i)$ and $P_j(e_j,e_j')$ then: (a) (one-step) upward navigation is enabled, from e_j' to e_j, when $e_j' \in \bar{e}_i$ (i.e. e_j' appears in $R_i(\bar{e}_i;\bar{a}_i)$) and $e_j \in \bar{e}_k$, i.e in the head), (b) (one-step) downward navigation is enabled, from e_j to e_j', when e_j occurs in R_i and e_j' occurs in R_k. Several occurrences of child-parent predicates in a body capture multi-step navigation. □

Example 4.1.4 (ex. 4.1.3 cont.) Rule σ_2 captures downward-navigation; and this is a general behavior with tgds of the form (4.10). That is when drilling down via σ_2, from a tuple, say $\mathit{WorkingSchedules}(u,d;n,t)$ via the category member u (for Unit), *for each* child w of u in the *Ward* category, a tuple for *Shifts* is generated, as specified in the body of σ_2.

For example, chasing σ_2 with the third tuple in *WorkingSchedules*, generates the new tuple $(\mathsf{W}_2, \mathsf{sep/6}, \mathsf{helen}, \zeta)$ in *Shifts*, with a fresh null, ζ, for the shift. This allows us to answer the query about the wards *Helen* works on *Sep/6*:

$$\mathcal{Q}'(w):\ \exists s\ \mathit{Shifts}(w, \mathsf{sep/6}, \mathsf{helen}, s).$$

We obtain W_1 and W_2.

Instead, the join between *Shifts* and *WardUnit* in σ_1 enables upward-dimensional navigation; and generates only one tuple for *WorkingSchedules* from each tuple in *Shifts*, because each *Ward* member has only one *Unit* parent. □

4.2 Computational Properties and Query Answering

Here, we first establish the membership of our MD ontologies, $\mathcal{M}$ (cf. Section 4.1) in a class of the Datalog$^\pm$ family. Membership is determined by the set $\Sigma_\mathcal{M}$ of its tgds. Next, we analyze the role of the constraints in $\kappa_\mathcal{M}$, in particular, of the set $\kappa_\mathcal{M}$ of egds.

Proposition 4.2.1 MD ontologies are WS Datalog$^\pm$ programs. □

Proof of Proposition 4.2.1: A MD ontology includes dimensional rules of the form (4.10): $R_1(\bar{e}_1;\bar{a}_1),...,R_n(\bar{e}_n;\bar{a}_n), P_1(e_1,e'_1),...,P_m(e_m,e'_m) \rightarrow \exists \bar{a}_z\ R_k(\bar{e}_k;\bar{a}_k)$, in which (a) $\bar{a}_z \subseteq \bar{a}_k$, (b) $\bar{e}_k \subseteq \bar{e}_1 \cup ... \cup \bar{e}_n \cup \{e_1,...,e_m,e'_1,...,e'_m\}$, (c) $\bar{a}_k \smallsetminus \bar{a}_z \subseteq \bar{a}_1 \cup ... \cup \bar{a}_n$, and (d) repeated variables in bodies are only in positions of categorical attributes.

In particular, (a) guarantees that no null values are invented in the categorical positions during the chase of $\mathcal{M}$. Also, (b) ensures that the variables in the non-categorical positions in the bodies do not appear in the heads in the categorical positions. As a result, no null value can replace a variable in a categorical position during the chase. This, in addition with (d), proves that no null value can replace a repeated body variable, which proves a set of dimensional rules is *WS*. □

A consequence of this result is that CQ answering from $\Sigma_{\mathcal{M}}$ is in polynomial-time in data complexity [Calì et al., 2012c]. The complexity stays the same if we add dimensional NCs, of the forms (4.9), because they can be checked through the CQs in their bodies (cf. Section 2.4.1). In case of dimensional egds of the form (4.8), we consider a syntactic condition that guarantees separability:

Proposition 4.2.2 For an MD ontology $\mathcal{M}$ with a set $\Sigma_{\mathcal{M}}$ of tgds as in (4.10) and set $\kappa_{\mathcal{M}}$ of egds as in (4.8), separability holds if, for every egd in $\kappa_{\mathcal{M}}$, the variables in the equality (in the head) occur in categorical positions in the body. □

Proof of Proposition 4.2.2: Let Π be the Datalog$^{\pm}$ program with database $D_{\mathcal{M}}$, Σ_M and κ_M as its rules Π^R and constraints Π^C, resp. Let Π' be Π with only Π^R (without Π^C).

In order to prove that Π^R and Π^C are separable, we need to show that if *chase*(Π) does not fail, then *chase*$(\Pi) \models \mathcal{Q}$ if and only if *chase*$(\Pi') \models \mathcal{Q}$. In the proof of Proposition 4.2.1, it is shown that no null value replaces a variable in a categorical position during the chase of $\mathcal{M}$. The variables in the head of every egd appear in the body only in the categorical positions. Therefore, they are *not* replaced by nulls. As a result, the egds can only equate constants, which results into hard violations and inconsistency. Since we assumed Π is consistent, the egds are never applicable during the chase, and they can be ignored: *chase*$(\Pi) \models \mathcal{Q}$ if and only if *chase*$(\Pi') \models \mathcal{Q}$. □

Adding egds and NCs in $\Omega_{\mathcal{M}}$ does not change the complexity of QA, as discussed in Sections 2.4.1 and 2.4.2. Note that egds of the form (4.3) are non-conflicting and separable. That is because they satisfy the first non-conflicting condition in Definition 2.4.3. In combination with Proposition 4.2.1, we obtain:

Corollary 4.2.1 Under the hypothesis of Proposition 4.2.2, CQ answering from an MD ontology can be done in polynomial-time in data. □

4.3 Discussion and Extensions

Here, we study some extensions of the OMD model.

4.3.1 Uncertain downward-navigation and closed predicates

In the OMD model, the dimensional rules of the form (4.10) do not allow existential quantifiers on variables in the categorical positions. Notice that sometimes in logic, we use existential quantifiers as a way of referring to an element in a specified set (that is as a disjunction on the set of elements). In dimensional rules, we can use existential quantifiers to refer to the parent of a child member without explicitly mentioning the former. This kind of existential quantification can be avoided, because the parent is unique as captured by the child-parent relation and the FD as stated by the egd (4.3). It might not be desirable to use an existential quantifier in this case (upward navigation), because in principle a new parent could be invented that would be forced by (4.3) to be the same as the original parent which only creates additional complexity and possible semantic collision.

If we allow existential quantifiers on variables in categorical positions while going downward, they stand for existing child members and possibly new ones. This is a more complex situation because we have more than one child members and there is no egd such as (4.3) in downward direction. All this becomes more relevant in real applications of MD ontologies where category predicates and child-parent predicates are considered closed.

Table 4.1: *DischargePatients*

	Inst.	Day	Patient
1	H_1	Sep/9	Tom Waits
2	H_1	Sep/6	Lou Reed
3	H_2	Oct/5	Elvis Costello
4	H_1	Dec/16	Elvis Costello

Table 4.2: *PatientUnit*

	Unit	Day	Patient
1	Standard	Sep/5	Tom Waits
2	Standard	Sep/9	Tom Waits
3	Intensive	Sep/6	Lou Reed

Example 4.3.1 Consider *DischargePatients* (Table 4.1) and *PatientUnit* (Table 4.2), that contain data on patients leaving an institution and on locations of patients, resp. Since, a patient was in a unit when discharged, we can use *DischargePatient* to generate data for *PatientUnit*, at the *Unit* level, down from the *Institution* level. For example, through the following rule:

$$\mathit{DischargePatients}(i,d;p) \ \rightarrow \ \exists u \ (\mathit{UnitInstitution}(u,i), \mathit{PatientUnit}(u,d;p)). \quad (4.13)$$

Notice that (4.13) is not of the form (4.10): (a) it can invent values in the non-categorical position *PatientUnit*[1], (b) it has the child-parent predicate *UnitInstitution* in its head, and (c) it has two head atoms. For (c), it can be resolved by transforming (4.13) into multiple rules with single head atoms.[4] We used two head atoms to better convey (a) and (b).

The ∃-variable u in (4.13) (and then, value invention) appears in the first, i.e. "downward" attribute of the child-parent relation *UnitInstitution*. Inventing such a value in this relation amounts to creating possibly new members in categories, which in many applications we would consider to be given by a finite and closed extension. Categories are normally "complete". □

In Example 4.1.3, we adopted the usual OWA semantics of Datalog$^\pm$. There was no problem with upward tgds, such as σ_1, nor with "regular" downward tgds,

[4] In this case, the rules are $\mathit{DischargePatients}(i,d;p) \ \rightarrow \ \exists u \ \mathit{TempPatient}(i,u,d;p)$, $\mathit{TempPatient}(i,u,d;p) \ \rightarrow \ \mathit{UnitInstitution}(u,i)$, and $\mathit{TempPatient}(i,u,d;p) \ \rightarrow \ \mathit{PatientUnit}(u,d;p)$.

such as σ_2: they invent values only for non-categorical attributes. However, as in Example 4.3.1, we might object value invention in complete child-parent relations and categories due to the application of non-deterministic downward rules, such as (4.13). If we accept this kind of tgds and, at the same time, we consider dimensional predicates, i.e. category predicates and child-parent predicates, as closed, then we start departing from the usual Datalog$^\pm$ semantics, and some of the results we reuse or provide for *WS* programs (with OWA semantics) have to be reconsidered (cf. Section 3.4.2).

A semantics with a combination of closed dimensional predicates and open categorical predicates greatly impacts CQ answering under MD ontologies (we discussed this semantics for DL and Datalog$^\pm$ ontologies in Section 3.4.2). In fact, the complexity results in Section 4.2 do not hold under this semantics, and the problem of CQ answering becomes intractable. Intuitively, this is because of the combinatorial choices of child members during uncertain downward-navigation from a parent member, in the non-deterministic downward rules, such as (4.13).

Example 4.3.2 Consider a MD ontology $\mathcal{M}$ with $\mathit{ThermType}(\mathit{Type};\mathit{Nurse})$ and $\mathit{ThermBrand}(\mathit{Brand};\mathit{Nurse})$ as categorical relations, specifying the types and brands of thermometers used by nurses. *ThermType* and *ThermBrand* have categorical attributes *Type* and *Brand* associated with categories *Type* and *Brand* in the Instrument dimension. $\mathcal{M}$ also includes the dimensional rule:

$$\mathit{ThermType}(t;n) \ \rightarrow \ \exists b \ (\mathit{BrandType}(b,t), \mathit{ThermBrand}(b;n)), \tag{4.14}$$

where *BrandType* is a child-parent predicate in Instrument (cf. Figure 4.1), which is closed, with its complete data in the extensional database. Table 4.3 shows the extension of *ThermType*, while *ThermBrand* is defined by Rule (4.14).

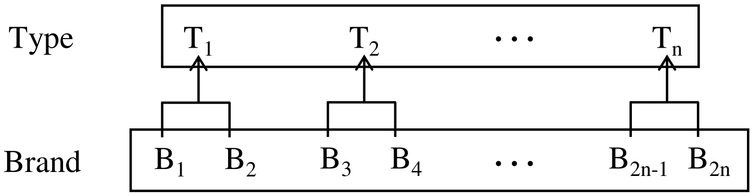


Figure 4.1: The *Brand* and *Type* categories.

$\mathcal{M}$ has exponentially many models (w.r.t. the size of the extensional data). That is because, moving from each parent member T_i in *Type* category during the chase with $\mathcal{M}$, there are two possible child members B_{2i-1} and B_{2i} in *Brand* category. There are n binary choices for n nurses in these relations, which make 2^n possible extensions for *ThermBrand*. □

Table 4.3: *ThermType*

	Type	**Nurse**
1	T_1	N_1
2	T_2	N_2
$\vdots$	$\vdots$	$\vdots$
n	T_n	N_n

The exponentially many models for the MD ontology in Example 4.3.2 suggests that CQ answering under MD ontologies with closed dimensional predicates semantics is intractable in data complexity. In fact, the problem is CoNP-hard even for inexpressive Datalog$^\pm$ and DL ontologies [Ahmetaj et al., 2016; Franconi et al., 2011].

4.3.2 Categorical keys

In the HM model, attributes in a fact-table, excluding the measure, form a key in the fact-table. In other words, a point in the multidimensional space is mapped by a fact-table to at most one measure. This is a natural assumption in the DWHs and OLAP applications. However, in the OMD model, categorical attributes are not necessarily key attributes for categorical relations. For example in *WorkingSchedules(Unit,Day;Nurse,Speciality)*, *Unit* and *Day* are not key attributes since multiple nurses might have working schedules in the same unit and on the same day.

There are still cases when categorical attributes define key attributes in categorical relations. For example in *InstitutionBoard(Institution;Chair,President,CEO)*, if there is only one board directory (i.e. chair, ceo, and president) for an institution, then attribute *Institution* is a key attribute for the relation.

Here, we discuss the effect of considering *categorical keys*, i.e. making categorical attributes form a key for their categorical relations. More precisely, we assume in every categorical relation $R(C_1, ..., C_n; A_1, ..., A_m)$, the set $\{C_1, ..., C_n\}$ is a key for R. This can be expressed in the MD ontology by egds ($y_i \in \bar{y}$ and $y_i' \in \bar{y}'$):

$$R(\bar{x}; \bar{y}), R(\bar{x}; \bar{y}') \ \rightarrow \ y_i' = y_i. \tag{4.15}$$

For example the categorical key for *InstitutionBoard* is captured by egds, among them the following:

$$\mathit{InstitutionBoard}(i; c, p, e), \mathit{InstitutionBoard}(i; c', p', e') \ \rightarrow \ c' = c.$$

The following example shows that dimensional rules and categorical keys defined by (4.15) are not necessarily separable (cf. Section 2.4.2).

Example 4.3.3 Consider the categorical relation *InstitutionBoard* with the key attribute *Institution*, and the following dimensional rules (they are different in the underlined ∃-variable in heads):

$$\mathit{PatientUnit}(u, d; p), \mathit{UnitInstiution}(u, i) \rightarrow \exists c \, \exists n \ \mathit{InstitutionBoard}(i; c, \underline{c}, n). \tag{4.16}$$

$$\mathit{PatientUnit}(u, d; p), \mathit{UnitInstiution}(u, i) \rightarrow \exists c \, \exists n \ \mathit{InstitutionBoard}(i; c, \underline{n}, n). \tag{4.17}$$

Let $(\mathsf{standard}, \mathsf{sep/5}; \mathsf{tom\ waits})$ be the only tuple in the extension of *PatientUnit*. The egds defining the key attribute *Institution* are not separable from dimensional tgds (4.16) and (4.17) because: (a) the chase does not fail since the egds only equate nulls invented by (4.16) and (4.17), and (b) the BCQ $\mathcal{Q}: \exists i \, \exists c \ \mathit{InstitutionBoard}(i, c, c, c)$

has a negative answer without the categorical key, but its answer is *true* with the categorical key. □

Although the dimensional rules and the categorical keys are not separable from each other, QA on MD ontologies with categorical keys is still in PTIME in data complexity. This is because no null value appears in the positions of categorical attributes, therefore there are polynomially many (w.r.t. the size of data) applicable pairs of tgd/assignment during the chase, and after each of the tgd-chase steps. This shows that the chase runs in polynomial time for an MD ontology under the categorical key assumption.

Proposition 4.3.1 The data complexity of CQ answering on MD ontologies with categorical keys is in PTIME. □

4.3.3 Inconsistency-tolerant multidimensional ontologies

In Section 2.4, we discussed QA in the presence of dimensional constraints (dimensional NCs and egds). So far, we have considered QA with consistent MD ontologies, where constraints are satisfied. If the ontology is inconsistent, i.e. dimensional constraints are not satisfied, CQ answering becomes trivial and pointless since every BCQ is answered positively under the ontology.

We consider an inconsistency-tolerant semantics that gives meaningful answers to queries under inconsistent MD ontologies. This semantics is based on repairing extensional database, as in the AR semantics for Datalog$^\pm$ and DL ontologies (cf. Section 3.4.3). According to this semantics, given an inconsistent ontology $\mathcal{M}$, a repair $\mathcal{M}_r$ is a consistent MD ontology with the same rules and constraints in $\mathcal{M}$, and a database $D^{\mathcal{M}_r}$ that is maximally contained in $D^{\mathcal{M}}$. Answers to a CQ $\mathcal{Q}$ are those that are obtained from every repair of $\mathcal{M}$.

QA under this semantics is intractable in the size of the extensional database as suggested by Example 4.3.4. In fact, there are results that show QA under the AR semantics is NP-hard in data complexity, even for inexpressive Datalog$^\pm$ ontologies such as linear and sticky Datalog$^\pm$ [Lukasiewicz et al., 2012, 2015]. This means the complexity results in Section 4.2 do not hold for inconsistency-tolerant QA under MD ontologies.

Table 4.4: *ThermBrand*

	Brand	**Nurse**
1	B_1	N_1
2	B_2	N_1
3	B_3	N_2
4	B_4	N_2
⋮	⋮	⋮
2*n*-1	B_{2n-1}	N_n
2*n*	B_{2n}	N_n

Example 4.3.4 Assume categorical relation *ThermBrand* with extensional data in Table 4.4 (in Example 4.3.2, it did not have extensional data and it was defined by (4.14)) and the following egd that states *"Every nurse uses thermometers of the same brand"*:

$$\epsilon: \mathit{ThermBrand}(b; n), \mathit{ThermBrand}(b'; n) \ \rightarrow \ b = b'. \tag{4.18}$$

There is no dimensional tgd, so ϵ is trivially separable. According to the extensional data of *ThermBrand*, ϵ does not hold since each nurse N_i uses thermometers of two brands B_{2i-1} and B_{2i}. The data in Table 4.4 can be repaired in different ways: for each pair of tuples (B_{2i-1}, N_i) and (B_{2i}, N_i), either of them can be removed. Since there are n pairs, 2^n different repairs are possible, which suggests that QA under these repairs is not tractable in the size of the extensional data. □

In order to obtain tractability of CQ answering under inconsistent ontologies, we suggest a novel and general approach for inconsistency-tolerant QA. The NCs (and egds, mainly in the separable case) can and are checked on the result of the chase (cf. Section 2.4). A possible more natural and practical approach would be to integrate constraint checking with data generation, restricting the latter process. We consider two ways to apply this approach: (a) by compiling the constraints into the tgds to restrict the data generation and prevent inconsistency, or (b) by checking the constraints continuously after each tgd-chase step [Calì et al., 2010a, 2012c] during the chase procedure in order to detect inconsistency sooner. In (a), a form of stratified negation [Alviano & Pieris, 2015; Calì et al., 2013] might be needed to impose the constraints using negation in the tgds bodies. Next, (a) and (b) are explained in Examples 4.3.5 and 4.3.6, resp.

Example 4.3.5 Consider the categorical relations *PatientUnit* and *PatientWard*, and the following dimensional rule:

$$PatientUnit(u, d; p),\ WardUnit(w, u)\ \rightarrow PatientWard(w, d; p), \tag{4.19}$$

and a dimensional NC that says there is no patient in the wards of the hospital during *September*:

$$PatientWard(w, d; p), DayMonth(d, \mathsf{september})\ \rightarrow \bot. \tag{4.20}$$

Here, (4.19) and (4.20) can be complied into the following tgd:

$$\begin{aligned} PatientUnit(u, d; p),\ WardUnit(w, u), \neg DayMonth(d, \mathsf{september})& \\ \rightarrow PatientWard(w, d; p).& \end{aligned} \tag{4.21}$$

that applies the constraint while performing dimensional navigation. The negation in the body of (4.21) is stratified since it is applied on a dimensional predicate with complete data in the extensional database. □

Example 4.3.6 (ex. 4.3.5 cont.) Consider the following additional dimensional rule:

$$PatientWard(w, d; p), DayMonth(d, m)\ \rightarrow \exists s\ PatientDiagnosis(w, m; p, s). \tag{4.22}$$

According to the data of *PatientUnit* in Table 4.2, the tuples $(\mathsf{intensive}, \mathsf{sep/1}, \mathsf{tom\ waits})$ and $(\mathsf{intensive}, \mathsf{sep/6}, \mathsf{lou\ reed})$ generate tuples in *PatientWard*, through (4.19), that violate (4.20). If we postpone checking (4.20) to after the complete data generation by the chase, these tuples in turn generate tuples in *PatientDiagnosis* using (4.22). However, by continuously checking (4.20), we can detect the error caused by these two tuples sooner (and resolve the inconsistency possibly by repair), which prevents additional erroneous data generation in *PatientDiagnosis*. □

4.3.4 Dimensional vs. static constraints

In the OMD model, dimensional constraints are egds and NCs of the form (4.8) and (4.9), resp., with body atoms of the child-parent predicates for dimensional navigation as explained in Remark 4.1.1. By static constraints we refer to egds and NCs without these child-parent atoms in their bodies (for example, key constraints and FDs). Here, we briefly study the connection between the two types of constraints.

Dimensional constraints can be transformed into static constraints and dimensional rules of the form (4.10), as it is shown by the next example.

Example 4.3.7 (ex. 4.1.3 cont.) Consider the dimensional egd (4.12):

$$\mathit{Therm}(w, t; n),\, \mathit{Therm}(w', t'; n'), \mathit{WardUnit}(w, u), \mathit{WardUnit}(w', u) \rightarrow t = t'.$$

We can split it into a dimensional rule of the form (4.10) and a static egd as follows:

$$\mathit{Therm}(w, t; n),\, \mathit{WardUnit}(w, u) \;\rightarrow\; \mathit{ThermTemp}(u, t; n).$$

$$\mathit{ThermTemp}(u, t; n),\, \mathit{ThermTemp}(u, t'; n') \;\rightarrow\; t = t'.$$

Similarly, the dimensional constraint (4.11),

$$\mathit{WorkingSchedules}(\mathsf{intensive}, d; n, s),\, \mathit{DayMonth}(d, \mathsf{jan}) \;\rightarrow\; \bot,$$

can be expressed by the following dimensional rule and static NC:

$$[\mathit{WorkingSchedules}(u,d;n,s),\mathit{WardUnit}(w,u),$$
$$\mathit{DayMonth}(d,m)] \ \rightarrow \mathit{SchedulesTemp}(u,m).$$
$$\mathit{SchedulesTemp}(\mathsf{intensive},\mathsf{jan}) \ \rightarrow \bot. \qquad \square$$

Dimensional constraints of the forms (4.8) and (4.9) prevent some unnecessary steps during the chase by avoiding data propagation for the additional predicates (e.g. *ThermTemp* and *SchedulesTemp*).

Dimensional constraints can represent static constraints since static NCs and egds are special cases of the general forms (4.8) and (4.9), resp., without child-parent atoms in their bodies. Notice that the only egds in the model are those obtained from the transformation we just illustrated in the example. However, these new egds are still expected to be separable from the dimensional rules for good computational properties. For example, they could satisfy the condition in Proposition 4.2.2.

4.3.5 Summarizability in multidimensional ontologies

Like in the relational data model, semantic constraints can be applied on the HM model. *Strictness* and *homogeneity* are two important constraints on dimensions that ensure the *summarizability property*, a desirable property that guarantees the correct computation of cube views [Hurtado & Mendelzon, 2002]. A dimension is *strict*, i.e. each member in a category has at most one parent in each higher category. It satisfies *homogeneity (a.k.a. covering)* if each member in a category has at least one parent in a parent category.

Example 4.3.8 The Hospital dimension (Figure 2.4) satisfies both strictness and homogeneity. In Figure 4.2, the dimension on the left-side is not strict, because the

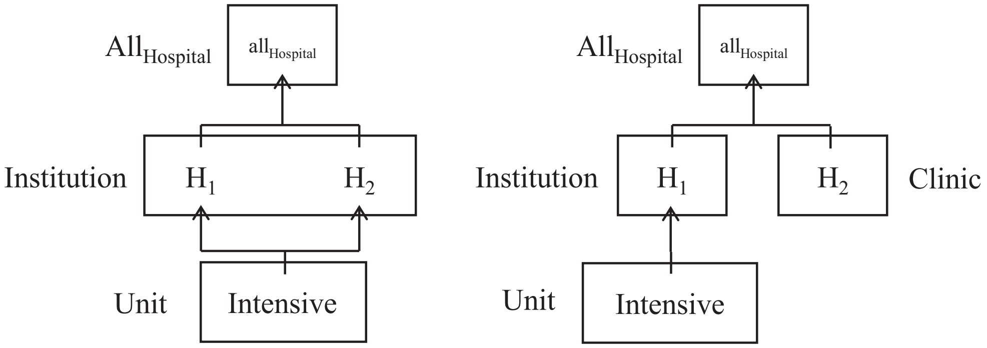


Figure 4.2: Homogenously and strictness of dimensions

Intensive child member in the *Unit* category has two parent members, H_1 and H_2 in the *Institution* category. Assuming that *Clinic* and *Institution* are both parent categories of the *Unit* category, the dimension on the right-side is not homogeneous, because the child member of *Intensive* does not have a parent member in the *Clinic* category. □

We can represent homogeneity and strictness in the MD ontology through tgds, egds, and NCs. To do that, we first introduce a binary predicate T_H defined by rules (4.23) and (4.24). There is one rule of this form for every child-parent predicate $P \in \mathcal{L}$.

$$P(e_1, e_2) \ \rightarrow \ T_H(e_1, e_2). \tag{4.23}$$

The transitivity of T_H is imposed by the following rule:

$$T_H(e_1, e_2), T_H(e_2, e_3) \ \rightarrow \ T_H(e_1, e_3). \tag{4.24}$$

The strictness constraint on dimension H can be captured by egds of the form (4.25). There is an egd for every *intermediate category* $K \in \mathcal{K}$.[5]

$$T_H(e_1, e_2), T_H(e_1, e_3), K(e_2), K(e_3) \ \rightarrow \ e_2 = e_3. \tag{4.25}$$

[5] An intermediate category is a category that is not a base or top category.

Under strictness and homogeneity, each member of a child category has a unique member in the parent category. Homogeneity of H can be represented by tgds of the form (4.26). There is a tgd for every child-parent predicate $P \in \mathcal{L}$, between the child category K and the parent category K'.

$$K(e) \ \rightarrow \ \exists e' \ P(e,e'). \tag{4.26}$$

Notice that (4.26) can also be expressed by the following tgd and NC:

$$P(e,e') \ \rightarrow C_K(e). \tag{4.27}$$

$$K(e), \neg C_K(e) \ \rightarrow \bot. \tag{4.28}$$

Here, C_K is an auxiliary, defined predicate that collects the child members from the child-parent predicate P, and the negation in (4.28) is stratified. In fact, this representations is preferred to (4.26) since it expresses homogeneity as a NC rather than as a data generation rule.

Example 4.3.9 (ex. 4.3.8 cont.) In the Hospital dimension, the following rules capture strictness:

$$T_{\mathit{Hospital}}(e_1,e_2),\, T_{\mathit{Hospital}}(e_1,e_3),\, \mathit{Unit}(e_2),\, \mathit{Unit}(e_3) \ \rightarrow \ e_2 = e_3.$$

$$T_{\mathit{Hospital}}(e_1,e_2),\, T_{\mathit{Hospital}}(e_1,e_3), \mathit{Institution}(e_2), \mathit{Institution}(e_3) \ \rightarrow \ e_2 = e_3.$$

The homogeneity constraint is imposed by the following rule, among others:

$$\mathit{Ward}(w) \ \rightarrow \ \exists u \ \mathit{WardUnit}(w,u),\, \mathit{Unit}(u).$$

This can be expressed by the following tgd and NC:

$$\mathit{WardUnit}(w,u) \ \rightarrow C_{\mathit{Ward}}(w).$$

$$\mathit{Ward}(w), \neg C_{\mathit{Ward}}(w) \ \rightarrow \bot.$$

□

4.3.6 Reconstruction of the context-aware databases

The *context-aware data model* [Martinenghi & Torlone, 2009, 2010, 2014] can be fully reconstructed in terms of the OMD data model. The standard operators of the relational algebra, i.e. selection, projection, and natural join, are supported by Datalog and inherited by the MD ontology. The upward and downward extensions are also expressible by means of dimensional rules of the form (4.10). The MD model can additionally express recursion and contain incomplete data, both not expressible in the context-aware databases.

Example 4.3.10 (ex. 3.2.3 cont.) The *Schedules* c-relation in the context-aware data model can be represented as a categorical relation with the categorical and non-categorical attributes that correspond to the contextual attributes and relation attributes of the c-relation, resp. In particular, $\mathit{Schedule}(\mathit{Loc},\mathit{Date};\mathit{Nurse},\mathit{Shift})$ is a categorical relation that represents the c-relation, $\mathit{Schedules}(\mathit{Nurse}\!:\!\mathit{String}, \mathit{Shift}\!:\! \mathit{String} \parallel \mathit{Loc}\!:\!\mathit{Unit}, \mathit{Date}\!:\!\mathit{Day})$. In the categorical relation, *Loc* and *Date* are categorical attributes taking values from *Unit* and *Day* categories in the Hospital and Time dimensions; and *Nurse* and *Shift* are non-categorical attributes.

The result of upward extension, $\hat{\mathcal{E}}^{\mathit{Loc}:\mathit{Inst}}_{\mathit{Loc}:\mathit{Unit}}(\mathit{Schedules})$, is a categorical predicate, $\mathit{Schedules}'$, which is defined by a dimensional rule of the form (4.10):

$$\mathit{Schedules}(u,d;n,s),\, \mathit{UnitInstitution}(u,i) \;\rightarrow\; \mathit{Schedules}'(u,d,i;n,s).$$

Similarly, the result of downward extension, $\check{\mathcal{E}}^{\mathit{Loc}:\mathit{Unit}}_{\mathit{Loc}:\mathit{Ward}}(\mathit{Schedules})$, is a categorical predicate, $\mathit{Schedules}''$, which is defined by a rule:

$$\mathit{Schedules}(u,d;n,s),\, \mathit{WardUnit}(w,u) \;\rightarrow\; \mathit{Schedules}''(u,d,w;n,s). \qquad \square$$

The context-aware data model and its query language inherits the limitations of relational algebra, including the following (that are necessary in many applications of the OMD data model [Milani et al., 2014; Milani & Bertossi, 2015b]): (1) It can not capture recursive queries on the hierarchical data, (2) It is unable to represent incomplete data.

Chapter 5

Multidimensional Ontologies and Data Quality

The OMD model provides a formal representation of the multidimensional context as a core MD ontology. This allows us to establish a framework for contextual data quality assessment.

5.1 Contextual Data Quality Assessment Revisited

We now show in detail the role of a MD context in quality data specification and extraction. We will at the same time, for illustration and fixing ideas, use an example (an extension of the running examples in Chapter 1), to put it in terms of the MD context elements.[1]

Example 5.1.1 The relational table *Temperatures* (Table 5.1) shows body temperatures of patients in a hospital. A doctor wants to know *"The body temperatures of Tom Waits for August 21 taken around noon with a thermometer of brand B_1 and by a certified nurse"*. Possibly a nurse, unaware of this requirement, took a measurement and stored the data in *Temperatures*. In this case, not all the measurements in the table are up to the expected quality. However, table *Temperatures* alone does not discriminate between the intended values (those taken with brand B_1 and by a certified nurse) and the others.

[1] Note that the tables in Chapter 1 reappear in this chapter, sometimes with a few changes in their data to convey the ideas in more detail.

Table 5.1: *Temperatures*

	Time	Patient	Value	Nurse
1	Sep/1-12:10	Tom Waits	38.2	Anna
2	Sep/6-11:50	Tom Waits	37.1	Helen
3	Nov/12-12:15	Tom Waits	37.7	Alan
4	Aug/21-12:00	Tom Waits	37.0	Sara
5	Sep/5-11:05	Lou Reed	37.5	Helen
6	Aug/21-12:15	Lou Reed	38.0	Sara

For assessing the quality of the data in *Temperatures* according to the doctor's quality requirement, extra contextual information about the thermometers and the nurses may help. In this case, the contextual information is in categorical relations *WorkingSchedules*, *Shifts*, and *Personnel* shown in Tables 5.2-5.4, resp. *WorkingSchedules* and *Shifts* have working schedules and shifts of nurses in units and wards of the hospital, resp. Table *Personnel* stores hiring dates of personnel in the hospital.

Furthermore, the institution has two *guidelines* prescribing that:

(a) *"Temperature measurements for patients in intensive care unit have to be taken with thermometers of Brand B_1".*

(b) *"Personnel hired after February are certified".*

Guideline (a) can be used for data quality assessment when combined with categorical table *WorkingSchedules*, which is linked to the *Unit* category. The data for *WorkingSchedules* is partial and can be completed by table *Shifts*, by upward navigation through the Hospital dimension from category *Ward* to category *Unit*. Tuples

Table 5.2: *WorkingSchedules*

	Unit	Day	Nurse	Speciality
1	Terminal	Sep/5	Cathy	Cardiac Care
2	Intensive	Nov/12	Alan	Critical Care
3	Standard	Sep/6	Helen	?
4	Intensive	Aug/21	Sara	?

Table 5.3: *Shifts*

	Ward	Day	Nurse	Shift
1	W_4	Sep/5	Cathy	Noon
2	W_1	Sep/6	Helen	Morning
3	W_3	Nov/12	Alan	Evening
4	W_3	Aug/21	Sara	Noon
5	W_2	Sep/6	Helen	?

that are obtained through dimensional navigation and data generation are shown shaded in Tables 5.2.

According to (a), it is possible to conclude that tuples 3,4, and 6 in *Temperatures* contain measurements taken with a thermometer of brand B_1. In particular, the nurses that took the measurements (*Alan* and *Sara*) were in the intensive care unit (according to *WorkingSchedules*).

Using guideline (b), only tuples 4 and 6 in *Temperatures* are measurements taken by a certified nurse, *Sara*, since she is hired after *February*, according to table *Personnel*. This "clean data" in relation to the doctor's expectations appear in relation *Temperatures*^{q} (Table 5.5) that can be seen as a quality version of *Temperatures*.

Table 5.4: *Personnel*

	Inst.	**Day**	**Name**
1	H_2	Sep/5	Anna
2	H_1	Mar/9	Helen
3	H_1	Jan/6	Alan
4	H_1	Mar/6	Sara

In the OMD model, there could be semantic constraints, represented as *dimensional constraints*. For example, a constraint that states *"No nurse in intensive care unit during January"*. It is satisfied by table *WorkingSchedules* (Table 5.2) since none of the tuples shows a working schedule during January. Another example is a constraint saying *"No nurse has working schedules in more than one institution on the same day"*, which is also satisfied by *WorkingSchedules*.

Table 5.5: *Temperatures*^{q}

	Time	**Patient**	**Value**	**Nurse**
1	Aug/21-12:00	Tom Waits	37.0	Sara
2	Aug/21-12:15	Lou Reed	38.0	Sara

According to the clean data in *Temperatures*^{q}, the second tuple provides the answer to the query. □

Figure 5.1 shows the overview of our general methodology for contextual data quality specification and extraction using MD ontologies. On the LHS, D is a database

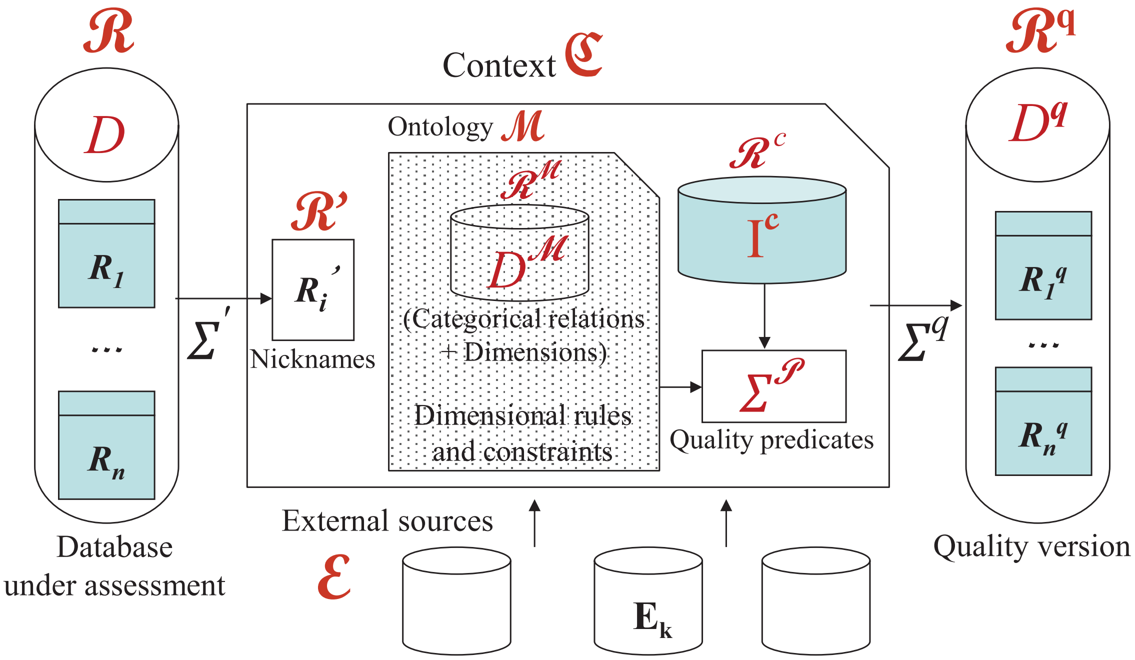


Figure 5.1: A multidimensional context

instance for a relational schema $\mathcal{R} = \{R_1, ..., R_n\}$ that is under quality data specification, assessment, and extraction.

The main element is a context $\mathfrak{C}$, shown in the middle of Figure 5.1. It contains the following:

1. Nickname predicates R' in a nickname schema $\mathcal{R}'$ for predicates R in $\mathcal{R}$. Predicates R' have the same extensions as the corresponding ones R in D, producing a material or virtual instance D' within $\mathfrak{C}$. These nickname predicates are defined by a set Σ' of non-recursive Datalog rules of the form:

$$R(\bar{x}) \ \rightarrow \ R'(\bar{x}). \tag{5.1}$$

   where $R \in \mathcal{R}$ and $R' \in \mathcal{R}'$.

2. The *core MD ontology*, $\mathcal{M}$, includes a partial instance, $D_{\mathcal{M}}$, containing dimensional data; and dimensional rules $\Sigma_{\mathcal{M}}$, and dimensional constraints $\kappa_{\mathcal{M}}$, among

them, egds and NCs as in Section 4.1.[2] We assume that application- dependent guidelines and constraints (guidelines (a) and (b) and semantic constraints in Example 5.1.1) are represented as dimensional rules and constraints in $\mathcal{M}$, resp. These are rules and constraints in $\Sigma_{\mathcal{M}}$ and $\kappa_{\mathcal{M}}$, resp., that unlike basic constraints $\Omega_{\mathcal{M}}$, are application-dependent (cf. Section 4.1).

3. A contextual relational schema $\mathcal{R}^c$, with an instance I^c, which contains possibly partial materialized data at the contextual level.

4. A set of *quality predicates*, $\mathcal{P}$, with their definitions with non-recursive Datalog rules $\Sigma^{\mathcal{P}}$ (possibly with negation, *not*), in terms of predicates $\mathcal{R}^{\mathcal{M}}$ (e.g. *WorkingScheduels* and *Personnel* in Example 5.1.1), predicates in $\mathcal{R}^c$, and built-in predicates.[3] A quality predicate reflects an application dependent specific quality concern. The definition of a quality predicate $P \in \mathcal{P}$ is a rule in $\Sigma^{\mathcal{P}}$ of the following form:

$$\phi^c_P(\bar{x}), \varphi^{\mathcal{M}}_P(\bar{x}) \ \rightarrow \ P(\bar{x}). \tag{5.2}$$

Here, $\phi^c_P(\bar{x})$ is a conjunction of atoms with predicates in $\mathcal{R}^c$ plus built-ins, and $\varphi^{\mathcal{M}}_P(\bar{x})$ is a conjunction of atoms with predicates in schema $\mathcal{R}^{\mathcal{M}}$ of the ontology $\mathcal{M}$.

Notice that the definition of quality predicates in $\mathcal{P}$ can be syntactically told apart from the dimensional rules in $\mathcal{M}$. Unlike quality predicates, the dimensional rules perform dimensional navigation through the join variables in their bodies that appear in categorical predicates and child-parent predicates (cf. Section 4.1 and Remark 4.1.1)

[2] The “core” ontology since it is within the context $\mathfrak{C}$ that can also be considered as an ontology.

[3] More general rules can be used, but their the interaction with the rest of the ontology may affect the complexity of QA.

Furthermore, and not strictly inside context $\mathfrak{C}$, there are predicates $R^q_1, ..., R^q_n \in \mathcal{R}^q$, the *quality versions* of $R_1, ..., R_n \in \mathcal{R}$. They are defined through *quality data extraction rules* Σ^q written in non-recursive Datalog, in terms of nickname predicates (in $\mathcal{R}'$), and the quality predicates (in $\mathcal{P}$), and built-in predicates. Their definitions (Σ^q in Figure 5.1) impose conditions corresponding to user's data quality profiles, and their extensions form the quality data (instance). The following is the general form for the rules in Σ^q:

$$R'(\bar{x}), \psi^{\mathcal{P}}_{R'}(\bar{x}) \ \rightarrow \ R^q(\bar{x}), \tag{5.3}$$

where $R' \in \mathcal{R}', R^q \in \mathcal{R}^q$ (R' and R^q are associated with $R \in \mathcal{R}$), and $\psi^{\mathcal{P}}_{R'}(\bar{x})$ is a conjunction of atoms with predicates in $\mathcal{P}$ and built-ins.

Notice that the connection between the quality versions in $\mathcal{R}^q$, categorical relations in $\mathcal{M}$, and contextual relations in I^c is through quality predicates $\mathcal{P}$. Since the latter are defined by general and flexible rules, through them we can also access the ontology $\mathcal{M}$ and the contextual instance I^c.

The external sources $\mathcal{E} = \{E_1, ..., E_j\}$ are of different types and contribute with data to the contextual schema. These data can be materialized and stored at the context level by the contextual instance I^c, or left at the sources and accessed through mappings.

Example 5.1.2 (ex. 5.1.1 cont.) *Temperatures*$' \in \mathcal{R}'$ is a nickname predicate for *Temperatures* $\in \mathcal{R}$, whose initial contents (in D) is under quality assessment.

In the core MD ontology $\mathcal{M}$, *WorkingSchedules*, *Shifts*, and *Personnel* are categorical relations. *WardUnit*, *TimeDay* are child-parent relations in the Hospital and Time dimensions, resp. The following are dimensional rules (tgds) of $\Sigma_{\mathcal{M}}$:

$$\sigma_1 : \mathit{Shifts}(w, d; n, s),\ \mathit{WardUnit}(w, u) \rightarrow \exists t\ \mathit{WorkingSchedules}(u, d; n, t).$$

$$\sigma_2 : \mathit{WorkingSchedules}(u, d; n, t),\ \mathit{WardUnit}(w, u) \rightarrow \exists s\ \mathit{Shifts}(w, d; n, s).$$

Categorical relations *WorkingTimes* and *PersonnelMonth* are defined below as views in terms of *WorkingSchedules* and *Personnel*, to bring their data to the levels of time and month, resp.

$$\mathit{WorkingSchedules}(u, d; n, s),\ \mathit{TimeDay}(t, d) \rightarrow \mathit{WorkingTimes}(u, t; n, s). \tag{5.4}$$

$$\mathit{Personnel}(i, d; p), \mathit{DayMonth}(d, m) \rightarrow \mathit{PersonnelMonth}(i, m; p). \tag{5.5}$$

The constraints in Example 5.1.1 are expressed by the following dimensional constraints:

$$\mathit{WorkingSchedules}(\mathsf{intensive}, d; n, s), \mathit{DayMonth}(d, \mathsf{jan})\ \rightarrow\ \bot.$$

$$[\mathit{WorkingSchedules}(u, d; n, s),\ \mathit{WorkingSchedules}(u', d; n, s),$$
$$\mathit{UnitInstitution}(u, i),\ \mathit{UnitInstitution}(u', i')]\ \rightarrow\ i = i'.$$

TakenWithTherm and *CertifiedNurse* are defined next as quality predicates, with definitions of the form (5.2) in $\Sigma^{\mathcal{P}}$ (cf. Figure 5.1). They address quality concerns about the certified nurses and the thermometers:

$$\mathit{WorkingTimes}(\mathsf{intensive}, t; n, y) \rightarrow \mathit{TakenWithTherm}(t, n, \mathsf{b1}). \tag{5.6}$$

$$\mathit{PersonnelMonth}(m, i; p), \mathsf{february} \leq m \rightarrow \mathit{CertifiedNurse}(p). \tag{5.7}$$

Here, (5.6) and (5.7) refer to (a) and (b) in Example 5.1.1.

The quality version of *Temperatures* is $\mathit{Temperatures}^q \in \mathcal{R}^q$, with the following definition of the form (5.3) in Σ^q, which captures the intended, clean contents of the

Algorithm 1 The QualityQA algorithm

Step 1: Replace each predicate R in $\mathcal{Q}$ with its corresponding quality version R^q, obtaining a CQ $\mathcal{Q}^q$ over schema $\mathcal{R}^q$.

Step 2: Unfold the definitions of quality versions R^q of predicate R in $\mathcal{R}$, given by rules of the form (5.3) in Σ^q. This results into a UCQ $\mathcal{Q}^{\mathfrak{C}}$ in terms of predicates in $\mathcal{R}' \cup \mathcal{P}$ and built-ins.

Step 3: Unfold the definitions of quality predicates, given by rules of the form (5.2) in $\Sigma^{\mathcal{P}}$, obtaining a UCQ $\mathcal{Q}^{\mathcal{M}}$ in terms of predicates in $\mathcal{R}' \cup \mathcal{R}^c \cup \mathcal{R}^{\mathcal{M}}$, and built-ins.

Step 4: Answer $\mathcal{Q}^{\mathcal{M}}$ by CQ answering over: (a) the database D' for schema $\mathcal{R}'$, (b) the instance I^c of $\mathcal{R}^c$, and (c) the MD ontology $\mathcal{M}$. For sub-queries in (c), use the CQ answering algorithm proposed in Chapter 7 (or is the algorithm in Chapter 9).

former:

$$[\mathit{Temperatures}'(t,p,v,n),\ \mathit{CertifiedNurse}(n),$$
$$\mathit{TakenWithTherm}(t,n,\mathsf{b1})] \ \rightarrow\ \mathit{Temperatures}^q(t,p,v,n). \tag{5.8}$$

□

Now, we present the QualityQA algorithm (Algorithm 1) that computes clean quality answers to a CQ $\mathcal{Q}$ posed to the initial dirty database D for schema $\mathcal{R}$.

In Steps 2 and 3, the results are UCQs because $\Sigma^{\mathcal{P}}$ and Σ^q contain non-recursive Datalog rules, in addition to the fact that $\mathcal{Q}$ is a CQ.[4] As a consequence of that, Step 4 starts with a UCQ $\mathcal{Q}^{\mathcal{M}}$, for which each conjunct can be answered by CQ answering under D', I^c, and $\mathcal{M}$. More complex rules than non-recursive Datalog in $\Sigma^{\mathcal{P}}$ and Σ^q

[4] A UCQ query is the unions of some CQs.

require different QA approaches, that might affect other steps. In particular, $\mathcal{Q}^{\mathcal{M}}$ in Step 4 might be a more complex query than a UCQ, for which QA may not be done by just answering CQs over $\mathcal{M}$, in addition to D' and I^c.

Regarding QA under $\mathcal{M}$ in Step 4, we can use the algorithm in Chapter 7. This algorithm is a chase-based QA algorithm that imposes CQs on a canonical model that represents multiple models of the ontology. This means, in general, there are multiple clean instances, $\mathcal{D}^q$, for which we implicitly use certain answers for quality query answering, by utilizing the QA algorithm under the MD ontology in QualityQA.

Remark 5.1.1 Notice that given the kind of predicate definitions we have, the QualityQA algorithm computes what we could define as the *clean answers* to query $\mathcal{Q}$, as follows:

$$QAns_D^{\mathfrak{C}}(\mathcal{Q}) \ = \ \{\bar{c} \mid D \cup \Sigma' \cup \mathcal{M} \cup I^c \cup \Sigma^{\mathcal{P}} \cup \Sigma^q \ \models \ \mathcal{Q}^q[\bar{c}]\}.$$

This formulation of clean answers corresponds to a *model-theoretic definition of clean answers*. Note that in this formulation, $\mathcal{M}$ can have multiple models that can only be represented by a canonical model (the chase) for the purpose of computing certain answers to CQs. □

The algorithm can be applied in particular to compute the clean version R^q of a table R in D.

Example 5.1.3 (ex. 5.1.2 cont.) This is the initial query asking for (quality) values for Tom Waits' temperature,

$$\mathcal{Q}(v)\colon \exists n\ \exists t\ (\mathit{Temperatures}(t, \textsf{tom waits}, v, n) \wedge \textsf{aug/21-11:45} \leq t \leq \textsf{aug/21-12:15}),$$

which, according to Step 1 of $\mathsf{QualityQA}$, has to be first rewritten into:

$$\mathcal{Q}^q(v)\colon \exists n\ \exists t\ (\mathit{Temperatures}^q(t, \mathsf{tom\ waits}, v, n)\ \wedge\ \mathsf{aug/21\text{-}11{:}45} \leq t \leq \mathsf{aug/21\text{-}12{:}15}).$$

To answer $\mathcal{Q}^q$, first (5.8) can be used (Step 2 of $\mathsf{QualityQA}$), obtaining a query:

$$\mathcal{Q}^{\mathfrak{C}}(v)\colon \exists n\ \exists t\ (\mathit{Temperatures}'(t, \mathsf{tom\ waits}, v, n) \wedge \mathit{TakenWithTherm}(t, n, \mathsf{b1}) \wedge$$
$$\mathit{CertifiedNurse}(n)\ \wedge\ \mathsf{aug/21\text{-}11{:}45} \leq t \leq \mathsf{aug/21\text{-}12{:}15}).$$

This query will in turn use the quality predicate definitions (5.6) and (5.7), that lead to $\mathcal{Q}^{\mathcal{M}}$ expressed in terms of $\mathit{Temperatures}'$ and the predicates in $\mathcal{R}^{\mathcal{M}}$, and built-ins, namely the following (this is Step 3 of $\mathsf{QualityQA}$):

$$\mathcal{Q}^{\mathcal{M}}(v)\colon \exists i\ \exists m\ \exists n\ \exists t\ \exists y\ (\mathit{Temperatures}'(t, \mathsf{tom\ waits}, v, n) \wedge \mathit{WorkingTimes}(\mathsf{intensive}, t; n, y)$$
$$\wedge\ \mathit{PersonnelMonth}(m, i; n)\ \wedge\ \mathsf{february} \leq m\ \wedge\ \mathsf{aug\text{-}21/11{:}45} \leq t \leq \mathsf{aug/21\text{-}12{:}15}).$$

At this point, according to Step 4 of $\mathsf{QualityQA}$, we answer $\mathcal{Q}^{\mathcal{M}}$ by CQ answering under $\mathcal{M}$ and database D'. The former is by the application of the QA algorithm in Chapter 7. □

Under our approach, data cleaning (or extraction of quality data from an initial table) amounts to obtaining a clean instance D^q from the dirty target instance D, and it is done by collecting clean extensions $R_1^q, ..., R_n^q$ of $R_1, ..., R_n \in \mathcal{R}$. The clean extension R^q of possibly dirty relation R in D is obtained by answering the atomic query $\mathcal{Q}(\bar{x}) : R(\bar{x})$ using $\mathsf{QualityQA}$. In particular, the algorithm uses a rule of the general form (5.3) to collect the clean data of R^q by applying conditions in $\psi^{\mathcal{P}}_{R'}$ on R'.

The quality of the target database instance D can be assessed through its "distance" to the quality instance D^q, that is the aggregate distance of every relation $R(D)$ from its quality version $R^q(D^q)$. Different notions of distance might be used as discussed in [Bertossi, 2011b; Bertossi et al., 2016].

We studied computational properties of MD ontologies and CQ answering over them in isolation, in Section 4.2. Adding the definitions of the quality predicates and quality versions, the result is a new ontology that might not preserve the syntactic properties of the core MD ontology. The next example shows that adding the definition of quality predicates as plain Datalog rules can break the *WS* syntactic property.

Example 5.1.4 (ex. 5.1.2 cont.) Consider dimensional rules σ_1, σ_2, and the following non-recursive Datalog rule that defines a quality predicate *Mornings*(*Unit*):

$$\sigma_3: \ \mathit{Shifts}(w, d; n, \mathsf{morning}), \mathit{WorkingSchedules}(u, d; n, \mathsf{critical\text{-}care}) \rightarrow \mathit{Mornings}(u).$$

Here, $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$ is not *WS* since n in the body of σ_3 is a repeated marked body variable that only appears in infinite-rank positions *Shifts*[3] and *WorkingSchedules*[3] of Σ. □

In general terms, combining existential rules that separately enjoy good computational properties might lead to a set of rules that does not inherit these properties [Baget et al., 2011b, 2015]. In the case of $\Sigma^{\mathcal{P}}$ and Σ^{q} define over the core MD ontology, although the weak-stickiness might not hold for the resulting ontology, CQ answering is still tractable. This is because $\Sigma^{\mathcal{P}}$ and Σ^{q} are sets of non-recursive Datalog rules, and as a result, a CQ can be rewritten using them in terms of predicates in $\mathcal{M}$ (and other predicates in context, i.e. $\mathcal{R}' \cup \mathcal{R}^c$) and answered by the ontology and the extensional data at context level (cf. Step 4 in Algorithm 1).

Since ontological predicates act as extensional predicates in the definitions of quality predicates, we can also accept quality predicates definitions in recursive Datalog, without extensional variables in their heads, while still enjoying the good computational properties of QA.

Chapter 6

Semantic Generalization of Stickiness

For practical QA under *WS* Datalog$^\pm$, which is essential for our MD ontology, we present a bottom-up chase-based QA algorithm in Chapter 7. Being bottom-up, we proposed the MagicD^+ rewriting algorithm (cf. Chapter 8) to optimize it. However, the class of *WS* programs is provably not closed under MagicD^+, meaning that the result of applying the rewriting to a *WS* program may not be *WS* anymore. This led us to search for a more general class of programs that (i) is closed under the MagicD^+, (ii) extends *WS* Datalog$^\pm$, (iii) still has tractable QA, and (iv) allows the application of the proposed bottom-up QA.

To find such a program class, in this chapter, we study the syntactic stickiness and the (semantic) shc-property (cf. Section 2.3.3) in detail. In particular, We generalize the sch-property using the notion of *finite and infinite positions* which results to specification of a range of semantic and syntactic classes, including sticky and *WS* Datalog$^\pm$, and a new class of *JWS* that satisfies the properties above.

6.1 Generalized Stickiness

The *generalized-stickiness of the chase (gsch-property)* is defined by relaxing the condition in the *sch-property*: the condition applies to values for the repeated body variables that do not appear in *finite positions*. A position in a program Π is *finite* if finitely many values appear in the position during the chase of Π. We denote the set of finite positions of Π by $\mathit{FinPoss}(\Pi)$.

Definition 6.1.1 A Datalog$^+$ program Π (including extensional data) has the *generalized-stickiness property* of the chase, in short, the *gsch-property*, if and only if for every chase step $I_i \xrightarrow{\sigma_i,\theta_i} I_i \cup \{A_i\}$ during the chase of Π, the following holds: If a variable x appears more than once in $\mathit{body}(\sigma_i)$ and *not* in $\mathit{FinPoss}(\Pi)$, $\theta_i(x)$ occurs in A_i and every atom B for which, $A_i \xrightarrow{\Pi}^{*} B$. *Generalized-stickiness of the chase (GSCh)* is the class of programs with the *gsch-property*. □

Example 6.1.1 (ex. 2.3.3 cont.) Π_1 and Π_2 have no infinite positions because for both programs the chase terminates. Consequently, they are *GSCh*. Let Π_3 be Π_2 with a new rule, $\sigma: \ R(x,y) \rightarrow \exists z \ R(z,x)$. $R[1]$ and $R[2]$ are infinite positions because, during the chase of Π_3, σ cyclically generates infinite null values in $r[2]$ that also propagate to $R[1]$. The chase of Π_3 does not have the *gsch-property* and it is not *GSCh* since the value b replaces the repeated body variable y that only appears in infinite positions ($R[1]$ and $R[2]$) and b does not propagate all the way down during the chase procedure. □

6.2 Selection Functions and Program Classes

The finite positions in the definition of the *gsch-property* are not computable for a given program which makes it impossible to decide if the program has the property. Here, we define *selection functions* that determine subsets of the finite positions of a program. We replace finite positions in the definition of the *gsch-property* with the results from selection functions in order to define new stickiness properties and program classes.

A *selection function* $\mathcal{S}$ (over a schema $\mathcal{R}$) is a function that takes a program Π and returns a subset of $\mathit{FinPoss}(\Pi)$. Particular functions are $\mathcal{S}^{\perp}$ and $\mathcal{S}^{\top}$, that given a program Π, return the empty set and $\mathit{FinPoss}(\Pi)$, respectively. The latter may not be computable, and depends on the program's data, which is not the case for the former. π_F also defines a data-independent selection function, $\mathcal{S}^{rank}$, that returns the finite-rank positions (there are finitely many values in them in the chase of Π, for any data set [Calì et al., 2012c, Lemma 5.1]). A selection function is "syntactically computable" if it only depends on the rules Π^R of a program Π, and we use the notation $\mathcal{S}(\Pi^R)$ if it is not clear from the context.

The $\mathcal{S}$-*stickiness* is defined by replacing the finite positions in the definition of the *gsch-property* with a selection function $\mathcal{S}$: The chase of a program Π has the $\mathcal{S}$-*stickiness property* if the stickiness condition applies only to values replacing the repeated body variables that do not appear in a position of $\mathcal{S}(\Pi)$. $\mathit{SCh}(\mathcal{S})$ is the semantic class of programs with the $\mathcal{S}$-*stickiness*. In particular, $\mathit{SCh} = \mathit{SCh}(\mathcal{S}^{\perp})$, $\mathit{GSCh} = \mathit{SCh}(\mathcal{S}^{\top})$. Also, $\mathit{WSCh} = \mathit{SCh}(\mathcal{S}^{rank})$ is the class of programs with *weak-stickiness of the chase*. $\mathit{SCh}(\mathcal{S})$ specifies a range of semantic classes of programs starting with SCh, ending with GSCh, and with WSCh in between.

$$\begin{tikzcd}
\mathcal{S}^{\perp} & \underset{(a)}{\subsetneq} & \mathcal{S}^{rank} & \underset{(b)}{\subsetneq} & \mathcal{S}^{\exists} & \underset{(c)}{\subsetneq} & \mathcal{S}^{\top} \\
\mathit{SCh}{:=}\mathit{SCh}(\mathcal{S}^{\perp}) \arrow[u] & \underset{(d)}{\subsetneq} & \mathit{WSCh}{:=}\mathit{SCh}(\mathcal{S}^{rank}) \arrow[u] & \underset{(e)}{\subsetneq} & \mathit{SCh}(\mathcal{S}^{\exists}) \arrow[u] & \underset{(f)}{\subsetneq} & \mathit{GSCh}{:=}\mathit{SCh}(\mathcal{S}^{\top}) \arrow[u] \\
\mathit{Sticky} \arrow[u, phantom, "\subsetneq" sloped, "(g)"'] & \underset{(j)}{\subsetneq} & \mathit{WS} \arrow[u, phantom, "\subsetneq" sloped, "(h)"'] & \underset{(k)}{\subsetneq} & \mathit{JWS} \arrow[u, phantom, "\subsetneq" sloped, "(i)"'] & &
\end{tikzcd}$$

Figure 6.1: Semantic and syntactic program classes, and selection functions

$\mathit{SCh}(\mathcal{S})$ grows monotonically with $\mathcal{S}$: For selection functions $\mathcal{S}_1$ and $\mathcal{S}_2$ over schema

$\mathcal{R}$, if $\mathcal{S}_1 \subseteq \mathcal{S}_2$, then $SCh(\mathcal{S}_1) \subseteq SCh(\mathcal{S}_2)$. Here, $\mathcal{S}_1 \subseteq \mathcal{S}_2$ if and only if for every program Π, $\mathcal{S}_1(\Pi) \subseteq \mathcal{S}_2(\Pi)$. In general, the more finite positions are (correctly) identified (and the consequently, the less finite positions are treated as infinite), the more general subclass of *GSCh* that is identified or characterized.

Sticky Datalog$^\pm$ uses the marking procedure to restrict the repeated body variables and impose the *sch-property*. Applying this syntactic restriction only on body variables specified by syntactic selection functions results in syntactic classes that extend sticky Datalog$^\pm$. These syntactic classes are subsumed by the semantic classes defined by the same selection functions; each of these syntactic classes only partially represents its corresponding semantic class. In particular, *SCh* subsumes sticky Datalog$^\pm$ [Calì et al., 2012c]; and *WS* is a syntactic subclass of *WSCh* (cf. (g) and (h) in Figure 6.1).

6.3 Joint Weakly-Sticky Programs

The definition of the class of *JWS* programs uses the syntactic selection function $\mathcal{S}^\exists$, which appeals to the *existential dependency graph* of a program [Krötzsch & Rudolph, 2011] (cf. Section 2.3.2).

Definition 6.3.1 For a program Π, the set of *finite-existential positions* of Π, denoted by $\pi^\exists_F(\Pi)$, is the set of positions that are not in the target set of any $\exists$-variable in a cycle in EDG(Π). □

Intuitively, a position in $\pi^\exists_F(\Pi)$ is not in the target of any $\exists$-variable that may invent infinite null values. Therefore, it specifies a subset of finite positions and $\pi^\exists_F(\Pi)$ characterise a syntactic selection function that we denote by $\mathcal{S}^\exists$. Since it is

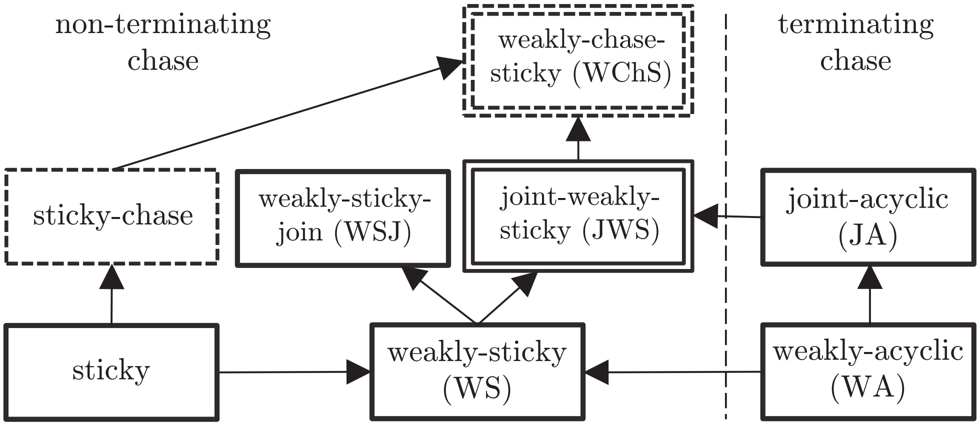


Figure 6.2: Generalization relationships between program classes

syntactic, it can also be denoted by $\pi_F^{\exists}(\Pi^R)$ but for simplicity of notation we use $\pi_F^{\exists}(\Pi)$.

Proposition 6.3.1 For every set of rules Π, $\pi_F(\Pi) \subseteq \pi_F^{\exists}(\Pi)$. □

Proof of Proposition 6.3.1: Using proof by contradiction, we assume there is a position p such that: $p \in \pi_F(\Pi)$ and $p \notin \pi_F^{\exists}(\Pi)$. The latter means there is a cycle in EDG(Π) that includes an $\exists$-variable z in a rule σ such that $p \in T_z$. The definition of EDG implies that, there is $\forall$-variable x in the body of σ for which $B_x \subseteq T_z$. Let p_z and p_x be the two positions where z and x appear in σ resp. Then, there is a path in DG(Π) from p_z to p_x and there is also a special edge from p_x to p_z making a cycle including p_z with a special edge. Therefore, p_z has infinite-rank, $p_z \notin \pi_F(\Pi)$. Since $p \in T_z$, we can conclude that p also has infinite-rank, $p \notin \pi_F(\Pi)$, which contradicts the assumption and completes the proof. □

$\pi_F^{\exists}$ defines a computable selection function $\mathcal{S}^{\exists}$ that returns finite-existential positions of a program (cf. (c) in Figure 6.1). *SCh*($\mathcal{S}^{\exists}$) is a new semantic subclass of *GSCh* that generalizes *SCh*($\mathcal{S}^{rank}$) since $\mathcal{S}^{\exists}$ provides a finer mechanism for capturing finite positions in comparison with $\mathcal{S}^{rank}$ (cf. (e) and (f) in Figure 6.1).

Definition 6.3.2 A program Π is *joint-weakly-sticky* (*JWS*) if for every rule in Π and every variable in its body that occurs more than once, the variable is either non-marked or appears in some positions in $\pi_F^{\exists}(\Pi)$. □

The class of *JWS* programs is a proper subset of $SCh(\mathcal{S}^{\exists})$ and extends *WS* (cf. (i) and (k) in Figure 6.1). The latter is shown by Example 6.3.1.

Example 6.3.1 Let Π be program with rules:

$$R(x,y), U(y) \ \rightarrow \exists z\ R(y,z). \tag{6.1}$$

$$R(x,y), R(y,z) \ \rightarrow R(x,z). \tag{6.2}$$

$\pi_F(\Pi) = \{U[1]\}$ and $\pi_F^{\exists}(\Pi) = \{U[1], R[1], R[2]\}$. After applying the marking procedure, all the body variables are marked. Π is not *WS* because of y in the second rule. It is *JWS* since every position is in $\pi_F^{\exists}(\Pi)$. □

Chapter 7

Query Answering for Semantically Sticky Classes

In this chapter, we present a bottom-up chase-based QA algorithm for programs in the semantic classes in Section 6.2, and their related syntactic classes.

7.1 The SChQA Algorithm

SChQA takes as input a computable selection function $\mathcal{S}$, a program $\Pi \in \mathit{SCh}(\mathcal{S})$, and a CQ $\mathcal{Q}$ over schema $\mathcal{R}$ and returns $\mathit{ans}(\mathcal{Q}, \Pi)$. Before describing SChQA, we need to introduce the notion of applicability that modifies the applicability condition in tgd-based chase step in Section 2.2.

Definition 7.1.1 Consider a Datalog$^+$ program Π, and an instance I of Π. A pair of rule/assingment (σ, θ), with $\sigma \in \Pi$, is *applicable* over I if: (a) $\theta(\mathit{body}(\sigma)) \subseteq I$; and (b) there is an assignment θ' that extends θ, maps the $\exists$-variables of σ into fresh nulls, and $\theta'(\mathit{head}(\sigma))$ is *not homomorphic* to any atom in I.[1] $\square$

For an instance I and a program Π, we can systematically compute the applicable pairs of rule/assignment by first finding $\sigma \in \Pi$ for which $\mathit{body}(\sigma)$ is satisfied by I. That gives an assignment θ for which $\theta(\mathit{body}(\sigma)) \in I$. Then, we construct θ' as in Definition 7.1.1 and we iterate over atoms in I and we check if they are homomorphic to $\theta'(\mathit{head}(\sigma))$.

[1] Atom A is homomorphic to atom B, iff there is a homomorphism h such that $h(A) = B$.

In SChQA, we use the notion of *freezing a null value* that is moving it from $\mathcal{N}$ into $\mathcal{C}$. It may cause new applicable pairs of rule/assignment because it changes homomorphic atoms. *Resumption* is freezing every null in the current instance I and continuing the algorithm steps. Notice that a pair of rule/assignment is applied only once in Step 2. Moreover, if there are more than one applicable pairs, then SChQA chooses the pair as in the chase using the notion of level and then lexicographic order (cf. Section 2.2).

SChQA is applicable to any Datalog$^+$ program and any computable selection function, and returns sound answers. However, completeness is guaranteed only when applied to programs in $\mathit{SCh}(\mathcal{S})$ with a computable $\mathcal{S}$.

Algorithm 2 The SChQA algorithm

Inputs: A selection function $\mathcal{S}$, a program $\Pi \in \mathit{SCh}(\mathcal{S})$, and a CQ $\mathcal{Q}$ over Π.

Output: $\mathit{ans}(\mathcal{Q}, \Pi)$.

Step 1: Initialize an instance I with the extensional database D.

Step 2: Choose an applicable rule/assignment σ and θ over I, add $\theta'(\mathit{head}(\sigma))$ into I (θ' is the assignment defined in Definition 7.1.1).

Step 3: Freeze nulls that appear in the new atom and in the positions of $\mathcal{S}(\Pi)$.

Step 4: Iteratively apply Steps 2-3 until all applicable pairs are applied.

Step 5: Resume Step 2, i.e. freeze nulls in I and continue with Steps 2. Repeat resumption $M_{\mathcal{Q}}$ times where $M_{\mathcal{Q}}$ is the number of $\exists$-variables in $\mathcal{Q}$

Step 6: Return the tuples in $\mathcal{Q}(I)$ that do not have null values (including the frozen nulls).

Example 7.1.1 Consider a program Π with $D = \{S(a, b, c), V(b), U(c)\}$, and a set of rules containing (the hat signs show the marked variables),

$$\begin{aligned}
\sigma_1: &\quad S(\hat{x},\hat{y},\hat{z}) \;\rightarrow\; \exists w\, S(y,z,w),\\
\sigma_2: &\quad U(\hat{x}) \;\rightarrow\; \exists y,z\, S(x,y,z),\\
\sigma_3: &\quad S(\hat{x},y,z), V(\hat{x}), S(y,z,\hat{w}) \;\rightarrow\; P(y,z),
\end{aligned}$$

and a BCQ $\mathcal{Q} : \exists y\; P(c,y)$. Π is in WS and so $\mathit{SCh}(\mathcal{S}^{rank})$. Specifically in σ_3, x occurs in $V[1]$ which is in $\mathcal{S}^{rank}(\Pi)$ and y and z are not marked.

The algorithm starts from $I := D$. At Step 2, σ_1 and $\theta_1 : x \mapsto a, y \mapsto b, z \mapsto c$, are applicable; and SChQA adds $S(b,c,\zeta_1)$ into I. σ_2 and $\theta_2 : x \mapsto c$, are also applicable and they add $S(c,\zeta_2,\zeta_3)$ into I. Step 3 does not freeze ζ_1, ζ_2, and ζ_3 since they are not in $\mathcal{S}^{rank}(\Pi)$.

There is no more applicable pairs and we continue with Step 5. Notice that σ_1 and $\theta_3 : \; x \mapsto b, y \mapsto c, y \mapsto \zeta_1$ are not applicable since any $\theta_3' : \; \theta_3 \cup \{w \mapsto \zeta_4\}$ generates $S(c,\zeta_1,\zeta_4)$ that is homomorphic to $S(c,\zeta_2,\zeta_3)$ in I. SChQA is resumed once since $\mathcal{Q}$ has one $\exists$-variable. This is done by freezing ζ_1,ζ_2,ζ_3 and returning to Step 2. Now, $S(c,\zeta_1,\zeta_4)$ and $S(c,\zeta_2,\zeta_3)$ are not homomorphic anymore and (σ_1,θ_3) is applied which results in $S(c,\zeta_1,\zeta_4)$. As a consequence, σ_3 and $\theta_4 : \; x \mapsto b, y \mapsto c, z \mapsto \zeta_1, w \mapsto \zeta_4$, are applicable, which generate $P(c,\zeta_1)$. The instance I in Step 6 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 $I \models \mathcal{Q}$. □

The number of resumptions with SChQA depends on the query. However, for practical purposes, we could run SChQA with N resumptions, to be able to answer queries with up to N $\exists$-variables. If a query has more than N variables, we can incrementally retake the already-computed instance I, adding the required number of resumptions.

7.2 Correctness of SChQA and Complexity Analysis

In this section, we prove that SChQA is sound and complete w.r.t CQ answers under programs in $\mathit{SCh}(\mathcal{S})$, and we analyse the complexity of running it for different program classes.

Theorem 7.2.1 Consider a computable selection function $\mathcal{S}$ over schema $\mathcal{R}$, a program $\Pi \in \mathit{SCh}(\mathcal{S})$, and a CQ $\mathcal{Q}$ over $\mathcal{R}$. Algorithm SChQA taking $\mathcal{S}$, Π, and $\mathcal{Q}$ as inputs, terminates returning $\mathit{ans}(\mathcal{Q}, \Pi)$. □

Proof of Theorem 7.2.1: Let I_i be the instance I in SChQA after the i-th resumption, c_i be the number of frozen nulls and constants in I_i during SChQA, r be the number of the predicates in Π, and w be the maximum arity of the predicates. Initial value c_0 is the number of constants in $\mathit{Adom}(D)$ plus the finite number of nulls in $\mathcal{S}(\Pi)$. Therefore there are $r \times (c_0 + 1)^w$ non-frozen nulls in I_0 since there is no homomorphic pair of atoms in I_0. As a result, there are at most $c_0 + r \times (c_0 + 1)^w$ possible terms in I_0. After the first resumption, every null value is frozen; so there are $c_1 = c_0 + r \times (c_0 + 1)^w$ and at most $r \times (c_1 + 1)^w$ new nulls are invented, which results to at most $c_1 + r \times (c_1 + 1)^w$ terms in I_1. Along the same line of reasoning, we conclude that there are at most $c_{M_\mathcal{Q}} + r \times (c_{M_\mathcal{Q}} + 1)^w$ terms in $I_{M_\mathcal{Q}}$, so it is a finite instance. SChQA always terminates since there are finitely many applicable pairs w.r.t the finite instance $I = I_{M_\mathcal{Q}}$.

For the rest of the proof, we assume $\mathcal{Q}$ is an atomic and BCQ, and the proof can be extended to free CQs.[2] To prove SChQA is sound $((I_{M_\mathcal{Q}} \models \mathcal{Q}) \Rightarrow (\mathit{chase}(\Pi) \models \mathcal{Q}))$, we show that $I_{M_\mathcal{Q}}$ is isomorphic to a subset of $\mathit{chase}(\Pi)$. We construct this isomorphism

[2] Non-atomic queries can be converted to atomic queries using a query answer collection rule that preserves the *sch-stickiness property*.

inductively while running SChQA. More precisely, if a pair (σ, θ) is applied during SChQA and generates atom A, there is an applicable pair (σ, θ') during the chase of Π. (σ, θ) and (σ, θ') have isomorphic body images and they generate isomorphic atoms as both invent fresh nulls. (σ, θ') is eventually applied and generates A' isomorphic to A.[3]

To prove SChQA is complete $((\Pi \models \mathcal{Q}) \Rightarrow (I_{M_\mathcal{Q}} \models \mathcal{Q}))$, assume that the antecedent holds. Let k be the minimum number of steps, such that $\mathit{chase}^{[k]}(\Pi) \models \mathcal{Q}$. $\mathcal{Q}$ is mapped to an atom A_m in $\mathit{chase}^{[k]}(\Pi)$. We prove that A_m is isomorphic to an atom A'_m in $I_{M_\mathcal{Q}}$, so $\mathcal{Q}$ is also mapped to $I_{M_\mathcal{Q}}$, and $I_{M_\mathcal{Q}} \models \mathcal{Q}$.

Assume A_m is not in D, otherwise the proof is trivial. Let $I_A = \{A_1, ..., A_m\}$ be the set of atoms that derive A_m and are not in D, including A_m ($A_i \xrightarrow{\Pi}^* A_m, i \neq m$), ordered by their appearance in the chase. Let $S_1, ..., S_m$ be the chase steps that generate $A_1, ..., A_m$, by applying the pairs $(\sigma_1, \theta_1), ..., (\sigma_m, \theta_m)$, resp. The null values in I_A either, (a) appear in the positions of $\mathcal{S}(\Pi)$, or (b) appear in the positions of non-$\mathcal{S}(\Pi)$ and replace join variables in the body images of an applied pair, or (c) not in (a) or (b).

We proof by induction that $A_i, 1 \leq i \leq m$ is isomorphic to A'_i in I_k, such that k is the number of null values of type (b) in $A_1, ..., A_i$.

<u>*Base case*</u>: Starting from S_1, θ_1 maps $\mathit{body}(\sigma_1)$ to D. (σ_1, θ_1) satisfies the first condition in Definition 7.1.1: $\theta_1(\mathit{body}(\sigma_1)) \subseteq D \subseteq I_{M_\mathcal{Q}}$. It also satisfies the second applicability condition, therefore, it is applied in SChQA, and generates the atom, A'_1. The second condition holds, since otherwise A'_1 is homomorphic to an atom B'_1 in $I_{M_\mathcal{Q}}$, which means we can find and atom $B_1 \in \mathit{chase}(\Pi)$ that corresponding to B'_1

[3] Note that the chase procedure in Section 2.2 is *fair*, i.e. every applicable pair is eventually applied [Calì et al., 2013].

and is obtained before A_1. A_1 and B_1 can only differ in nulls on type (c), since nulls of type (a) and (b) are frozen and equal in both. Specially, if there is at least one null of type (b) in A_1 and B_1, there is at least one $\exists$-variable in $\mathcal{Q}$, since that null value also appear in A_m, and so, there is at least one resumption which freezes that null. B_1 can replace A_1 to derive A_m which contradicts our assumption that A_1 derives A_m. Therefore, there is no B'_1 and B_1. As a result, A'_1 is in I_1 if A_1 contains nulls of type (b) and it is in I_0 if there is no null of type (b).

Inductive step: Assume $A_1, ..., A_{i-1}, i \leq m$ are isomorphic to $A'_1, ..., A'_{i-1}$ in I_k, such that k is the number of nulls of type (b) in $A_1, ..., A_{i-1}$. We proof A_i is also isomorphic to A'_i in I'_k, where k' is the number of values of values of type (b) in $A_1, ..., A_i$. θ_i in S_i maps $\mathit{body}(\sigma_i)$ to $D \cup \{A_1, ..., A_{i-1}\}$. Consider the pair (σ_i, θ'_i), in which θ'_i is obtained from θ_i by replacing nulls with their corresponding nulls in $I_{M_\mathcal{Q}}$. (σ_i, θ'_i) satisfies the first applicability condition in Definition 7.1.1, since $\theta'_i(\mathit{body}(\sigma_i)) \subseteq D_A \cup \{A'_1, ..., A'_{i-1}\}$ (inductive hypothesis). It also satisfies the second applicability condition, and the pair is applied and generates A'_i.

If the second applicability condition does not hold, A'_i is homomorphic to an atom B'_i in I_k, that corresponds to an atom $B_i \in \mathit{chase}(\Pi)$ that is obtained before A_i and only differs from A_i in nulls of type (c). Specially for the nulls of type (b), they either all correspond to frozen nulls in I_k, or they are frozen later in I_{k+1}. Therefore A'_i is either obtained in I_k (in which case $k' = k$), or it is obtained in I_{k+1} (in which case $k' = k + 1$). This completes the inductive proof.

We also need to show that k in the proof never goes beyond $M_\mathcal{Q}$. This is because there are at most $M_\mathcal{Q}$ nulls in (b): *sch-stickiness property of the chase* implies that those nulls continue to appear in the subsequent atoms and therefore in A_m, that can only contain $M_\mathcal{Q}$ nulls. As a result, k never proceeds $M_\mathcal{Q}$ which shows $A_1, ..., A_m$ are mapped to atoms $A'_1, ..., A'_m$ in $I_{M_\mathcal{Q}}$. □

Proposition 7.2.1 Algorithm SChQA runs in polynomial time in data if the following holds for $\mathcal{S}$: for any program Π, the number of values appearing in $\mathcal{S}(\Pi)$-positions during the chase is polynomial in the size of the extensional data. □

Proof of Proposition 7.2.1: $c_{M_\mathcal{Q}} + r \times (c_{M_\mathcal{Q}} + 1)^w$ is a series with a closed form that is polynomial in c_0. The condition in the proposition means c_0, i.e. the number of frozen nulls before any resumption plus the number of constants in the extensional database, is polynomial w.r.t the size of the extensional data. As a result, $c_{M_\mathcal{Q}} + r \times (c_{M_\mathcal{Q}} + 1)^w$, which is the number of terms in $I_{M_\mathcal{Q}}$, is polynomial in the size of the extensional database which proves the proposition.

Lemma 7.2.1 During the chase of a Datalog$^+$ program Π, the number of distinct values in $\mathcal{S}^\exists(\Pi)$-positions is polynomial in the size of the extensional data. □

Proof of Lemma 7.2.1: We first define the notion of $\exists$-rank of a position p in Π. Let Z_p be the set of $\exists$-variables z in Π, such that $p \in T_z$. Then, the $\exists$-rank of p is the maximum length of any path in the existential dependency graph of Π that ends with any $\exists$-variable in Z_p. A position in $\pi_F^\exists(\Pi)$ has finite $\exists$-rank, since it is not in the target of any $\exists$-variable that appears in a cycle in the existential dependency graph of (Π). We prove by induction that there are polynomially many values w.r.t. d (i.e. the size of the extensional database), that appear during the chase in the positions with $\exists$-rank at most i. In the inductive proof, d_i is the number of values in positions with $\exists$-rank of i.

Base case: Only values from $Adom(D)$ appear in the positions with $\exists$-rank of 0, so $d_0 = d$.

Inductive step: The values that appear in a position of $\exists$-rank i are either (a) from other positions with $\exists$-rank i, or (b) from the positions with $\exists$-rank $j < i$. For (b), they are at most d_{i-1} which, by inductive hypothesis, is polynomial in d. For (a), the values are invented by $\exists$-variables that appear at the end of paths of length i in the existential dependency graph of Π. Let σ be a rule containing such a $\exists$-variable, z. The values in $\mathit{body}(\sigma)$ are in positions with $\exists$-rank less than i. Let v be the maximum number of variables in the body of any rule in Π. Then, σ can invent d_{i-1}^{v} new values for the positions with $\exists$-rank i. There are at most r such rules where r is the number of rules in Π. Therefore, there are at most $r \times d_{i-1}^{v} + d_{i-1}$ distinct values in the positions of rank at most i, and since r and v are independent of data, the number is a polynomial w.r.t. d. Considering that the maximum $\exists$-rank in Π is independent of the data of Π, we conclude that d_k is also polynomial w.r.t. d.[4]

Corollary 7.2.1 SChQA runs in polynomial time in data with programs in $\mathit{SCh}(\mathcal{S}^{\exists})$, in particular for the programs in the *JWS* and *WS* syntactic classes. □

This proves *JWS* has the desirable property mentioned at the beginning of Chapter 6: it extends *WS* programs and also allows the application of the proposed bottom-up QA, SChQA. Now, it remains to show that *JWS* has the first property: SChQA for QA under *JWS* programs can be optimized through magic-sets rewriting, which is addressed in the next chapter.

[4] The proof is similar to the proof of [Fagin et al., 2005, Theorem 3.9], which shows the chase of a *WA* program runs in polynomial time in data complexity.

Chapter 8

Magic-Sets Optimization for Datalog$^+$ Programs

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 & Ramakrishnan, 1987; Ceri et al., 1990]. The advantage of such a rewriting technique is that, by working bottom-up, we can take advantage of the structure of the query and the data values in it, optimizing the data generation process. In this chapter, we present a magic-sets rewriting for Datalog$^+$ programs, denoted by MagicD^+.

8.1 The MagicD^+ Rewriting Algorithm

MagicD^+ 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) work with $\exists$-variables in tgds, and (b) consider the extensional data of the predicates that also have intensional data defined by the rules. For (a), we apply the solution proposed in [Alviano et al., 2012]. However (b) is specifically relevant for Datalog$^+$ programs that allow predicates with both extensional and intentional data, and we address it in MagicD^+.

To present MagicD^+, we first introduce *adornments*, a convenient way for representing binding information for intentional predicates [Ceri et al., 1990].

Definition 8.1.1 Let P be a predicate of arity k in a program Π. 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 α for P, the magic atom of A w.r.t. α is the atom $\mathit{mg\_P}^{\alpha}(\bar{t})$, where $\mathit{mg\_P}^{\alpha}$ is a predicate not in Π, and $\bar{t}$ contains all the terms in $a_1...a_k$ that correspond to bound positions according to α. □

Example 8.1.1 "*bfb*" is a possible adornment for ternary predicate S, and $\mathit{mg\_S}^{bfb}(x, z)$ is the magic atom of $S(x, y, z)$ w.r.t. "*bfb*". □

Binding information can be propagated in rule bodies according to a *side-way information passing strategy (SIPS)* [Beeri & Ramakrishnan, 1987].

Definition 8.1.2 Let σ be a tgd and α be an adornment for the predicate of P in $\mathit{head}(\sigma)$. A *side-way information passing strategy (SIPS)* for σ w.r.t. α is a pair $(\prec^{\alpha}_{\sigma}, f^{\alpha}_{\sigma})$, where:

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

2. f^{α}_{σ} is a function assigning to each atom A in σ, a subset of the variables in A that are bound after processing A. f^{α}_{σ} must guarantee that if $A = \mathit{head}(\sigma)$, then $f^{\alpha}_{\sigma}(A)$ contains only and all the variables in $\mathit{head}(\sigma)$ that correspond to the bound arguments of α. □

Now, we present MagicD^+ using running Example 8.1.2.

Example 8.1.2 Let Π be a program with $D = \{U(b), R(a, b)\}$ and the rules,

$$R(x, y), R(y, z) \rightarrow P(x, z), \tag{8.1}$$

$$U(y), R(x, y) \rightarrow \exists z\ R(y, z), \tag{8.2}$$

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

The MagicD^+ rewriting technique takes a Datalog$^+$ program Π and a CQ $\mathcal{Q}$ of schema $\mathcal{R}$, and returns a program Π_m and a CQ $\mathcal{Q}_m$ of schema $\mathcal{R}_m$, such that $ans_{\mathcal{Q}}(\mathcal{Q}, \Pi) = ans_{\mathcal{Q}_m}(\mathcal{Q}_m, \Pi_m)$. It 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^α, MagicD^+ finds every rule σ with the head predicate P and it generates an adorned rule σ' as follows and adds it to Π_m. According to a pre-determined SIPS, MagicD^+ replaces every body atom in σ with its adorned atom and the head of σ with P^α. The adornment of the body atoms is obtained from the SIPS and its function f_σ^α. This possibly generates new adorned predicates for which we repeat this step.

Example 8.1.3 (ex. 8.1.2 cont.) P^{bf} is the new adorned predicate obtained from $\mathcal{Q}$. MagicD^+ considers P^{bf} and (8.1). It generates the rule,

$$R^{bf}(x, y), R^{bf}(y, z) \ \rightarrow \ P^{bf}(x, z), \tag{8.3}$$

and adds it to Π_m. This makes new adorned predicate R^{bf}. MagicD^+ generates the adorned rule,

$$U(y), R^{fb}(x, y) \ \rightarrow \ \exists z\, R^{bf}(y, z), \tag{8.4}$$

and adds it to Π_m. Here, (8.2) is not adorned w.r.t. R^{fb}, because this bounds the position $R[2]$ that holds the $\exists$-variable z. The following are the result adorned rules:

$$R^{bf}(x, y), R^{bf}(y, z) \ \rightarrow \ P^{bf}(x, z). \tag{8.5}$$

$$U(y), R^{fb}(x, y) \ \rightarrow \ \exists z\, R^{bf}(y, z). \tag{8.6}$$

□

2. Adding magic atoms and magic rules: Let σ be an adorned rule in Π_m with the head predicate P^α. MagicD^+ adds magic atom of $\mathit{head}(\sigma)$ (cf. Definition 8.1.1) to the body of σ. Additionally, it generates magic rules as follows. For every occurrence of an adorned predicate P^α in σ, it constructs a magic rule σ' that defines $\mathit{mg\_}P^\alpha$ (a magic predicate might have more than one definition). We assume that the atoms in σ' are ordered according to the partial order in the SIPS of σ and α. If the occurrence of P^α is in atom A and there are $A_1, ..., A_n$ on the left hand side of A in σ, the body of σ' 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 8.1.4 (ex. 8.1.3 cont.) Adding the magic atoms to the adorned rules, we obtain the following rules:

$$\mathit{mg\_}P^{bf}(x), R^{bf}(x,y), R^{bf}(y,z) \ \rightarrow P^{bf}(x,z). \tag{8.7}$$

$$\mathit{mg\_}R^{bf}(y), U(y), R^{fb}(x,y) \ \rightarrow \exists z\ R^{bf}(y,z). \tag{8.8}$$

The following magic rules define the magic predicates:

$$\mathit{mg\_}P^{bf}(x) \ \rightarrow \mathit{mg\_}R^{bf}(x). \tag{8.9}$$

$$\mathit{mg\_}R^{bf}(x), R^{bf}(x,y) \ \rightarrow \mathit{mg\_}R^{bf}(y). \tag{8.10}$$

□

3. Adding rules to load extensional data: This step applies only if Π 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. In Example 8.1.3, 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}:

$$mg\_R^{bf}(x), R(x,y) \ \rightarrow \ R^{bf}(x). \tag{8.11}$$

$$mg\_R^{fb}(x), R(x,y) \ \rightarrow \ R^{fb}(x). \tag{8.12}$$

MagicD^+ differs from the rewriting algorithm of [Alviano et al., 2012] in Step 3. In particular, in the latter Step 3 is not needed since, unlike the former, it assumes the intentional predicates in Π and the adorned predicates in Π_m do not have extensional data. Therefore, the correctness of MagicD^+, i.e. $ans(\mathcal{Q}, \Pi) = ans(\mathcal{Q}_m, \Pi_m)$, follows from both the correctness of the rewriting algorithm in [Alviano et al., 2012] and Step 3.

Π_m has certain syntactic properties. The magic rules do not have $\exists$-variables. Also as mentioned in Step 1, the positions of $\exists$-variables in the head of a rule never become bounded. Applying MagicD^+ over a *WS* program Π, Π_m is not necessarily *WS* or in $SCh(\mathcal{S}^{rank})$ as shown in the following example.

Example 8.1.5 Consider BCQ $\mathcal{Q} : \exists x \ R(x,a)$ over program Π with extensional database $D = \{R(a,b), V(b)\}$ and rules:

$$R(x,y) \ \rightarrow \ \exists z \ R(y,z). \tag{8.13}$$

$$R(x,y) \ \rightarrow \ \exists z \ R(z,x). \tag{8.14}$$

$$R(x,y), R(y,z), V(y) \ \rightarrow \ R(y,x). \tag{8.15}$$

Π is *WS* since the only repeated marked variable, y in (8.15), appears in $V[1] \in \pi_F(\Pi)$. Note that every body variable is marked. The result of the magic-sets rewriting Π_m contains the adorned rules:

$$R^{fb}(x,a) \ \rightarrow \ ans_{\mathcal{Q}}. \tag{8.16}$$

$$mg\_R(y), R^{fb}(x,y) \rightarrow \exists z \ R^{bf}(y,z). \tag{8.17}$$

$$mg\_R(x), R^{bf}(x,y) \rightarrow \exists z \ R^{fb}(z,x). \tag{8.18}$$

$$mg\_R(x), R^{bf}(x,y), R^{bf}(y,z), V(y) \rightarrow R^{fb}(y,x). \tag{8.19}$$

$$mg\_R(y), R^{fb}(x,y), R^{bf}(y,z), V(y) \rightarrow R^{bf}(y,x). \tag{8.20}$$

and the magic rules:

$$mg\_R(a). \tag{8.21}$$

$$mg\_R(x), R^{bf}(x,y) \ \rightarrow \ mg\_R(y). \tag{8.22}$$

$$mg\_R(y), R^{fb}(x,y) \ \rightarrow \ mg\_R(x). \tag{8.23}$$

Here, every body variable is marked. Note that according to the description of MagicD$^+$, the magic predicates mg_R^{fb} and mg_R^{bf} are equivalent and so we replace them with a single predicates, mg_R.

Π_m is *not WS*, since $R^{fb}[1], R^{fb}[2], R^{bf}[1], R^{bf}[2], mg\_R[1]$ are not in $\pi_F(\Pi_m)$; and (8.17), (8.18), (8.22) break the syntactic property of *WS*. The chase of Π_m shows that the program is not in $SCh(\mathcal{S}^{rank})$. That is because in a chase step of (8.22) that "a" replaces variable x that appears only in infinite-rank positions $mg\_R[1]$ and $R^{bf}[1]$. Π_m is *JWS*. That is because, $R^{fb}[2], R^{bf}[1]$ are in $\pi_F^{\exists}(\Pi_m)$ and every repeated marked variable appears at least once in one of these two positions. □

The above example proves that $SCh(\mathcal{S}^{rank})$ and *WS* are not closed under MagicD$^+$. This is because MagicD$^+$ introduces new join variables between the magic predicates and the adorned predicates, and these variables might be marked and appear only in the infinite-rank positions. That means the joins may break the $\mathcal{S}^{rank}$-stickiness as it happens in Example 14. Specifically it turned out to be because $\mathcal{S}^{rank}$ decides some finite positions of Π_m^R as infinite-rank positions. In fact, the positions of the new join variables are always bounded and are finite. Therefore, MagicD^+ does not break $\mathcal{S}$-stickiness if we consider a finer selection function $\mathcal{S}$ that decides the bounded positions as finite.

We show in Theorem 8.1.1 that the class of *SCh*($\mathcal{S}^\exists$) and its subclass of *JWS* are closed under MagicD^+ since they apply $\mathcal{S}^\exists$ that better specifies finite positions compared to $\mathcal{S}^{rank}$.

Theorem 8.1.1 Let Π and Π_m be the input and the result programs of MagicD^+, resp. If Π is *JWS*, then Π_m is *JWS*. □

Proof of Theorem 8.1.1: To prove Π_m is in *JWS*, we show every repeated marked variable in Π_m appears at least once in a position of $\pi_F^\exists(\Pi_m)$. The repeated variables in Π_m either: (a) are in adorned rules and correspond to the repeated variables in Π, or (b) appear in magic predicates. For example, y in $mg\_R(x), R^{bf}(x,y), R^{bf}(y,z) \rightarrow R^{fb}(y,x)$ is of type (a) since it corresponds to y in $R(x,y), R(y,z) \rightarrow R(y,x)$. x is a variable of type (b), because it appears in the magic predicate mg_R.

The bounded positions in Π_m are in $\pi_F^\exists(\Pi_m)$. That is because an $\exists$-variable never gets bounded during MagicD^+, and if a position in the head is bounded the corresponding variable appears in the body only in the bounded positions. As a result, a bounded position is *not* in the target of any $\exists$-variable, so it is in $\pi_F^\exists(\Pi_m)$.

The join variables in (a) do not break the $\mathcal{S}^\exists$-stickiness property since they correspond to join variables in Π and Π is *JWS*. This follows two facts: first, a variable in Π_m that corresponds to a marked variable in Π is marked, second, variables in

Π_m that correspond to variables in $\pi_F(\Pi)$ are in $\pi_F(\Pi_m)$. As a result if a repeated variable is not marked or appears at least once in a $\pi_F(\Pi)$, its corresponding variable in Π_m also has these properties. The join variables in (b), also satisfy the *JWS* syntactic condition, because they appear in positions of the magic predicates that are in $\pi_F(\Pi)$. $\square$

As a result of Theorem 8.1.1, we are able to apply MagicD^+ in order to optimize SChQA for the class of *JWS* and its subclasses sticky and *WS*. This shows the class of *JWS* programs has the desirable properties w.r.t. QA while generalizing the class of *WS* programs and sticky programs.

Chapter 9

Partial Grounding and Rewriting for WS Datalog$^{\pm}$

An alternative approach to chased-based bottom-up approach is query rewriting in which a given query is rewritten in terms of rules and constraints in a program and efficiently answered on the extensional database. Sticky Datalog$^{\pm}$ enjoy FO rewritability (cf. Section 2.3.4) and rewriting algorithms are proposed for these programs [Gottlob et al., 2011, 2014].

WS programs, on the other hand, are not FO rewritable, and there is no pure query rewriting algorithm for them. In this chapter, we propose a combined approach that first applies a partial grounding algorithm to convert a *WS* program to a sticky program, for which we can use query rewriting for QA.

9.1 Query Answering based on Partial Grounding

We propose a partial grounding algorithm, called PartialGroundingWS, that takes a *WS* Datalog$^{\pm}$ program Π and transforms it into a sticky Datalog$^{\pm}$ program Π_s such that Π_s is equivalent to Π for CQ answering. PartialGroundingWS selectively replaces certain variables in positions of finite-rank with constants from the active domain of the underlying database.

Our algorithm requires that Π satisfies the condition that there is no $\exists$-variable in Π in any finite-rank position; therefore each position in Π will have rank either 0 or ∞. The reason for this requirement is the convenience of grounding variables at zero-rank positions by replacing them by constants rather than by labeled nulls. This does not really restrict the input programs since, as we will show, an arbitrary program can be transformed by the ReduceRank algorithm to a program that has the requirement.

9.2 The ReduceRank Rewriting Algorithm

ReduceRank takes a program Π and compiles it into an equivalent program $\Pi_{0,\infty}$ that has only zero-rank or infinite-rank positions. The algorithm is inspired by the reduction method in [Krötzsch & Rudolph, 2011] for transforming a weakly-acyclic program into an existential-free Datalog program. Given a program Π, ReduceRank executes the following steps:

1. Initialize $\Pi_{0,\infty}$ with rules and extensional database of Π.

2. Choose a rule σ in $\Pi_{0,\infty}$ with an $\exists$-variable in a position with rank 1. Notice that if there are $\exists$-variables in the finite-rank positions, at least one of them has rank 1.

3. Generate σ' by replacing the $\exists$-variable in σ with a functional term. For example, $\sigma : P(x, y) \rightarrow \exists z\, R(y, z)$, becomes $\sigma' : P(x, y) \ \rightarrow \ R(x, f(x))$.

4. Replace the predicate with functional term with a new expanded predicate of higher arity and introduce a *fresh* constant to represent the function symbol. The constant precedes its arguments in a newly introduced position. For example, $R(x, f(x))$ becomes, $R'(x, \mathsf{f}, x)$, where the position $R[2]$ is expanded.

5. Replace the expanded predicate in other rules. That might expand other predicates in positions where repeated variables appear. For example, if $R[2]$

in $R(x, y), T(y, z) \rightarrow S(x, y, z)$ gets expanded, $T[1]$ and $S[2]$ both get expanded, because of the variable y, and it results in $R'(x, y, y'), T'(y, y', z) \rightarrow S'(x, y, y', z)$.

6. Add new rules to $\Pi_{0,\infty}$ to "load" the extensional data of the expanded predicates. For example, if R has extensional data, we add a rule, $R(x, y) \rightarrow R'(x, y, \triangle)$. $\triangle$ is a fresh constant that is used to fill the new positions in the expanded predicates since they do not carry extensional data.

7. Repeat Steps 2 to 6 until there is no $\exists$-variable in a finite-rank position.

Remark 9.2.1 If a predicate is expanded in a head-atom in a position where an $\exists$-variable occurs, the new positions are not required and are filled with the special symbol $\triangle$. For example, $U(x) \rightarrow \exists y\, R(x, y)$ becomes $U(x) \rightarrow \exists y\, R'(x, y, \triangle)$, if $R[2]$ is expanded.

In Step 3, only the body variables that also appear in the head participate as arguments of the function term. For example, in $P(x, y) \rightarrow \exists z\, R(y, z)$, the function term that replaces z does not include x since the rule can be broken down into $P(x, y) \rightarrow U(y)$ and $U(y) \rightarrow \exists z\, R(y, z)$.

Given a CQ $\mathcal{Q}$ over Π, Steps 2 to 6 are also applied on $\mathcal{Q}$ obtaining a new CQ $\mathcal{Q}_{0,\infty}$ over $\Pi_{0,\infty}$. □

Example 9.2.1 Let Π be a program with the following rules:

$$V(x) \rightarrow \exists y\, R(x, y). \tag{9.1}$$

$$T(x, y), V(x) \rightarrow P(x, y). \tag{9.2}$$

$$R(x, y) \rightarrow \exists z\, T(x, z). \tag{9.3}$$

$$P(x, y) \rightarrow \exists z\, P(y, z). \tag{9.4}$$

In Π, $\pi_F(\Pi) = \{V[1], R[1], R[2], T[1], T[2]\}$. ReduceRank will eliminate y in σ_1 and z in σ_2, but not z in σ_4 since the later is in an infinite-rank position. ReduceRank chooses y in σ_1 over z in σ_2 since y is in $R[2]$ with rank 1, and z is in $T[2]$ with rank 2. After applying Steps 2-6, the following rules are obtained:

$$V(x) \rightarrow R'(x, \mathsf{f}, x). \tag{9.5}$$

$$R'(x, y, y') \rightarrow \exists z\ T(x, z). \tag{9.6}$$

(9.5) and (9.6) replace (9.1) and (9.3), resp. Now, z in (9.6) is placed in a position with rank 1, and ReduceRank repeats Steps 2-6 to eliminate it which results into $\Pi_{0,\infty}$:

$$V(x) \rightarrow R'(x, \mathsf{f}, x). \tag{9.7}$$

$$T'(x, y, y'), V(x) \rightarrow p'(x, \triangle, y, y'). \tag{9.8}$$

$$R'(x, y, y') \rightarrow T'(x, \mathsf{g}, x). \tag{9.9}$$

$$P'(x, x', y, y') \rightarrow \exists z\ P'(y, y', z, \triangle). \tag{9.10}$$

Notice that ReduceRank does not try to remove z in the last rule, since it is in the infinite-rank position $P[3]$. Note also that P is expanded twice since both its positions can host labeled nulls generated by z in σ_2. □

Proposition 9.2.1 Given a CQ $\mathcal{Q}$ over a program Π, ReduceRank runs in EXPTIME to the size of the rules in Π, and returns a CQ $\mathcal{Q}_{0,\infty}$ over a program $\Pi_{0,\infty}$, such that $\Pi_{0,\infty}$ has no $\exists$-variable in $\pi_F(\Pi_{0,\infty})$, and $ans(Q, \Pi) = ans(\mathcal{Q}_{0,\infty}, \Pi_{0,\infty})$. □

Proof of Theorem 9.2.1: Consider each iteration of ReduceRank, i.e. Steps 2-6, that transforms Π_i into Π_{i+1} and removes the $\exists$-variable z_i in σ_i. It expands the predicate P_i in $head(\sigma_i)$ to P_i'. An iteration does not introduce new $\exists$-variables in the finite-rank positions, therefore there are k iterations, such that k is the number of $\exists$-variables in the positions of $\pi_F(\Pi)$.

The variable z_i is in a position with the rank 1, so expanding P_i does not expand any predicate in $body(\sigma_i)$ during Step 5. As a result, every position in the program only gets expanded once during an iteration. Let r and b the number of rules and the maximum number of body atoms in Π, resp. r and b do not change after running each iteration. Let w_i be and the maximum arity of atoms in Π_i. The arity of P_i' (the expanded predicate of P_i) is at most increased by $b \times w_i$ (the maximum possible number of variables in $body(\sigma_i)$). Therefore, after propagating the expanded position in other rules, the maximum arity of the predicates is $w_{i+1} = b \times w_i^2$. After k iterations the maximum arity of the predicates is $b^k \times {w_i}^{2k}$. This shows the size of the result program $\Pi_{0,\infty}$ is EXPTIME to the size of the rules in Π.

The prove $ans(\mathcal{Q}, \Pi) = ans(\mathcal{Q}_{0,\infty}, \Pi_{0,\infty})$, we prove $ans(\mathcal{Q}_i, \Pi_i) = ans(\mathcal{Q}_{i+1}, \Pi_{i+1})$, for each iteration of ReduceRank. That is by constructing an instance $I_{i+1} \models \Pi_{i+1} \cup \mathcal{Q}_{i+1}$, for every instance $I_i \models \Pi_i \cup \mathcal{Q}_i$.

Let assume that removing z_i introduces a function symbol f. For every assignment θ that maps $body(\sigma_i)$ and $head(\sigma_i)$ to I_i, let $\mu_i(\theta(z_i))$ be the list of terms in $\theta(body(\sigma_i))$. Now, for every atom $A = P(t_1, ..., t_n) \in I_i$, we add an atom A' into I_{i+1} that is constructed as follows: (a) if P is not expanded in Π_i then $A' = A$, (b) if P is expanded to P', in its k-th position, there are two possibilities, t_k is either a null value or a constant. If t_k is a constant, expand it into $t_k, \triangle, ..., \triangle$ to fill the expanded positions, and if t_k is a null value, expand t_k into f following by $\mu(t_k)$. $I_{i+1} \models \Pi_{i+1} \cup \mathcal{Q}_{i+1}$, because for every assignment θ' that maps the body of a rule σ into I_{i+1}, we can make an extension of θ'', using μ, that maps the head also into I_{i+1}.

Now, for an instance $I_{i+1} \models \Pi_{i+1} \cup \mathcal{Q}_{i+1}$, we construct a instance $I_i \models \Pi_i \cup \mathcal{Q}_i$. This is simply by replacing any extended predicate P' with its original predicate P and removing additional terms, i.e. removing $\triangle$ symbols and the function symbols and their consequent terms with null values. For example, $P'(a, b, \mathsf{f}, c)$ becomes $P(a, b, \zeta)$ if P expanded by one, and $P'(a, b, d, \triangle)$ becomes $P(a, b, d)$. Again, it is straightforward to prove that $I_i \models \Pi_i \cup \mathcal{Q}_i$ by showing that for any assignment that maps the body of a rule into I_i, there is an extension of it that maps the head into I_i.[1] □

Lemma 9.2.1 The class of *WS* programs is closed under ReduceRank. □

Proof of Lemma 9.2.1: To prove the lemma, we show ReduceRank is closed under each iteration of ReduceRank. Let Π_i be *WS*, the the following hold for Π_{i+1}: (a) If position p is in $\pi_F(\Pi_i)$, and p' is one of the positions resulted by expanding p in Π_{i+1}, then p' is in $\pi_F(\Pi_{i+1})$. (b) If a body variable x is not marked in Π_i, the corresponding variables in Π_{i+1} (resulted from expanding a predicate in the position of x) are not marked in Π_{i+1}.

If Π_{i+1} is not *WS* then there is a repeated marked variable in Π_{i+1} that does not appear in $\pi_F(\Pi_{i+1})$. As a result (a) and (b), there is also a repeated marked variable in Π_i that does not appear in $\pi_F(\Pi_i)$, and Π_i is not *WS*. Since Π_i is *WS*, Π_{i+1} must be also *WS*. This proves each iteration preserves the *WS* syntactic property, so, ReduceRank also preserves the property. □

9.3 The PartialGroundingWS Algorithm

Now that we explained the ReduceRank algorithm, we continue and present the PartialGroundingWS algorithm. Given a WS program Π, let us call *weak rules* the rules of Π in which some repeated marked body variables (which we call *weak variables*) appear at least once in a position with finite-rank. PartialGroundingWS transforms Π into a sticky program Π_s, that has the same extensional database as Π, i.e. $D_s := D$, and

[1] The proof is similar to the proof of Theorem 1 in [Krötzsch & Rudolph, 2011] for EXPTIME combined complexity of reduction from *JA* programs to Datalog programs.

its set of rules is obtained by replacing the weak variables of Π with every constants from the active domain of D and constants in rules of Π. Example 9.3.1 illustrates the PartialGroundingWS algorithm.

Example 9.3.1 Consider a *WS* program Π with $D = \{P(a, b), R(a, b)\}$ and rules:

$$\sigma_1\colon\ \ P(\hat{x}, \hat{y})\ \rightarrow\ \exists z\, P(y, z).$$

$$\sigma_2\colon\ \ P(\hat{x}, y), P(y, z)\ \rightarrow\ S(x, y, z).$$

$$\sigma_3\colon\ \ S(\hat{x}, y, z), R(\hat{x}, y) \rightarrow T(y, z).$$

Here, σ_3 is a weak rule with x as its weak variable. Notice that y in σ_2 and σ_3 are not weak since they are not marked (the hat signs show the marked variables). We replace x with constants a and b from D. The result is a sticky program Π_s that contains σ_1 and σ_2 as well as the following rules, σ_3' : $S(a, y, z), R(a, y) \rightarrow T(y, z)$ and σ_3'' : $S(b, y, z), R(b, y) \rightarrow T(y, z)$. □

Theorem 9.3.1 Let Π be a *WS* program with extensional database D such that there is no $\exists$-variable in $\pi_F(\Pi)$, and let $\mathcal{Q}$ be a CQ over Π. PartialGroundingWS runs in polynomial time with respect to the size of D and it transforms Π and $\mathcal{Q}$ into a sticky program Π_s such that: $\mathit{ans}(\mathcal{Q}, \Pi) = \mathit{ans}(\mathcal{Q}, \Pi_s)$. □

Proof of Theorem 9.3.1: Π_s is sticky since every weak variable that breaks the weakly-stickiness syntactic property is replaced with constants. Also, $\mathit{ans}(\mathcal{Q}, \Pi) = \mathit{ans}(\mathcal{Q}, \Pi_s)$ holds because the weak variables in Π are replaced with every possible constant from D. It is important that no null value can appear in the positions of the weak variables in Π. The algorithm runs in polynomial time with respect to the size of the database because the partial grounding replaces weak variables with polynomially many values from the database, and the number of weak variables is independent of the size of the database. □

A possible optimization for PartialGroundingWS is to narrow down the values for replacing the weak variables. That is to ignore those constants in the active domain of D that can not appear in the positions where weak variables appear during the chase of Π. In Example 9.3.1, σ_3' is not useful since the value "a" can never be assigned to x in σ_3.

The hybrid approach for CQ answering over *WS* programs combines ReduceRank and PartialGroundingWS with a query rewriting algorithm for sticky programs [Gottlob et al., 2011, 2014]. Given a *WS* program Π and a CQ $\mathcal{Q}$, the hybrid algorithm proceeds as follows:

1. Use ReduceRank to compile Π into a *WS* program $\Pi_{0,\infty}$ without $\exists$-variable in finite-rank positions. This also transforms $\mathcal{Q}$ into a new query $\mathcal{Q}_{0,\infty}$.

2. Apply PartialGroundingWS on $\Pi_{0,\infty}$ that results to a sticky program Π_s.

3. Rewrite $\mathcal{Q}_{0,\infty}$ into a FO query $\mathcal{Q}_s$ using the rewriting algorithm proposed in [Gottlob et al., 2011] and answer $\mathcal{Q}_s$ over D (any other sound and complete rewriting algorithm for sticky programs is also applicable at this step).

Example 9.3.2 Consider a *WS* program Π with database $D = \{V(a)\}$ and rules:

$$\begin{aligned}
\sigma_1: &\quad P(x,y) \ \rightarrow\ \exists z\, P(y,z).\\
\sigma_2: &\quad P(x,y), P(y,z) \ \rightarrow\ U(y).\\
\sigma_3: &\quad V(x) \rightarrow\ \exists y\, R(x,y).\\
\sigma_4: &\quad R(x,y), S(x,z) \rightarrow C(z).\\
\sigma_5: &\quad C(x) \ \rightarrow\ \exists y\, P(x,y).
\end{aligned}$$

The ReduceRank method removes the $\exists$-variable y in σ_3. The result is a *WS* program $\Pi_{0,\infty}$ with rules:

$$\begin{aligned} P(x,y) &\rightarrow \exists z\ P(y,z). \\ R'(x,y,y'), S(x,z) &\rightarrow C(z). \\ P(x,y), P(y,z) &\rightarrow U(y). \\ C(x) &\rightarrow \exists y\ P(x,y). \\ V(x) &\rightarrow R'(x,\mathsf{f},x). \end{aligned}$$

Next, PartialGroundingWS grounds the only weak variable, x in σ_4' with constant a which results into sticky program Π_s with $\Pi_s = \{\sigma_1, \sigma_2, \sigma_3', \sigma_4'', \sigma_5\}$, in which $\sigma_4'':\ R'(a,y,y'), S(a,z) \rightarrow C(z)$. Π_s is sticky and a CQs can be answered by rewriting it in terms of Π_s and answered directly on D. □

Corollary 9.3.1 Given a *WS* program Π and a CQ $\mathcal{Q}$, the set of answers obtained from the hybrid approach is $ans(\mathcal{Q}, \Pi)$. □

The corollary concludes the results of this chapter on QA under *WS* Datalog$^\pm$ based on query rewriting. It shows that the combination of partial grounding and rewriting can be used for QA under *WS* programs. This approach can serve as an alternative to SChQA in Chapter 7, while the comparison of the two approaches, and their implementations, remains the future extensions of this work (cf. Chapter 5).

Chapter 10

Related Work

In this chapter, we review some relevant research on data quality and context modeling.

10.1 Declarative Approaches to Data Quality Assessment

Existing solutions for data quality assessment and data cleaning are mostly ad hoc, rigid, and application-dependent. Most approaches to data cleaning are procedural, and provided in terms of specific mechanisms. Their semantics and scope of applicability are not fully understood [Batini & Scannapieco, 2006].

Declarative approaches to data cleaning intend to be more general [Bertossi & Bravo, 2013]. They specify, usually by means of a logic-based formalism, what is the intended result of a data cleaning process. The semantics of the specification tells us what the result should look like, if there are alternative solutions, and what are the conclusions that can be derived from the process and results. They also allow us, in principle, to better understand the range of applicability and the complexity of the declaratively specified cleaning mechanism.

Declarative data quality assessment is focused on using classic ICs, such as functional dependencies and inclusion dependencies, and denial constraints.[1] One can

[1] NCs are denial constraints in Datalog$^{\pm}$.

specify the semantics of quality data with ICs, in a declarative way, and catch inconsistencies and errors that emerge as violations of them. Since they can be violated by the database, the latter can be cleaned by repairing it based on ICs [Chiang & Miller, 2011; Volkovs et al., 2014; Fan, 2009; Kolahi & Lakshmanan, 2009]. The ICs could also be imposed at query answering time, seeing them more like constraints on query answers than on database states [Arenas et al., 1999; Bertossi, 2011b].

The limited expressiveness of classic ICs often does not allow to represent data quality requirements that are commonly found in real life databases. Newer classes of ICs are introduced that extend classic ICs, and are particularly intended to capture data quality issues or conditions, to directly support data cleaning processes [Fan, 2008]. Some examples are *conditional dependencies (conditional FDs and IDs)*, and *matching dependencies* [Fan et al., 2009, 2011]. The latter are applied to *entity resolution (ER)*, which is the problem of discovering and matching database records that represent the same entity in the application domain, and detecting duplicates [Fan et al., 2011].

10.2 Comparison with Data Quality Approaches

Our approach to quality data specification and extraction in Section 5.1 is declarative: it uses logic-based languages, i.e. Datalog and Datalog$^{\pm}$, in order to define and specify quality data. Therefore, compared to procedural approaches [Batini & Scannapieco, 2006], it has the following advantages: it has clear semantics and its scope of applicability can be easily understood and analysed. It is also independent of any procedural mechanism for quality data extraction and data cleaning.

In comparison to the declarative approaches to data quality assessment (cf. Section 10.1), our approach is more general and comprehensive. In particular, those declarative approaches are based on checking some forms of IC (classic ICs such as FDs or newer classes of ICs such as conditional and matching dependencies), while considering the data under assessment as complete (CWA, cf. Section 2.1). Our approach is able to represent rules and constraints, in particular classic ICs. Also, the logic-based languages in our approach can be replaced with any other logic-based formalism, which makes it possible to represent more complex constructs depending on an application. In addition, the approach in our work supports the OWA and provides data completion through value invention as part of the OMD model. This is not supported by the declarative approaches we found in the literature (cf. Section 10.1).

10.3 Data Quality Dimensions Revisited

Regarding the data quality dimensions that we mentioned in Chapter 1 (cf. [Fan, 2008; Jiang et al., 2008] for more details about these dimensions), our approach to quality data specification and extraction is specifically directed at data completeness, a data quality dimension that characterises data quality in terms of the presence/absence of values. Our approach allows the representation of incomplete data (OWA in MD ontologies) with missing contextual information and provides a mechanism to complete the data (using dimensional rules and constraints) and additional contextual data.

Our approach also relates to data consistency quality dimension, which is about the validity and integrity of data representing real-world entities typically identified as satisfaction of integrity constraints. However, our approach goes beyond consistency checking of ICs (CWA in relational databases) and further support the OWA and data completion through rules and constraints.

The data accuracy dimension refers to the closeness of values in a database to the true values for the entities that the data in the database represents. Data accuracy is mostly evaluated in terms of the reliability and trustworthiness of data sources, that is characterised by mechanisms such as using dependencies [Luna Dong et al., 2009] and lineage information [Agrawal et al., 2006a] of data sources in order to detect copy relationships, and employing vote counting [Galland et al., 2010] and probabilistic analysis [Zhao et al., 2012]. There is also a connection between data accuracy and data consistency since certain forms of accuracy can be enforced by ICs and consistency checking [Cong et al., 2007; Batini & Scannapieco, 2006]. For example, some cases of syntactic data inaccuracy can be detected by checking the range or type of values of an attribute using ICs. Therefore, certain forms of data accuracy can be addressed in our approach by means of rules and constraints.

Data currency (timeliness) aims to identify the current values of entities represented by tuples in a (possibly stale) database, and to answer queries with the current values. With respect to data currency, our approach lacks the necessary elements to address this data quality dimension. In particular, the MD ontologies are not able to represent data that are associated with a temporal validity period, something that is necessary for addressing the data quality assessment regarding data currency [Batini & Scannapieco, 2006]. This can be resolved possibly by extending the MD ontologies and our context model with *temporal ontologies* [Borgwardt et al., 2016; Calvanese et al., 2016].

10.4 Context Modeling

Many formalizations and implementations of the notion of context have emerged in various areas of computer science, including *artificial intelligence (AI)*, *knowledge representation* and *data management*. The study of a formal notion of context has a long history in AI; however, it became more widely discussed in the late 1980s, when J. McCarthy proposed the formalization of context in his Turing award lecture [McCarthy, 1987], as a crucial step towards the solution of *the problem of generality*, and devising axioms to express common sense. He raised the issue that no formal theory of common sense can succeed without some formalization of context, since the representation of common sense axioms crucially depend on the context in which they are asserted.

McCarthy elaborated his views in a paper on formalizing context [McCarthy, 1993] where several important concepts around context modeling were presented. Specifically, he introduced *the notion of contexts as first class objects*, expressed by the formula $ist(c, p)$, meaning a proposition p is true in context c; and also operations for entering and exiting contexts. Following [McCarthy, 1987], Guha –under McCarthy's supervision– proposed in his PhD dissertation [Guha, 1992] a formalization of context. In particular, he introduced a formal semantics for formulas of the form $ist(c, p)$. He also discussed several important concepts, such as the notion of context structure and vocabulary, the concept of having a universal well-formed grammar and local vocabularies and their semantics within a given context, and the notion of lifting axioms. In addition, he discussed several applications and techniques of context-based problem-solving techniques.

McCarthy and Guha's work is the basis for Buvač and Mason's *Propositional Logic of Context (PLC)* [Buvač et al., 1995]. PLC intended to formalize McCarthy's views on context, while giving a more traditional, model-theoretic approach to Guha's semantics. Particular relevance is given to the idea that contexts must be formalized as first class objects (i.e. the logical language must contain terms for contexts, and the interpretation domain contains objects for contexts), and to the mechanisms of entering and exiting a context, which are identified as the two main mechanisms of contextual reasoning. [Buvač, 1996] is a generalization of PLC to FO languages.

Following a different line of research, [Giunchiglia, 1993] formalized contexts with motivation in *the locality problem*, namely the problem of modeling reasoning that uses only a subset of what reasoners know about the world. The idea is that in solving a problem on a given occasion, people do not use all their knowledge, but construct a "local theory", and use it as if it contained all relevant facts about the problem at hand. While reasoning, people can switch from one context to another, for example when the original context is not adequate to solve the problem. Under this approach, unlike McCarthy's, the emphasis is more on formalizing contextual reasoning than on formalizing contexts as first class objects.

In [Giunchiglia & Serafini, 1994], *Multi Context Systems (MCS)* are presented as a proof-theoretic framework for contextual reasoning. They introduce the notion of bridge rule, i.e. a special kind of inference rule whose premises and conclusion hold in different contexts. They later proposed *Local Models Semantics (LMS)* as a model-theoretic framework for contextual reasoning, and used MCS to axiomatize many important classes of LMS [Ghidini & Giunchiglia, 2001]. From a conceptual point of view, they argued that contextual reasoning can be analyzed as the result of the interaction of two very general principles: the principle of locality (reasoning always happens in a context); and the principle of compatibility (there can be relationships between reasoning processes in different contexts). In other words, contextual reasoning is the result of the interaction between distinct local structures. More recently, *MCS* have been also investigated, and the problem of bridging them, e.g. using logic programs [Dao-Tran et al., 2010], is matter of recent and ongoing research.

In the area of data management, the notion of context is usually implicit. It is of form of context-awareness, associated to the notion of data dimensions, usually time, user, and location [Bolchini et al., 2007a, 2009, 2007b, 2013; Martinenghi & Torlone, 2009, 2010, 2014]. In context-aware systems, context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including location, time, activities, and the preferences of each entity. A system is context-aware if it can extract, interpret and use contextual information, and adapt its functionality to the current context of use. In information management, context-aware systems are devoted to determining what portion of the entire information is relevant with respect to the ambient conditions of an agent or user [Bolchini et al., 2007a].

In [Ghidini & Serafini, 1998], ideas from [Giunchiglia & Serafini, 1994] are applied to information integration. Specially LMS is used for dealing with different problems in the management of federated databases, where each database may have its own local semantics, which can be formalized by a *Local Model Semantics for federated databases* as an extension of LMS.

In [Analyti et al., 2007; Theodorakis et al., 2002] an interesting formalization of contexts is presented and applied to conceptual modeling. Contexts are sets of named objects, not theories, that allow the context to be structured through the traditional abstraction mechanisms of classification, generalization, and attribution.

A general framework is proposed in [Motschnig, 1995, 2000] for decomposing *information bases* into possibly overlapping fragments, called *contexts*, in order to be able to better manage and customize information. Examples of information bases are databases, knowledge bases, softwares, and programming languages, for which contexts are defined as database views, knowledge base partitions, software components, and program scopes, resp. The framework provides general mechanisms for partitioning and coping with a fragmented information base.

According to *the context relational data model* [Rousoss et al., 2005], a context is a first-class citizen at the level of the data model and its query language. It is defined as a set of *worlds*, where each world is characterised by pairs of dimension names and their values. A relation in this data model, called a *context-relation*, is a collection of classic relations (as in the relational data model), and each classic relation is assigned to a possible world, representing the context-relation in that world. Accordingly, an attribute of a context-relation may not exist in some worlds, or the same attribute may have different values under different worlds. A set of basic operations are provided that extend relational algebra for querying context-relations, taking into account the contexts and possible worlds.

A *preference database system* is presented in [Stefanidis et al., 2005, 2007] that supports context-aware queries; that is, queries whose results depend on the context at the time of their submission. Here, a context is modeled as a set of multidimensional attributes; and data cubes (as in the MD data model) are used to store the dependencies between context-dependant preferences and database relations. That makes it possible to apply OLAP techniques for processing context-aware queries. Auxiliary data structures, called *context-trees*, store results of past context-aware queries indexed by the context of their execution.

10.5 Comparison with Related Context Models

Here, we make comparisons between our notion of context in Section 5.1 and those that we reviewed in Sections 3.2, 3.3, and 10.4. Notice that some of the the context models reviewed in Section 10.4 are from other areas, such as AI and knowledge representation, and not strictly for data management, and they are not easily comparable with our approach.

1. The multidimensional aspect of context is not considered in [Motschnig, 1995, 2000]. In [Rousoss et al., 2005], ambient dimensions are used solely as names/labels, and values from some domain sets are assigned to these dimensions to characterise context. Context-aware data tailoring [Bolchini et al., 2007a,b, 2009] further allows sub-dimensions with values of finer granularity. In [Martinenghi & Torlone, 2009, 2014; Stefanidis et al., 2005, 2007], similar to our approach, dimensions are defined as in the HM data model.

2. With regard to the compatibility with the relational data model, we gave a complete relational representation of an extension of the HM data model: data in our model is modeled as relations only (cf. Section 4.1). However, the relational context models that we reviewed in this thesis, namely [Bolchini et al., 2007a,b, 2009; Martinenghi & Torlone, 2009, 2014; Stefanidis et al., 2005, 2007; Rousoss et al., 2005] (that are the closest ones we found in the literature to the database discipline) are not completely relational: they use an extension of relations with new data entities. In [Rousoss et al., 2005], a collection of relations represents a context relation, and creating, manipulating and querying these context relations needs additional care with respect to the underlying collection of relations. In [Martinenghi & Torlone, 2009, 2014; Stefanidis et al., 2005, 2007; Bolchini et al., 2007a,b, 2009], no relational representation of dimensions is given. In particular, [Martinenghi & Torlone, 2009, 2014; Stefanidis et al., 2005, 2007] use the MD data model for modeling dimensions, and [Bolchini et al., 2007a,b, 2009] propose context dimension trees (CDTs) and chunk configurations, which are not represented by relational terms.

3. In terms of languages for querying context, [Bolchini et al., 2007a,b, 2009; Martinenghi & Torlone, 2009, 2014; Stefanidis et al., 2005, 2007] use extensions of relational algebra that enable context querying, and they inherit the shortcomings of relational algebra. For example, they can not express recursive queries that are supported in our MD context.

4. The notion of context is explicit and represented by a first class entity in [Rousoss et al., 2005; Bolchini et al., 2007a,b, 2009]. In [Martinenghi & Torlone, 2009, 2014], similar to our work, context is implicitly modeled as certain "contextual" attributes that take dimensional values. In [Motschnig, 1995, 2000], the notion of context is abstract and is captured by partitions over an information base, e.g. views over a database.

5. Concerning the applications of these context models, the one in [Stefanidis et al., 2005, 2007] is in particular for context-aware preference databases. Context-aware data tailoring [Bolchini et al., 2007a,b, 2009] is designed as a methodology for managing small databases aimed at being hosted by portable devices. For both of these context models, it is not clear how they can be adapted for other purposes. The work on context-aware databases in [Martinenghi & Torlone, 2009, 2014] is fairly general and can be applied in many applications in data management. But, still there are necessary useful and necessary constructs for some applications that are not supported, in particular, recursive queries and capturing incomplete data, that are both supported by our notion of context.

Chapter 11

Conclusions and Future Work

11.1 Conclusions

In this thesis, we started from the idea that data quality is context-dependent. As a consequence, we needed a formal model of context for context-based data quality assessment and quality data extraction. For that we followed and extended the approach in [Bertossi et al., 2011a, 2016]. In that work, context is represented as a database, or as a database schema with partial information, or, more generally, as a virtual data integration system [Lenzerini, 2002] that receives and processes the data under quality assessment. However, contexts have a dimensional nature, e.g. representing information about the time or location, which is not considered in [Bertossi et al., 2011a, 2016].

Here, in order to capture general dimensional aspects of data for inclusion in contexts, we started from the HM data model [Hurtado & Mendelzon, 2002; Hurtado et al., 2005]. The HM model has shortcomings when it comes to applications beyond DWHs and OLAP, including context modeling. We resolved this by the use of MD ontologies in our proposed OMD model.

The proposed OMD model extends the HM model while replacing fact-tables with more general categorical relations. Unlike fact-tables, they can store non-numerical data, and can be linked to different levels of dimensions, other than the base level.

The model is also enriched with dimensional rules and constraints to express additional knowledge, and adding the capability of navigating multiple dimensions in both upward and downward directions.

We represented MD ontologies using the Datalog$^\pm$ ontological language, and we showed that the result falls in the syntactic class of *WS* Datalog$^\pm$ programs, for which CQ answering is tractable. We also studied several issues around the OMD model, among others: adding a form on uncertain downward navigation, consistent QA when facing inconsistent MD ontologies, and reconstruction of other context models.

We used the MD ontologies and propose a general methodology for contextual and multidimensional data quality specification and extraction.

In the second part of the thesis, the first being the OMD model and data quality assessment, we analysed *WS* Datalog$^\pm$ by investigating its syntactic and semantics properties that lead to the characterization of a range of syntactic and semantic programs classes that extend *WS* programs. This includes the new syntactic class of *JWS* that is more general than *WS* and its programs inherit good computational properties of *WS* Datalog$^\pm$ programs. We proposed a bottom-up chase-based QA algorithm for those programs, and presented a magic-sets optimization for QA under *JWS*, which is closed under magic-sets.

We also introduced a hybrid approach to CQ answering based on combining the query rewriting and grounding CQ answering paradigms. This hybrid approach transforms a *WS* program using the underlying data into a sticky program that is then used for query rewriting.

11.2 Future Work

We conclude this thesis with a list of problems for further research, and the sections they are related to:

1. We presented a syntactic condition in Proposition 4.2.2 that guarantees separability of dimensional rules and constraints in the MD ontology. There are two different directions that are interesting to explore: (a) studying other syntactic conditions (such as a non-conflicting condition) that can guarantee separability in MD ontologies; (b) studying non-separable rules and constraints for which QA is still decidable.

2. The quality predicates and quality versions in Section 5.1 are defined as non-recursive Datalog rules over the MD ontology, which guarantees tractable QA over the result ontology. We intend to further study the definition of quality predicates and quality versions using more expressive rules and its impact on QA.

3. The problem of representing and reasoning about Datalog with aggregates has received considerable interest in the database and the logic community and different extensions of Datalog have been proposed to support aggregate rules. We can combine the results and methods in the literature with MD ontologies.

4. With regard to QA under *WS* programs and the algorithms in Chapters 7 and 10, we will work on further optimizations and implementations of them and on experiments using real world data. In particular, for our hybrid algorithm in Chapter 10, a possible improvement can be the use of only necessary constants

from the program's extensional database for partial grounding. With this we can decrease the number of sticky rules resulting from the algorithm.

5. We discussed a novel approach to the enforcement of constraints along with data propagation (cf. Section 4.3.3). In our opinion, this is an interesting approach for dealing with constraints and generating repairs in ontologies. We intend to formalize this approach and its semantics, and study the properties of the generated repairs, and also the connections with other inconsistency-tolerant semantics in the literature.

6. We showed in Section 4.3.1 a form of dimensional navigation that requires mixed closed/open predicates in the MD ontologies for representing, which provably leads to intractability of QA. We intend to investigate the following possible solutions to retain tractability of QA: (a) imposing syntactic restrictions on dimensional rules, e.g. to navigate in one direction, (b) restriction on the hierarchy of dimensions, e.g. the number of levels in a dimension, (c) considering simpler CQs such as atomic queries.

Bibliography

Abiteboul, S., Hull, R. and Vianu, V. *Foundations of Databases.* Addison-Wesley, 1995.

Agrawal, P., Benjelloun, O., Das Sarma, A., Hayworth, C., Nabar, S., Sugihara, T. and Widom, J. Trio: A System for Data, Uncertainty, and Lineage. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, 2006, pp. 1151-1154.

Ahmetaj, S., Ortiz, M. and Šimkus, M. Polynomial Datalog Rewritings for Ontology Mediated Queries with Closed Predicates. In *Proc. of the Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, CEUR-WS Proc. Vol. 1644, 2016.

Aho, A. V., Beeri, C. and Ullman, J. D. The Theory of Joins in Relational Databases. *ACM Transactions on Database Systems (TODS)*, 1979, 4(3): 297-314.

Alviano, M., Faber, W., Leone, N. and Manna, M. Disjunctive Datalog with Existential Quantifiers: Semantics, Decidability, and Complexity Issues. *Theory and Practice of Logic Programming (TPLP)*, 2012, 12(4-5): 701-718.

Alviano, M., Leone, N., Manna, M., Terracina, G. and Veltri, P. Magic-Sets for Datalog with Existential Quantifiers. In *Proc. of the International Conference on Datalog in Academia and Industry 2.0*, 2012, Springer LNCS 7494, pp. 31-43, DOI: 10.1007/978-3-642-32925-8_5.

Alviano, M. and Pieris, A. Default Negation for Non-Guarded Existential Rules. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2015, pp. 79-90.

Anality, A., Theodorakis, M., Spyratos, N. and Constantopoulos, P. Contextualization as an Independent Abstraction Mechanism for Conceptual Modeling. *Information Systems*, 2007, 32(1): 24-60.

Arenas, M., Bertossi, L. and Chomicki, J. Consistent Query Answers in Inconsistent Databases. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 1999, pp. 68-79.

Arenas, M., Gottlob, G. and Pieris, A. Expressive Languages for Querying the Semantic Web. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2014, pp. 14-26.

Artale, A., Calvanese, D., Kontchakov, R. and Zakharyaschev, M. DL-Lite in the Light of First-Order Logic. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2007, pp. 361-366.

Artale, A., Calvanese, D., Kontchakov, R. and Zakharyaschev, M. The DL-lite Family and Relations. *J. of Artificial Intelligence Research (JAIR)*, 2009, 36(1): 1-69.

Baader, F., Brandt, S. and Lutz, C. Pushing the $\mathcal{EL}$ Envelope. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2005.

Baader, F., Calvanese, D., McGuinness, L., Nardi, D. and Patel-Schneider, P. F. *Description Logic Handbook*, 2nd Edition. *Cambridge Univinverity Press*, 2007.

Baget, J. F., Leclére, M., Mugnier, M. L. and Salvat, E. Extending Decidable Cases for Rules with Existential Variables. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2009, pp. 677-682.

Baget, J. F., Mugnier, M. L., Rulolph, S. and Thomazo, M. Walking the Complexity Lines for Generalized Guarded Existential Rules. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011, pp. 712-717.

Baget, J. F., Leclére, M., Mugnier, M.L. and Salvat, E. On Rules with Existential Variables: Walking the Decidability Line. *Artificial Intelligence*, 2011, 175(9-10): 1620-1654.

Baget, J. F., Bienvenu, M., Mugnier, M.L. and Rocher, S. Combining Existential Rules and Transitivity: Next Steps. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2015, pp. 2720-2726.

Barcelo, P. Logical Foundations of Relational Data Exchange. *ACM SIGMOD Record*, 2009, 38(1):49-58.

Batini, C. and Scannapieco, M. *Data Quality: Concepts, Methodologies and Techniques.* Springer, 2006.

Beeri, C. and Vardi, M. Y. The Implication Problem for Data Dependencies. In *Proc. of the Colloquium on Automata, Languages and Programming (ICALP)*, 1981, Springer LNCS 115, pp. 73-85.

Beeri, C. and Vardi, M. Y. A Proof Procedure for Data Dependencies. *Journal of the ACM (JACM)*, 1984, 31(4): 718-741.

Beeri, C. and Ramakrishnan, R. On the Power of Magic. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 1987, pp. 269-284.

Bertossi, L., Rizzolo, F. and Lei, J. Data Quality is Context Dependent. In *Proc. of the Workshop on Enabling Real-Time Business Intelligence (BIRTE) Collocated with the International Conference on Very Large Data Bases (VLDB)*, Springer LNBIP 84, 2011, pp. 52-67.

Bertossi, L. *Database Repairing and Consistent Query Answering.* Morgan & Claypool, 2011.

Bertossi, L. and Bravo, L. Generic and Declarative Approaches to Data Quality Management. In *Handbook of Data Quality Research and Practice*, 2013, Springer, pp. 181-211, DOI: 10.1007/978-3-642-36257-6_9.

Bertossi, L. and Rizzolo, F. Contexts and Data Quality Assessment. Corr Arxiv Paper cs.DB/1608.04142, 2016.

Bienvenu, M. On the Complexity of Consistent Query Answering in the Presence of Simple Ontologies. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2012, pp 705-711.

Bienvenu, M. and Rosati, R. Tractable Approximations of Consistent Query Answering for Robust Ontology-Based Data Access. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2013, pp. 775-781.

Bienvenu M., Bourgaux, C. and Goasdouè, F. Querying Inconsistent Description Logic Knowledge Bases under Preferred Repair Semantics. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2014, pp 996-1002.

Bolchini, C., Schreiber, F. and Tanca, L. A Methodology for a Very Small Data Base Design. *Information Systems*, 2007, 32(1): 61-82.

Bolchini, C., Curino, C. A., Quintarelli, E., Schreiber, F. A. and Tanca, L. A Data-Oriented Survey of Context Models. *ACM SIGMOD Record*, 2007, 36(4): 19-26.

Bolchini, C., Quintarelli, E., Rossato, R. and Tanca, L. Using Context for the Extraction of Relational Views. In *Proc. of the International and Interdisciplinary Conference on Modeling and Using Context*, 2007, pp 108-121.

Bolchini, C., Curino, C. A., Quintarelli, E., Schreiber, F. A. and Tanca, L. Context Information for Knowledge Reshaping. *International Journal of Web Engineering and Technology*, 2009, 5(1): 88-103.

Bolchini, C., Quintarelli, E. and Tanca, L. CARVE: Context-Aware Automatic View Definition over Relational Databases. *Information Systems*, 2007, 38(1): 45-67.

Borgwardt, S., Lippmann, M. and Thost, V. Temporalizing Rewritable Query Languages. *Web Semantics*, 2015, 33:50-70.

Buvač, S., Buvač, V. and Mason, I. Metamathematics of Contexts. *Fundamenta Informaticae*, 1995, 23(1): 263-301.

Buvač, S. Quantificational Logic of Context. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1996, pp. 600-606.

Calì, A., Lembo, D. and Rosati, R. On the Decidability and Complexity of Query Answering over Inconsistent and Incomplete Databases. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2003, pp. 260-271.

Calì, A., Gottlob, G. and Lukasiewicz, T. Datalog$^{\pm}$: A Unified Approach to Ontologies and Integrity Constraints. In *Proc. of the International Conference on Database Theory (ICDT)*, 2009, pp. 14-30.

Calì, A., Gottlob, G. and Pieris, A. Advanced Processing for Ontological Queries. In *Proc. VLDB Endowment (PVLDB)*, 2010, 3(1-2): 554-565.

Calì, A., Gottlob, G., Lukasiewicz, T., Marnette, B. and Pieris, A. Datalog$^{\pm}$: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *Proc. of the Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2010, pp. 228-242.

Calì, A., Gottlob, G., Lukasiewicz, T. and Pieris, A. A Logical Toolbox for Ontological Reasoning. *ACM SIGMOD Record*, 2011, 40(3): 5-14.

Calì, A., Gottlob, G. and Pieris, A. Ontological Query Answering under Expressive Entity-Relationship Schemata. *Information Systems*, 2012, 37(4): 320-335.

Calì, A., Gottlob, G. and Lukasiewicz, T. A General Datalog-Based Framework for Tractable Query Answering over Ontologies. *Web Semantics*, 2012, 14:57-83.

Calì, A., Gottlob, G. and Pieris, A. Towards More Expressive Ontology Languages: The Query Answering Problem. *Artificial Intelligence*, 2012, 193:87-128.

Calì, A., Console, M. and Frosini, R. On Separability of Ontological Constraints. In *Proc. of the Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, 2012, CEUR-WS Proc. Vol. 866, pp. 48-61.

Calì, A., Gottlob, G. and Kifer, M. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *J. of Artificial Intelligence Research (JAIR)*, 2013, 48(1): 115-174.

Calvanese, D., Giacomo, G.C., Lembo, D., Lenzerini, M. and Rosati, R. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *J. of Automated Reasoning*, 2007, 39(3): 385-429.

Calvanese, D., Kalayci, G.E., Ryzhikov, V. and Guohui, X. Towards Practical OBDA with Temporal Ontologies. In *Proc. of the International Conference on Web Reasoning and Rule Systems (RR)*, 2016, pp. 18-24.

Ceri, S., Gottlob, G. and Tanca, L. *Logic Programming and Databases.* Springer, 1990.

Chandra, A.K. and Vardi, M.Y. The Implication Problem for Functional and Inclusion Dependencies. *SIAM Journal of Computing*, 1985, 14(3): 671-677.

Chiang, F. and Miller, R. A Unified Model for Data and Constraint Repair. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2011, pp. 446-457.

Cong, G., Fan, W., Geerts, F., Xibei, J. and Shuai, M. Improving Data Quality: Consistency and Accuracy. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, 2007, pp. 315-326.

Dao-Tran, M., Eiter, T., Fink, M. and Krennwallner, T. Distributed Nonmonotonic Multi-Context Systems. In *Proc. of the International Conference on Principles of Knowledge Represenattion and Reasoning (KR)*, 2010, pp. 60-70.

Deutsch, A., Nash, A. and Remmel, J. The Chase Revisited. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2008, pp. 149-158.

Eckerson, W. Data Quality and the Bottom Line: Achieving Business Success Through a Commitment to High Quality Data. *Report of the Data Warehousing Institute*, 2002.

Enderton, H. B. *A Mathematical Introduction to Logic*. 2nd Edition, Academic Press, 2001.

Fagin, R., Kolaitis, P. G., Miller, R. J. and Popa, L. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science (TCS)*, 2005, 336(1): 89-124.

Fan, W. Dependencies Revisited for Improving Data Quality. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2008, pp 159-170.

Fan, W. Constraint-Driven Database Repair. In *Encyclopedia of Database Systems, Springer US*, 2009, pp 458-463.

Fan, W., Jia, X., Li, J. and Ma, S. Reasoning about Record Matching Rules. In *Proc. VLDB Endowment (PVLDB)*, 2009, 2(1): 407-418.

Fan, W., Gao, H., Ji, X., Li, J. and Ma, S. Dynamic Constraints for Record Matching. *The International Journal on Very Large Data Bases (VLDBJ)*, 2009, 20(4): 495-520.

Franconi, E., Garcia, Y. and Seylan, I. Query Answering with DBoxes is Hard. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2011, 278(1): 71-84.

Galland, A., Abiteboul, S., Marian, A. and Senellart, P. Corroborating Information from Disagreeing Views. In *Proc. of the International Conference on Web Search and Data Mining (WSDM)*, 2010, pp. 131-140.

Giunchiglia, F. Contextual Reasoning. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1993, pp. 39-49.

Giunchiglia, F. and Serafini, L. Multilanguage Hierarchical Logics, or: How We Can Do without Modal Logics. *Artificial Intelligence*, 1994, 65(1): 29-70.

Ghidini, C. and Serafini, L. Model Theoretic Semantics for Information Integration. In *Proc. of the International Conference on Artificial Intelligence, Methodology, Systems, and Applications (AIMSA)*, 1998, Springer LNAI Vol. 1480, pp. 267-280.

Ghidini, C. and Giunchiglia, F. Local Models Semantics, or Contextual Reasoning = Locality + Compatibility. *Artificial Intelligence*, 2001, 127(1): 221-259.

Guha, R. V. Contexts: A Formalization and Some Applications. *Ph.D. Dissertation, Stanford University*, 1992.

Gottlob, G., Orsi, G. and Pieris, A. Ontological Queries: Rewriting and Optimization. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2011, pp. 2-13.

Gottlob, G., Orsi, G. and Pieris, A. Query Rewriting and Optimization for Ontological Databases. *ACM Transactions on Database Systems (TODS)*, 2014, 39(3): Article No. 25.

Gottlob, G., Kikot, S., Kontchakov, R., Podolskii, V., Schwentick, T. and Zakharyaschev. M. The Price of Query Rewriting in Ontology-Based Data Access. *Artificial Intelligence.*, 2015, 213(1): 42-59.

Herzog, T., Scheuren, F. and Winkler, W. *Data Quality and Record Linkage Techniques.* Springer, 2009.

Horrocks, I., Kutz, O. and Sattler, S. The even more Irresistible SROIQ. In *Proc. of the International Conference on Principles of Knowledge Represenattion and Reasongin (KR)*, 2006, pp. 57-67.

Hurtado, C. and Mendelzon, A. OLAP Dimension Constraints. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2002, pp. 169-179.

Hurtado, C., Gutierrez, C. and Mendelzon, A. Capturing Summarizability with Integrity Constraints in OLAP. *ACM Transactions on Database Systems (TODS)*, 2005, 30(3): 854-886.

Imielinski, T. and Lipski, W. Incomplete Information in Relational Databases. *J. of the ACM*, 1984, 31(4): 761-791.

Jensen, Ch. S., Bach Pedersen, T. and Thomsen, Ch. *Multidimensional Databases and Data Warehousing.* Morgan & Claypool, 2010.

Jiang, L., Borgida, A. and Mylopoulos, J. Towards a Compositional Semantic Account of Data Quality Attributes. In *Proc. International Conference on Conceptual Modeling (ER)*, 2008, pp. 55-68.

Johnson, D. S. and Klug, A. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 1984, pp. 164-169.

Kolahi, S. and Lakshmanan, L. On Approximating Optimum Repairs for Functional Dependency Violations. In *Proc. of the International Conference on Database Theory (ICDT)*, 2009, pp. 53-62.

Kolaitis, P. G., Tan, W. C. and Panttaja, J. The Complexity of Data Exchange. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2006, pp. 30-39.

Krötzsch, M. and Rudolph, S. Extending Decidable Existential Rules by Joining Acyclicity and Guardedness. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011, pp. 963-968.

Lembo, D., Lenzerini M., Rusati, R., Ruzzi, M. and Savo, D. F. Inconsistency-Tolerant Semantics for Description Logics. In *Proc. of the International Conference on Web Reasoning and Rule Systems (RR)*, 2010, pp. 103-117.

Lenzerini, M. Data Integration: A Theoretical Perspective. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2002, pp. 233-246.

Leone, N., Manna, M., Terracina, G. and Veltri, P. Efficiently Computable Datalog$^{\exists}$ Programs. In *Proc. of the International Conference on Principles of Knowledge Represenattion and Reasongin (KR)*, 2012, pp. 13-23.

Lukasiewicz, T., Martinez, M., Pieris, A. and Simari, G. Inconsistency Handling in Datalog+/- Ontologies. In *Proc. of the European Conference on Artificial Intelligence (ECAI)*, 2012, pp. 558-563.

Lukasiewicz, T., Martinez, M., Pieris, A. and Simari, G. From Classical to Consistent Query Answering under Existential Rules. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 2015, pp. 1546-1552.

Luna Dong, X., Berti-Equille, L. and Srivastava, D. Reasoning about Record Matching Rules. In *Proc. VLDB Endowment (PVLDB)*, 2009, 2(1): 562-573.

Lutz, C. and Wolter. F. Lutz, C., Seylan, I. and Wolter, F. Non-uniform Data Complexity of Query Answering in Description Logics. In *Proc. of the International Workshop on Description Logics (DL)*, 2012, CEUR-WS Proc. Vol 745.

Lutz, C., Seylan, I. and Wolter, F. Ontology-Based Data Access with Closed Predicates is Inherently Intractable (Sometimes). In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2013, pp. 1024-1030.

Lutz, C., Seylan, I. and Wolter, F. Ontology-Mediated Queries with Closed Predicates. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2015, pp. 3120-3126.

Maier, D., Mendelzon, A. and Sagiv, Y. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems (TODS)*, 1979, 4(4): 455-469.

Meier, M., Schmidt, M. and Lausen, G. On Chase Termination Beyond Stratification. In *Proc. VLDB Endowment (PVLDB)*, 2009, 2(1): 970-981.

Marnette, B. Generalized Schema-Mappings: from Termination to Tractability. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 2009, pp. 13-22.

Martinenghi, D. and Torlone, R. Querying Context-Aware Databases. In *Proc. of the International Conference on Flexible Query Answering Systems (FQAS)*, 2009, pp. 76-87.

Martinenghi, D. and Torlone, R. Querying Databases with Taxonomies. In *Proc. of the International Conference on Conceptual Modeling (ER)*, 2010, pp. 377-390.

Martinenghi, D. and Torlone, R. Taxonomy-Based Relaxation of Query Answering in Relational Databases. *The International Journal on Very Large Data Bases (VLDBJ)*, 2014, 23(5): 747-769.

McCarthy, J. Generality in Artificial Intelligence. *Communications of the ACM*, 1987, 30(12): 1030-1035.

McCarthy, J. Notes on Formalizing Context. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 1993, pp. 555-560.

Milani, M., Bertossi, L. and Ariyan, S. Extending Contexts with Ontologies for Multidimensional Data Quality Assessment. In *Proc. of the International Workshop on Data Engineering meets the Semantic Web (DESWeb) collocated with the International Conference on Data Engineering (ICDE)*, 2014, pp. 242-247, DOI:10.1109/ICDEW.2014.6818333.

Milani, M. and Bertossi, L. Tractable Query Answering and Optimization for Extensions of Weakly-Sticky Datalog±. In *Proc. of the Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, CEUR-WS Proc. Vol. 1378, 2015, pp. 101-105.

Milani, M. and Bertossi, L. Ontology-Based Multidimensional Contexts with Applications to Quality Data Specification and Extraction. In *Proc. of the International Symposium on Rules and Rule Markup Languages for the Semantic Web (RuleML)*, Springer LNCS 9202, 2015, pp. 277-293.

Milani, M., Bertossi, L. and Calì, A. Query Answering on Expressive Datalog$^{\pm}$ Ontologies. In *Proc. of the Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, CEUR-WS Proc. Vol. 1644, 2016.

Milani, M. and Bertossi, L. Extending Weakly-Sticky Datalog$^{\pm}$: Query-Answering Tractability and Optimizations. In *Proc. of the International Conference on Web Reasoning and Rule Systems (RR)*, Springer LNCS 9898, 2016, pp. 128-143.

Milani, M., Bertossi, L. and Calì, A. A Hybrid Approach to Query Answering under Expressive Datalog$^{\pm}$. In *Proc. of the International Conference on Web Reasoning and Rule Systems (RR)*, Springer LNCS 9898, 2016, pp. 144-158.

Motschnig-Pitrik, R. An Integrating View on the Viewing Abstraction: Contexts and Perspectives in Software Development, AI, and Databases. *Systems Integration*, 1995, 5(1): 23-60.

Motschnig-Pitrik, R. A Generic Framework for the Modeling of Contexts and its Applications. *Data & Knowledge Engineering*, 2000, 32(2): 145-180.

Ngo, N., Ortiz, M. and Šimkus, M. The Combined Complexity of Reasoning with Closed Predicates. In *Proc. of the International Workshop on Description Logic (DL)*, CEUR-WS Proc. Vol. 1350, 2015.

Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M. and Rosati, R. Linking Data to Ontologies. *Data Semantics*, 2008, 10(1): 133-173.

Rabin, M. O. A Simple Method for Undecidability Proofs and Some Applications. In *Logic, Methodology and Philosophy of Science, Proceedings of the 1964 International Congress*, Bar-Hillel, Y. (ed.). Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, Amsterdam 1965, pp. 38-68.

Rajugan, R., Dillon, T. S., Chang, E. and Feng, L. Modeling Views in the Layered View Model for XML using UML. *International Journal of Web Information Systems (IJWIS)*, 2006, 2(2): 95-117.

Redman, T. The Impact of Poor Data Quality on the Typical Enterprise. *Communications of the ACM*, 1998, 41(2): 79-82.

Reiter, R. Towards a Logical Reconstruction of Relational Database Theory. In *On Conceptual Modelling*, Springer, 1984, pp. 191-233.

Rosati, R. On the Complexity of Dealing with Inconsistency in Description Logic Ontologies. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2011, pp. 1057-1062.

Rousoss, Y., Stavrakas, Y. and Pavlaki, V. Towards a Context-Aware Relational Model. In *Proc. International Workshop on Context Representation and Reasoning (CRR)*, 2005, pp. 5-17.

Schmidt-Schauß, M. and Smolka, G. Attributive Concept Descriptions with Complements. *Artificial Intelligence*, 1991, 48(1): 1-26.

Stefanidis, K., Pitoura, E. and Vassiliadis, P. A Context-Aware Preference Database System. *Pervasive Computing and Communications*, 2005, 3(4): 439-460.

Stefanidis, K., Pitoura, E. and Vassiliadis, P. Adding Context to Preferences. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2007, pp. 846-855.

Theodorakis, M., Anality, A., Constantopoulos, P. and Spyratos, N. A Theory of Contexts in Information Bases. *Information Systems*, 2002, 27(3): 151-191.

Vardi, M. On the Complexity of Bounded-Variable Queries. In *Proc. of the ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (PODS)*, 1995, pp. 266-276.

Volkovs, M., Chiang, F., Szlichta, J., and Miller, R. Continuous Data Cleaning. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2014, pp. 244-255.

Zhao, B., Rubinstein, I. P. R., Gemmell, J. and Han, J. A Bayesian Approach to Discovering Truth from Conflicting Sources for Data Integration. In *Proc. VLDB Endowment (PVLDB)*, 2012, 5(6): 550-561.