Roskilde University
Department of Computer Science

Advanced Techniques for Efficient Data Integrity Checking

Ph.D. Dissertation

Davide Martinenghi

Supervisor:
Prof. Henning Christiansen

October 2005

Abstract

Integrity constraint checking, understood as the verification of data correctness and well-formedness conditions that must be satisfied in any state of a database, is not fully supported by current database technology. In a typical scenario, a database is required to comply with given semantic criteria (the integrity constraints) and to maintain the compliance each time data are updated.

Since the introduction of the SQL2 standard, the SQL language started supporting *assertions*, which allow one to define general data consistency requirements expressing arbitrarily complex "business rules" that may go beyond predefined constraints such as primary keys and foreign keys. General integrity constraints are, however, far from being widely available in commercial systems; in fact, their usage is commonly not encouraged, since the database management system would not be able to provide their incremental evaluation. Given the size of today's data repositories and the frequency at which updates may occur, any non-incremental approach, even for conditions whose complexity is only linear in the size of the database, may prove unfeasible in practice.

Typically it is the database designer and the application programmer who take care of enforcing integrity via hand-coded pieces of programs that run either at the application level or within the DBMS (e.g., triggers). These solutions are, however, both difficult to maintain and error prone: small changes in a database schema may require subtle modifications in such programs.

In this respect, database management systems need to be extended with means to verify, *automatically* and *incrementally*, that no violation of integrity is introduced by database updates. For this purpose we develop a procedure aimed at producing incremental checks whose satisfaction guarantees data integrity. A so-called *simplification* procedure takes in input a set of constraints and a pattern of updates to be executed on the data and outputs a set of optimized constraints which are as incremental as possible with respect to the hypothesis that the database is initially consistent. In particular, the proposed approach allows the compilation of incremental checks at database *design time*, thus without burdening database run time performance with expensive optimization operations. Furthermore, integrity verification may take place *before* the execution of the update, which means that the database will never reach illegal states and, thus, rollback as well as repair actions are virtually unneeded.

The simplification process is unavoidably bound to a function that gives an approximate measure of the cost of evaluating the simplified constraints in actual database states and it is natural to characterize as *optimal* a simplification with a minimal cost. It is shown that, for any sensible cost function, no simplification procedure exists that returns optimal results in all cases. In spite of this negative result, that holds for the most general setting, important contexts can be found in which optimality can indeed always be guaranteed. Furthermore, non-optimal simplification may imply a slight loss of efficiency, but still is a great improvement with respect to non-incremental checking.

Finally, we extend the applicability of simplification to a number of different contexts, such as recursive databases, concurrent database systems, data integration systems and XML document collections, and provide a performance evaluation of the proposed model.

Acknowledgements

I would like to express my first words of gratitude to my supervisor, Henning Christiansen, knowing that my thanks cannot compensate the enormous dedication and the long hours of discussion that he devoted to my work. His patience and sharpness of mind have been invaluable means for my understanding of the subject and for the improvement of my drafts.

I am also very grateful to Stefano Ceri, who let me join the database group at Politecnico di Milano for a period of six months for a fruitful and pleasurable collaboration. Among the people in Milano with whom I had insightful discussions and exchanged ideas, I would also like to thank Daniele Braga, Alessandro Campi, Marco Colombetti, Carlo Alberto Furia, Stefano Paraboschi, Alessandro Raffio, Damiano Salvi, and Paola Spoletini.

I am also indebted to Hendrik Decker, with whom I spent a week of very intensive and productive work at the Instituto Tecnológico de Informática in Valencia. The elegance and precision of his writings have been a model for me during these months.

Roskilde University was also a precious source of information and learning. I particularly wish to thank Torben Braüner for his lectures on modal and hybrid logics, John Gallagher for introducing me to partial evaluation, and Jørgen Villadsen for sharing his interest in paraconsistent logics and higher-order logics.

I would also like to thank the members of the Program Committee of the First International Workshop on Logical Aspects and Applications of Integrity Constraints (LAAIC'05), who agreed to provide their expertise for the success of an event that is very close to my own research: Marcelo Arenas, Andrea Calì, Stefano Ceri, Henning Christiansen, Hendrik Decker, Parke Godfrey, Mohand-Said Hacid, Maurizio Lenzerini, Rainer Manthey, Rosa Meo, and Jack Minker.

Special thanks are due to Amos Scisci, who, even when submerged in his activities as a man of letters, found the time to provide me with useful comments on some of my manuscripts. I am very grateful to Anders Winther, who helped me with the Danish translation of the abstract.

And of course to Céline, who was the first one to believe in my project.

Dansk resumé

Kontrol af en databases integritet, forstået som verifikation af betingelser for korrekthed og "velformethed" af data, understøttes kun i ringe omfang af den databaseteknologi, som anvendes i dag. Typisk må en database forventes at overholde givne semantiske betingelser (integritetsbegrænsningerne) og at opretholde disse, hver gang databasen opdateres. Allerede ved fastsættelsen af SQL2-standarden i 1992, som er en anerkendt standard for definition af og interaktion med relationelle databaser, har sproget SQL indeholdt såkaldte assertions, som gør det muligt at beskrive vilkårligt komplekse "business rules" som rækker ud over de prædefinerede typer af begrænsninger så som primær- og fremmednøgler. Generelle integritetsbegrænsninger er desværre ikke særligt anvendelige i kommercielle databasesystemer. Faktisk er det ikke ualmindeligt at producenterne opfordrer til at man ikke benytter dem, da teknologien ikke understøtter en inkrementel evaluering. Set i forhold til størrelsen af typiske datasamlinger af i dag, og den hyppighed med hvilken de opdateres, er en ikke-inkrementel tilgang i de fleste tilfælde ikke brugbar i praksis, selv for begrænsninger som "kun" er lineære i forhold til størrelsen af databasen.

I de fleste tilfælde må databasedesigneren i samarbejde med applikationsprogrammøren håndtere integriteten ved at håndkode programstumper som indlejres i applikationsprogrammet eller databasen (f.eks. som såkaldte triggers). Denne praksis er forbundet med nogle problemer i og med at vedligeholdelse er besværlig og medfører risiko for programmeringsfejl: selv små ændringer i databasens skema kan kræve subtile ændringer af disse programmer. Der er således et behov for at databasesystemerne udvides med faciliteter til, *automatisk* og *inkrementelt*, at kunne verificere at ingen opdatering får lov at ødelægge integriteten. Med udgangspunkt i denne problemstilling har vi udviklet en metode til automatisk at producere betingelser rettet mod inkrementel kontrol som garanterer at integriteten bevares. En procedure for såkaldt *simplifikation* tager som input et sæt af integritetsbegrænsninger plus et mønster for mulige opdateringer, og den producerer afledte begrænsninger, som er optimerede med hensyn til inkrementel evaluering baseret på en hypotese om at databasen er konsistent fra starten. Det skal fremhæves, at den foreslåede teknik muliggør konstruktion af inkrementelle betingelser på et tidspunkt, hvor databasen designes, dvs. før den sættes i drift, således at potentielt tidskrævende optimeringer ikke belaster når databasen er i drift. Desuden kan disse betingelser kontrolleres *før* en påtænkt opdatering udføres, så databases aldrig, end ikke midlertidigt, mister sin integritet; således kan besværlige reetableringer og såkaldte rollbacks undgås.

Det at foretage denne simplifikation så godt som muligt, er uomgængeligt forbundet med en funktion som giver en tilnærmet mål af omkostningen for efterfølgende at evaluere de simplificerede betingelser i konkrete tilstande af databasen. Det er naturligt at karakterisere en simplifikation som *optimal*, hvis den minimerer værdien af denne funktion. Vi er i stand til at vise, med enhver rimelig omkostningsfunktion, at der ikke kan findes simplifikationsprocedurer som altid producerer optimale resultater. Dette generelle resultat forhindrer dog ikke at vi kan udpege vigtige og relevante områder, hvor optimale resultater kan garanteres. I de tilfælde, hvor en ikke-optimal simplifikation kan medføre et tab i forhold til en ideel effektivitet, er det dog stadig tale om en essentiel forbedring i forhold til en ikke-inkrementel evaluering.

Endelig undersøger vi anvendelsen af simplifikation til en række andre kontekster så som rekursive databaser, samtidighed i databaser, systemer til dataintegration og baser

af XML-dokumenter, og der foretages målinger af køretider til vurdering af den foreslåede model til simplifikation.

Contents

Chapter 1

Introduction

Semantic information in databases is conventionally represented under the form of integrity constraints. Integrity constraints are properties, typically depending on the nature of the application domain, that must always be satisfied for the data to be considered *consistent*. Besides simple forms of predefined constraints such as primary and foreign keys, real-world applications may involve nontrivial integrity requirements that capture complex data dependencies and "business logic". The need for advanced integrity verification tools is testified by the introduction of several standard constructs for integrity support in the SQL language, such as *check* constraints and *assertions*. In spite of a long recognition of the importance of such practices, which are part of the SQL standard since 1992, today's database management systems are hardly capable of efficiently handling other than predefined constraints.

Maintaining compliance of data with respect to integrity constraints is an essential requirement, since, if some data lack integrity, then answers to queries cannot be trusted. Furthermore, once semantic properties of the data are known to hold, they can be exploited to improve query evaluation performance by means of so-called semantic query optimization. Databases, however, usually contain very large collections of data that quickly evolve over time; this makes unabridged checking at each update too time consuming a task to be feasible. Efficiency is indeed a crucial issue, because the complexity of a complete integrity check is often linear (or worse) with respect to the size of some database table, which is prohibitive in any nontrivial case. In this regard, DBMSs need to be extended with the ability to automatically verify, in an *optimized* way, that database updates do not introduce any violation of integrity.

To date, the common practice in database applications is still based on *ad hoc* techniques. The two main approaches are triggers, at the database level, and hand-coding of tests, at the application level. By their procedural nature, both methods have major disadvantages, as they are prone to errors, require advanced programming skills and have little flexibility with respect to changes in the schema of the database. This motivates the need for automated simplification methods.

In response to this, the database as well as the logic programming and artificial intelligence communities have developed a number of techniques for optimization of integrity checking. Main approaches to efficient integrity checking that have been proposed since

the early eighties include extensions of the SLD(NF) proof procedure, partial evaluation, update propagation, incremental view maintenance and several others. The way we pursue here is the so-called *simplification* of integrity constraints — a principle that has been recognized for more than two decades, dating back to at least [137, 24], and then elaborated by several other authors, e.g., [100, 102, 122, 143, 147, 48, 71, 114, 155, 70]. Our work is an attempt to reconcile and generalize such ideas in a systematic way that may promote practical applications with current database management technology.

This thesis presents a general characterization of the simplification problem.

On one hand, simplification means to generate a set of integrity constraints whose satisfaction implies the satisfaction of the original constraints in the updated state. The input of the procedure is a set of integrity constraints to be maintained on the database as well as an update pattern describing a kind of updates that the database can receive; the produced output is the set of simplified integrity constraints that should be checked upon reception of an update matching the given pattern. We find it important that a proposed simplification algorithm can work on parametric update patterns, not only specific updates. This means that such patterns can be simplified at design time, when only the schema exists and not yet any database state. At runtime the simplified integrity constraints can be instantiated with respect to the specific updates and tested in the actual state.

On the other hand, the main interest of the simplification process is to obtain a set of constraints that are as easy to evaluate as possible. In this sense, simplification proper is only feasible by assuming that the database conforms to the integrity constraints in the state prior to the update. In this respect, we identify as "ideal" a simplification procedure that outputs a set of integrity constraints that is minimal with respect to an ordering that represents an approximation of the cost of evaluating the constraints. Although there is no ultimate criterion that, independently of the actual database state, perfectly measures the evaluation effort, natural requirements can be imposed that should be met by any sensible ordering — in particular, that "nothing to check" is the best possible simplification one can hope for. With this assumption, it can be proved that ideal simplification is equivalent to decidability of query containment, which is known not to hold in general (query containment is not decidable, e.g., already for pure DATALOG without negation). In fact, ideal simplification is possible in a class of databases if and only if query containment is decidable in that class.

In spite of this limitation, it can be argued that simplification procedures that are "almost ideal" can still be of practical use and certainly improve upon non-optimized integrity checking.

A number of earlier approaches to simplification produce constraints that need to be checked in the updated database. This is not completely satisfactory because, in most cases, it is possible to decide, in the current state, whether a proposed update will introduce inconsistency (i.e., if it were executed). In this way, illegal database states are completely avoided and therefore expensive rollback operations as well as repair actions become unnecessary.

Another important aspect is the expressiveness of the language at hand. We present a framework that can handle a very general class of updates specified with a language that includes additions, deletions and changes as well as transactions and any kind of bulk operation expressible with a rule or query. The different steps of the simplification

procedure are transformations that operate on pieces of database programs formulated in DATALOG$^\neg$, the extension of DATALOG with negation. The choice of this logical language instead of the more commonly used SQL is based on its syntactic simplicity and clarity that make it particularly well suited for the application of proof techniques. Integrity constraints are written as denial clauses, that roughly correspond to SQL queries for which an empty answer indicates consistency.

Several challenging problems are posed in a number of different contexts. The presented transformations almost directly apply to integrity checking for data integration systems, where a mediator provides a unified view over multiple, autonomous data sources. In concurrent database systems, simplification techniques need to be strengthened with locks or similar constructs, because the guarantee of consistency of a transaction may be affected by another, interleaved transaction. More in line with today's focus in database research, integrity requirements need to be enforced in semi-structured contexts. The growing importance of XML, the *de facto* standard of semi-structured data, together with the high availability of online content and the increasing need for data quality in these settings suggests that simplification techniques should be adapted to these cases.

The availability of methods for efficient integrity constraint checking constitutes an important step forward to improving performance and reliability of database systems and applications. The integration of such techniques with commercial DBMSs raises interesting implementation and technology-related issues that, however, will not be the core of this thesis.

1.1 Goals and main results

The problem of efficiently checking integrity in relational and deductive databases has been addressed by a number of heterogeneous techniques. A first objective of this work is the development of a uniform tool of sufficient generality that covers the domains of applicability of previous solutions.

- **Goal 1:** To develop a general framework for the symbolic simplification of integrity constraints enabling incremental integrity checking.

In order to improve the efficiency of the whole integrity checking process, the output simplifications must be minimized with respect to a cost function expressing (an approximation of) the effort needed to actually evaluate them.

- **Goal 2:** To identify possible cost models related to the efficiency of integrity checking and to study feasibility of algorithms that produce optima with respect to those cost models.

In order to achieve the above goals, notions from Hoare's logic, such as weakest preconditions, are extended to the context of deductive databases and applied transformation techniques based on proof procedures, namely resolution, subsumption and factoring. The following main results were obtained.

- We introduce a framework for the simplification of integrity constraints that has the following features:

  - The procedure on which it is based produces a set of integrity constraints that are a *necessary and sufficient condition* of consistency of the database in the updated state.
  - It uses a compiled approach: the simplified integrity constraints are derived at design time. No run time needs to be spent on optimization.
  - Consistency of the updated state is tested *before* the update is executed. Only consistency-preserving transactions will eventually be given to the database and no illegal state will ever be reached.
  - The update language at hand includes tuple additions, deletions and changes; transactions and transaction patterns are also supported. In particular, any set of tuples that can be expressed as the result of a query can be added, deleted or modified.
  - It supports integrity constraints referring to special forms of recursive rules (a generalization of linear right- or left-recursion).
  - It includes special treatment for aggregates.

- Several criteria for measuring the quality of a simplification are proposed and studied; consequently, ideality of simplification is defined. It is shown that for any acceptable criterion, decidability of query containment and ideality of simplification are equivalent.

- Practically relevant classes are shown for which the procedure is guaranteed to return an ideal simplification.

- On top of the simplification procedure, a schedule construction policy based on transaction modification is developed that ensures serializability as well as consistency. This is achieved without using the simplifying assumption, common in the treatment of concurrent database systems, that serial schedules always produce states that are consistent with the integrity constraints.

1.2 Thesis organization

The thesis is organized as follows. Chapter 2 presents an overall perspective over the problem of integrity control and provides motivation for this work. In particular, after introducing the basic notions and notation used throughout the thesis, we describe the framework of deductive databases and present relevant related problems, including the important notion of query containment.

Chapter 3 specifies the simplification problem and defines the core components of a procedure for the simplification of integrity constraints. After identifying a subclass of DATALOG$^\neg$ as the initial context of study, the chapter presents a series of syntactic transformations that comply with certain syntactic and semantic requirements and that are used to compose the simplification procedure. Criteria to measure the quality of the procedure's output are then defined and its ability to produce optimal results is explored. In particular, the chapter discusses the relationship between optimal simplification and

decidability of query containment. Finally, it reviews related work in the field and emphasizes the differences and improvements of the present method with respect to existing approaches.

The limitations introduced in chapter 3 are progressively relaxed in chapter 4. Firstly, the simplification procedure is extended to the context of non-recursive DATALOG$^\neg$ databases. Recursion is then dealt with and suitable adjustments of the procedure are considered that provide improved results for specific recursive patterns in rules and integrity constraints. Aggregate functions fall outside DATALOG as well as first-order logic; however, they are a tool of prime importance in terms of expressiveness of a database language. An extension of the simplification procedure to deal with such constructs as well as with arithmetic expressions is also discussed in chapter 4.

Chapter 5 provides an overview on orthogonal aspects related to simplification as well as on other contexts of applicability of the described transformations. We put integrity checking in the perspective of concurrent database systems and investigate the relationships between simplified integrity constraints and efficient evaluation in this setting. Integrity verification can also be regarded from the point of view of a "global" logical database (the mediator) in a data integration system consisting of several local data sources. Since integrity constraints do not necessarily hold in the mediator even though they do locally at the sources, this chapter addresses this problem by adapting the methods that were applied to a single database. A structurally different context is that of databases containing hierarchically ordered data, as is the case for collections of XML documents. The chapter discusses an approach to efficient integrity checking in this setting, based on mappings of the XML model to a (flat) relational model.

An overall assessment from an experimental viewpoint of the results presented in this thesis is given in chapter 6. We discuss implementation issues and report on the efficiency of the simplification procedure based on the comparison with previous techniques developed by other researchers.

1.2.1 Origin of the chapters

An earlier version of parts of the material contained in chapter 3 has appeared in [58]. Chapter 4 incorporates results taken from [125] and [127]. Chapter 5 includes and revisits material that has been published in [128] and [59]. The experiments on recursive databases described in chapter 6 were also published in [127].

Papers [58, 59, 128, 127] were written together with Henning Christiansen. The parts regarding integrity checking for XML in chapter 5 and 6 are an adaptation of unpublished material written in collaboration with Daniele Braga and Alessandro Campi.

Chapter 2

Integrity Control in Relational and Deductive Databases

In this chapter we present the basic concepts related to deductive databases. Section 2.1 introduces the notation used throughout the thesis. The main components of a deductive database are described in section 2.2, along with an overview of its model-based semantics and query and update languages. We refer to standard texts in logic, logic programming and databases such as [163, 45, 139] for further discussion on foundational aspects of deductive databases. We then take a closer look at integrity constraints. The meaning of integrity constraints and different kinds thereof are discussed in section 2.3, together with different languages for the specification of integrity constraints. Finally, strategies for integrity control are discussed in section 2.4.

2.1 Preliminaries

We base our discourse throughout this thesis on the framework of deductive databases and use the syntax of the DATALOG language [163] as a basis to formulate the main concepts.

This choice is motivated by different reasons. First of all, DATALOG is recognized as a standard logical language that has evolved from Prolog (the most popular language for PROgramming in LOGic) into a paradigm designed for use as a database language. Among the distinctive features of DATALOG with respect to Prolog we mention:

- set-at-a-time processing (as opposed to Prolog's tuple-at-a-time), which is more in line with the fact that queries over a database should return a set of tuples, rather than individual tuples.
- declarativity: database languages are nonprocedural, i.e., the execution of queries does not depend on the order of retrieval of tuples. The procedure for accessing data is left as a system task and is not specified at the language level.
- absence of function symbols, which are typically used to model complex data structures and objects, and thus may be of little relevance in the context of (flat) deductive/relational databases.

These aspects, together with DATALOG's syntactic simplicity, make it a suitable choice as a database language. Our preference is also determined by the direct applicability of proof techniques and transformations to DATALOG rules, which would be much more problematic with SQL, although since the standard SQL:1999, SQL and DATALOG have a very similar expressive power[1]. A further advantage of DATALOG with respect to other database languages is that it has a well understood declarative semantics. This makes it easier to reason about queries and answers.

2.1.1 Formulas

We assume an alphabet including infinite sets of symbols for *predicates*, *constants*, and *variables*. As a notational convention, we generally use lowercase letters to denote predicates (p, q, ...) and constants (a, b, ...) and uppercase letters (X, Y, ...) to denote variables.

Each predicate has an associated nonnegative *arity* and the notation p/n indicates that predicate p has arity n. A *term* is either a variable or a constant[2]; conventionally, terms are also denoted by lowercase letters (t, s, ...) and sequences of terms are indicated by vector notation, e.g., $\vec{t}$.

Besides terms and predicates, the alphabet includes:

- Logical connectives ($\neg, \wedge, \vee, \leftarrow$), including the 0-ary connectives *true* and *false*.
- Quantifiers ($\forall, \exists$).
- Parentheses.
- Commas.

Definition 2.1.1 (Formula) *The set $\mathcal{F}$ of* well-formed formulas *is the smallest set such that:*

- *$true \in \mathcal{F}$ and $false \in \mathcal{F}$;*
- *if p/n is a predicate and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n) \in \mathcal{F}$;*
- *if both $F \in \mathcal{F}$ and $G \in \mathcal{F}$ then $(\neg F) \in \mathcal{F}$, $(F \wedge G) \in \mathcal{F}$, $(F \vee G) \in \mathcal{F}$, $(F \leftarrow G) \in \mathcal{F}$;*
- *if $F \in \mathcal{F}$ and X is a variable then $(\forall X F) \in \mathcal{F}$ and $(\exists X F) \in \mathcal{F}$.*

In $(\forall X F)$ and in $(\exists X F)$, the formula F is called the *scope* of the quantifier; any occurrence of X in F is said to be *bound*; it is bound by the quantifier of smallest scope that causes it to be bound; a variable which is not bound in F is said to occur *free* in F. A formula in which all variables are bound is said to be *closed*. Whenever we have

[1] In particular, the extension of DATALOG with negation, known as DATALOG$^\neg$, allows the expression of *multi-linear* recursive rules, which are not (yet) allowed in the SQL standard; on the other hand, SQL includes additional concepts, such as NULL values, aggregates and arithmetic operators and has an underlying bag semantics.

[2] The described language, as customary for databases, does not contain any (non-nullary) function symbols.

a sequence of variables $\vec{X} = \langle X_1, \ldots, X_n \rangle$, the notation $\forall \vec{X} F$ is taken as an abbreviation for $\forall X_1(\forall X_2(\cdots(\forall X_n F)\cdots))$; similarly for $\exists \vec{X}$. For convenience, in the following, free variables are indicated in boldface (**a**, **b**, ...) and referred to as *parameters*; the other variables, unless explicitly quantified existentially, are assumed to be universally quantified. An expression containing parameters is called *parametric*.

Formulas of the form $p(t_1, \ldots, t_n)$, where p is a predicate of arity n and the t_i's are terms, are called *atoms*. An atom preceded by a $\neg$ symbol is a *negated atom*; a *literal* is either an atom or a negated atom. In the following, we will only consider formulas that are well-formed and will omit parentheses whenever possible by assuming that the connectives have the following binding-order: $\neg$ (strongest), $\wedge$, $\vee$, $\leftarrow$ (weakest). For example, the formula $p \wedge q \vee r$ is read as $(p \wedge q) \vee r$.

Predicates are divided into three pairwise disjoint sets: *intensional*, *extensional*, and *built-in* predicates. Intensional and extensional predicates are collectively called *database predicates*; atoms and literals are classified similarly according to their predicate symbol. There is one built-in binary predicate for term equality ($=$), written using infix notation; $t_1 \neq t_2$ is a shorthand for $\neg(t_1 = t_2)$ for any two terms t_1, t_2.

2.1.2 Substitutions

A *substitution* is a mapping from bound variables to terms. A substitution σ is also written as $\{\vec{X}/\vec{t}\}$ to indicate that the variables in $\vec{X}$ are orderly mapped to the terms in $\vec{t}$; the notation $\mathrm{dom}(\sigma)$ refers to the set of variables in $\vec{X}$. Whenever E is a term (resp. formula) and σ is a substitution $\{\vec{X}/\vec{t}\}$, the notation $E\sigma$ denotes the term (resp. formula) that arises from E when each occurrence of a variable in $\vec{X}$ is simultaneously replaced by the corresponding term in $\vec{t}$; $E\sigma$ is called an *instance* of E. A formula or term which contains no variables is called *ground*. A substitution $\{\vec{X}/\vec{Y}\}$ is called a *renaming* iff $\vec{Y}$ is a permutation of $\vec{X}$. Formulas F, G are *variants* of one another if $F = G\rho$ for some renaming ρ. A substitution σ is said to be more general than a substitution θ iff there exists a substitution η such that $\theta = \sigma\eta$. A *unifier* of terms (resp. formulas) $t_1, \ldots, t_n$ is a substitution σ such that $t_1\sigma = \cdots = t_n\sigma$; σ is a *most general unifier* (mgu) of $t_1, \ldots, t_n$ if it is more general than any other unifier of these terms (resp. formulas).

Formulas that are equal modulo renaming or differ for orders of operands of commutative and associative connectives are considered identical; for any formula F, $\neg\neg F$ and F are also considered identical.

2.1.3 Clauses

A distinctive feature of deductive database is the ability to express implicit information in terms of deductive units called *clauses*.

A clause is usually defined as a disjunction of literals $L_1 \vee \cdots \vee L_n$; as mentioned, all the variables in a clause, unless indicated as parameters, are implicitly universally quantified at the outmost level. Conventionally, clauses defined in this way can be represented as a set of literals $\{L_1, \ldots, L_n\}$. Clauses with at most one positive literal are called *Horn clauses*; a clause with no literals is called *empty clause*.

In the context of deductive databases, however, it is more common to express clauses in an equivalent implicational form (using $\leftarrow$) that represents inferential knowledge of the

type "if some premises hold, then a given consequence also holds".

Definition 2.1.2 *A* clause *is a formula of the form*

$$A \leftarrow L_1 \wedge \cdots \wedge L_n$$

where A is an atom and $L_1, \ldots, L_n$ are literals; A is called the head *and $L_1 \wedge \cdots \wedge L_n$ the* body *of the clause and if $L_i = \neg A_i$ for some atom A_i, A_i is said to occur* negatively*. The head is optional and when it is omitted the clause is called a* denial*.*

A *rule* is a clause whose head is intensional, and a *fact* is a clause whose head is extensional and ground and whose body is empty (understood as *true*). Two clauses are said to be *standardized apart* if they have no bound variable in common.

Denials can be read as clauses in which the omitted head represents the 0-ary connective *false*. The intuition behind a denial is that its body indicates a condition that must not hold. The empty clause is a denial with an empty body, which is conventionally indicated with the 0-ary connective *false*. For convenience, in the transformations introduced in chapter 3, denials that are recognized to be tautological will be replaced by the 0-ary connective *true*, although, strictly speaking, this is not a denial.

The presence of negation or built-in predicates (which in most cases correspond to infinite relations) may compromise the so-called *domain independence*. Consider, e.g., the rule $q(X) \leftarrow \neg p(X)$; this indicates that q contains *all* values that are not in p. The application of such a rule would, thus, require a complete search on the domain of values that may apply to p, which in most cases is extremely inefficient. The common measure used to avoid such behavior is to introduce a syntactic property that forces clauses to be range restricted[3], as defined below.

Definition 2.1.3 (Range restriction) *A bound variable in a clause is* range bound *if it appears in a positive database literal in the body. A clause is* range restricted *if all bound variables in it are range bound.*

In the remainder of the thesis we will only consider range restricted clauses.

To simplify the notation, we introduce the *pred* operator. For a literal L, $pred(L)$ refers to the predicate of L. For a clause C with non-empty head A, $pred(C)$ refers to $pred(A)$; for a set of non-headless clauses $\mathcal{C}$, $pred(\mathcal{C})$ refers to the set of predicates $\{pred(C) \mid C \in \mathcal{C}\}$.

2.2 Deductive databases

Deductive databases are characterized by three components: facts, rules and integrity constraints. An *integrity constraint* can, in general, be any (closed) formula. In the context of deductive databases, however, it is customary to express integrity constraints in some canonical form; we adopt here the denial form, that gives a clear indication of what must not occur in the database. We return on the expressiveness of integrity constraints again in section 4.1.

[3]Sometimes also indicated as *safe* or *allowed* clauses.

Definition 2.2.1 (Schema, database) *A database* schema *S is a pair $\langle IDB, IC\rangle$, where IDB (the* intensional database*) is a finite set of range restricted rules, and IC (the* constraint theory*) is a finite set of integrity constraints. The predicates in pred(IDB) are said to be* defined by *IDB and it is assumed that any intensional predicate occurring in S is defined by IDB.*

A database state *(or, simply, database) D on schema S is a pair $\langle IDB, EDB\rangle$, where EDB (the* extensional database*) is a finite set of facts; D is said to be* based on *IDB.*

When the IDB is understood, the database may be identified with EDB and the schema with IC.

Definition 2.2.2 (Language) *Let $\mathcal{S}$ be the set of all schemata. Any set $\mathcal{L} \subseteq \mathcal{S}$ is called a* database language.

We observe here that, from the point of view of formulas, a set of clauses (rules, facts or integrity constraints) is logically intended as the conjunction of its elements; under this assumption, an empty set of clauses will interchangeably be indicated as $\emptyset$ or as *true*. Parameters are not expected to be part of any actual database or integrity constraint, but the transformations that will be introduced in chapter 3 may generate parametric versions of these categories. These transformations may introduce new auxiliary intensional predicates in IC. This means that such new IC components are only meaningful as part of a new schema with an extended IDB part. In order to state that such schema modifications do not interfere with existing definitions, we define the following notion of compatible schemata.

Definition 2.2.3 *Two schemata $S_1 = \langle IDB_1, IC_1\rangle$ and $S_2 = \langle IDB_2, IC_2\rangle$ are* compatible *whenever pred($IDB_1 \setminus IDB_2$) $\cap$ pred(IDB_2) $= \emptyset$ and pred($IDB_2 \setminus IDB_1$) $\cap$ pred(IDB_1) $= \emptyset$. They are* disjoint *if pred(IDB_1) $\cap$ pred(IDB_2) $= \emptyset$.*

Trivially, disjoint schemata are compatible. As a convention, we write $S_1 \cup S_2$ as a shorthand for $\langle IDB_1 \cup IDB_2, IC_1 \cup IC_2\rangle$ whenever S_1 and S_2 are compatible.

2.2.1 Semantics

We now introduce the semantics of deductive databases based on Herbrand models. For further details we refer to [121]. When reasoning about the models of a database, it is customary to restrict the alphabet to exactly those symbols that occur in the database. It is also assumed that there is at least one constant, since, otherwise, the domain represented by the database would be empty.

Definition 2.2.4 (Herbrand base) *The* Herbrand universe *$\mathcal{U}_D$ of a database D is the set of constants in D (plus one constant c, if D contains no constants). The* Herbrand base *$\mathcal{H}_D$ of D is the set of all ground atoms that can be constructed from the predicate symbols in D and constants in $\mathcal{U}_D$.*

We note that, since we are in a function-free setting, Herbrand bases will always be finite.

Definition 2.2.5 (Herbrand interpretation) *Let D be a database and $\mathcal{H}_D$ its Herbrand base. Any subset I of $\mathcal{H}_D$ is a (Herbrand)* interpretation *of D.*

Definition 2.2.6 *Let I be an interpretation for a database D. A closed formula F is true in I, written $\models_I F$, according to the following inductive definition. We write $\not\models_I F$ as a shorthand to indicate that it is not the case that $\models_I F$.*

- *For any I, $\models_I$ true and $\not\models_I$ false.*
- *For a ground database atom A, $\models_I A$ iff $A \in I$.*
- *For any constant c, $\models_I c = c$.*
- *For any closed formulas F, F_1, F_2,*

  $\models_I \neg F$ *iff* $\not\models_I F$;
  $\models_I F_1 \wedge F_2$ *iff both* $\models_I F_1$ *and* $\models_I F_2$;
  $\models_I F_1 \vee F_2$ *iff* $\models_I F_1$ *or* $\models_I F_2$;
  $\models_I F_1 \leftarrow F_2$ *iff* $\models_I F_1$ *or* $\not\models_I F_2$;
  $\models_I \forall X F$ *iff, for all constant* $c \in \mathcal{U}_D$, $\models_I F\{X/c\}$.
  $\models_I \exists X F$ *iff, for some constant* $c \in \mathcal{U}_D$, $\models_I F\{X/c\}$.

Definition 2.2.7 (Herbrand model) *An interpretation I of a database $D = \langle IDB, EDB\rangle$ is a (Herbrand)* model *of D if $\models_I C$ for every clause $C \in IDB \cup EDB$. A model of D is* minimal *if none of its subsets is a model of D.*

The notion of Herbrand model can be defined in a similar way for any arbitrary closed formula. In particular, we say that a closed formula F is a *logical consequence* of a closed formula G, written $G \models F$, iff F is true in every model of G; $F \equiv G$ means that both $F \models G$ and $G \models F$ hold.

Generally, a database does not necessarily have a unique minimal model. However, there are database classes for which the existence of a unique *intended* minimal Herbrand model is guaranteed. In order to identify such classes we introduce the notions of dependency graph and predicate dependencies.

Definition 2.2.8 (Dependency graph) *Given a set of clauses $\mathcal{C}$, the* dependency graph *$\mathcal{D}_\mathcal{C}$ of $\mathcal{C}$ is a directed graph with labelled arcs. Its nodes are the predicates occurring in $\mathcal{C}$. For every clause in $\mathcal{C}$ with predicate p in the head and predicate q in a positive (resp. negative) literal in the body there is an arc $q \hookrightarrow_\mathcal{C} p$ with label "+" (resp. "−"). No other arc is in $\mathcal{D}_\mathcal{C}$.*

The notation $q \dashrightarrow_\mathcal{C} p$ indicates that there is a nonempty path from q to p in the dependency graph $\mathcal{D}_\mathcal{C}$ of $\mathcal{C}$, and in this case we say that p depends on *q (in $\mathcal{C}$); if there is at least one "−" label in some path from q to p, then p* depends negatively on *q, indicated $q \overset{-}{\dashrightarrow}_\mathcal{C} p$. For a formula F, we say that F* depends (negatively) on *q if F contains a predicate that depends (negatively) on q.*

Recursion is one of the distinctive features of deductive databases that is captured by the dependency graph. In fact, databases can be classified according to the dependency graph of their *IDB*.

Definition 2.2.9 (Recursion) *A predicate p is* recursive *in a set of clauses $\mathcal{C}$ iff p depends on p in $\mathcal{C}$. Two predicates p and q are* mutually recursive *in $\mathcal{C}$ whenever both p depends on q and q on p. A rule is recursive whenever there exists a literal in the rule body whose predicate is mutually recursive with the head predicate; if there is only one such literal, the rule is* linear*; if there are exactly two such literals, the rule is* bilinear*. A set of rules is linear if every recursive rule in it is linear; a set of rules is bilinear if every recursive rule in it is linear or bilinear and there is at least one bilinear rule.*

Definition 2.2.10 *Let $D = \langle IDB, EDB \rangle$ be a database. Then IDB and D are called*

- positive *if there are no predicates p, q such that $q \overset{-}{\dashrightarrow}_{IDB} p$.*
- semi-positive *if there are no predicates p, q such that $q \overset{-}{\dashrightarrow}_{IDB} p$ and $q \in pred(IDB)$.*
- hierarchical *if there is no predicate p such that $p \dashrightarrow_{IDB} p$.*
- stratified *if there is no predicate p such that $p \overset{-}{\dashrightarrow}_{IDB} p$.*

A schema $S = \langle IDB, IC \rangle$ is positive if IDB is positive and no negation symbol occurs in IC. S is semi-positive if IDB is semi-positive and negation symbols in IC only occur immediately before extensional predicates. S is hierarchical (stratified) if IDB is hierarchical (stratified).

Problems arise when trying to soundly derive negative information from a database. A standard strategy for deriving negative knowledge from a positive database is to assume that one can draw negative conclusions based on the lack of positive information, known as *closed world assumption* (CWA) [145]. Given a set of clauses C of a positive database, this is expressed as the meta-rule "if A cannot be proved from C, then $\neg A$ is assumed to hold", provided that A is a ground literal. This definition is stated proof-theoretically; however, in case of a sound and complete proof system, it can be replaced by "if A is not a logical consequence of C, then $\neg A$ is assumed to hold". A first problem with this approach is that non-provability is in general undecidable, even for positive databases, thus it is not possible to determine whether the rule is applicable or not. However, a weaker version of the CWA, known as *negation as finite failure* (NaF), makes this rule decidable. Furthermore, the inferences derived via the CWA may not be logical consequences of the database: for example, the Herbrand base (in which all ground atoms are true) is always a model, albeit not minimal, of a positive database D, and thus no negative literal is a logical consequence of D.

The solution initially proposed by Clark [60] was to discard "uninteresting" models from consideration. Intuitively, this is done by completing the inferential knowledge contained in a database in clausal form by adding the only-if part of the rules. Technically, the clauses in $IDB \cup EDB$ are transformed according to the following steps:

- Transform each clause of the form $p(\vec{t}) \leftarrow B$ into $p(\vec{X}) \leftarrow \exists \vec{Y}(\vec{X} = \vec{t} \wedge B)$, where $\vec{Y}$ are the variables in the original clause and $\vec{X}$ are new variables.
- The formula $\forall \vec{X}(p(\vec{X}) \leftrightarrow F_1 \vee \cdots \vee F_n)$ replaces all the formulas $p(\vec{X}) \leftarrow F_1, \ldots,$ $p(\vec{X}) \leftarrow F_n$ obtained in the previous step[4].

[4] $A \leftrightarrow B$ is a shorthand for $(A \leftarrow B) \wedge (B \leftarrow A)$

- The formula $\forall\vec{X}(q(\vec{X}) \leftrightarrow \mathit{false})$ is added for each database predicate symbol not occurring in the head of a clause in the original database.
- Finally, the following *free equality axioms*, adapted for our function-free setting, are added to define the equalities introduced in the first step (see, e.g., [139] for a formal account including axioms that are needed in the presence of function symbols).
  - $\forall X(X = X)$
  - $\forall X, Y(X = Y \leftarrow Y = X)$
  - $\forall X, Y, Z(X = Z \leftarrow X = Y \wedge Y = Z)$
  - $\forall\vec{X}, \vec{Y}(p(\vec{X}) = p(\vec{Y}) \leftarrow \vec{X} = \vec{Y})$ for any predicate p.
  - $c_1 \neq c_2$ for any two different constants c_1, c_2.

The resulting formula, called the *completion* of a database $D = \langle IDB, EDB\rangle$, is indicated $comp(D)$.

Example 2.2.11 *Consider the database*

$$D = \langle\{p(X,Y) \leftarrow e(X,Y), p(X,Y) \leftarrow e(X,Z) \wedge p(Z,Y)\}, \emptyset\rangle.$$

Its completion $comp(D)$ is

$$\begin{aligned}\forall X_1, Y_1(p(X_1,Y_1) \leftrightarrow \quad & X_1 = X \wedge Y_1 = Y \wedge e(X,Y) \vee \\ & \exists Z(X_1 = X \wedge Y_1 = Y \wedge e(X,Z) \wedge p(Z,Y))).\end{aligned}$$

Completion provides a logical basis for NaF, but has problems of its own once we allow negations to occur in clauses. For example the clause $p \leftarrow \neg p$ is completed as $p \leftrightarrow \neg p$, which is false in all interpretations.

Such situations are completely avoided in the class of stratified databases, which is the most general of the four classes of definition 2.2.10 and that prevents mixing negation and recursion. Stratified databases admit a unique "intended" minimal Herbrand model, called the *standard model*. In order to show the construction of the standard model of a stratified database [8], we need to introduce the *immediate consequence operator*.

Definition 2.2.12 *Let I be a Herbrand interpretation for a database $D = \langle IDB, EDB\rangle$ and let $gr(D)$ indicate the set of ground instances of clauses in EDB∪IDB. The* immediate consequence operator *T_D is defined as follows.*

$$T_D(I) = \{A | A \leftarrow B \in gr(D) \text{ and } \models_I B\}.$$

The notation $T_D^\omega(I)$ denotes the limit of the sequence

$$T_D^0(I) = I, \ldots, T_D^{n+1}(I) = T_D(T_D^n(I)) \cup T_D^n(I).$$

A stratified database can be partitioned in layers, called *strata*, according to the following definition. To simplify notation, for two databases $D_1 = \langle IDB_1, EDB_1\rangle$ and $D_2 = \langle IDB_2, EDB_2\rangle$, we indicate with $D_1 \cup D_2$ the database $\langle IDB_1 \cup IDB_2, EDB_1 \cup EDB_2\rangle$.

Definition 2.2.13 *A database D has a* stratification $D_1 \cup \cdots \cup D_n = D$ *if, for* $1 \leq i \leq n$, *in the body of clauses in* D_i*:*

- *only D's extensional predicates or predicates in $\cup_{j=1}^{i} D_j$ occur positively*
- *only D's extensional predicates or predicates in $\cup_{j=1}^{i-1} D_j$ occur negatively*

It turns out that a program is stratified iff it admits a stratification. We now have all the elements to compute the standard model.

Definition 2.2.14 *Let D be a stratified database and $D_1 \cup \cdots \cup D_n$ a stratification for D. Consider the sequence $M_1 = T^{\omega}_{D_1}(\emptyset), \ldots, M_n = T^{\omega}_{D_n}(M_{n-1}) = M_D$. M_D is called the* standard model *of D.*

In [8] it was shown that the standard model is a minimal Herbrand model and does not depend on the stratification. Furthermore, the standard model is *supported*.

Definition 2.2.15 *An interpretation I of a database D is said to be* supported *if, for any ground atom A, $A \in I$ implies that there is a clause $A \leftarrow B \in gr(D)$ such that $\models_I B$.*

Example 2.2.16 *Consider the sets of rules $IDB_1 = \{p \leftarrow \neg q, \quad q \leftarrow r\}$ and $IDB_2 = \{p \leftarrow \neg p\}$. Any database based on IDB_1 is stratified, whereas any database based on IDB_2 is not stratified. The standard model of the database $\langle IDB_1, \emptyset \rangle$ is $\{p\}$; another minimal, but not standard model of this database is $\{q, r\}$.*

It was also shown by Apt, Blair and Walker [8] that the completion of a stratified program always admits a model. Determining whether a database is stratified is decidable, but the problem of determining whether the completion of a database admits a model is undecidable. Furthermore, it is known that the class of possibly non-stratified databases is strictly more expressive than that of stratified databases [111]. Other approaches to the calculation of a minimal model are known in the literature and for slightly more general classes than stratified databases, e.g., the *perfect model* for *locally stratified deductive databases* [142], the *stable model* semantics [87] and the *well-founded model* semantics [165]. For a survey on these model-theoretic issues we refer to [9]. In the following, we will focus on stratified databases. We define database semantics in terms of the standard model.

The notation $D \models \phi$, where D is a (stratified) database and ϕ is a closed formula, indicates that ϕ holds in D's standard model M_D, i.e., $\models_{M_D} \phi$. For any schemata $S_1 = \langle IDB_1, IC_1 \rangle$ and $S_2 = \langle IDB_2, IC_2 \rangle$ defined on the same extensional predicates, we write $S_1 \equiv S_2$ to indicate that their constraint theories evaluate in "equivalent ways", i.e., for every extensional database EDB, $\langle IDB_1, EDB \rangle \models IC_1$ iff $\langle IDB_2, EDB \rangle \models IC_2$. According to this semantics, the negation symbol ($\neg$) used throughout this thesis indicates what is commonly referred to as *default negation*, since it is essentially defined by assuming $\neg A$ "by default", that is, in the absence of sufficient evidence to the contrary[5]. Specifically, here $\neg A$ is assumed if A is false in the standard model of the database.

[5]Default negation is sometimes written as **not** to distinguish it from logical negation. In this thesis, the $\neg$ symbol will always refer to default negation; when needed, we will use the symbol $\neg_\ell$ to indicate logical negation.

In general, we will express our definitions and operators on schemata, so that integrity constraints are always in the context of an *IDB*; however, when the *IDB* is understood, the schema may be identified with the integrity constraints and the database with the extensional database.

2.2.2 Queries and updates

We now introduce a rule-based language for the modification of the relations of a deductive database.

Definition 2.2.17 (Query) *A* query *for a database $D = \langle IDB, EDB \rangle$ is an expression of the form $\Leftarrow A$ where A is an atom such that $pred(A) \in pred(IDB)$.*

For convenience, we include queries in intensional predicates, i.e., if $\Leftarrow p(\vec{X})$ is a query for a database D then p is defined in D's *IDB*. But when no ambiguity arises, a given query may be indicated directly by means of its *defining formula* (instead of the predicate name).

Definition 2.2.18 (Defining formula) *The* disjunctive predicate definition *for an intensional predicate p in a database schema S is the set of all rules $H \leftarrow B$ is S such that $pred(H) = p$. Without loss of generality, we assume that they are expressed as follows:*

$$\begin{aligned} \{p(\vec{X}) &\leftarrow B_1, \\ &\vdots \\ p(\vec{X}) &\leftarrow B_n\}, \end{aligned}$$

where $\vec{X}$ is a sequence of distinct variables, called distinguished variables*, and the B_i's are conjunctions of literals. Variables that are not distinguished are called* non-distinguished variables*. The* defining formula *of p is*

$$B_1\rho_1 \vee ... \vee B_n\rho_n,$$

where each ρ_i is a renaming giving fresh new names to the non-distinguished variables of B_i to avoid name clashes. The disjunctive predicate definition is also written as

$$p(\vec{X}) \leftarrow B_1\rho_1 \vee ... \vee B_n\rho_n,$$

Using the terminology of relational databases, facts correspond to *tuples* and intensional predicates that are not queries are called *views*; views can occur anywhere in the body of *IDB* rules, whereas query predicates can only be used as such (i.e., not in rule bodies). To complete the mapping with the relational world, a position in a predicate corresponds to a *relational attribute*.

Definition 2.2.19 (Extension) *The* extension *of a database predicate p in a database D is defined as the set of ground tuples $\{\vec{a} \mid D \models p(\vec{a})\}$; if $\Leftarrow p(\vec{X})$ is a query, we refer also to p's extension as the* answer *to $\Leftarrow p(\vec{X})$ in D and denote it $\mathcal{A}^p_D$.*

Definition 2.2.20 (Update) *A* predicate update *for an extensional predicate p in a database D is an expression of the form $p(\vec{X}) \Leftarrow p'(\vec{X})$ where $\Leftarrow p'(\vec{X})$ is a query for D; p is said to be* affected *by the update. A* (database) update *is a set of predicate updates for distinct predicates. For a given database $D = \langle IDB, EDB \rangle$ and an update U, the* updated *database D^U is defined as $\langle IDB, EDB' \rangle$, where EDB' is as EDB but in which, for every extensional predicate p affected by a predicate update $p(\vec{X}) \Leftarrow p'(\vec{X})$ in U, the subset $\{p(\vec{t}) \mid D \models p(\vec{t})\}$ of EDB is replaced by the set $\{p(\vec{t}) \mid D \models p'(\vec{t})\}$.*

As mentioned, integrity constraints need to be specialized for update patterns rather than for specific updates. This is achieved by using parameters. In general, a parametric formula does not have a truth value but its meaning may be understood as a mapping from sequences of constants to truth values. Parameters are not part of any query or update actually given to a database, but, again, we may have parametric expressions of these categories.

Definition 2.2.21 (Parametric instance, equivalence) *A* parameter substitution *is a mapping from parameters to constants. Whenever E is an expression containing parameters, and π is a parameter substitution for all the parameters in E, the notation $E\pi$ denotes the parameter-free expression that arises from E when each occurrence of a parameter is replaced by its value specified by π; $E\pi$ is called a* parametric instance *of E. We use the notation $\{\vec{\mathbf{a}}/\vec{c}\}$ for a parameter substitution in order to indicate that parameters $\vec{\mathbf{a}}$ are orderly mapped to constants $\vec{c}$.*

The notation $A \models B$ is extended to parametric expressions with the meaning that $A\pi \models B\pi$ holds for all parameter substitutions π such that $A\pi$ and $B\pi$ are parameter-free; similarly for $A \equiv B$.

The notation $D_1 \models \phi \Leftrightarrow D_2 \models \psi$, for databases D_1 and D_2 and constraint theories ϕ and ψ, indicates that, for any parametric instance $(D_1', \phi', D_2', \psi')$ of (D_1, ϕ, D_2, ψ), $D_1' \models \phi'$ if and only if $D_2' \models \psi'$.

Parametric updates will be used in the following chapters as input to the transformations that compose the simplification framework; however, there is no sense in directly applying a parametric update to a database.

Example 2.2.22 *Consider the intensional database*

$$IDB_1 = \{p'(X) \leftarrow p(X), \quad p'(X) \leftarrow X = a\}.$$

The following update U_1 describes the addition of fact $p(a)$ (p is an extensional predicate):

$$\{p(X) \Leftarrow p'(X)\}.$$

As mentioned, for convenience, defining formulas instead of queries will be written in the body of predicate updates wherever possible. Update U_1 will, e.g., be indicated as follows:

$$\{p(X) \Leftarrow p(X) \vee X = a\}.$$

The following U_2 means "change any $r(a, X)$ into $r(b, X)$", where r is an extensional predicate.

$$\{r(X, Y) \Leftarrow (r(X, Y) \wedge X \neq a) \vee (r(a, Y) \wedge X = b)\}.$$

The following U_3 is a parametric update that looks very much like U_2, but refers to two parameters instead of constants.

$$\{r(X,Y) \Leftarrow (r(X,Y) \wedge X \neq \mathbf{a}) \vee (r(\mathbf{a},Y) \wedge X = \mathbf{b})\}.$$

Notice that if **a** *and* **b** *are instantiated to the same constant, U_3 does not perform any changes. The following U_4 exchanges the contents of extensional predicates p and q.*

$$\{p(X) \Leftarrow q(X), \quad q(X) \Leftarrow p(X)\}.$$

2.2.3 Query containment

Once a query language is available, a crucial database issue is to answer queries efficiently. In this respect, optimizing techniques arise naturally by answering a simple decision problem: given two queries $\Leftarrow p(\vec{X})$ and $\Leftarrow q(\vec{X})$, is $\mathcal{A}^p_D \subseteq \mathcal{A}^q_D$ for all database D? This problem, known as *query containment* (QC), has been studied extensively in the literature.

Definition 2.2.23 (Query containment problem) *Let $S = \langle IDB, IC \rangle$ be a database schema and $\Leftarrow p(\vec{X})$, $\Leftarrow q(\vec{X})$ two queries with identical arity. The QC problem for p and q, denoted $S : p \subseteq q$, is the problem of deciding whether $\mathcal{A}^p_D \subseteq \mathcal{A}^q_D$ for all consistent databases D with schema S. QC over a language $\mathcal{L}$ is the problem of deciding, for all schemata $S \in \mathcal{L}$ and queries $\Leftarrow p(\vec{X})$, $\Leftarrow q(\vec{X})$ in S, whether $S : p \subseteq q$.*

Example 2.2.24 *Suppose that p and q are defined in the IDB of a schema S and that it is known that $S : p(\vec{X}) \subseteq q(\vec{X})$, i.e., that query $\Leftarrow p(\vec{X})$ is contained in query $\Leftarrow q(\vec{X})$. Consider a query Q defined as $\Leftarrow p(\vec{X}) \wedge q(\vec{X})$ posed on a database with schema S. Then, for all databases with schema S, Q is equivalent to the query $\Leftarrow p(\vec{X})$, in the sense that they will always produce identical answers. However, the latter contains fewer predicate symbols and fewer variable bindings than Q, which correspond, respectively, to relations and joins in the relational setting.*

Unfortunately, QC is undecidable in the general case. Shmueli [158], relating QC to the containment problem for context-free languages, proved that QC is already undecidable for DATALOG databases without negation. This means that QC is undecidable also for larger database classes, such as that of stratified databases. Since integrity constraints can be regarded as queries whose answer must always be empty, this undecidability will have some repercussions on the realization of a "perfectly" efficient integrity checker, i.e., a tool that is able to eliminate all possible redundancies for query evaluation in all cases. We will delve into these issues in chapter 3.

2.3 Integrity constraints

Although integrity constraints are a common as well as central notion in the context of databases and information systems, there is no general consensus on the meaning of "integrity". Certainly integrity constraints contribute to the characterization of the semantics of the data, by imposing dependencies and restrictions that must always be

complied with. The very notion of data integrity is usually regarded as correctness and accuracy of the data in the database. For example, Motro [133] with his equation "Integrity = Validity + Completeness" looks at integrity from the point of view of queries, stating metaphorically that their "answers have integrity if they contain the whole truth (completeness) and nothing but the truth (validity)".

The lack of consensus is apparent once we examine the notion of constraint satisfaction. In [88], two different views are presented: satisfaction of integrity constraints *by entailment* (IC must hold in all models of the database) and *by consistency* (IC must hold in at least one model of the database, i.e., there exists a model of $EDB \cup IC \cup IDB$, where $\langle IDB, EDB \rangle$ is the database in question). These approaches coincide for positive databases without inequality, but may differ if negation is present. Furthermore, in the presence of negation, there may be several minimal models of the database. Two extreme approaches in this case are the *pure entailment approach* (IC must hold in all minimal models of the database) and the *pure consistency approach* (IC holds in at least one minimal model of the database). For stratified databases, satisfaction of IC in the standard model is the usual way of reconciling these approaches.

Definition 2.3.1 (Consistency) *A database D is* consistent[6] *with a constraint theory IC whenever $D \models IC$.*

We chose to represent integrity constraints as denials, instead of *any* clause, according to the intuition that they are not used to infer knowledge, but rather to discard models that are considered unacceptable.

2.3.1 Static vs. dynamic constraints

The kind of integrity constraints that are considered in this thesis are the so-called *static* integrity constraints, that indicate properties that must be met by the data in each database state.

A disjoint class is that of the so-called dynamic constraints, that are used to impose restrictions on the way the database states can evolve over time. A typical example of dynamic constraint is "the marital status of a person cannot change from *single* to *divorced*". This condition is not static in that it does not refer to a single state, but rather to an old and a new state at the same time. Dynamic constraints, like this one, referring to the change from a state to the successive state are called *transitional constraints*. More complicated conditions involving (possibly infinite) sets of states may be expressed by resorting to temporal logics that have constructs such as "since", "until", "before", etc. Dynamic constraints have been considered, e.g., in [53, 65].

2.3.2 Hard vs. soft constraints

Another important distinction can be made between *hard* (or *strong*) constraints and *soft* (or *deontic*) constraints. The former ones are used to model necessary requirements of the world that the database represents. For example, the fact that "every person must

[6] Note that we use here the terms "consistency" and "integrity" as synonyms, whereas for other authors [30] the word "consistency" refers to the *satisfiability* of a set of integrity constraints independently of any state, i.e., whether there exists any database state in which the integrity constraints are satisfied.

be either male or female" can be reasonably considered an intrinsic truth of the world represented, e.g., by a database containing information on the personnel of a company. Deontic constraints govern what is obligatory but not necessary of the world. For instance, the fact that "every ordered item is in the stock" is certainly a requirement for the good running of an organization but may not correspond to a necessary truth of the world; however, if the company adheres to a policy that guarantees that ordered items will always be available, then this might be considered a hard constraint.

In other words, violations of deontic constraints correspond to violations of obligations that have to be reported to the user or administrator of the database rather than inconsistencies proper. The wording "soft constraint" sometimes refers to a condition that should preferably be satisfied, but may also be violated, while with "deontic constraint" one usually emphasizes the distinction between what the database *believes* is true of the world it represents and what *is* actually true of the world. In this thesis we concentrate on hard constraints; deontic constraints have been considered, e.g., in [40] and soft constraints in [89]. Applications of so-called soft constraints seem to be possible, albeit to a lesser extent, for integrity checking and semantic query optimization (see [89] and references therein).

2.3.3 Classification of integrity constraints

Static, hard constraints can be further divided in several subclasses. In the relational setting, it is common to classify constraints according to the entities that they involve.

At the most specific level are found *tuple constraints*, that impose restrictions on the combinations of elements in a tuple. Among these, *domain constraints* indicate the possible values that an attribute of a tuple can take.

Constraints that refer to the tuples of a single relation are referred to as *intra-relation constraints* and include different kinds of data dependencies that constrain equalities or inequalities between values of attributes of a relation. Among these, *functional dependencies* [61] and *multi-valued dependencies* [78] are the most common ones. A functional dependency (FD) involves sets of attributes, say X and Y, of a relation R and is written $X \rightarrow Y$ to indicate that every pair of tuples in R that agrees on the attributes in X also agrees on the attributes in Y. While a functional dependency $X \rightarrow Y$ relates one value of X to one value of Y, a multi-valued dependency (MVD) , indicated $X \twoheadrightarrow Y$, defines a relationship in which a set of values of attribute Y is determined by a single value of X. The multi-valued dependency $X \twoheadrightarrow Y$ is said to hold for a relation R with (sets of) attributes X, Y, Z, written $R(X, Y, Z)$, if for a given set of values for attributes X, there is a set of associated values for the set of attributes Y and the Y values depend only on X values and have no dependence on the set of attributes Z. This also indicates that relation $R(X, Y, Z)$ is the join of its projections $R_1(X, Y)$ and $R_2(X, Z)$, i.e., it can be decomposed in R_1 and R_2 without loss of information. *Key dependencies* are functional dependencies that impose a *key*, i.e., a subset of attributes of a relation that uniquely identify a tuple in that relation.

Constraints involving two or more relations are known as *inter-relation constraints*. Among these are *inclusion dependencies* (also known as *referential constraints*), that involve two relations and constrain the set of values assumed by some attributes on the first relation to be a subset of the values assumed by the same attributes on the second

relation. In particular, when these attributes are a key for the second relation, the constraint is called a *foreign key constraint*.

All the constraints mentioned so far can be expressed in the setting of deductive databases as a finite set of denials (see definition 2.1.2), apart from domain constraints involving infinite domains. These last ones would be easily captured by a *typed* version of DATALOG; however, such constraints are often considered as *structural* rather than *semantic constraints*, and thus have little interest for this work.

A more complex, and yet very common kind of constraints is that of *aggregate constraints*, that involve several (possibly all) tuples of a relation simultaneously. For example, constraints on the number of occurrences of certain tuples in a relation or the average of values assumed by certain attributes belong to this category. However, aggregates are not part of standard DATALOG; we will introduce an extension of the language that takes into account these constructs in section 4.2.

More detailed classifications of integrity constraints for deductive databases have been attempted by several authors, e.g., [163, 97].

2.3.4 Constraint specification languages

Different languages for expressing integrity constraints exist. Common paradigms are, e.g., relational algebra and relational calculus; these (or proper extensions thereof) are known to have exactly the same expressive power as DATALOG$^\neg$ [3] (in particular, the algebra plus while-statements, the calculus plus fixpoint and DATALOG with negation are all equivalent). Our choice to express integrity constraints as DATALOG denials rather than relational algebraic expressions is motivated by the availability of proof tools for first-order logic that allow one to reason easily about formulas. The preference granted to DATALOG with respect to relational calculus is, however, only made for uniformity with recent works in the field (such as [70, 20]) and for the closer similarity with Prolog, that will prove a useful test platform for some of our experiments (see chapter 6).

The techniques presented in this work can also be adapted to the context of SQL. A mapping between DATALOG and SQL, although technically nontrivial, is of little theoretical interest and will not be discussed in detail in this thesis[7]. A technique for translating DATALOG into (subsets of) SQL has been described in [70].

SQL provides a series of built-in constructs for specifying integrity constraints, such as `PRIMARY KEY`, `FOREIGN KEY` and `UNIQUE`. These have obvious DATALOG counterparts and can be checked and maintained very efficiently by any relational database management system. SQL also allows one to express more general constraints, such as conditions referring to exactly one or several attributes of a relation, called respectively *column check constraints* and *table check constraints*; both are specified with the `CHECK` keyword. Since 1992, the SQL standard is equipped with the `ASSERTION` construct that allows the specification of general inter-relation constraints. These may contain any condition that can be expressed with an SQL query and are therefore very powerful. However, typically, DBMSs are not able to verify `CHECK` and `ASSERTION` constraints in an incremental way.

[7]There may be, however, both syntactic and semantic discrepancies between the two paradigms. We note that SQL admits null values, which do not have a counterpart in the model-theoretical view of databases. Furthermore, SQL is based on a multi-set semantics instead of a set semantics, that we will consider in section 4.2.

For this reason, assertions are not commonly supported by today's vendors, and, even when they are [140], their use is not encouraged.

The main purpose of this thesis is to provide techniques for efficiently and incrementally checking general integrity constraints. Such techniques are aimed at providing a realistic framework for extending the capability of relational database systems. It should be noted at this point that, in some cases, the presence of complex general constraints may be regarded as a symptom of a badly defined schema, and so schema redesign proves to be an effective practical means of reducing their complexity. However, there are several interesting application scenarios for which the specification of complex constraints is natural or even necessary; examples will be given throughout this thesis.

2.4 Integrity control

Integrity control addresses the problem of possible integrity violations upon modifications of the database state determined by updates. We now formalize the problem and specify when the control takes place and what can be done to maintain the database in a consistent state.

2.4.1 Immediate vs. deferred semantics

An update transaction (in short: update) can be imagined as a sequence of operations that modify the state of a database. In this regard, satisfaction of integrity constraints with respect to the update can be considered under two different perspectives. The *immediate* semantics requires satisfaction after each single modification of the state; the opposite approach is that of the *deferred* semantics, that requires satisfaction after the whole update has executed, but not in the intermediate states. We refer in this thesis to the deferred semantics only, as, from the point of view of integrity checking alone, the immediate semantics is a special case of deferred semantics when the update transaction consists of a single operation.

2.4.2 Constraint verification

The constraint verification problem may be formulated as follows. Given a database state D, a constraint theory Γ, such that $D \models \Gamma$, and an update U, does $D^U \models \Gamma$ hold too?

However, checking directly whether $D^U \models \Gamma$ holds may be too expensive, so a suitable reformulation of the problem is called for. A first improvement is achieved if another constraint theory Δ, informally referred to as a *post-test*, can be found such that $D^U \models \Gamma$ iff $D^U \models \Delta$ and Δ is somehow "easier" to evaluate than Γ. A characterization of the efficiency of evaluation of integrity constraints with respect to given cost models is provided in chapter 3.

The way we pursue here is slightly different, in that we prefer to check integrity in the current state D, i.e., before the execution of a potentially offensive update U. With our approach we look for a constraint theory Γ^U such that $D^U \models \Gamma$ iff $D \models \Gamma^U$ and Γ^U is easier to evaluate than Γ. In other words, condition Γ^U, called a *pre-test* of the original constraints Γ, should specialize the original Γ, as specific information coming from U is available, and avoid redundant checks by exploiting the fact that $D \models$

Γ holds. Sometimes, post-tests and pre-tests are referred to as *simplifications* of the original constraints. We observe that reasoning about the future database state D^U with a condition (Γ^U) that is tested in the present state D, complies with the deferred semantics and allows avoiding the execution of illegal updates completely.

2.4.3 Checking, maintenance, and paraconsistency

Once illegal updates are detected, it must be decided how to restore a consistent database state.

The approach described in the previous subsection, that we adopt in this thesis, is based on a complete *prevention* of inconsistencies: integrity of the updated state is checked in the state preceding the update and, whenever an illegal update is detected, it is not executed.

As mentioned, integrity checking can also be done in the state following the update. In this case, upon inconsistencies, integrity needs to be restored via corrective actions. Most typically, such action is a rollback, that simply cancels the effects of the unwanted update and usually requires costly bookkeeping in order to restore the old state.

In other circumstances, the update can be considered as an operation that is not completely illegal, but that can only be accepted provided that other parts of the database be modified. In this case a repairing action is needed to change, add or delete tuples of the database in order to satisfy the integrity constraints again. The obtained database is called a *repair*.

Whereas approaches based on prevention or rollback can be completely automated, repairs typically require the intervention of the database designer to indicate how to react upon detection of inconsistencies. For very simple (yet common) cases, SQL itself provides some standard reaction policies such as `cascade` and `no action` for foreign keys. For more complex circumstances, the DBMS might need to calculate a corrective action according to some criterion of minimality perhaps corresponding to preferences specified by the user. The generation of repairs is a nontrivial issue; see, e.g., [10, 36, 11, 96].

Active rules provide a flexible tool that encompasses all these three approaches. According to the event-condition-action paradigm, triggers react upon (either before or after) a given event (e.g., the update); then they check a given condition specific for that event, such as a (possibly simplified) integrity constraint; finally, if the condition is satisfied, they execute some action, such as aborting the transaction or repairing the database. The main drawback of current trigger-based solutions is that triggers are generated by hand and are typically very difficult to maintain. Approaches based on automatic or semi-automatic generation of triggers for constraint maintenance have appeared since the 1990s [46].

In some scenarios, a temporary violation of integrity constraints may be accepted provided that consistency is quickly repaired; if, by nature of the database application, the data are particularly unreliable, inconsistencies may be even considered unavoidable. Approaches that cope with the presence of inconsistencies are commonly referred to as *paraconsistent* (see, e.g., [72]). In this direction, a problem that has been studied in the literature is that of allowing inconsistencies to occur in databases but to filter the data during query processing so as to provide a *consistent query answer* [10], i.e., the set of tuples that answer a query in all possible repairs (of course without actually having to

compute all the repairs).

Chapter 3

Simplification of Integrity Constraints

This chapter formally specifies the simplification problem and defines the components of a procedure that produces a simplification of integrity constraints with respect to a given update pattern. The key ideas are based on the After and Optimize operations. The former allows the construction of a schema that represents the given integrity constraints in the state after the update, but applying to the current state; in this way, integrity can be checked before performing the update. The latter aims to eliminate redundancies from After's output so as to speed up integrity checking.

In section 3.1 we formalize the simplification problem and we also characterize the notion of ideal simplification, i.e., the ability of a simplification procedure to produce results that are optimal according to some criterion. Then, in section 3.2, we describe a procedure by means of transformations that can be used to achieve simplification in a non-recursive database language called $\mathcal{L}_{\mathsf{s}}$. Before assessing the quality of the results obtained with the described transformations, we first explore the theoretical limits of simplification procedures in general. In particular, in section 3.3, we show the relationship between query containment and ideal simplification procedures. In section 3.4, we discuss different criteria of "ideality" of simplification and describe their relationship with efficiency of evaluation of integrity constraints. In section 3.5, we identify the cases in which the simplification procedure described for $\mathcal{L}_{\mathsf{s}}$ behaves ideally and add some considerations about the complexity. Finally, we conclude the chapter by discussing related work.

3.1 The simplification problem

As discussed in the previous chapter, our goal is to produce a series of checks that can be evaluated in the *present* database state and tell whether the database will remain consistent *after* a prospective update has been performed. This is achieved in two steps. First we produce a new constraint theory that holds in the present state if and only if the original integrity constraints hold following the update. Then we attempt to remove redundancies from the new theory possibly by exploiting the hypothesis that the

database was consistent before the update; the obtained theory, as mentioned, is called a *simplification*.

In this section we discuss the conditions that must be satisfied in order to achieve a simplification of a constraint theory with respect to a given update pattern; later, in section 3.2, we present a concrete simplification framework that operates in terms of syntactic transformations of integrity constraints.

3.1.1 Weakest preconditions

As already emphasized, it is important to be able to test that a prospective database update does not violate the integrity constraints — without actually executing the update, i.e., a test is needed that can be checked in the present state but indicating properties of the prospective new state. A semantic correctness criterion for such a test is given by the notion of weakest precondition.

Definition 3.1.1 (Weakest precondition) *Let $S = \langle IDB, \Gamma \rangle$ be a schema and U an update. A schema $S' = \langle IDB', \Gamma' \rangle$ compatible with S is a* weakest precondition (WP) *of S with respect to U whenever $D \models \Gamma' \Leftrightarrow D^U \models \Gamma$ for any database D based on $IDB \cup IDB'$.*

As also noticed by Qian [143], this definition of WP is similar to the standard axiom for defining assignment statements in a programming language, whose side effects are analogous to a database update. Hoare's [101] original version of the axiom used only implication from pre- to postcondition; the notion of *weakest* precondition that we need is due to Dijkstra [75]. This definition is here given in terms of schemata instead of constraint theories, since new intensional predicates might be needed in order to properly characterize a WP; conversely, in other cases, some predicates originally defined in *IDB* might be irrelevant for that purpose. The notion of schema compatibility ensures that no predicate name clashes occur between the original constraint theory and that of a WP. In the remainder of the thesis, whenever the *IDB* is clear from the context, we shall omit it when indicating a schema or database; in these cases a WP may be simply written as a constraint theory.

It is also important to notice that parametric updates can result in WPs that may be parametric.

The notion of weakest precondition is a sufficient but coarse semantic correctness condition for integrity checks to be performed before an update. In order to also employ the knowledge that the database is consistent before each update, we extend the class of checks of interest as follows.

Definition 3.1.2 (Conditional weakest precondition) *Let $S = \langle IDB, \Gamma \rangle$ be a schema and U an update. A schema $S' = \langle IDB', \Gamma' \rangle$ compatible with S is a* conditional weakest precondition (CWP) *of S with respect to U whenever $D \models \Gamma' \Leftrightarrow D^U \models \Gamma$ for any database D based on $IDB \cup IDB'$ and consistent with Γ.*

A WP is also a CWP but the reverse does not necessarily hold. In fact, we expect that the optimal condition in many cases is among those CWPs that are not WPs.

Example 3.1.3 *Consider U_1 and IDB_1 from example 2.2.22 and let Γ_1 be the constraint theory $\{\leftarrow p(X) \wedge q(X)\}$ stating that p and q are mutually exclusive. Then $\{\leftarrow q(a)\}$ is a CWP (but not a WP) of Γ_1 with respect to U_1.*

The following simple but important property indicates how parameters interact with these notions and follows directly from definition 3.1.2.

Proposition 3.1.4 *Let S and S' be schemata, U a parametric update, and π a parameter substitution for the parameters in U. If S' is a CWP of S with respect to U then $S'\pi$ is a CWP of S with respect to $U\pi$.*

This indicates that it is meaningful to consider general procedures searching for CWPs that are used so to speak "statically" on parametric updates corresponding to update patterns allowed by the database designer before the database contains any data.

3.1.2 Definition of simplification and ideal simplification

A CWP is a necessary and sufficient condition for consistency of a database in the updated state to be checked in the state prior to the update. In order to qualify as a simplification, a CWP should be considered preferable than any non-optimized WP.

For this purpose, we assume, for any schema S and update U, the existence of a fixed reference schema $\bar{S}^U$ representing a non-optimized WP of S with respect to U; we will show the construction of such a WP in section 3.2. We further assume an ordering to sort the different CWPs so that the smallest element in this ordering represents an optimum. For such orderings, we state here a set of natural requirements that are used in our proofs; in section 3.4, we will review several different orderings.

Definition 3.1.5 *An* ordering *between schemata is a reflexive and transitive binary relation $\preceq$. The ordering is* enumerative *if, for any two schemata S and S':*

1. *$\langle \emptyset, \emptyset \rangle \preceq S$.*
2. *It is decidable whether $S \preceq S'$.*
3. *$S \neq S'$ iff either $S \prec S'$ or $S' \prec S$ ($S \prec S'$ means $S \preceq S'$ but not $S' \preceq S$).*
4. *For a given S, the set $\{S'' \mid S'' \preceq S\}$ is finite and the schemata in it can be enumerated.*

In order to have conditions 3 and 4 satisfied in interesting cases it is essential to consider as identical any two expressions that differ only by a consistent renaming of bound variables or differ for orders of commutative and associative operators. Unless differently stated, we always refer to enumerative orderings; see section 3.4 for further discussion.

Definition 3.1.6 (Simplification) *A function from a schema S and an update U to another schema S' is called a* simplification function *for an ordering $\preceq$, written $\mathsf{Simp}^U(S) = S'$, whenever*

- *S' is a CWP of S with respect to U (and thus S' is compatible with S),*
- *$S' \preceq \bar{S}^U$.*

Schema S' is called a simplification *of S wrt U. We say that a simplification is* ideal *whenever there is no other schema S'' that is a CWP of S with respect to U such that $S'' \prec S'$. A simplification function is* ideal *if, for any S, U, $\mathsf{Simp}^U(S)$ is an ideal simplification of S wrt U.*

A(n ideal) simplification procedure is an algorithm that implements a(n ideal) simplification function.

According to this definition, any CWP that is at least as small as $\bar{S}^U$ in the $\preceq$ ordering is considered a simplification; the minimal one, among those, is the ideal simplification. This distinction makes sense because, as we shall see, it is not always possible to obtain an ideal simplification in all cases, but even a non-ideal simplification can be a significant improvement with respect to a non-optimized WP. This is particularly important when the ordering somehow reflects the effort of checking the satisfaction of the constraint theory in any database state: ideal simplifications do then express the best possible way of checking integrity constraints. An immediate consequence of definitions 3.1.5 and 3.1.6 is that, if the update cannot affect the integrity, the ideally simplified constraint theory must be $\emptyset$.

Proposition 3.1.7 *Let U be an update and $S = \langle IDB, IC\rangle$ a schema such that, for any database D based on IDB and consistent with IC, D^U is also consistent with IC. Let* Simp *be an ideal simplification procedure. Then $\mathsf{Simp}^U(S) = \langle\emptyset, \emptyset\rangle$.*

Proof *Trivial, as the schema $\langle\emptyset, \emptyset\rangle$ is a CWP of S with respect to U and is minimal with respect to $\preceq$ (first assumption on the ordering in definition 3.1.5).* □

As mentioned, simplification can be performed statically, but further simplification can be achieved, in some cases, once the parameters are instantiated, even when the initial simplification was ideal.

Example 3.1.8 *Consider the following constraint theory and update:*

$$\Gamma = \{\leftarrow p(X,Y) \wedge X \neq Y\}$$

$$U = \{p(X,Y) \Leftarrow p(X,Y) \vee (X = \mathbf{a} \wedge Y = \mathbf{b})\}.$$

The schema $S' = \langle\emptyset, \{\leftarrow \mathbf{a} \neq \mathbf{b}\}\rangle$ is a CWP of $\langle\emptyset, \Gamma\rangle$ wrt. U; S' is an ideal simplification with respect to an enumerative ordering counting the number of literals in a constraint theory, as S' cannot be reduced to an expression with fewer literals. However, for $\pi = \{\mathbf{a}/c, \mathbf{b}/c\}$ we get $S'\pi \equiv \langle\emptyset, \emptyset\rangle \prec S'\pi$ (by assumptions 1 and 3 on $\prec$ in definition 3.1.5).

We introduce later in proposition 3.1.13 a simple correction for the indicated problem.

When the *IDB* is clear from the context, we may omit it and write more simply that $\mathsf{Simp}^U(\Gamma) = \Gamma'$, where Γ and Γ' are constraint theories. If U is parametric, the resulting Γ' may be parametric. Note, however, that Γ' is not necessarily parametric even though U is. This happens, for instance, when U does not affect Γ or when (in)equalities between parameters become eliminable, as in example 3.1.9.

Example 3.1.9 *Consider the constraint theory $\Gamma = \{\leftarrow p(X,Y) \wedge X \neq Y\}$ and the parametric update $U = \{p(X,Y) \Leftarrow p(X,Y) \vee (X = \mathbf{a} \wedge Y = \mathbf{a})\}$. Then $\Gamma' = \emptyset$ is a (non-parametric) CWP of Γ with respect to U.*

3.1.3 Simplification based on optimization

The basic idea in our approach to produce simplified integrity constraints is to start with the reference schema and optimize it as much as possible.

Definition 3.1.10 (Conditional equivalence) *Let $S_\Delta = \langle IDB_\Delta, \Delta\rangle$, $S = \langle IDB, \Gamma\rangle$, $S' = \langle IDB', \Gamma'\rangle$ be compatible schemata. Then S and S' are* conditionally equivalent *with respect to S_Δ, denoted $S \overset{S_\Delta}{\equiv} S'$, if $D \models \Gamma \Leftrightarrow D \models \Gamma'$ in any database D based on $IDB_\Delta \cup IDB \cup IDB'$ and consistent with Δ.*

Conditional equivalence characterizes the class of constraint theories/schemata that are equivalent under the hypothesis that another constraint theory/schema holds. In definition 3.1.10, S_Δ typically includes the constraints to be simplified and may also contain other hypotheses that are trusted. Now we can precisely specify the notion of optimization with respect to given hypotheses.

Definition 3.1.11 *An* optimization function *takes two schemata S_Δ and S and produces another schema S', written $\mathsf{Optimize}^{S_\Delta}(S) = S'$, such that*

- $S \overset{S_\Delta}{\equiv} S'$
- $S' \preceq S$
- $\mathsf{Optimize}^{S_\Delta}(S') = S'$, *i.e., the function is* idempotent.

An optimization function is ideal *whenever there is no other schema $S'' \overset{S_\Delta}{\equiv} S$ such that $S'' \prec S'$. An (ideal) optimization procedure is an algorithm that implements an (ideal) optimization function.*

Proposition 3.1.12 follows immediately from the definitions.

Proposition 3.1.12 *For a given optimization function (procedure)* Optimize*, the function (procedure) defined by*

$$S \mapsto \mathsf{Optimize}^S(\bar{S}^U)$$

is a simplification function (procedure). If Optimize *is an ideal function (procedure), so is the indicated simplification function (procedure).*

We end this section showing how an extra round of optimization can repair the slight lack of perfection in the application of (even ideal) simplification for parametric updates indicated in example 3.1.8.

Proposition 3.1.13 *Let $S = \langle IDB, \Gamma\rangle$ be a schema, U a parametric update, and π a parameter substitution for the parameters in U. Assume an ideal simplification procedure* Simp *and an ideal optimization procedure* Optimize*. Then $\mathsf{Optimize}^S((\mathsf{Simp}^U(S))\pi)$ is a minimal CWP of S with respect to $U\pi$.*

Proof *Let $S' = \mathsf{Simp}^U(S)$. By proposition 3.1.4 we have that $S'\pi$ is a CWP of S with respect to $U\pi$. By definition 3.1.11, $\mathsf{Optimize}^S(S'\pi)$ is minimal according to $\prec$.* □

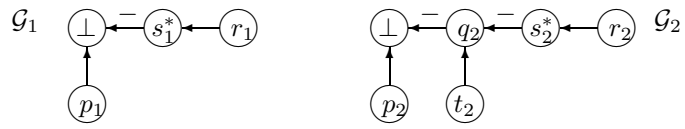

Figure 3.1: The starred dependency graphs for the schemata of example 3.2.2.

3.2 Transformations of integrity constraints

We start this section by formally characterizing a language, that we call $\mathcal{L}_s$, on which a core simplification procedure can be applied. The language $\mathcal{L}_s$ is described in section 3.2.1. Later, in section 3.2.2, we show a procedure, called After, that, for any given schema S and update U, effectively generates the reference weakest precondition $\bar{S}^U$ that was only assumed to exist in the previous section. This procedure is then refined for schemata and updates in $\mathcal{L}_s$. In section 3.2.3, we give a concrete implementation for the operators Optimize and Simp for the language $\mathcal{L}_s$. Their usage is then demonstrated in several examples. Finally, in section 3.2.4 we further discuss differences between pre-tests and post-tests.

3.2.1 The language $\mathcal{L}_s$

We introduce now a few technical notions that are needed in order to precisely identify the class of predicates, integrity constraints and updates that are part of $\mathcal{L}_s$, which is a non-recursive function-free language equipped with negation.

Definition 3.2.1 (Starred dependency graph) *Let $S = \langle IDB, \Gamma \rangle$ be a schema in which the IDB consists of a set of disjunctive (range restricted) predicate definitions and Γ is a set of range restricted denials. Let $\mathcal{G}$ be a graph that has a node N^p for each predicate p in S plus another node named $\bot$, and no other node; if p's defining formula has non-distinguished variables, then N^p is marked with a "*". For any two predicates p and p' in S, $\mathcal{G}$ has an arc from N^p to $N^{p'}$ for each occurrence of p in an IDB rule in which p' occurs in the head; similarly, there is an arc from N^p to $\bot$ for each occurrence of p in the body of a denial in Γ. In both cases the arc is labelled with a "–" (and said to be* negative*) iff p occurs negatively. $\mathcal{G}$ is the* starred dependency graph *for S.*

Example 3.2.2 *Let $S_1 = \langle IDB_1, \Gamma_1 \rangle$ and $S_2 = \langle IDB_2, \Gamma_2 \rangle$ be schemata with*

$$\begin{aligned} IDB_1 &= \{ \quad s_1(X) \leftarrow r_1(X,Y)\}, \\ \Gamma_1 &= \{ \quad \leftarrow p_1(X) \wedge \neg s_1(X)\}, \\ IDB_2 &= \{ \quad s_2(X) \leftarrow r_2(X,Y), \\ &\qquad\quad q_2(X) \leftarrow \neg s_2(X) \wedge t_2(X)\}, \\ \Gamma_2 &= \{ \quad \leftarrow p_2(X) \wedge \neg q_2(X)\}. \end{aligned}$$

The starred dependency graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ of, respectively, S_1 and S_2 are shown in figure 3.1.

Definition 3.2.3 ($\mathcal{L}_s$) *Let $S = \langle IDB, \Gamma \rangle$ be a database schema in which the IDB consists of a set of disjunctive predicate definitions and Γ is a set of range restricted denials. Then S is in $\mathcal{L}_s$ if:*

1. *its starred dependency graph $\mathcal{G}$ is acyclic;*

2. *in every path in $\mathcal{G}$ from a starred node to $\bot$ the number of arcs labelled with "$-$" is even.*

Consider an update U for S of the form $\{p_1(\vec{X}_1) \Leftarrow q_1(\vec{X}_1),\ \ldots,\ p_m(\vec{X}_m) \Leftarrow q_m(\vec{X}_m)\}$, where the q_i's are intensional predicates in S. Then U is in $\mathcal{L}_\mathsf{s}$ with respect to S if the graph obtained from $\mathcal{G}$ by adding an arc from N^{q_i} to N^{p_i}, for $1 \leq i \leq m$, still satisfies condition 2 above.

The acyclicity of the starred dependency graph corresponds to the absence of recursion in the database. The second condition in definition 3.2.3 requires that the unfolding of the intensional predicates in the constraint theory with respect to their definitions does not introduce any negated existentially quantified variable, as explained below. A similar requirement is imposed when the database is updated. In particular, starred nodes correspond to literals with non-distinguished variables, that are existentially quantified; therefore, in order to avoid negated existential quantifiers, that cannot be represented in $\mathcal{L}_\mathsf{s}$ denials, the number of encountered "$-$" signs must be even, as an even number of negations means no negation.

Example 3.2.4 *Consider S_1 and S_2 from example 3.2.2. Clearly S_1 is not in $\mathcal{L}_\mathsf{s}$, as in its starred dependency graph $\mathcal{G}_1$ in figure 3.1 there is a path from s_1^* to $\bot$ containing one "$-$" arc. On the contrary, S_2 is in $\mathcal{L}_\mathsf{s}$, as in $\mathcal{G}_2$ the only path from s_2^* to $\bot$ contains two "$-$" arcs.*

In semi-positive schemata, negation can only occur before extensional predicates. Therefore, in the dependency graph of a semi-positive schema, no path from a starred node to $\bot$ can contain negative arcs. This indicates that the language of non-recursive semi-positive schemata is contained in $\mathcal{L}_\mathsf{s}$. In fact, the containment is strict, since, e.g., schema S_2 is in $\mathcal{L}_\mathsf{s}$, but is not semi-positive. According to these considerations, the class of schemata captured by $\mathcal{L}_\mathsf{s}$ is already sufficiently broad as to include many practically relevant cases. The extension to more expressive languages will be considered in chapter 4.

3.2.2 Generating weakest preconditions

The following straightforward syntactic transformation After translates a constraint theory A referring to the updated database state into another constraint theory B that holds in the present state if and only if A holds following the update. In other words, After generates a WP.

Definition 3.2.5 (After) *Let $S = \langle IDB, \Gamma \rangle$ be a schema and U an update such that, for each predicate update $p(\vec{X}) \Leftarrow p^U(\vec{X})$ in U, p^U is defined in IDB.*

- *Let us indicate with Γ^U a copy of Γ in which any atom $p(\vec{t})$ whose predicate is affected by a predicate update $p(\vec{X}) \Leftarrow p^U(\vec{X})$ in U is simultaneously replaced by the expression $p^U(\vec{t})$ and every intensional predicate q is simultaneously replaced by a new intensional predicate q^U defined in IDB^U below.*

- *Similarly, let us indicate with IDB^U a copy of IDB in which the same replacements are simultaneously made, and let IDB^* be the biggest subset of $IDB \cup IDB^U$ including only definitions of predicates on which Γ^U depends.*

We define $\mathsf{After}^U(S) = \langle IDB^*, \Gamma^U \rangle$.

The IDB^U used in the construction of definition 3.2.5 indicates auxiliary views that are needed in order to properly characterize the resulting constraint theory. As we shall see, often no such views are strictly necessary, whereas, in some other cases, they cannot be avoided; in the former case, when this is clear from the context, we may omit the specification of the intensional database from schemata. The definition given here is not limited to constraint theories in $\mathcal{L}_\mathsf{s}$; we specialize it for $\mathcal{L}_\mathsf{s}$ in definition 3.2.13.

We notice the following trivial property which is paramount for our use of parameters.

Proposition 3.2.6 *Let S be a schema, U an update, and π a parameter substitution. Then* $(\mathsf{After}^U(S))\pi \equiv \mathsf{After}^{U\pi}(S)$.

Proof *The claim holds by construction: the parameters present in U are included in p^U's defining formula for each predicate update $p(\vec{X}) \Leftarrow p^U(\vec{X})$ in U. No p^U definition is modified by* After. *The effect of applying π either to U or to* $\mathsf{After}^U(S)$ *is therefore to replace the parameters in each p^U definition. But the construction in* $(\mathsf{After}^U(S))\pi$ *and in* $\mathsf{After}^{U\pi}(S)$ *is the same. The results are, thus, identical.* □

Example 3.2.7 *Consider the updates and IDB definitions of example 2.2.22 on page 17. Let schema S_1 be $\langle IDB_1, \Gamma_1 \rangle$, where $\Gamma_1 = \{\leftarrow p(X) \wedge q(X)\}$ states that p and q are mutually exclusive. We have* $\mathsf{After}^{U_1}(S_1) = \langle IDB_1, \Gamma_1^{U_1} \rangle$, *where:*

$$\Gamma_1^{U_1} = \{\leftarrow p'(X) \wedge q(X)\}.$$

Note that the occurrence of $p'(X)$ in $\Gamma_1^{U_1}$ can be replaced by its defining formula without affecting the semantics of $\Gamma_1^{U_1}$. As mentioned, we may therefore omit the intensional database and more conveniently represent the final result of the After *transformation as follows*

$$\mathsf{After}^{U_1}(\Gamma_1) = \Sigma = \quad \{ \quad \leftarrow \quad p(X) \wedge q(X), \\ \leftarrow \quad X = a \wedge q(X) \quad \},$$

observing that $\mathsf{After}^{U_1}(S_1) \equiv \langle \emptyset, \Sigma \rangle$. *Here the disjunction in U_1 was split in two clauses that only contain conjunctions as to conform to the denial format for integrity constraints.*

For $\Gamma_2 = \{\leftarrow r(b, X) \wedge q(X)\}$ and update U_2 we get in a similar way the following.

$$\mathsf{After}^{U_2}(\Gamma_2) = \quad \{ \quad \leftarrow \quad r(b, X) \wedge b \neq a \wedge q(X), \\ \leftarrow \quad r(a, X) \wedge b = b \wedge q(X) \quad \}.$$

As above, the operation resulted in a disjunction that was split into two denials. Note that $b \neq a$ and $b = b$ are consequences of the free equality axioms and could be removed.

However, when applying instead the parametric update U_3 to Γ_2 we cannot remove equalities and non-equalities, as they evaluate to different truth values depending on different instantiations of the parameters. We get:

$$\mathsf{After}^{U_3}(\Gamma_2) = \{ \begin{array}{ll} \leftarrow & r(b,X) \wedge b \neq \mathbf{a} \wedge q(X), \\ \leftarrow & r(\mathbf{a},X) \wedge b = \mathbf{b} \wedge q(X) \end{array} \}.$$

If the parameters $\mathbf{a}$, $\mathbf{b}$ are respectively instantiated to the two constants a and b, the result can be reduced to $\mathsf{After}^{U_2}(\Gamma_2)$. In case both parameters are instantiated to the same constant a, the result collapses to Γ_2 which is consistent with the observation that the update in this case is neutral.

The following example shows that After may introduce new predicates that cannot be eliminated from the resulting constraint theory by replacing them with their defining formula, as is the case in example 3.2.8 below.

Example 3.2.8 *Let D be a database containing information about the nodes of a directed graph. The extensional relation $e/2$ contains pairs of nodes that are connected by a directed arc. Directed paths between two nodes are expressed by the recursive intensional relation $p/2$ defined by the following rules:*

$$IDB^p = \{ \begin{array}{ll} & p(X,Y) \leftarrow e(X,Y), \\ & p(X,Y) \leftarrow e(X,Z) \wedge p(Z,Y) \end{array} \}.$$

Acyclicity of the graph can be imposed with the following constraint theory

$$\Gamma = \{\leftarrow p(X,X)\},$$

indicating that there must be no path from a node to itself. Let $U = \{e(X,Y) \Leftarrow e'(X,Y)\}$ be an update pattern that adds the tuple $\langle \mathbf{a}, \mathbf{b} \rangle$ to relation e, where $\mathbf{a}$ and $\mathbf{b}$ are parameters and e' is defined as follows.

$$IDB^{e'} = \{ \begin{array}{ll} & e'(X,Y) \leftarrow e(X,Y), \\ & e'(X,Y) \leftarrow X = \mathbf{a} \wedge Y = \mathbf{b} \end{array} \}.$$

Let $IDB = IDB^p \cup IDB^{e'}$ and S be the schema $\langle IDB, \Gamma \rangle$ (which is not in $\mathcal{L}_\mathsf{s}$). Then $\mathsf{After}^U(S) = \langle IDB^U, \Gamma^U \rangle$, where

$$\begin{array}{rll} IDB^U = & \{ & p^U(X,Y) \leftarrow e(X,Y), \\ & & p^U(X,Y) \leftarrow X = \mathbf{a} \wedge Y = \mathbf{b}, \\ & & p^U(X,Y) \leftarrow e(X,Z) \wedge p^U(Z,Y), \\ & & p^U(X,Y) \leftarrow X = \mathbf{a} \wedge Z = \mathbf{b} \wedge p^U(Z,Y) \quad \}, \\ \Gamma^U = & \{ & \leftarrow p^U(X,X) \ \}. \end{array}$$

In this case, p^U is recursively defined and therefore it is not possible to disregard IDB^U by replacing occurrences of p^U in Γ^U with its defining formula.

The treatment of such recursive cases will be described in chapter 4.

The characteristic property of the After transformation is that of producing a WP.

Proposition 3.2.9 *Let S be a schema and U an update. Then $\mathsf{After}^U(S)$ is a WP of S with respect to U.*

Proof *Trivially $\mathsf{After}^U(S)$ is compatible with S, as only new intensional predicates are introduced. Assume $S = \langle IDB, \Gamma \rangle$; assume for now that neither U nor S contain parameters. Let D be a database based on $IDB \cup IDB^U$. By construction, we have that*

1. *for any database predicate p in Γ, its extension in D^U coincides with the extension of p^U in D.*

Hence the claim $D^U \models \Gamma \Leftrightarrow D \models \Gamma^U$.

To see that (1) is the case, we note that:

- *If p is extensional, (1) coincides with the definition of update.*
- *If p is intensional, then p^U in IDB^U is defined exactly as p in IDB modulo renaming of predicates (adding superscript U). As claim (1) holds for the first stratum of the standard model (extensional predicates, by the previous point) and the rules are isomorphic, the argument can be repeated for the second stratum, and so on for all strata.*

To see that this generalizes to the case where U contains parameters, observe that the arguments above hold for any instance of U and that $\mathsf{After}^{U\pi}(S) \equiv (\mathsf{After}^U(S))\pi$ for any parameter substitution π by proposition 3.2.6. □

Since a WP is also a CWP, the result produced by After is also trivially a CWP. However, the hypothesis of consistency of the database in the original state is not used at all in the calculation of After, so we may assume that the reference WP $\bar{S}^U$ of a schema S with respect to update U is given by $\mathsf{After}^U(S)$. Therefore, a simplification procedure must produce results that are at least as good (wrt. $\preceq$) as those produced by After. We note that, in some trivial cases, the result of the After transformation itself can be an ideal simplification, e.g., when the initial constraint theory is $\emptyset$.

The After operator simultaneously replaces all occurrences of an updated predicate with the corresponding query and possibly performs some equivalence-preserving transformations. For $\mathcal{L}_\mathsf{s}$, we can specialize the After transformation so as to use *unfolding* [26] to repeatedly replace every intensional predicate by its definition until only extensional predicates appear in the constraint theory. These replacements may determine the presence of disjunctions and negated conjunctions in the formula. The denial form needs then to be restored in order for the database language to be closed under the After transformation. For this purpose, we use the $\mathsf{Unfold}_{\mathcal{L}_\mathsf{s}}$ operator below, which is defined in terms of unification and syntactic simplifications based on de Morgan's laws to remove all disjunctions and negated conjunctions.

Definition 3.2.10 (Unfolding) *Let $S = \langle IDB, \Gamma \rangle$ be a database schema in $\mathcal{L}_\mathsf{s}$. Then, $\mathsf{Unfold}_{\mathcal{L}_\mathsf{s}}(S)$ is the schema $\langle \emptyset, \Gamma' \rangle$, where Γ' is the set of denials obtained by iterating the two following steps as long as possible:*

1. *replace, in Γ, each occurrence of an atom of the form $p(\vec{t})$ by $F^p\{\vec{X}/\vec{t}\}$, where $p \in pred(IDB)$, F^p is p's defining formula and $\vec{X}$ its distinguished variables. If no replacement was made, then stop;*

2. *transform the resulting formula into a set of denials according to the following patterns:*

   - $\leftarrow A \wedge (B_1 \vee B_2)$ *is replaced by* $\leftarrow A \wedge B_1$ *and* $\leftarrow A \wedge B_2$*;*
   - $\leftarrow A \wedge \neg(B_1 \vee B_2)$ *is replaced by* $\leftarrow A \wedge \neg B_1 \wedge \neg B_2$*;*
   - $\leftarrow A \wedge \neg(B_1 \wedge B_2)$ *is replaced by* $\leftarrow A \wedge \neg B_1$ *and* $\leftarrow A \wedge \neg B_2$*.*

Due to the implicit outermost universal quantification of the variables, non-distinguished variables in a predicate definition are existentially quantified to the right-hand side of the arrow, as shown in example 3.2.11 below. For this reason, with no indication of the quantifiers, the replacements in definition 3.2.10 preserve equivalence iff no predicate containing non-distinguished variables occurs negated in the resulting expression.

Example 3.2.11 *Consider S_1 from example 3.2.2, which, as shown, is not in $\mathcal{L}_s$. With the explicit indication of the quantifiers we have*

$$IDB_1 \equiv \{\forall X(s_1(X) \leftarrow \exists Y\ r_1(X,Y))\}.$$

The replacement, in Γ_1, of $s_1(X)$ by its definition in IDB_1 would determine the formula

$$\Gamma_1' = \{\forall X,Y(\leftarrow p_1(X) \wedge \neg r_1(X,Y))\}.$$

However, this replacement is not equivalence-preserving, because a predicate (r_1) containing a non-distinguished variable occurs negated:

$$\Gamma_1 \equiv \{\forall X(\leftarrow p_1(X) \wedge \neg\exists Y\ r_1(X,Y))\} \not\equiv \Gamma_1'.$$

An extension of the language to cases where $\neg\exists$ may occur in denials is discussed in chapter 4.

The language $\mathcal{L}_s$ is closed under unfolding and $\mathsf{Unfold}_{\mathcal{L}_s}$ preserves equivalence.

Proposition 3.2.12 *Let $S = \langle IDB, \Gamma\rangle \in \mathcal{L}_s$. Then $\mathsf{Unfold}_{\mathcal{L}_s}(S) \in \mathcal{L}_s$ and $\mathsf{Unfold}_{\mathcal{L}_s}(S) \equiv S$.*

Proof *Each iteration in definition 3.2.10 preserves range restriction: if an atom $p(\vec{t})$ occurs negatively, then, by condition 2 in definition 3.2.3, it is replaced by an expression that does not contain any extra (non-distinguished) variable not in $\vec{t}$; if it occurs positively, then it is replaced by the body of a range restricted rule. The constraint theory in $\mathsf{Unfold}_{\mathcal{L}_s}(S)$ is a set of denials not containing intensional predicates, and therefore $\mathsf{Unfold}_{\mathcal{L}_s}(S) \in \mathcal{L}_s$. Furthermore, each iteration in definition 3.2.10 is equivalence-preserving. In step 1, condition 2 in definition 3.2.3 ensures that each predicate containing non-distinguished variables is preceded by an even number of $\neg$ signs, and thus not negated; step 2 is a collection of trivial equivalence-preserving rewrites.* □

The After transformation can now be refined for $\mathcal{L}_s$, by unfolding the resulting theory.

Definition 3.2.13 *Let $S \in \mathcal{L}_s$ and U be an update in $\mathcal{L}_s$ with respect to S. We define $\mathsf{After}^U_{\mathcal{L}_s}(S)$ as $\mathsf{Unfold}_{\mathcal{L}_s}(\mathsf{After}^U(S))$.*

The closure of $\mathcal{L}_s$ under $\mathsf{After}_{\mathcal{L}_s}$ follows immediately from the definitions and proposition 3.2.12 and is, thus, stated without a proof.

Proposition 3.2.14 (Closure of $\mathcal{L}_s$ under $\mathsf{After}_{\mathcal{L}_s}$) *Let $S \in \mathcal{L}_s$ and U be an update in $\mathcal{L}_s$ with respect to S; then $\mathsf{After}^U_{\mathcal{L}_s}(S) \in \mathcal{L}_s$.*

In the following, when *IDB* is understood or empty, we indicate the constraint theory instead of the schema in $\mathsf{After}_{\mathcal{L}_s}$.

Example 3.2.15 *Consider a database containing information about marriages, where the binary predicate m indicates that a husband (first argument) is married to a wife (second argument). We expect for this database updates of the form $U = \{m(X,Y) \Leftarrow m^U(X,Y)\}$, where m^U is a query defined by the predicate definition $m^U(X,Y) \leftarrow m(X,Y) \vee (X = \mathbf{a} \wedge Y = \mathbf{b})$, i.e., U is the addition of the tuple $\langle \mathbf{a}, \mathbf{b} \rangle$ to m. The following integrity constraint is given:*

$$\phi = \leftarrow m(X,Y) \wedge m(X,Z) \wedge Y \neq Z$$

meaning that no husband can be married to two different wives. Following definition 3.2.13 in the calculation of $\mathsf{After}^U_{\mathcal{L}_s}(\{\phi\})$, first each occurrence of m is replaced by m^U, obtaining

$$\leftarrow m^U(X,Y) \wedge m^U(X,Z) \wedge Y \neq Z.$$

Then $\mathsf{Unfold}_{\mathcal{L}_s}$ is applied to this integrity constraint. The first step of definition 3.2.10 generates the following:

$$\{\leftarrow (m(X,Y) \vee (X = \mathbf{a} \wedge Y = \mathbf{b})) \wedge (m(X,Z) \vee (X = \mathbf{a} \wedge Z = \mathbf{b})) \wedge Y \neq Z\}.$$

The second step translates it to clausal form:

$$\begin{aligned} \mathsf{After}^U_{\mathcal{L}_s}(\{\phi\}) = \{ \quad & \leftarrow m(X,Y) \wedge m(X,Z) \wedge Y \neq Z, \\ & \leftarrow m(X,Y) \wedge X = \mathbf{a} \wedge Z = \mathbf{b} \wedge Y \neq Z, \\ & \leftarrow X = \mathbf{a} \wedge Y = \mathbf{b} \wedge m(X,Z) \wedge Y \neq Z, \\ & \leftarrow X = \mathbf{a} \wedge Y = \mathbf{b} \wedge X = \mathbf{a} \wedge Z = \mathbf{b} \wedge Y \neq Z \quad \}. \end{aligned}$$

Example 3.2.16 *We shall now consider an example of referential integrity, where a relation f (father) is only meaningful if its first argument (the father) is recorded in a relation p (person) with a specific constant value concerning the gender (m for "male"):*

$$\phi = \leftarrow f(X,Y) \wedge \neg p(X,m).$$

The following U and IDB indicate an update transaction adding a father/child tuple $\langle \mathbf{a}, \mathbf{b} \rangle$ to f and a tuple indicating that $\mathbf{a}$'s gender is "male" to p.

$$U = \{f(X,Y) \Leftarrow f^U(X,Y),\ p(X,Y) \Leftarrow p^U(X,Y)\},$$

$$\begin{aligned} IDB = \{ \quad & f^U(X,Y) \leftarrow f(X,Y) \vee (X = \mathbf{a} \wedge Y = \mathbf{b}), \\ & p^U(X,Y) \leftarrow p(X,Y) \vee (X = \mathbf{a} \wedge Y = m) \quad \}. \end{aligned}$$

We have:

$$\begin{aligned} \mathsf{After}^U_{\mathcal{L}_s}(\{\phi\}) = \{ \quad & \leftarrow X = \mathbf{a} \wedge Y = \mathbf{b} \wedge X \neq \mathbf{a} \wedge \neg p(X,m), \\ & \leftarrow X = \mathbf{a} \wedge Y = \mathbf{b} \wedge m \neq m \wedge \neg p(X,m), \\ & \leftarrow f(X,Y) \wedge X \neq \mathbf{a} \wedge \neg p(X,m), \\ & \leftarrow f(X,Y) \wedge m \neq m \wedge \neg p(X,m) \qquad \}. \end{aligned}$$

3.2.3 Simplification in $\mathcal{L}_s$

The theory returned by $\mathsf{After}_{\mathcal{L}_s}$ refers to extensional predicates only, so, from now on, the *IDB* component of a database can be completely disregarded.

Clearly, the result returned by $\mathsf{After}_{\mathcal{L}_s}$ may contain redundant denials and sub-formulas (such as, e.g., $a = a$). Moreover, the fact that the original integrity constraints hold in the current database state can be used to achieve further simplification. For this purpose, we define a transformation $\mathsf{Optimize}_{\mathcal{L}_s}$ that optimizes a given constraint theory using a set of trusted hypotheses. Typically, the input to $\mathsf{Optimize}_{\mathcal{L}_s}$ is $\mathsf{After}_{\mathcal{L}_s}$'s output theory and the hypotheses are $\mathsf{After}_{\mathcal{L}_s}$'s input theory. We describe here an implementation in terms of sound and terminating rewrite rules. An application of a rewrite rule to a constraint theory always produces a theory that has fewer literals[1]. $\mathsf{Optimize}_{\mathcal{L}_s}$ applies the rules as long as possible in order to remove from the input theory all denials and all literals that, with the hypotheses, can be proved to be redundant. Termination is then guaranteed, since the size of the theory is reduced at each rule application and the conditions of applicability of the rules are based on decidable criteria.

Subsumption, reduction and resolution as constraint modification tools

We introduce in this subsection a series of syntactic tools that will be used to compose the optimization step of the simplification procedure.

Definition 3.2.17 (Subsumption) *Given two denials D_1 and D_2, D_1* subsumes *D_2 (with substitution σ) iff there is a substitution σ such that each literal in $D_1\sigma$ occurs in D_2. The subsumption is* strict *if D_1 is not a variant of D_2.*

The subsumption algorithm (see, e.g., [94]), besides checking subsumption, also returns such substitution σ and, for convenience, with the symbols of the definition above, we say that D_1 subsumes D_2 with substitution σ.

Example 3.2.18 *The denial $\leftarrow p(X,Y) \wedge q(Y)$ (strictly) subsumes $\leftarrow p(X,b) \wedge X \neq a \wedge q(b)$ with substitution $\{Y/b\}$.*

The definition of subsumption is syntactic but has the semantic property that the subsuming denial implies the subsumed one. We also note that an ordering of denials based on strict subsumption is well-founded, i.e., there is no infinite descending chain $\phi_1, \phi_2, \ldots$ such that ϕ_1 is strictly subsumed by ϕ_2, ϕ_2 by ϕ_3, and so on, since the initial denial is finite.

The notion of *reduction* [93] characterizes the elimination of redundancies within a single denial.

Definition 3.2.19 (Reduction) *For a denial ϕ, the reduction ϕ^- of ϕ is the result of applying on ϕ the following rules as long as possible, where L is a literal, c_1, c_2 are distinct constants, X a bound variable, t a term, A an atom, C, D (possibly empty) conjunctions*

[1] To be precise, the produced theory is either strictly smaller or its expansion is strictly smaller than the expansion of the original theory. See below in the text and definition 3.2.28.

of literals and vars *indicates the set of bound variables occurring in its argument.*

$$
\begin{array}{rcl}
\leftarrow L \wedge C & \Rightarrow & \leftarrow C \quad \textit{if } L \textit{ is of the form } t = t \textit{ or } c_1 \neq c_2 \\
\leftarrow L \wedge C & \Rightarrow & \textit{true} \quad \textit{if } L \textit{ is of the form } t \neq t \textit{ or } c_1 = c_2 \\
\leftarrow X = t \wedge C & \Rightarrow & \leftarrow C\{X/t\} \\
\leftarrow A \wedge \neg A \wedge C & \Rightarrow & \textit{true} \\
\leftarrow C \wedge D & \Rightarrow & \leftarrow D \quad \textit{if} \leftarrow C \textit{ subsumes} \leftarrow D \textit{ with a substitution } \sigma \textit{ s.t.} \\
 & & \mathrm{dom}(\sigma) \cap \mathrm{vars}(D) = \emptyset
\end{array}
$$

Clearly, for any denial ϕ we have $\phi^- \equiv \phi$ and the number of literals in ϕ^- is less than the number of literals in ϕ. Obviously, an ordering of denials based on the number of literals is well-founded, i.e., there is no infinite descending chain $\phi_1, \phi_2, \ldots$ such that ϕ_1 has more literals than ϕ_2, ϕ_2 has more literals than ϕ_3, and so on, since the initial denial is finite. The last rewrite rule is called subsumption factoring [77] and includes the elimination of duplicate literals from a denial as a special case. We also note that each rule preserves range restriction. In the last rule, a variable occurring in a negative literal in D and in a positive literal in C also occurs in a positive literal in D, otherwise this variable would be in $\mathrm{dom}(\sigma)$. An additional rule for handling parameters may be considered in the reduction process:

$$\leftarrow \mathbf{a} = c_1 \wedge C \quad \Rightarrow \quad \leftarrow \mathbf{a} = c_1 \wedge C\{\mathbf{a}/c_1\} \tag{3.1}$$

This may replace parameters with constants and possibly allow further reduction. For example, the denial $\leftarrow \mathbf{a} = c \wedge b = \mathbf{a}$ would be transformed into $\leftarrow \mathbf{a} = c \wedge b = c$, and, thus, into *true*, since b and c are different constants. However, rule 3.1 does not reduce the number of literals, which is a condition for termination that we use in the proofs to follow. To this end, we may assume that rule 3.1 is applied before the other reduction rules and that each equality between a parameter and a constant is only processed once.

Although, in most cases, reduction eliminates all redundancies from a denial, this may not happen in the presence of parameters. For example, the denial

$$\leftarrow \mathbf{a} \neq \mathbf{b} \wedge \mathbf{b} = \mathbf{c} \wedge \mathbf{c} = \mathbf{d} \wedge \mathbf{d} = \mathbf{a}$$

is clearly a consequence of the free equality axioms, but reduction leaves it unchanged. To detect this redundancy, a transitivity axiom is needed, as will be discussed in section 3.5.2.

We now briefly recall the definition of resolvent and derivation for the well-known principle of resolution in the context of clauses with logical negation ($\neg_\ell$); we refer to [146, 50] for other standard related notions. To stress that the usual context of application of resolution is not that of deductive databases, we first state the following definition for clauses expressed as sets of literals of the form $\{L_1, \ldots, L_n\}$, as shown on page 9.

Definition 3.2.20 *Let C_1 and C_2 be two clauses (called* parent clauses*) with no variables in common. Let L_1 and L_2 be two literals in C_1 and C_2, respectively. If L_1 and $\neg_\ell L_2$ have an mgu σ, then the clause $(C_1\sigma \setminus \{L_1\sigma\}) \cup (C_2\sigma \setminus \{L_2\sigma\})$ is called a* binary resolvent *of C_1 and C_2. The literals L_1 and L_2 are called the literals* resolved upon.

If two or more literals of a clause C have an mgu σ, then $C\sigma$ is called a factor *of C.*

A resolvent *of (parent) clauses C_1 and C_2 is one of the following binary resolvents:*

1. *a binary resolvent of* C_1 *and* C_2,

2. *a binary resolvent of* C_1 *and a factor of* C_2,

3. *a binary resolvent of a factor of* C_1 *and* C_2,

4. *a binary resolvent of a factor of* C_1 *and a factor of* C_2.

The resolution principle is a sound inference rule, in that a resolvent is a logical consequence of its parent clauses. In the context of denials with default negation the notion of binary resolvent can be formulated as in definition 3.2.21, where, apart from the different syntactic representation, the $\neg$ symbol replaces the $\neg_\ell$ symbol. The notions of factor and resolvent for denials with default negation are the same as those for clauses with logical negation.

Definition 3.2.21 *Let* $\phi_1' = \leftarrow L_1 \wedge \cdots \wedge L_m, \phi_2' = \leftarrow M_1 \wedge \cdots \wedge M_n$ *be two standardized apart variants of denials* ϕ_1, ϕ_2. *If* θ *is a mgu of* $\{L_i, \neg M_j\}$ *then the clause*

$$\leftarrow (L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \ldots L_m \wedge M_1 \wedge \cdots \wedge M_{j-1} \wedge M_{j+1} \wedge \cdots \wedge M_n)\theta$$

is called a binary resolvent *of* ϕ_1 *and* ϕ_2 *and* L_i, M_j *are said to be the literals* resolved upon.

The resolution principle as defined in definition 3.2.20 relies on the fact that the literals resolved upon are "complementary" in the sense of logical negation. In definition 3.2.21 the literals resolved upon are not complementary in the same sense. Therefore we prove below that resolution is sound also in the context of range restricted denials with default negation, i.e., if two denials are satisfied in a given standard model M, then their resolvent is also satisfied in M.

Proposition 3.2.22 *Let* ϕ_1, ϕ_2 *be denials and* I *an interpretation such that* $\models_I \phi_1$ *and* $\models_I \phi_2$. *Let* ϕ *be a resolvent of* ϕ_1 *and* ϕ_2. *Then* $\models_I \phi$.

Proof *Assume that* $\phi_1 = \leftarrow L_1 \wedge \cdots \wedge L_m$ *and* $\phi_2 = \leftarrow M_1 \wedge \cdots \wedge M_n$ *are standardized apart (as* ϕ_1' *and* ϕ_2' *in definition 3.2.21) and* θ *is a mgu of* $\{L_i, \neg M_j\}$. *Assume, for now, that* ϕ_1 *and* ϕ_2 *are ground. To satisfy the parent clauses,* I *has to dissatisfy at least one literal in each of them. Let us first consider the case in which* I *dissatisfies* L_i. *Then it cannot dissatisfy* M_j *in* ϕ_2 *(*$L_i = \neg M_j$ *in the ground case) and hence dissatisfies one of the* M_k*'s,* $k \neq j$. *This literal belongs to the resolvent, which is therefore also satisfied. In the other case,* I *satisfies* L_i *and the proof is symmetric.*

Suppose now that ϕ_1 *and* ϕ_2 *are not ground. We show that every ground instance of* ϕ *holds in* I. *Let* σ *be a grounding substitution for* ϕ. *Then, by construction,* $\phi\sigma$ *is a resolvent of* $\phi_1\theta\sigma$ *and* $\phi_2\theta\sigma$. *Both* $\phi_1\theta\sigma$ *and* $\phi_2\theta\sigma$ *hold in* I *and both are ground, so the proof reduces to the previous case. To see that they are ground, consider that* σ *grounds all literals in* $\phi_1\theta$ *and* $\phi_2\theta$, *except perhaps for* $L_i\theta$ *and* $M_j\theta$. *However the denials are range restricted and thus all variables occurring in a negative literal also occur in a positive literal in the body. Suppose* $L_i\theta$ *is the negative literal. Then its variables must also occur in other literals in* $\phi_1\theta$*; but such literals are all grounded by* σ, *so* $L_i\theta$ *is also grounded by* σ. *Now,* $M_j\theta$ *has the same variables as* $L_i\theta$, *so it is also grounded by* σ. □

We note that if the parent clauses are range restricted, then the resolvent is also range restricted [162], since the variables that occur in the positive literal resolved upon are unified with the terms in the negative literal resolved upon, which are either constants, parameters or range bound variables.

Definition 3.2.23 (Derivation) *Let Γ be a constraint theory and ϕ a denial. A* derivation *of ϕ from Γ is a finite sequence of denials $\psi_1, \ldots, \psi_k = \phi$, such that each ψ_i is either in Γ or is a resolvent of two denials in $\{\psi_1, \ldots, \psi_{i-1}\}$ and such that no two resolvents are resolvents of the same denials or variants thereof. If such a derivation exists, we write $\Gamma \vdash_r \phi$. A derivation of the empty clause from Γ is called a* refutation *of Γ.*

Definition 3.2.24 (Deduction) *Let Γ be a constraint theory and ϕ a denial. There is a* deduction *of ϕ from Γ, written $\Gamma \vdash_d \phi$ if there exists a denial ψ such that $\Gamma \vdash_r \phi$ and ψ subsumes ϕ.*

We also refer to the notion of *expansion* [48]: the expansion of a clause consists in replacing every constant or parameter in a database predicate (or variable already appearing elsewhere in database predicates) by a new variable and adding the equality between the new variable and the replaced item. We indicate the expansion of a (set of) denial(s) with a "+" superscript.

Example 3.2.25 *Let $\phi = \leftarrow p(X, a, X)$. Then $\phi^+ = \leftarrow p(X, Y, Z) \wedge Y = a \wedge Z = X$.*

We now describe a resolution-based procedure that limits the size of each resolvent to the size of the biggest denial in Γ.

Definition 3.2.26 *For a constraint theory Γ in $\mathcal{L}_s$ and a denial ϕ in $\mathcal{L}_s$, the notation $\Gamma \vdash_R \phi$ indicates that there is a resolution derivation of a denial ψ from Γ^+ such that in each resolution step the resolvent has at most n literals and ψ^- subsumes ϕ, where n is the number of literals of the largest denial in Γ^+.*

The advantages and disadvantages of such a limit on the size of the resolvents in the resolution proofs will be discussed in section 3.5.2. Informally, the motivation for this choice is that the denials whose size is less than n are the only usable clauses for subsequent elimination of literals by subsumption (and reduction).

Proposition 3.2.27 *$\vdash_R$ is sound and terminates on any input.*

Proof *Soundness follows from soundness of resolution and subsumption. Termination follows from the fact that the set of denials in Γ^+ is finite, so there is only a finite number of clauses of at most n literals that can be generated by resolution steps in Γ^+, as Γ^+ is function-free.* □

Optimization of integrity constraints

The tools introduced in the previous section can be used to compose a procedure that eliminates redundant literals and denials from a given constraint theory Γ assuming that another theory Δ holds. Definition 3.2.28 below introduces an operator that allows us to do this. Informally, $\vdash_R$, subsumption and reduction are used here to approximate entailment; a discussion on how good an approximation is achieved will be provided in section 3.5.2. In the following, the notation $A \sqcup B$ indicates union of disjoint sets, i.e., it denotes $A \cup B$ and indicates that $A \cap B = \emptyset$.

Definition 3.2.28 *Given two constraint theories Δ and Γ in $\mathcal{L}_s$, $\mathsf{Optimize}^{\Delta}_{\mathcal{L}_s}(\Gamma)$ is the result of applying the following rewrite rules on Γ as long as possible. In the following, ϕ, ψ are denials in $\mathcal{L}_s$, Γ' is a constraint theory in $\mathcal{L}_s$.*

$$\begin{array}{lcl} \{\phi\} \sqcup \Gamma' & \Rightarrow & \Gamma' \textit{ if } \phi^- = \textit{true} \\ \{\phi\} \sqcup \Gamma' & \Rightarrow & \Gamma' \textit{ if } (\Gamma' \cup \Delta) \vdash_R \phi \\ \{\phi\} \sqcup \Gamma' & \Rightarrow & \{\phi^-\} \cup \Gamma' \textit{ if } \phi \neq \phi^- \neq \textit{true} \\ \{\phi\} \sqcup \Gamma' & \Rightarrow & \{\psi^-\} \cup \Gamma' \textit{ if } (\{\phi\} \sqcup \Gamma' \cup \Delta) \vdash_R \psi \textit{ and } \psi^- \textit{ strictly subsumes } \phi \end{array}$$

The first two rules attempt the elimination of a whole denial, whereas the last two try to remove literals from a denial[2]. Notice that if the empty clause is produced during the process, then $\mathsf{Optimize}_{\mathcal{L}_s}$ returns *false*, as it subsumes every other denial.

Proposition 3.2.29 *For any two constraint theories Γ and Δ in $\mathcal{L}_s$, the execution of $\mathsf{Optimize}^{\Delta}_{\mathcal{L}_s}(\Gamma)$ terminates. Furthermore the result is in $\mathcal{L}_s$ and $\Gamma \overset{\Delta}{\equiv} \mathsf{Optimize}^{\Delta}_{\mathcal{L}_s}(\Gamma)$.*

Proof *Each condition of applicability in the rules of definition 3.2.28 is decidable, as subsumption, reduction and $\vdash_R$ are. Termination follows from the fact that each rule in the procedure either reduces the number of literals in the constraint theory (first three rules) or replaces a denial with one that strictly subsumes it (last rule), which is obviously a well-founded ordering. Every rule preserves range restriction: the first two rules eliminate a whole denial; reduction preserves range restriction and so does each resolution step. Denial form is also preserved, and therefore the result is in $\mathcal{L}_s$. By soundness of subsumption, reduction and $\vdash_R$, all rules preserve Δ-conditional equivalence.* □

The simplification transformation for $\mathcal{L}_s$ is straightforwardly defined in terms of $\mathsf{After}_{\mathcal{L}_s}$ and $\mathsf{Optimize}_{\mathcal{L}_s}$.

Definition 3.2.30 *Let $S = \langle IDB, \Gamma \rangle \in \mathcal{L}_s$ and U be an update in $\mathcal{L}_s$ with respect to S. Let $\mathsf{Unfold}_{\mathcal{L}_s}(S) = \langle \emptyset, \Gamma' \rangle$. We define*

$$\mathsf{Simp}^{U}_{\mathcal{L}_s}(S) = \mathsf{Optimize}^{\Gamma'}_{\mathcal{L}_s}(\mathsf{After}^{U}_{\mathcal{L}_s}(S)).$$

From the previous results we get immediately the following.

[2]The last rule does not necessarily reduce the number of literals, but note that the expansion of the replacing denial will anyhow contain fewer literals than the expansion of the original one.

Proposition 3.2.31 *Let $S \in \mathcal{L}_{\mathsf{s}}$ and U be an update in $\mathcal{L}_{\mathsf{s}}$ with respect to S. Then $\mathsf{Simp}^U_{\mathcal{L}_{\mathsf{s}}}(S)$ is in $\mathcal{L}_{\mathsf{s}}$ and is a CWP of S with respect to U.*

Example 3.2.32 *Consider again the update and the constraint theory from example 3.2.15 on page 36, where we showed the transformation $\mathsf{After}^U_{\mathcal{L}_{\mathsf{s}}}(\{\phi\})$. In order to obtain $\mathsf{Simp}^U_{\mathcal{L}_{\mathsf{s}}}(\{\phi\})$, we apply the rewrite rules of $\mathsf{Optimize}_{\mathcal{L}_{\mathsf{s}}}$ as follows. The reduction of each denial in $\mathsf{After}^U_{\mathcal{L}_{\mathsf{s}}}(\{\phi\})$ generates the following set.*

$$\begin{array}{lll} \{ & \leftarrow & m(X,Y) \wedge m(X,Z) \wedge Y \neq Z, \\ & \leftarrow & m(\mathbf{a},Y) \wedge Y \neq \mathbf{b}, \\ & \leftarrow & m(\mathbf{a},Z) \wedge \mathbf{b} \neq Z \qquad\qquad \}. \end{array}$$

Then, the third denial is removed, as it is subsumed by the second one[3]*. Similarly, the first constraint is subsumed by ϕ and, thus, removed.*

$$\mathsf{Simp}^U_{\mathcal{L}_{\mathsf{s}}}(\{\phi\}) = \{\leftarrow m(\mathbf{a},Y) \wedge Y \neq \mathbf{b}\}.$$

This result indicates that, for the database to be consistent after update U, husband $\mathbf{a}$ must not be already married to a wife Y different from $\mathbf{b}$.

Example 3.2.33 *We continue example 3.2.32 and suppose now that no duplicate entries are allowed in the database, i.e., $\Delta = \{\leftarrow m(\mathbf{a},\mathbf{b})\}$ is a trusted set of hypotheses when update U is made. Then, using Δ, we can achieve further simplification, since $\Delta^+ = \{\leftarrow m(X,Y) \wedge X = \mathbf{a} \wedge Y = \mathbf{b}\}$ and with a resolution step with the expansion of the previous result, i.e., $\{\leftarrow m(X,Y) \wedge X = \mathbf{a} \wedge Y \neq \mathbf{b}\}$, we obtain*

$$\mathsf{Optimize}^{\Delta}_{\mathcal{L}_{\mathsf{s}}}(\mathsf{Simp}^U_{\mathcal{L}_{\mathsf{s}}}(\{\phi\})) = \{\leftarrow m(\mathbf{a},Y)\},$$

i.e., now we only need to check that $\mathbf{a}$ is not already married.

Example 3.2.34 *Reconsider now the update and the constraint theory from example 3.2.16 on page 36. We have here:*

$$\mathsf{Simp}^U_{\mathcal{L}_{\mathsf{s}}}(\{\phi\}) = \emptyset.$$

This indicates that database integrity cannot be violated by this update.

In the following example we show different combinations of tuple updates for predicates occurring in positive and negative literals. In order to simplify the notation for tuple additions and deletions, we write $p(\vec{\mathbf{a}})$ as a shorthand for the database update $p(\vec{X}) \Leftarrow p(\vec{X}) \vee \vec{X} = \vec{\mathbf{a}}$ and $\neg p(\vec{\mathbf{a}})$ for $p(\vec{X}) \Leftarrow p(\vec{X}) \wedge \vec{X} \neq \vec{\mathbf{a}}$.

Example 3.2.35 *Consider the integrity constraint $\phi = \leftarrow p(X) \wedge q(X) \wedge \neg r(X)$, parameters $\mathbf{a}$ and $\mathbf{b}$ and constants a and b. The following transformations show how different*

[3] Alternatively, the second constraint could be removed instead of the third one, as they subsume one another.

instances of the parameters may modify the result of a parametric simplification.

$$\begin{aligned}
\mathsf{Simp}^{\{p(\mathbf{a}),r(\mathbf{b})\}}_{\mathcal{L}_\mathsf{S}}(\{\phi\}) &= \{\leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a}) \wedge \mathbf{a} \neq \mathbf{b}\} \\
\mathsf{Simp}^{\{p(a),r(b)\}}_{\mathcal{L}_\mathsf{S}}(\{\phi\}) &= \{\leftarrow q(a) \wedge \neg r(a)\} \\
\mathsf{Simp}^{\{p(a),r(a)\}}_{\mathcal{L}_\mathsf{S}}(\{\phi\}) &= \emptyset \\
\mathsf{Simp}^{\{p(\mathbf{a}),\neg r(\mathbf{b})\}}_{\mathcal{L}_\mathsf{S}}(\{\phi\}) &= \{ \leftarrow q(\mathbf{a}) \wedge \neg r(\mathbf{a}), \\
&\qquad \leftarrow p(\mathbf{b}) \wedge q(\mathbf{b}), \\
&\qquad \leftarrow \mathbf{a} = \mathbf{b} \wedge q(\mathbf{a}) \ \} \\
\mathsf{Simp}^{\{p(a),\neg r(b)\}}_{\mathcal{L}_\mathsf{S}}(\{\phi\}) &= \{ \leftarrow q(a) \wedge \neg r(a), \\
&\qquad \leftarrow p(b) \wedge q(b) \ \} \\
\mathsf{Simp}^{\{p(a),\neg r(a)\}}_{\mathcal{L}_\mathsf{S}}(\{\phi\}) &= \{\leftarrow q(a)\}
\end{aligned}$$

Next we show an example of update in which the tuples to be modified are not mentioned explicitly, but rather specified intensionally with a complex defining formula.

Example 3.2.36 *Consider a database of employees $e(EMP, DEPT)$ (an employee works in a department), and skills $s(EMP, SKILL)$ (an employee has a skill). It is required that the computer science personnel (cs) has programming skills (pr): $\phi = \leftarrow e(X, cs) \wedge \neg s(X, pr)$. The following update indicates that all employees of department $\mathbf{a}$ migrate to department $\mathbf{b}$: $U = \{e(X, Y) \Leftarrow (e(X, Y) \wedge Y \neq \mathbf{a}) \vee (e(X, \mathbf{a}) \wedge Y = \mathbf{b})\}$. We have:*

$$\begin{aligned}
\mathsf{After}^U_{\mathcal{L}_\mathsf{S}}(\{\phi\}) &\equiv \{ \leftarrow ((e(X, cs) \wedge cs \neq \mathbf{a}) \vee (e(X, \mathbf{a}) \wedge cs = \mathbf{b})) \wedge \neg s(X, pr) \ \} \\
&\equiv \{ \leftarrow e(X, cs) \wedge cs \neq \mathbf{a} \wedge \neg s(X, pr), \\
&\qquad \leftarrow e(X, \mathbf{a}) \wedge cs = \mathbf{b} \wedge \neg s(X, pr) \ \} \\
\mathsf{Simp}^U_{\mathcal{L}_\mathsf{S}}(\{\phi\}) &= \{ \leftarrow e(X, \mathbf{a}) \wedge cs = \mathbf{b} \wedge \neg s(X, pr) \ \}.
\end{aligned}$$

So, if $\mathbf{b}$ is computer science, each employee in department $\mathbf{a}$ must have programming skills.

Example 3.2.37 *Consider the following constraint theories.*

$$\begin{aligned}
\Gamma_1 &= \{ \leftarrow \neg p(X) \wedge q(X) \wedge r(X), \\
&\qquad \leftarrow p(X) \wedge \neg q(X), \\
&\qquad \leftarrow p(X) \wedge \neg r(X) \ \}, \\
\Gamma_2 &= \{ \leftarrow s(X) \wedge q(X) \wedge r(X) \ \}, \\
\Gamma_3 &= \{ \leftarrow s(X) \wedge p(X) \ \}.
\end{aligned}$$

Let $\Gamma_{1,2} = \Gamma_1 \cup \Gamma_2$ and $\Gamma_{1,3} = \Gamma_1 \cup \Gamma_3$. For the update $U = \{p(a)\}$, the theory obtained in the calculation of $\mathsf{Simp}^U_{\mathcal{L}_\mathsf{S}}(\Gamma_{1,3})$ after all possible reduction and subsumption steps in $\mathsf{Optimize}_{\mathcal{L}_\mathsf{S}}$ is as follows:

$$\{\leftarrow s(a), \leftarrow \neg q(a), \leftarrow \neg r(a)\}.$$

The denial $\leftarrow s(a)$ can be eliminated, as $\Gamma_{1,3} \cup \{\leftarrow \neg q(a), \leftarrow \neg r(a)\} \vdash_R (\leftarrow s(a))$. We have:

$$\mathsf{Simp}^U_{\mathcal{L}_\mathsf{S}}(\Gamma_{1,3}) = \{\leftarrow \neg q(a), \leftarrow \neg r(a)\}.$$

Note that $\Gamma_{1,2} \equiv \Gamma_{1,3}$ *and* $\mathsf{Simp}^U_{\mathcal{L}_S}(\Gamma_{1,2}) = \mathsf{Simp}^U_{\mathcal{L}_S}(\Gamma_{1,3})$*; however, in the calculation of* $\mathsf{Simp}^U_{\mathcal{L}_S}(\Gamma_{1,2})$*, subsumption and reduction are sufficient to obtain the same (minimal) result.*

In examples 3.2.32–3.2.37, it appears that the simplified integrity constraints are minimal in the number of literals and instantiated as much as possible. In examples 3.2.35 and 3.2.36, some parametric simplifications are minimal but could become smaller once the parameters are instantiated, as was shown in example 3.1.8 on page 28.

3.2.4 On pre-tests and post-tests

In section 2.4.2 we introduced the notions of post- and pre-test. Now that we have tools for effectively generating pre-tests, we can take a closer look at the relationship between pre-tests and post-tests.

First, we show with a counter-example that, in general, a pre-test (resp. post-test) cannot be used as a post-test (resp. pre-test).

Example 3.2.38 *Consider the constraint theory* $\Gamma = \{\leftarrow p(a) \wedge q(b)\}$ *and the update* $U = \{p(X) \Leftarrow q(X),\ q(X) \Leftarrow p(X)\}$ *that exchanges* p *and* q*. Then* $\Sigma_1 = \{\leftarrow q(a) \wedge p(b)\}$ *is a pre-test (returned by* $\mathsf{Simp}_{\mathcal{L}_S}$*) but clearly it is not a correct post-test. Consider, e.g., a database* $D = \{p(a), p(b), q(a)\}$*; we have* $D \models \Gamma, D^U \not\models \Gamma, D^U \models \Sigma_1$*, i.e.,* $D^U \models \Sigma_1 \not\Leftrightarrow D^U \models \Gamma$*, although* D *is consistent with* Γ*. Similarly,* $\Sigma_2 = \{\leftarrow p(a) \wedge q(b)\}$ *is a post-test, but not a pre-test.*

We now introduce a restricted class of updates that excludes an update such as U of example 3.2.38.

Definition 3.2.39 *An update* U *is* idempotent *if* $D^U = (D^U)^U$ *for any database* D*.*

For idempotent updates, we can prove that a WP is always a valid post-test.

Proposition 3.2.40 *Let* Γ *be a constraint theory and* U *an idempotent update. Let* Σ *be a WP of* Γ *with respect to* U*. Then* $D^U \models \Sigma \Leftrightarrow D^U \models \Gamma$ *for any database* D*, i.e.,* Σ *is a post-test of* Γ *with respect to* U*.*

Proof *Since* U *is idempotent, i.e.,* $D^U = (D^U)^U$ *for any* D*, we have*

(1) $D^U \models \Gamma \Leftrightarrow (D^U)^U \models \Gamma$ *for any* D*.*

Since Σ *is a WP of* Γ *wrt.* U*, we have*

(2) $D^U \models \Sigma \Leftrightarrow (D^U)^U \models \Gamma$ *for any* D*.*

By transitivity between (1) and (2) we obtain the thesis. □

It is an open question whether, for idempotent updates, any CWP is also a valid post-test. We conjecture that this claim holds, but we were neither able to prove it nor to disprove it.

Conjecture 3.2.41 *Let Γ be a constraint theory and U an idempotent update. Let Σ be a CWP of Γ with respect to U. Then $D^U \models \Sigma \Leftrightarrow D^U \models \Gamma$ for any database D consistent with Γ, i.e., Σ is a post-test of Γ with respect to U.*

We can easily prove that, if D is a consistent state, there is no update U such that $D \models \Sigma$, $D^U \not\models \Sigma$, and $D^U \models \Gamma$. However we were not able to exclude the existence of an update U such that $D \not\models \Sigma$, $D^U \models \Sigma$, and $D^U \not\models \Gamma$.

On the other hand, we can show with a counter-example that the reverse does not hold, i.e., for idempotent updates, there are post-tests that are not valid pre-tests.

Example 3.2.42 *Consider the constraint theory $\Gamma = \{\leftarrow p(X,Y) \wedge p(Y,X)\}$ and the update $U = \{p(\mathbf{a},\mathbf{b})\}$. Clearly, the theory $\Sigma = \{\leftarrow p(\mathbf{b},\mathbf{a})\}$ is a post-test, but not a correct pre-test. Suppose, e.g., that $\mathbf{a} = \mathbf{b} = c$; then Σ can succeed even though the updated state is necessarily inconsistent. This is due to the presence of parameters. Note that the theory returned by $\mathsf{Simp}^U_{\mathcal{L}_\mathsf{S}}(\Gamma)$, which is a pre-test, is $\Sigma' = \{\leftarrow p(\mathbf{b},\mathbf{a}), \leftarrow \mathbf{b} = \mathbf{a}\}$, that is also a valid post-test.*

To substantiate with empirical evidence the claim of conjecture 3.2.41 we used a generator of random constraint theories and updates of the following form

$$\{p_1(\vec{c}_1), \ldots, p_n(\vec{c}_n), \neg q_1(\vec{d}_1), \ldots, \neg q_m(\vec{d}_m)\}, \tag{3.2}$$

where $p_1, \ldots, p_n, q_1, \ldots, q_m$ are (not necessarily distinct) predicates, $\vec{c}_1, \ldots, \vec{c}_n$, $\vec{d}_1, \ldots, \vec{d}_m$ are vectors of constants, $m + n > 0$, and there are no i, j such that $p_i(\vec{c}_i) = q_j(\vec{d}_j)$. Our tests showed that all simplifications produced by $\mathsf{Simp}_{\mathcal{L}_\mathsf{S}}$ in these cases were also valid post-tests. This would suggest that $\mathsf{Simp}_{\mathcal{L}_\mathsf{S}}$ can also be used to produce post-tests for updates of the form (3.2), a claim that follows directly from conjecture 3.2.41.

Another interesting aspect regarding pre-tests and post-tests is whether their evaluation is at all affected by the update. Proposition 3.2.40 immediately implies that the evaluation of a WP is not affected by the update, as stated in the following corollary.

Corollary 3.2.43 *Let Σ be a WP of a constraint theory Γ with respect to an idempotent update U. Then $D \models \Sigma \Leftrightarrow D^U \models \Sigma$ for any D.*

Proof *Since Σ is a WP of Γ wrt. U, we have*

$$D \models \Sigma \Leftrightarrow D^U \models \Gamma \text{ for any } D,$$

and, by transitivity with the claim of proposition 3.2.40, we have the thesis. □

This does not hold in general for CWPs, as demonstrated in the following counter-example, i.e., the evaluation of a CWP may be affected by the update.

Example 3.2.44 *Consider a constraint theory $\Gamma = \{\leftarrow p(X) \wedge q(X) \wedge r(X),\ \leftarrow \neg q(X) \wedge r(X)\}$ and an update $U = \{p(a)\}$. A CWP of Γ wrt. U is $\Sigma = \{\leftarrow q(a) \wedge r(a),\ \leftarrow p(X) \wedge r(X)\}$, which is obtained from the WP $\{\leftarrow p(X) \wedge q(X) \wedge r(X),\ \leftarrow q(a) \wedge r(a),\ \leftarrow \neg q(X) \wedge r(X)\}$ by removing the literal $q(X)$ in the first denial by a resolution step with the last denial, and then by removing the last denial (which also occurs in Γ). The atom $p(a)$ may affect the evaluation of Σ. Consider, e.g., $D = \{r(a)\}$; then $D \models \Sigma$ but $D^U \not\models \Sigma$.*

However, example 3.2.44 is not a counter-example for the claim of conjecture 3.2.41, since the initial state D is not consistent with Γ.

The following table summarizes the results and open problems presented in this subsection. We indicate with $\text{pre}^U(\Gamma)$ a pre-test of constraint theory Γ with respect to U, with $\text{post}^U(\Gamma)$ a post-test of Γ with respect to U, and with $\text{WP}^U(\Gamma)$ a WP of Γ with respect to U. We enclose between curly brackets to indicate the set of all such constraint theories. Known results are in boldface, whereas conjectures are in italics and followed by a question mark.

for any Γ and for any D	any U	U idempotent	U (3.2)
$\{\text{pre}^U(\Gamma)\} \subseteq \{\text{post}^U(\Gamma)\}$?	**no**	*yes?*	*yes?*
$\{\text{post}^U(\Gamma)\} \subseteq \{\text{pre}^U(\Gamma)\}$?	**no**	**no**	*?*
$\{\text{WP}^U(\Gamma)\} \subseteq \{\text{post}^U(\Gamma)\}$?	**no**	**yes**	**yes**
$D \models \text{pre}^U(\Gamma) \Leftrightarrow D^U \models \text{pre}^U(\Gamma)$?	**no**	**no**	**no**
$D \models \text{WP}^U(\Gamma) \Leftrightarrow D^U \models \text{WP}^U(\Gamma)$?	**no**	**yes**	**yes**

3.3 On the equivalence of ideal simplification and query containment

There is a direct correspondence between the problem of ideal simplification (definition 3.1.6 on page 27) and query containment (definition 2.2.23 on page 18). In order to characterize this relationship, we introduce the following lemma.

Lemma 3.3.1 *Consider the following constraint theories*

$$\begin{aligned}
&\Gamma = \{\leftarrow A_1, \ldots, \leftarrow A_n\},\\
&\Gamma_1 = \Gamma \cup \{\leftarrow p(\vec{X}), \leftarrow q(\vec{X})\},\\
&\Gamma_2 = \Gamma \cup \{\leftarrow q(\vec{X})\},\\
&\Gamma^r = \{\leftarrow A_1 \wedge r, \ldots, \leftarrow A_n \wedge r\},\\
&\Gamma_1^r = \Gamma^r \cup \{\leftarrow p(\vec{X}) \wedge r, \leftarrow q(\vec{X}) \wedge r\},\\
&\Gamma_2^r = \Gamma^r \cup \{\leftarrow q(\vec{X}) \wedge r\}
\end{aligned}$$

and the schemata

$$S_1 = \langle I, \Gamma_1\rangle, S_2 = \langle I, \Gamma_2\rangle, S_1^r = \langle I, \Gamma_1^r\rangle, S_2^r = \langle I, \Gamma_2^r\rangle,$$

where $\Leftarrow p(\vec{X})$ and $\Leftarrow q(\vec{X})$ are queries, r is a nullary predicate not occurring in S_1 or S_2 and the A_i's are conjunctions of literals. Assume Simp *is an ideal simplification procedure and let U be the update $\{r \Leftarrow true\}$ and let $T_i = \langle I_i, \Sigma_i\rangle = \mathsf{Simp}^U(S_i^r)$ for $i = 1, 2$. Then $T_1 = T_2$ iff $S_1 \equiv S_2$.*

Proof *Preliminary considerations.*
T_i is a CWP of S_i^r with respect to U for $i = 1, 2$, i.e.,

(1) $D \models \Sigma_i$ iff $D^U \models \Gamma_i^r$ for any database D based on $I_i \cup I$ and consistent with Γ_i^r.

If part. If r is false then Γ_1^r and Γ_2^r are both true; if r is true then $S_1^r \equiv S_1$ and $S_2^r \equiv S_2$ (and $S_1 \equiv S_2$ by hypothesis). Therefore $S_1^r \equiv S_2^r$, i.e.,

(2) $D \models \Gamma_1^r$ iff $D \models \Gamma_2^r$ for any D based on I.

By transitivity, we obtain from (1) and (2) that T_1 is also a CWP of S_2^r wrt U (and, similarly, T_2 is also a CWP of S_1^r wrt U). Assume now, by contradiction, that $T_1 \neq T_2$. If, for example, $T_1 \prec T_2$, then T_2 would not be an ideal simplification of S_2^r wrt U, against the hypothesis that Simp *was ideal.*
Only if part. Assume, by contradiction, that $S_1 \not\equiv S_2$. Then there exists a state D' such that

(3) $D' \not\models \Gamma_1$ and $D' \models \Gamma_2$[4].

Consider a state $D'' = D' \setminus \{r\}$. Both Γ_1^r and Γ_2^r hold in D'' (since r is false in it) and we have $D''^U = D'$. But since T_i, for $i = 1, 2$, is a CWP of S_i^r wrt U and $D'' \models \Gamma_i^r$, we have, from (1),

(4) $D'' \models \Sigma_i$ iff $D' \models \Gamma_i^r$,

and, since $\{r\} \in D'$, we have

(5) $D' \models \Gamma_i^r$ iff $D' \models \Gamma_i$.

Now, using (3), (4) and (5), we obtain $D'' \not\models \Sigma_1$. Similarly, we obtain $D'' \models \Sigma_2$. But this is a contradiction, since $\Sigma_1 = \Sigma_2$ by hypothesis. □

Lemma 3.3.2 *Let $\Gamma, \Gamma_1, \Gamma_2, S_1, S_2$ be as in lemma 3.3.1 and consider a schema $S = \langle I, \Gamma \rangle$. Then $S : p \subseteq q$ iff $S_1 \equiv S_2$.*

Proof *By definition of answer and QC, the problem $S : p \subseteq q$ is equivalent to the following*

(1) for any ground tuple $\vec{a}$, $D \models p(\vec{a})$ implies $D \models q(\vec{a})$, in any database D based on I and consistent with Γ.

Similarly, the equivalence between S_1 and S_2 indicates that,

(2) for any ground tuple $\vec{a}$, $D \models (\Gamma \wedge \leftarrow p(\vec{a}) \wedge \leftarrow q(\vec{a}))$ iff $D \models (\Gamma \wedge \leftarrow q(\vec{a}))$, in any database D based on I.

Clearly, (2) always holds if $D \not\models \Gamma$. If $D \models \Gamma$, (2) amounts to the following.

(3) for any ground tuple $\vec{a}$, $D \models (\leftarrow p(\vec{a}) \wedge \leftarrow q(\vec{a}))$ iff $D \models (\leftarrow q(\vec{a}))$, in any database D based on I.

[4] We can already exclude the case where $D' \models \Gamma_1$ and $D' \not\models \Gamma_2$ since $\Gamma_1 \models \Gamma_2$ by construction.

i.e., (2) and (3) are equivalent. But also (1) and (3) are equivalent, as can be seen by noticing that the truth tables of $p \rightarrow q$ *and* $(\neg p \wedge \neg q) \leftrightarrow \neg q$ *are identical. Therefore (1) and (2) are equivalent.* □

We can now state the main theorem of this section, which shows the correspondence between query containment and ideal simplification. We implicitly assumed so far that nullary predicates are available in the predicate language at hand[5]. In theorem 3.3.3 we will also assume that, in any given language $\mathcal{L}$, whenever some schema has an integrity constraint of the form $\leftarrow A$, then the schema with *IDB* extended with $p^A \leftarrow A$ is also in $\mathcal{L}$. Although, strictly speaking, this is not true of all database languages, it is convenient to assume that any reasonably expressive database language enjoys these properties. The theorem below refers to such database languages.

Theorem 3.3.3 *For any database language* $\mathcal{L}$*, QC is decidable in* $\mathcal{L}$ *if and only if* $\mathcal{L}$ *admits an ideal simplification procedure.*

Proof *If part. Assume an ideal simplification procedure* Simp *for* $\mathcal{L}$ *and consider the QC problem* $S : p \subseteq q$ *over a schema* $S = \langle I, \Gamma \rangle$*. Consider the schemata and constraint theories defined in lemma 3.3.1. By lemma 3.3.2,* $S : p \subseteq q$ *iff* $S_1 \equiv S_2$*. But the last equivalence can be decided with an ideal simplification procedure, as shown in lemma 3.3.1, and therefore also the QC problem.*

Only if part. For simplicity, we start considering the case with parameter-free updates. Let $S_A = \langle IDB_A, \Gamma_A \rangle = \mathsf{After}^U(S)$ *for some parameter-free update* U*. By proposition 3.2.9,* S_A *is a CWP of* $S = \langle IDB, \Gamma \rangle$ *wrt* U*. Let* $S_B = \langle IDB_B, \Gamma_B \rangle$[6] *be any schema such that* $S_B \preceq S_A$*. By definition 3.1.2 (of CWP) and transitivity of* $\Leftrightarrow$*,* S_B *is a CWP of* S *wrt* U *iff the following holds:*

$$D \models \Gamma_A \text{ iff } D \models \Gamma_B \quad \text{for every } D \text{ based on } IDB_A \cup IDB_B \text{ such that } D \models \Gamma.$$

This still holds after adding the following IDB_{q^A} *to* IDB_A *and* IDB_{q^B} *to* IDB_B*:*

$$IDB_{q^A} = \{q^A \leftarrow \phi_1, \cdots, q^A \leftarrow \phi_n\},$$
$$IDB_{q^B} = \{q^B \leftarrow \psi_1, \cdots, q^B \leftarrow \psi_m\},$$

where $\Gamma_A = \{\leftarrow \phi_1, \cdots, \leftarrow \phi_n\}$*,* $\Gamma_B = \{\leftarrow \psi_1, \cdots, \leftarrow \psi_m\}$ *and* q^A*,* q^B *are nullary predicates not occurring in* S_A *or* S_B*. These definitions state that* q^A *holds iff* Γ_A *does not hold, and similarly for* q^B *and* Γ_B*. Let* $IDB' = IDB_A \cup IDB_B \cup IDB_{q^A} \cup IDB_{q^B}$*; then* S_B *is a CWP of* S *wrt* U *iff*

$$D \models \Gamma_A \text{ iff } D \models \Gamma_B \quad \text{for every } D \text{ based on } IDB' \text{ such that } D \models \Gamma.$$

But the if and, respectively, only-if parts of this condition are the query containment problems $S' : q^A \subseteq q^B$ *and* $S' : q^B \subseteq q^A$ *for the schema* $S' = \langle IDB', \Gamma \rangle$*. By properties 2 and 4 of definition 3.1.5 we can enumerate all schemata* S_B *such that* $S_B \preceq S_A$ *and test*

[5] Alternatively, we could have used (non-nullary) ground atoms instead of nullary predicates.

[6] We can assume without loss of generality that S_B is compatible with S_A, for, if it is not, the predicates in $pred(IDB_B \setminus IDB_A)$ can be renamed.

the two conditions above by query containment. This process is then repeated until the minimal CWP is found. This provides an ideal simplification procedure for the parameter-free case.

Suppose now that U contains n different parameters. The construction of IDB' is done as in the non-parametric case, but this time IDB' also contains parameters. The two containment problems above can be decided for every instance of these parameters: the two QC relationships above hold for the parametric case iff they do for every instance of the parameters. Furthermore, we only need to consider a finite number of instances. To see this, let C be a set containing all constants occurring in IDB' plus n (fixed) constants not occurring in IDB'. We only need to consider all possible parametric instances mapping the parameters in IDB' to constants in C. Considering other mappings (to constants not in C) amounts to reformulating the same containment problem modulo renaming of symbols. Since C is finite, the described procedure is terminating. □

Query containment has been studied extensively in the literature, including identification of a number of decidable sub-cases. However, the sort of procedure indicated in the proof is unlikely to be applicable in practice, since many CWP candidates will be generated in a typical case. A different strategy to achieve ideal simplifications is described in section 3.4.2 below. It follows immediately from proposition 3.1.12 on page 29, in which simplification was obtained in terms of optimization, and the if part of theorem 3.3.3 that no ideal optimization procedure exists for a language for which query containment is undecidable.

Analogously to the only-if part of theorem 3.3.3, it follows that an optimization procedure can be constructed from a query containment decision procedure (if it exists) using enumeration.

Theorem 3.3.4 *There exists an ideal optimization procedure for a language $\mathcal{L}$ if and only if query containment is decidable in $\mathcal{L}$.*

Proof *The only-if part holds as, if there is an ideal optimization procedure, then, by proposition 3.1.12, there also exists an ideal simplification procedure; then, by theorem 3.3.3, query containment must be decidable in $\mathcal{L}$. For the other part, it suffices to note that, given the schema S to be optimized with respect to schema S_Δ, we can enumerate all schemata S' such that $S' \prec S$, decide $S' \overset{S_\Delta}{\equiv} S$ by query containment as in the proof of theorem 3.3.3 and then select a minimal one among them.* □

3.4 Ordering and minimality

3.4.1 Ordering and efficiency

In order to characterize a transformed integrity constraint as an optimal simplification, it must represent a minimum in some ordering that reflects the effort of actually evaluating it. This can only be an estimate, as the actual execution times depend on the database state, which is not available at the time of the simplification process. Furthermore, it

is highly dependent on the applied database technology that may perform optimizations that cannot be included in a general definition. We restrict our discourse in this section to constraint theories in which only extensional predicates may occur, i.e., schemata with empty *IDB*, as was the case for unfolded theories in $\mathcal{L}_s$.

To find an optimal simplification means then to choose among all the conditionally equivalent CWPs according to an optimality criterion. Several different criteria can be defined. A natural choice is a syntactic order based on the number of literals: the optimal theories are those in which this number is minimal (and when the number of literals is the same, another standard ordering, such as the alphabetical ordering, is used). This ordering, indicated as $\preceq_\ell$, may appear a bit coarse, as the number of literals in, say, $\leftarrow 1 = 2$, $\leftarrow p(a)$, and $\leftarrow p(X)$ is the same. However, it should be kept in mind that it applies within a class of constraint theories that are pairwise conditionally equivalent (with respect to a given theory).

Another possibility is to define a semantic order based on the *weakness* of a theory. We say that a theory Γ_1 is weaker than a theory Γ_2 if $D \models \Gamma_1$ for all database states D for which $D \models \Gamma_2$ (and Γ_2 is said to be stronger than Γ_1). The interest of a weakest constraint theory is that it typically contains fewer constraints to check than a stronger theory (adding constraints means strengthening the theory) and a weaker constraint is more likely to evaluate to *true* than a stronger one, and therefore may be checked faster with an optimistic approach. We indicate this ordering as $\preceq_w$. Given two constraint theories Γ and Δ, it trivially follows from the definition of conditional equivalence (definition 3.1.10) that the strongest theory Σ, such that $\Sigma \overset{\Delta}{\equiv} \Gamma$, is $\Gamma \cup \Delta$; similarly, the weakest one is a theory equivalent to $\Gamma \vee \neg\Delta$, provided that this formula can be expressed as a set of denials. However, testing whether, for two given theories Γ and Σ, we have $\Gamma \preceq_w \Sigma$ corresponds to checking entailment, which is undecidable in general (see, e.g., [3]); therefore $\preceq_w$ is not an enumerative ordering according to the second point of definition 3.1.5.

Another interesting ordering is based on the notion of resource set, i.e., the set of extensional atoms that affect the evaluation of a given constraint theory: the smaller the resource set, the better the CWP[7]. This notion is formalized below.

Definition 3.4.1 (Uncovered set) *Let $S = \langle IDB, \Gamma \rangle$ be a schema and $\mathcal{R}$ a set of extensional atoms. Then $\mathcal{R}$ is an* uncovered set *for Γ whenever*

$$D \models \Gamma \Leftrightarrow D' \models \Gamma$$

for any two databases D, D' based on IDB with extensional parts, resp., E, E' such that $E \cap \mathcal{R} = E' \cap \mathcal{R}$. If, furthermore, Γ admits no other uncovered set $\mathcal{R}' \subset \mathcal{R}$, $\mathcal{R}$ is a minimal uncovered set *for Γ.*

Proposition 3.4.2 *For any constraint theory Γ there exists a unique minimal uncovered set.*

The minimal uncovered set of a constraint theory Γ is called the *resource set* of Γ and is indicated as $\mathcal{R}(\Gamma)$.

[7] In [59] the notion of "cover" was used instead (the bigger the cover, the better), as opposed to the dual notion of uncovered set which is used here.

Example 3.4.3 *Let* $\Gamma = \{\leftarrow p(X) \wedge X \neq a\}$ *be a constraint theory,* p *an extensional predicate, and* P *the set of all ground atoms constructible with predicate* p*. The resource set of* Γ *is the set* $P \setminus \{p(a)\}$*. Any set of ground extensional atoms that is a (strict) superset of* $P \setminus \{p(a)\}$ *is a (non-minimal) uncovered set for* Γ*.*

Consider now $\Sigma = \{\leftarrow p(X) \wedge \neg q(X),\ \leftarrow p(a)\}$ *and let* Q *be the set of all ground atoms constructible with predicate* q*. The resource set of* Σ *is the set* $(P \cup Q) \setminus \{q(a)\}$*, since the evaluation of* Σ *is not affected by the presence of* $q(a)$*, whereas it is affected by all other tuples in* Q *and all tuples in* P*.*

In order to prove proposition 3.4.2, we introduce lemma 3.4.4 below.

Lemma 3.4.4 *Consider a schema* $S = \langle IDB, \Gamma \rangle$*. A set of extensional atoms* $\mathcal{R}$ *is an uncovered set of* Γ *iff* Γ *is indifferent to any single extensional atom* A *not in* $\mathcal{R}$*, i.e., iff*

$$\langle IDB, (F \setminus \{A\}) \rangle \models \Gamma \text{ iff } \langle IDB, (F \cup \{A\}) \rangle \models \Gamma$$

for any extensional database F *and for any atom* $A \notin \mathcal{R}$*.*

Proof
Only-if part
Assume $\mathcal{R}$ *is an uncovered set of* Γ *and consider any two sets of extensional atoms* E *and* E' *that only differ by an atom* $A \notin \mathcal{R}$*, i.e.,*

$$E = F \cup \{A\} \text{ and } E' = F \setminus \{A\},$$

where F *is a set of extensional atoms. Clearly,* $E \cap \mathcal{R} = E' \cap \mathcal{R}$ *and, therefore, by definition of uncovered set, we have the thesis.*
If part
Assume Γ *is indifferent to any atom* $A \notin \mathcal{R}$ *and consider two sets of extensional atoms* E *and* E' *such that* $E \cap \mathcal{R} = E' \cap \mathcal{R}$*. Then* E *and* E' *differ by a finite number of atoms that do not belong to* $\mathcal{R}$*. In other words,* $E' = E \cup E^+ \setminus E^-$*, where* $E^+ \cap E^- = E^+ \cap \mathcal{R} = E^- \cap \mathcal{R} = \emptyset$*. But since* Γ *is indifferent to any single extensional atom* $A \notin \mathcal{R}$*, we can add (resp. remove) any single atom in* E^+ *(resp.* E^-*) at a time without affecting the evaluation of* Γ*. Hence the conclusion.* □

Corollary 3.4.5 *Let* Γ *be a constraint theory and* $\mathcal{R}_1$*,* $\mathcal{R}_2$ *two uncovered sets for* Γ*. Then their intersection* $\mathcal{R}_1 \cap \mathcal{R}_2$ *is also an uncovered set for* Γ*.*

Proof *For lemma 3.4.4,* Γ *is indifferent to all extensional atoms in* $\mathcal{B} \setminus \mathcal{R}_1$ *as well as of all extensional atoms in* $\mathcal{B} \setminus \mathcal{R}_2$*, and thus of all extensional atoms in their union* $\mathcal{B} \setminus (\mathcal{R}_1 \cap \mathcal{R}_2)$*. So* $\mathcal{R}_1 \cap \mathcal{R}_2$ *is also an uncovered set.* □

Now proposition 3.4.2 can be proved as follows.
Proof *Suppose that there exist two minimal uncovered sets* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *such that* $\mathcal{R}_1 \neq \mathcal{R}_2$*. But then for corollary 3.4.5 their intersection* $\mathcal{R}_1 \cap \mathcal{R}_2$ *is also an uncovered set and therefore at least one of them is not minimal, against the hypotheses.* □

In the ordering based on resource sets, a theory Γ_1 precedes another theory Γ_2, indicated $\Gamma_1 \preceq_r \Gamma_2$, whenever $\mathcal{R}(\Gamma_1) \subseteq \mathcal{R}(\Gamma_2)$. This notion is particularly relevant in the context of concurrent database systems. In chapter 5 we will discuss a schedule construction policy that guarantees conflict serializability as well as correctness. If a simplification procedure for integrity constraints which is ideal in the $\preceq_r$ ordering is available, then the amount of database resources that need to be locked for this to happen is minimal.

It is possible to find examples of simplification where the optimal constraint theories found according to these three criteria (minimal resource set, minimal number of literals, semantic weakness) differ.

Example 3.4.6 *Consider the following constraint theories:*

$$\begin{array}{rcl} \Gamma & = & \{\leftarrow p(X),\ \leftarrow q(a)\}, \\ \Delta & = & \{\leftarrow p(a),\ \leftarrow q(a)\}. \end{array}$$

Among the theories that are Δ-conditionally equivalent to Γ, the minimal ones with respect to the described orderings are shown in the following table.

$\preceq_\ell$-*minimal*	$\preceq_w$-*minimal*	$\preceq_r$-*minimal*
$\{\leftarrow p(X)\}$	$\{\leftarrow p(X) \wedge X \neq a \wedge \neg q(a)\}$	$\{\leftarrow p(X) \wedge X \neq a\}$

The first column has been calculated by removing $\leftarrow q(a)$ from Γ as it is true in Δ and the only smaller theory is $\emptyset$, which is not Δ-conditionally equivalent to Γ. The second column corresponds to the formula $\Gamma \vee \neg\Delta$. The last column is derived from $\leftarrow p(X)$ (first column), whose resource set is the entire relation p; the resource set can be reduced by removing the tuple $p(a)$ from it, whose truth value is already constrained (to be false) by Δ.

We note that the constraint theories indicated in the two last columns of example 3.4.6 are arbitrary, as any of the infinitely many equivalent constraint theories (obtained, e.g., by inserting tautologies in it) would be minimal in the respective ordering. Thus, $\preceq_w$ and $\preceq_r$ are not enumerative, as they do not comply with the fourth requirement of definition 3.1.5[8].

We stress, however, that, with respect to efficient query evaluation, each criterion can only approximate optimality. For example, a syntactically minimal query does not necessarily evaluate faster than an equivalent non-minimal query in all database states; the amount of computation required to answer a query can be reduced, for instance, by adding a join with a very small relation. Several refinements of the syntactic order $\preceq_\ell$ described in this section can be considered, such as preferring more specific constraints to more general ones. However, for all such improvements there will be cases in which efficiency is not measured precisely. For example, $\leftarrow p(X) \wedge q(Y)$ is likely to be evaluated faster than the more specific $\leftarrow p(X) \wedge q(X)$, as the former can be checked by verifying that either p or q are empty, whereas the latter introduces a join that potentially requires that all tuples in p be looked up in q. Even if we limit such criterion to the preference of ground literals to non-ground ones, we still do not capture the notion of efficiency correctly. For

[8]However, one could of course choose to represent the whole class of equivalent constraint theories with the syntactically minimal one, so that the only possible choices in the table of example 3.4.6 would be those we have indicated.

example, $\leftarrow p(X)$ will typically run faster than $\leftarrow p(a)$, as for the former it is sufficient to verify that p is empty, whereas for the latter a lookup in p is needed. However, it can be argued that a syntactic ordering such as $\preceq_\ell$ captures efficiency for *most* cases, as will be demonstrated in the experimental tests discussed in chapter 6. Furthermore, the simplification procedure also conforms to the strategy of specializing integrity constraint as much as possible, in that variable/constant equalities are removed by substituting the variable by the constant. So, for example, a denial such as $\phi = \leftarrow X = a \wedge p(X,Y) \wedge q(Y)$ is not transformed into $\leftarrow p(X,Y) \wedge q(Y)$,[9] which has fewer literals but is arguably less efficient to evaluate than ϕ, but to $\leftarrow p(a,Y) \wedge q(Y)$, which contains fewer literals *and* is more specialized than ϕ.

In the literature, most methods do not explicitly refer to any particular ordering; some refer to syntactic minimality criteria such as $\preceq_\ell$ [47, 93, 94]; others [102, 143, 70] use resource sets (sometimes called *checking space*).

3.4.2 Achieving minimal theories

In the remainder of the thesis, unless differently stated, we refer to the $\preceq_\ell$ ordering, that has a clear enumeration procedure for all constraint theories. However, enumerating theories, although theoretically possible, might be practically unfeasible. We describe here another strategy that tries to progressively eliminate constraints and parts of constraints from a starting, possibly non-optimal simplification, such as the one returned by After.

A local minimum for a set of denials is recursively defined below. We recall that, given two denials ϕ and ψ, ϕ is a subclause of ψ iff there is a renaming ρ such that each literal in $\phi\rho$ occurs in ψ.

Definition 3.4.7 *A theory Σ is* locally below *a theory Γ (written $\Sigma \prec_{loc} \Gamma$) whenever*

1. *$\Sigma = \emptyset$ and $\Gamma \neq \emptyset$ or*
2. *$\Sigma = \{\phi\} \sqcup \Sigma'$, $\Gamma = \{\psi\} \sqcup \Gamma'$, ϕ is a subclause of ψ and ($\Sigma' \prec_{loc} \Gamma'$ or $\Sigma' = \Gamma'$)*

where ϕ and ψ are denials. Σ is a local *(resp.* global*)* minimum *of Γ with respect to a constraint theory Δ if $\Sigma \stackrel{\Delta}{\equiv} \Gamma$ and there is no other theory $\Sigma'' \stackrel{\Delta}{\equiv} \Gamma$ such that $\Sigma'' \prec_{loc} \Sigma$ (resp. $\Sigma'' \prec_\ell \Sigma$).*

A general procedure to find a local minimum of Γ with respect to Δ consists in repeating the following steps as long as possible.

Procedure 3.4.8

1. *If for a denial $\phi \in \Gamma$ it holds that $\Delta \cup (\Gamma \setminus \phi) \models \phi$ then ϕ is removed from Γ.*
2. *If for a denial $\leftarrow L_1 \wedge \cdots \wedge L_n = \phi \in \Gamma$ it holds that $\Delta \cup \Gamma \models \leftarrow L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n = \psi$ for some i such that $1 \leq i \leq n$ then ϕ is replaced by ψ.*

[9] Unless a constraint such as $\leftarrow X \neq a \wedge p(X,Y)$ is known to hold, which could then be used by a query optimizer to evaluate $\leftarrow p(X,Y) \wedge q(Y)$ as fast as $\leftarrow p(a,Y) \wedge q(Y)$.

Each step preserves conditional equivalence and the strategy guarantees that a local minimum is found, as, when the procedure stops, no denial or literal can be removed from the resulting theory. Furthermore, if it was possible to remove more than a literal or denial at a time, then it would also be possible to remove a single literal or denial at a time.

Example 3.4.9 *Consider the following constraint theories from example 3.2.37.*

$$\begin{array}{rcll} \Delta & = & \{ & \leftarrow \neg p(X) \wedge q(X) \wedge r(X), \\ & & & \leftarrow p(X) \wedge \neg q(X), \\ & & & \leftarrow p(X) \wedge \neg r(X) \qquad \}, \\ \Gamma & = & \{ & \leftarrow s(X) \wedge q(X) \wedge r(X) \qquad \}, \\ \Sigma & = & \{ & \leftarrow s(X) \wedge p(X) \qquad \}. \end{array}$$

We have that $\Sigma \stackrel{\Delta}{\equiv} \Gamma$*, as* Δ *is clearly an encoding of the equivalence between* $p(X)$ *and* $q(X) \wedge r(X)$*. Both* Γ *and* Σ *are local minima of* $\Gamma \cup \Sigma$ *with respect to* Δ *and only* Σ *is a global minimum.*

In practice there is often only one local minimum, which coincides with the global minimum. However, when particular dependencies are encoded in the integrity constraints, such as equivalences between (sets of) predicates, like in example 3.4.9, then there may be several local minima. On the other hand, by definition 3.1.5, the global minimum is unique.

The procedure depicted in this section is, however, based on entailment, which is in general undecidable; furthermore, sound and complete proof procedures, based, e.g., on resolution, are not guaranteed to terminate[10].

The simplification framework described in section 3.2 implements a practically relevant approximation of this strategy in which entailment is replaced by specialized sound and terminating proof procedures. In particular, the first two rewrite rules in the definition of $\mathsf{Optimize}_{\mathcal{L}_\mathsf{s}}$ (3.2.28 on page 41) implement step 1, whereas the last two implement step 2.

3.5 Ideality and $\mathsf{Simp}_{\mathcal{L}_\mathsf{s}}$

Before discussing how well $\mathsf{Simp}_{\mathcal{L}_\mathsf{s}}$ approximates the described simplification strategy for finding a local minimum, we emphasize the completeness of resolution and subcases thereof.

3.5.1 Completeness of resolution

As shown in proposition 3.2.22 on page 39, resolution is sound; this property was used in the construction of $\mathsf{Simp}_{\mathcal{L}_\mathsf{s}}$. Resolution in the context of clauses with logical negation is also *refutation-complete*, in the sense specified below.

[10] In principle, entailment can be implemented by any general theorem prover, properly extended with a time-out facility to ensure termination. This may provide a procedure that produces minimal or close-to-minimal theories, but generality and quality would be problematic to characterize.

Theorem 3.5.1 *[146] Resolution is refutation-complete, i.e., if* Γ *is an unsatisfiable set of clauses then there exists a refutation of* Γ.

However, we applied resolution in the context of denials with default negation. To see that refutation-completeness also holds in this context, we note that the notion of unsatisfiability of a set of clauses with logical negation coincides with unsatisfiability of a set of denials with default negation. Indeed, a set is unsatisfiable if it admits no model. In the setting of denials with default negation, a set Γ is unsatisfiable if no standard model (for any possible attached database) is a model of Γ, i.e., if Γ admits no model. The latter can thus be checked via standard resolution by syntactically replacing default negation with logical negation. Therefore resolution applied to denials with default negation is also refutation-complete, i.e., if Γ is an unsatisfiable set of denials with default negation then there exists a refutation of Γ.

Thanks to theorem 3.5.1, proofs can be performed by *reductio ad absurdum*, i.e., to prove ϕ from Γ, one checks whether there is a refutation of $\Gamma \cup \neg\phi$. Of course, $\neg\phi$ has to be transformed into clausal form for the proof to be made by resolution. It is possible to transform any first-order formula F into an equivalent formula G expressed in *negation normal form* (NNF), i.e., a formula whose connectives are only $\wedge, \vee, \neg$ and $\neg$ can only occur in front of atoms. An NNF formula can then be transformed into *prenex normal form* (PNF) by moving all the quantifiers outwards. Finally, from a PNF formula we can eliminate the existential quantifiers by performing the so-called *skolemization*: if $\exists X$ occurs in G in the scope of universal quantifiers $\forall Y_1, \ldots, \forall Y_m$, then in $sk(G)$ the $\exists X$ is removed and X is replaced by $f(Y_1, \ldots, Y_m)$, where f is a new m-ary function symbol not occurring in G, called a Skolem function. Note that, if $\exists X$ is not in the scope of universal quantifiers, then f is a nullary function symbol, i.e., a (Skolem) constant. In general $sk(G)$ is not equivalent to G; however, $sk(G)$ is satisfiable iff G is. The transformation from $sk(G)$ to clausal form is trivial, and this means that skolemized formulas can be used in proofs by resolution. A detailed explanation of skolemization and its application to resolution can be found in [82].

With the addition of subsumption, completeness of resolution can be stated directly using deduction ($\vdash_d$) by the following theorem.

Theorem 3.5.2 (Subsumption theorem [138]) *Let* Γ *be a constraint theory and* ϕ *a denial. Then* $\Gamma \models \phi$ *iff* $\Gamma \vdash_d \phi$.

Specialized versions of resolution can be used to carry out proofs more efficiently. For example, a deduction by *unit resolution* is a deduction in which each resolvent is obtained by using at least one *unit clause*, i.e., a clause consisting of a single literal. Unit resolution is not (refutation-)complete in general.

Example 3.5.3 *The theory* $\{\leftarrow p(X) \wedge p(Y), \leftarrow \neg p(Z) \wedge \neg p(W)\}$ *is unsatisfiable, but the empty clause cannot be derived by unit resolution, as there is no unit clause in the set.*

However, when all clauses in the set are Horn clauses, unit resolution is refutation-complete [131]. Besides, in the presence of a Horn set, the subsumption theorem holds also if resolution is replaced by unit resolution.

Theorem 3.5.4 *Let* Γ *be a Horn constraint theory and* ϕ *a Horn denial. Then* $\Gamma \models \phi$ *iff there is a unit derivation of a denial* ψ *from* Γ *such that* ψ *subsumes* ϕ.

3.5.2 Local minima in $\mathsf{Simp}_{\mathcal{L}_\mathsf{s}}$

As mentioned, the $\mathsf{Optimize}_{\mathcal{L}_\mathsf{s}}$ procedure gives an approximation of the entailment-based procedure 3.4.8 described on page 53. The quality of the results produced by $\mathsf{Simp}_{\mathcal{L}_\mathsf{s}}$ depends on how well the described proof procedure implements entailment. It is known that for certain classes of languages, such as the monadic class, Herbrand's class and the one-variable class [115], sound and complete procedures based on resolution refinements are guaranteed to terminate. In these cases an ideal simplification can be found.

In the described procedure for $\mathcal{L}_\mathsf{s}$, we chose to limit the size of the resolvents in the resolution proofs to the number n of literals of the largest denial in the starting theory, but other techniques could have been used, such as a limit in the depth of the proof tree or even a time limit. However, our choice is relevant if resolution proofs are performed in a data driven fashion; in the deductive closure of the original theory, the denials whose size is less than n are the only usable clauses for elimination of literals by subsumption (and reduction). The principles of subsumption and reduction are, in practice, sufficient for most cases of denial elimination, and resolution proper is only needed when the integrity constraints encode circularity and recursive definitions or extra hypotheses are provided (see examples 3.2.33 and 3.2.37).

We now discuss how and when $\mathsf{Simp}_{\mathcal{L}_\mathsf{s}}$ reaches a local minimum. To see this, we first state when a local minimum can be found by procedures based on unit resolution, and then we relate this result to $\mathsf{Optimize}_{\mathcal{L}_\mathsf{s}}$.

There are interesting cases in which entailment can be replaced by a terminating proof procedure. We recall that a clause is Horn if, when expressed as a disjunction of literals, it contains at most one positive literal. Equivalently, we say that a denial is Horn if it has at most one negative literal.

Proposition 3.5.5 *Let Γ, Δ be two sets of Horn denials containing no non-nullary function symbol, no parameters and no equalities. Then:*

- *Γ is unsatisfiable iff there is a unit derivation of the empty clause from Γ,*
- *it is decidable whether there is a unit derivation of a given clause from Γ, and*
- *there is a terminating procedure that produces Γ's local minimum wrt Δ.*

Proof *The first claim is trivial because unit resolution is complete for the Horn fragment, and here the theory is Horn.*

For the second claim consider that in every unit resolution step there is one unit clause and another clause with, say, n literals. The number of literals in the resolvent is $n-1$. Since there is no function symbol and the number of clauses is finite, there is only a finite number of possible unit resolution steps. Therefore one can generate all possible unit derivations from Γ in finite time and check whether the given clause has been produced.

From the first two claims we can conclude that satisfiability of a set of Horn denials containing no non-nullary function symbol, no parameters and no equalities can be decided by unit resolution, by checking whether there is a unit derivation of the empty clause.

For the last claim consider that a local minimum is found by checking the two entailment properties described in procedure 3.4.8 on page 53. In both cases the entailed formula is a (subclause of a) denial in Γ, i.e., a formula of the form $\forall\vec{X}(\leftarrow L_1 \wedge \cdots \wedge L_n)$, where

$\vec{X}$ are the variables in $L_1, \ldots, L_n$. In order to check such entailments by resolution, the target denial needs to be negated, transformed into skolemized clausal form, and added to the initial set. But the skolemized clausal form of the negation of a denial is a formula of the form: $\{\leftarrow \neg L_1\sigma, \ldots, \leftarrow \neg L_n\sigma\}$, where σ is a grounding substitution that instantiates every different variable in $\vec{X}$ to a different Skolem constant. Therefore, after the addition of the negated denial, the set is still Horn and function-free and therefore its satisfiability can be decided by unit resolution, as was concluded above. □

We note that this result in not in contrast with the fact that query containment is undecidable for DATALOG, since the Horn constraint theory considered in proposition 3.5.5 is part of a schema with an empty *IDB*, whereas a DATALOG program may have a recursive (and thus not eliminable) *IDB*. So, this result is actually in accordance with the decidability of query containment for non-recursive DATALOG without negation [3].

Proposition 3.5.6 *The* $\mathsf{Optimize}_{\mathcal{L}_S}$ *procedure always returns a local minimum when the input theories are Horn, parameter-free and with no equalities.*

Proof *As shown in theorem 3.5.4, for Horn theories, the subsumption theorem holds when unit resolution is applied instead of resolution. In such case, unit deduction and* $\vdash_R$ *(which is used in the* $\mathsf{Optimize}_{\mathcal{L}_S}$ *procedure) behave in the same way. Therefore the result of proposition 3.5.5 applies also to* $\mathsf{Optimize}_{\mathcal{L}_S}$*.* □

The result of proposition 3.5.5 may be extended to Horn theories with equalities and parameters provided that proper equality axioms (reflexivity, symmetry, transitivity and substitutivity), such as the free equality axioms shown on page 14, are added to the input set. Alternatively, resolution can be extended with *paramodulation* and the reflexivity axiom [110] in order to be refutation-complete. The paramodulation rule is an inference rule that handles equalities in the presence of function symbols and allows replacing terms in equations. In a function-free settings, however, this generality is not needed, since reduction takes care of reflexivity and expansion provides substitutivity, whereas symmetry was assumed to be an implicit syntactic property of equality. However, the transitivity axiom ($X = Y \leftarrow X = Z \wedge Z = Y$) would need to be added to the input set to obtain refutation-completeness and, thus, further refine the simplification procedure.

Example 3.5.7 *Consider the constraint theory*

$$\Gamma = \{\leftarrow p(X) \wedge X \neq a, \ \leftarrow \ p(X) \wedge \ X \neq \ b\}.$$

It is clearly equivalent to $\{\leftarrow p(X)\}$ *since* $(X \neq a \vee X \neq b)$ *is a consequence of the equality axioms. The first denial resolves with the transitivity axiom into* $X = Y \leftarrow p(X) \wedge a = Y$*, which resolves with the second denial into* $X = b \leftarrow p(X) \wedge p(a)$*, a factor of which is* $a = b \leftarrow p(a)$*, which reduces to* $\leftarrow p(a)$ *and expands to* $\leftarrow p(X) \wedge X = a$*. A resolution step with the first denial produces* $\leftarrow p(X)$*, which subsumes all the denials and is thus the result.*

Without the inclusion of the transitivity axiom, there are cases of redundancies that are not detected by $\mathsf{Optimize}_{\mathcal{L}_S}$ that may occur when there are cyclic parameter dependencies. For example, consider the denial

$$\phi = \leftarrow \mathbf{a} \neq \mathbf{b} \wedge \mathbf{b} = \mathbf{c} \wedge \mathbf{c} = \mathbf{d} \wedge \mathbf{d} = \mathbf{a}.$$

With a resolution step of the transitivity axiom with itself, one gets $\leftarrow X \neq Y \wedge X = T \wedge T = Z \wedge Z = Y$, which subsumes ϕ; ϕ can therefore be eliminated. Without the transitivity axiom, one would not be able to infer this.

3.5.3 Complexity

We cannot hope to obtain an ideal procedure in all non-recursive cases, as QC is already undecidable for non-recursive DATALOG with negation [3].

Although QC is decidable for non-recursive DATALOG programs without negation, it is known to be decidable in exponential time (see [38] for an overview of this and other results on QC). The search for a local minimum in these cases, as sketched in procedure 3.4.8 on page 53, is thus also exponential, since it may require solving $n+m$ QC problems, where n is the number of literals and m is the number of denials in the constraint theory. These results would suggest that the problem is intractable; however, the complexity is here measured with respect to the size of the query and not of the data in the database.

Another aspect to be considered is that simplification is a static process, therefore it is worthwhile to invest resources for compiling the constraints at design time so as to improve run time efficiency.

Finally, we point out that subsumption, which is used in the simplification procedure, is an incomplete procedure for entailment. However it becomes complete (i.e., C subsumes D iff $C \models D$) if C is neither recursive nor tautological [90]. Subsumption is already NP-complete in general [108]. This complexity is due to the ambiguity of variable identification and amounts to $O(|\text{vars}(D)|^{|\text{vars}(C)|})$ to check whether C subsumes D. In practice, if the domain is *weakly structured*, i.e., there is only a small number of predicates occurring very often in the clauses, only short clauses are affordable; conversely, for *strongly structured* domains, complexity decreases dramatically [152]. This observation proves very important for data representation issues. For example, in section 5.3, this will indicate what relational representation of the nested data structure should be chosen in order to keep the application of the simplification operators tractable.

3.6 Related work

Simplification of integrity constraints is highly relevant for optimizations in database integrity checking. Typically it gives a speed-up of a linear factor (in the size of the database state) for singleton updates, but for certain transactions an even higher speed-up can be gained. We find crucial the ability to check consistency of a possibly updated database *before* execution of the transaction under consideration so that inconsistent states are completely avoided. As mentioned, pre-testing the feasibility of an update with respect to the integrity constraints is particularly valuable, as it allows one to avoid both the execution of the update and, especially, the restoration of the database state before the update, which is typically done by means of expensive rollback or repair operations. Rollbacks usually require costly bookkeeping in order to restore the old state; as for repairs, repaired states need to be calculated according to given preference criteria and then repairing actions need to be executed. Several approaches to simplification first require the transaction to be performed, and *then* the resulting state to be checked for

consistency [137, 124, 147, 71, 93, 116]. Methods that, as ours, are based on pre-tests are, e.g., [143, 100, 102, 114], including a few industrial attempts, e.g., [21, 41].

Simplification of integrity constraints with respect to given parametric update patterns resembles the notion of program *specialization* used in partial evaluation [106], which is the process of creating a specialized version of a given program with respect to known input data. Applications of these techniques to integrity checking have been investigated in [116], where partial evaluation of a meta-interpreter is used to produce logic programs that correspond to simplified constraints. A partial evaluator is given a meta-interpreter that constitutes a general integrity checker and produces as output a version of the meta-interpreter specialized to specific update patterns to be checked in the updated state (and employing the hypothesis that integrity holds before the update). Generally, it is difficult to evaluate such a method as it depends on a number of heuristics in the partial evaluator as well as in the meta-interpreter.

The proposal of [100] presents several analogies with our method, although it does not apply to deductive databases, but only to relational databases without views. A series of tests, expressed in a DATALOG-like language, is generated from an integrity constraint C and an update U; if one of these tests succeeds, then U is legal with respect to C. This method is based on *resolution* and *transition axioms*. The the update language is limited to single additions, deletions and changes, i.e., no transactions can be specified. Updates can contain so-called *dummy constants*, which correspond to our notion of parameters. A disadvantage of this approach is that, once the set of tests is generated, strategies have to be employed to decide which specific tests to execute and in which order, i.e., redundancies are kept within the generated tests. The authors do not develop this aspect further. Furthermore, and more importantly, failure of all tests does not necessarily imply inconsistency: this means that, in such a case, a necessary and sufficient condition must be used. Instead, our simplification algorithm can handle more general updates and generates tests that are a necessary and sufficient condition for determining consistency of the database if the update were performed[11].

In [47, 93], the authors introduce a principle called *partial subsumption* applied, among other things, to produce simplified integrity constraints. Their method can handle singleton additions or deletions and produces conditions to be applied *after* the update. Some parametric updates can be expressed using variables. However, the described approach does not consider several occurrences of the same parameter in the same relation or in different relations within the same update. For compound updates (transactions) the principle is explained in terms of examples, but no general procedure is described. Changes are modeled as deletions plus insertions; however, this is not completely satisfactory, as all deleted and inserted tuples must be known in advance (unlike, e.g., update U_2 of example 2.2.22 on page 17) and for any insertion (resp. deletion) of a tuple, it must be assumed that the tuple was not present (resp. was present) in the database. Partial subsumption applies also to semantic query optimization; see [94, 88] for an overview.

Qian [143] observes the relationship between Hoare's logic [101, 75] for imperative languages and integrity checking, identifying a simplified integrity constraint as a weakest precondition for having a consistent updated state. This notion is enforced by assuming consistency of the database before the update. Qian's method works for a variety of

[11] If their tests were a necessary and sufficient condition, they would logically correspond to the negation of our tests.

SQL-like ways of updating relations but with the impractical limitation that it does not allow more than one update action in a transaction to operate on the same relation; furthermore, no mechanism corresponding to parameters is present, thus requiring to execute the procedure for each update. We have no such restrictions.

Integrity checking is often regarded as an instance of materialized view maintenance: integrity constraints are defined as views that must always remain empty for the database to be consistent. The database literature is rich in methods that deal with relational view/integrity maintenance; the book [99] and the survey [76] provide insightful discussion on the subject.

Abduction in logic programming (e.g., [107]), which concerns the generation of hypotheses necessary to explain given observations, typically involves integrity constraints on the possible hypotheses to be generated, and a classical example of abduction is that of database updates through view updates. Most abductive methods perform a complete check of the integrity constraints when a new hypothesis (i.e., a tuple update) is proposed by the inference engine. By the use of constraint logic programming techniques (such as Constraint Handling Rules [85]) to implement abduction, incremental evaluation of integrity constraints may arise without an explicit simplification algorithm: each time an abducible atomic update a arises, the current representation of the integrity constraints wakes up, checks a's dependencies and, in case of success, delays a specialized version of the integrity constraints waiting for the next update. This principle is applied in the DemoII system [54, 57] and in the approaches of [1, 56] using Constraint Handling Rules for abduction; [55] is an attempt to relate such methods to database applications. However, a common drawback of these techniques is that the delayed constraints typically unroll to a size proportional to the database and that, occasionally, an unsatisfiable set of constraints is delayed where a failure should be reported.

As we noticed, ideal simplification is possible if and only if a query containment decision procedure exists for the class of databases under consideration. Although even non-ideal simplification procedures prove very useful in practice, progress in the field of query containment can inspire the construction of more refined simplification algorithms; we refer to [84, 37, 38, 27] and the references therein for an overview of decidable cases and decision algorithms that have been studied in the literature.

Chapter 4

Extensions

In this chapter we discuss three orthogonal extensions of the language $\mathcal{L}_{\mathsf{S}}$ and the simplification procedure $\mathsf{Simp}_{\mathcal{L}_{\mathsf{S}}}$.

In the definition of $\mathcal{L}_{\mathsf{S}}$ we limited the level of interaction between negation and existential quantification in the constraint theories. In section 4.1, we relax this limitation and extend the syntax of denials so as to allow the presence of negated existential quantifiers. The simplification procedure can be adapted to such cases, provided that its components are adjusted as to handle conjuncts starting with a negated existential quantifier.

In section 4.2, we describe an extension of the syntax of denials that includes arithmetic operators and aggregates. This requires extending the components of the simplification procedure with new rewrite rules that possibly interact with a constraint solver for arithmetic in order to properly handle such constructs.

Finally, in section 4.3, we consider an extension of $\mathcal{L}_{\mathsf{S}}$ that allows the presence of recursive predicates. With the precaution that such predicates cannot be unfolded, $\mathsf{Simp}_{\mathcal{L}_{\mathsf{S}}}$ can be used for such cases. However, for certain recursive patterns, it is possible to refine the simplification procedure so as to obtain a much higher degree of optimization than was available with $\mathsf{Simp}_{\mathcal{L}_{\mathsf{S}}}$.

4.1 Nested denials

In this section we address the problem of simplification of integrity constraints in hierarchical databases. As mentioned in section 2.2, all clauses are range restricted and the schemata of hierarchical databases are assumed to be non-recursive, but there is no other restriction on the occurrence of negation. We refer to the language of such schemata as $\mathcal{L}_{\mathsf{H}}$.

Definition 4.1.1 ($\mathcal{L}_{\mathsf{H}}$) *Let S be a schema. S is in $\mathcal{L}_{\mathsf{H}}$ if its starred dependency graph is acyclic.*

Unfolding predicates in integrity constraints with respect to their definitions cannot be done in the same way as $\mathsf{Unfold}_{\mathcal{L}_{\mathsf{S}}}$. For this purpose, we extend the syntax of denials so as to allow negated existential quantifiers to occur in literals.

Definition 4.1.2 (Extended denials) *A* negated existential expression *or* NEE *is an expression of the form* $\neg\exists\vec{X}B$*, where* B *is called the* body *of the NEE,* $\vec{X}$ *are some (possibly all) of the variables occurring in* B *and* B *has the form* $L_1 \wedge \cdots \wedge L_n$*, where each* L_i *is a general literal.*
A general literal *is either a literal or a NEE.*
A formula of the form $\forall\vec{X}(\leftarrow B)$*, where* B *is the body of a NEE and* $\vec{X}$ *are some (possibly all) of the free variables in* B*, is called an* extended denial*. When there is no ambiguity on the variables in* $\vec{X}$*, extended denials are simply written* $\leftarrow B$*.*

Example 4.1.3 *The formula* $\leftarrow parent(X) \wedge \neg\exists Y child\_of(X,Y)$ *is an extended denial. It reads as follows: there is inconsistency if there is a parent* X *that does not have a child. Note that this is different from the (non-range restricted) denial* $\leftarrow parent(X) \wedge \neg child\_of(X,Y)$*, which states that if* X *is a parent then all individuals must be his/her children.*

We observe that variables under a negated existential quantifier conform with the intuition behind safeness, so we could conclude that the first formula in example 4.1.3 is safe, whereas the second one is not. We can now apply unfolding in $\mathcal{L}_{\mathsf{H}}$ to obtain extended denials. In doing so, attention needs to be paid when replacing negated intensional predicates by their definition, since they may contain non-distinguished variables and thus existential quantifiers have to be explicitly indicated. As was the case for $\mathsf{Unfold}_{\mathcal{L}_{\mathsf{S}}}$, the replacements may result in disjunctions and negated conjunctions. Therefore, additional steps are needed to restore the extended denial form.

Definition 4.1.4 *Let* $S = \langle IDB, \Gamma\rangle$ *be a database schema in* $\mathcal{L}_{\mathsf{H}}$*. We define* $\mathsf{Unfold}_{\mathcal{L}_{\mathsf{H}}}(S)$ *as the set of extended denials obtained by iterating the two following steps as long as possible:*

1. *replace, in* Γ*, each occurrence of a literal of the form* $\neg p(\vec{t})$ *by* $\neg\exists\vec{Y}F^p\{\vec{X}/\vec{t}\}$ *and of a literal of the form* $p(\vec{t})$ *by* $F^p\{\vec{X}/\vec{t}\}$*, where* $p \in pred(IDB)$*,* F^p *is* p*'s defining formula,* $\vec{X}$ *its distinguished variables and* $\vec{Y}$ *its non-distinguished variables. If no replacement was made, then stop;*

2. *transform the resulting formula into a set of extended denials according to the following patterns;* $\Phi(Arg)$ *is an expression indicating the body of a NEE in which* Arg *occurs;* $\vec{X}$ *and* $\vec{Y}$ *are disjoint sequences of variables:*

   - $\leftarrow A \wedge (B \vee C)$ *is replaced by* $\leftarrow A \wedge B$ *and* $\leftarrow A \wedge C$*;*
   - $\leftarrow A \wedge \neg(B \vee C)$ *is replaced by* $\leftarrow A \wedge \neg B \wedge \neg C$*;*
   - $\leftarrow A \wedge \neg(B \wedge C)$ *is replaced by* $\leftarrow A \wedge \neg B$ *and* $\leftarrow A \wedge \neg C$*;*
   - $\leftarrow A \wedge \neg\exists\vec{X}\Phi(\neg\exists\vec{Y}[B \wedge (C \vee D)])$ *is replaced by* $\leftarrow A \wedge \neg\exists\vec{X}\Phi(\neg\exists\vec{Y}[B \wedge C] \wedge \neg\exists\vec{Y}[B \wedge D])$*.*

Without loss of generality, we can assume that, for any NEE $N = \neg\exists\vec{X}B$ occurring in an extended denial ϕ, the variables $\vec{X}$ do not occur outside N in ϕ. This can simply be obtained by renaming the variables appropriately and we refer to such an extended

denial as *standardized*. The *level* of a NEE in an extended denial is the number of NEEs that contain it, plus 1. The level of a variable X in a standardized extended denial is the level of the NEE starting with $\neg\exists\vec{X}$, where X is one of the variables in $\vec{X}$, or 0 if there is no such NEE. The level of an extended denial is the maximum level of its NEEs, or 0 if there is no NEE. In example 4.1.3, the extended denial has level 1, X has level 0 and Y has level 1.

With a slight abuse of notation, we write in the following $S \equiv \Psi$ (or $\Psi \equiv S$), where $S = \langle IDB, IC\rangle$ is a schema and Ψ is a set of extended denials, to indicate that, for every database D based on IDB, $D \models IC$ iff $D \models \Psi$. We can now claim the correctness of $\mathsf{Unfold}_{\mathcal{L}_\mathsf{H}}$. We state the following proposition without a proof, since all steps in definition 4.1.4 are trivially equivalence-preserving.

Proposition 4.1.5 *Let $S \in \mathcal{L}_\mathsf{H}$. Then $\mathsf{Unfold}_{\mathcal{L}_\mathsf{H}}(S) \equiv S$.*

Since the variables under a negated existential quantifier conform with the intuition behind safeness, the unfolding of a schema in which all the clauses are range restricted yields a set of extended denials that still are safe in this sense. We also note that the language of extended denials is very expressive. In [123], it was shown that *any* closed formula of the form $\forall\vec{X}(\leftarrow B)$, where B is a first-order formula and $\vec{X}$ are its open variables, can be equivalently expressed by a set of Prolog rules plus the denial $\leftarrow q(\vec{X})$, where q is a fresh predicate symbol. The construction is such that, if B is function-free, then the resulting rule set is also function-free (no skolemization is needed), i.e., it is that of a schema in $\mathcal{L}_\mathsf{H}$. The unfolding of the obtained schema is therefore an equivalent set of extended denials.

A simplification procedure can now be constructed for extended denials in a way similar to what was done in $\mathcal{L}_\mathsf{S}$.

Definition 4.1.6 *Let S be a schema in $\mathcal{L}_\mathsf{H}$ and U an update. We define $\mathsf{After}^U_{\mathcal{L}_\mathsf{H}}(S)$ as $\mathsf{Unfold}_{\mathcal{L}_\mathsf{H}}(\mathsf{After}^U(S))$.*

The optimization step needs to take into account the nesting of NEEs in extended denials. Besides the elimination of disjunctions within NEEs, which is performed by $\mathsf{Unfold}_{\mathcal{L}_\mathsf{H}}$, we can also eliminate, from a NEE, equalities and non-equalities referring to variables of lower level with respect to the NEE.

Definition 4.1.7 *Let A, B, C be (possibly empty) conjunctions of general literals, $\vec{Y}, \vec{Z}$ disjoint (sequences of) variables, W a variable of level lower than the level of $\vec{Z}$, and $\Phi(Arg)$ an expression indicating a NEE in which Arg occurs The following rewrite rules are, respectively, the* equality elimination *and* non-equality elimination *rules.*

$$\leftarrow A \wedge \Phi(\neg\exists\vec{Y}[B \wedge \neg\exists\vec{Z}(C \wedge W = c)]) \;\Rightarrow$$
$$\leftarrow A \wedge \Phi(\neg\exists\vec{Y}[B \wedge W = c \wedge \neg\exists\vec{Z}(C)] \wedge \neg\exists\vec{Y}[B \wedge W \neq c])$$

$$\leftarrow A \wedge \Phi(\neg\exists\vec{Y}[B \wedge \neg\exists\vec{Z}(C \wedge W \neq c)]) \;\Rightarrow$$
$$\leftarrow A \wedge \Phi(\neg\exists\vec{Y}[B \wedge W \neq c \wedge \neg\exists\vec{Z}(C)] \wedge \neg\exists\vec{Y}[B \wedge W = c])$$

The above (non-)equality elimination rewrite rules are equivalence preserving, as stated below.

Proposition 4.1.8 *Let ψ be an extended denial and ψ' (resp. ψ'') be the extended denial obtained after an application of the equality (resp. non-equality) elimination rule. Then $\psi \equiv \psi'$ and $\psi \equiv \psi''$.*

Proof *Using the notation of definition 4.1.7, we have:*

$$\begin{array}{rl} & \neg\exists\vec{Y}[B \wedge \neg\exists\vec{Z}(C \wedge W = c)] \\ \equiv & \neg\exists\vec{Y}[B \wedge (W = c \vee W \neq c) \wedge \neg\exists\vec{Z}(C \wedge W = c)] \\ \equiv & \neg\exists\vec{Y}[B \wedge W = c \wedge \neg\exists\vec{Z}(C \wedge W = c)] \wedge \\ & \neg\exists\vec{Y}[B \wedge W \neq c \wedge \neg\exists\vec{Z}(C \wedge W = c)] \\ \equiv & \neg\exists\vec{Y}[B \wedge W = c \wedge \neg\exists\vec{Z}(C)] \wedge \neg\exists\vec{Y}[B \wedge W \neq c] \end{array}$$

In the first step, we added the tautological conjunct $(W = c \vee W \neq c)$. In the second step we used de Morgan's laws in order to eliminate the disjunction (as in the definition of $\mathsf{Unfold}_{\mathcal{L}_\mathsf{H}}$). The formula $W = c \wedge \neg\exists\vec{Z}(C \wedge W = c)$ can be rewritten as $W = c \wedge \neg(\exists\vec{Z}C \wedge W = c)$, and then as $W = c \wedge (\neg\exists\vec{Z}(C) \vee W \neq c)$, which, with a resolution step, results in the first NEE in the last extended denial. Similarly, $W \neq c \wedge \neg\exists\vec{Z}(C \wedge W = c)$ can be rewritten as $W \neq c \wedge (\neg\exists\vec{Z}(C) \vee W \neq c)$, which results (by absorption) into the second NEE in the last extended denial.

The proof is similar for non-equality elimination. □

In case only levels 0 and 1 are involved, the rules look simpler. For example, equality elimination can be conveniently formulated as follows.

$$\leftarrow B \wedge \neg\exists\vec{X}[C \wedge W = c] \quad \Rightarrow \quad \{ \begin{array}{l} \leftarrow B\{W/c\} \wedge \neg\exists\vec{X}C\{W/c\} \\ \leftarrow B \wedge W \neq c \end{array} \}. \tag{4.1}$$

Although (non-)equality elimination does not necessarily shorten the input formula (in fact, it can also lengthen it), it always reduces the number of literals in higher level NEEs. Therefore, convergence to termination can still be guaranteed if this rewrite rule is applied during optimization. Repeated application of such rules "pushes" outwards the involved (non-)equalities until they reach an NEE whose level is the same as the level of the variable in the (non-)equality. Then, in case of an equality, the usual equality elimination step of reduction can be applied.

Example 4.1.9 *The following rewrites show the propagation of a variable of level 0 (X) from level 2 to level 0 via two equality eliminations and one non-equality elimination.*

$$
\begin{array}{lll}
 & \leftarrow & p(X) \wedge \neg\exists Y\{q(X,Y) \wedge \neg\exists Z[r(X,Y,Z) \wedge X = a]\} \\
 & & \\
\equiv & \leftarrow & p(X) \wedge \neg\exists Y\{q(X,Y) \wedge X = a \wedge \neg\exists Z[r(X,Y,Z)]\} \\
 & & \wedge\neg\exists Y[q(X,Y) \wedge X \neq a] \\
 & & \\
\equiv \{ & \leftarrow & p(a) \wedge \neg\exists Y[q(a,Y) \wedge \neg\exists Z r(a,Y,Z)] \wedge \neg\exists Y[q(a,Y) \wedge a \neq a], \\
 & \leftarrow & p(X) \wedge X \neq a \wedge \neg\exists Y[X \neq a \wedge q(X,Y)] \quad \} \\
 & & \\
\equiv \{ & \leftarrow & p(a) \wedge \neg\exists Y[q(a,Y) \wedge \neg\exists Z r(a,Y,Z)] \wedge \neg\exists Y[q(a,Y) \wedge a \neq a], \\
 & \leftarrow & p(X) \wedge X \neq a \wedge \neg\exists Y q(X,Y), \\
 & \leftarrow & p(a) \wedge X \neq a \wedge X = a \quad \} \\
 & & \\
\equiv \{ & \leftarrow & p(a) \wedge \neg\exists Y[q(a,Y) \wedge \neg\exists Z r(a,Y,Z)], \\
 & \leftarrow & p(X) \wedge X \neq a \wedge \neg\exists Y q(X,Y) \quad \}
\end{array}
$$

In the first step we applied equality elimination to $X = a$ at level 2. In the second step we applied the rewrite rule (4.1) for equality elimination to $X = a$ at level 1. Then we applied non-equality elimination to $X \neq a$ at level 1 in the second denial. In the last step, we removed, by standard application of reduction, the last extended denial and the last NEE in the first extended denial, which are clearly tautological.

We observe that the body of a NEE is structurally similar to the body of an extended denial. The only difference is that, in the former, there are variables that are quantified at a lower level. According to this observation, such (free) variables in the body of a NEE are to be treated as parameters during the different optimization steps, since, as was indicated on page 9, parameters are free variables.

Reduction (definition 3.2.19 on page 37) can then take place in NEE bodies exactly as in ordinary denials, with the proviso above of treating free variables as parameters.

The definition of resolution (3.2.21 on page 39 and following definitions) can be adapted for extended denials by applying it to general literals instead of literals.

Subsumption (definition 3.2.17 on page 37) can also be applied to extended denials without changing the definition. However, we can slightly modify the notion of subsumption to explore the different levels of NEEs in an extended denial. This is captured by the following definition.

Definition 4.1.10 *Let $\phi =\leftarrow A \wedge B$ and $\psi =\leftarrow C \wedge D$ be two extended denials, where A and C are (possibly empty) conjunctions of literals and B and D are (possibly empty) conjunctions of NEEs. Then ϕ extended-subsumes ψ if both conditions (1) and (2) below hold.*

(1) $\leftarrow A$ subsumes $\leftarrow C$ with substitution σ.

(2) For every NEE $\neg\exists\vec{X}_N N$ in B, there is a NEE $\neg\exists\vec{X}_M M$ in D such that $\leftarrow N\sigma$ is extended-subsumed by $\leftarrow M$.

Example 4.1.11 *The extended denial $\leftarrow p(X) \wedge \neg\exists Y, Z[q(X,Y) \wedge r(Y,Z)]$ extended-subsumes the extended denial $\leftarrow p(a)\wedge\neg\exists T[q(a,T)]\wedge\neg\exists W[s(T)]$, since $\leftarrow p(X)$ subsumes $\leftarrow p(a)$ with substitution $\{X/a\}$ and, in turn, $\leftarrow q(a,T)$ subsumes $\leftarrow q(a,Y) \wedge r(Y,Z)$.*

This definition encompasses ordinary subsumption, in that it coincides with it if B and D are empty. Furthermore, it captures the desired property that if ϕ extended-subsumes ψ, then $\phi \models \psi$; the reverse, as in subsumption, does not necessarily hold.

Proposition 4.1.12 *Let ϕ and ψ be extended denials. If ϕ extended-subsumes ψ then $\phi \models \psi$.*

Proof *Let ϕ, ψ, A, B, C, D be as in definition 4.1.10. If B and D are empty the claim holds, since ϕ and ψ are ordinary denials. The claim also holds if D is not empty, since $\leftarrow C$ entails $\leftarrow C \wedge D$. We now show the general claim with an inductive proof on the level of extended denials.*
The base case (level 0) is already proven.
Inductive step. Suppose now that ϕ is of level $n+1$ and that the claim holds for extended denials of level n or less. Assume as a first case that B is empty. Then ϕ entails ψ, since ϕ entails $\leftarrow C$ ($\leftarrow A$ subsumes $\leftarrow C$ by hypothesis). Assume for the moment that $B = \neg\exists\vec{X}_N N$ is a NEE of level 1 in ϕ and that D contains a NEE (of level 1) $\neg\exists\vec{X}_M M$, such that $\phi' =\!\leftarrow N\sigma$ is extended-subsumed by $\psi' =\!\leftarrow M$, as assumed in the hypotheses. But ϕ' and ψ' are extended denials of level n and, therefore, if ψ' subsumes ϕ' then ψ' entails ϕ' by inductive hypothesis. Clearly, since $\leftarrow A$ entails $\leftarrow C \wedge D$ (by hypothesis) and $\leftarrow M$ entails $\leftarrow N\sigma$ (as a consequence of the inductive hypothesis), then $\leftarrow A \wedge B$ entails $\leftarrow C \wedge D$, which is our claim. If B contains more than one NEE of level 1, the argument is iterated by adding one NEE at a time. □

The inductive proof also shows how to check extended subsumption with a finite number of subsumption tests. This implies that extended subsumption is decidable, since subsumption is. Now that a correct extended subsumption is introduced, it can be used instead of subsumption in the subsumption factoring rule of reduction (definition 3.2.19 on page 37). In the following, when referring to a NEE $N = \neg\exists\vec{X}B$ we can also write it as a denial $\leftarrow B$, with the understanding that the free variables in N are considered parameters.

Definition 4.1.13 *For an extended denial ϕ, the reduction ϕ^- of ϕ is the result of applying on ϕ equality and non-equality elimination as long as possible, and then the rules of definition 3.2.19 (reduction on page 37) on ϕ and its NEEs as long as possible, where "literal" is replaced by "general literal", "subsumes" by "extended-subsumes" and "denial" by "extended denial".*

Without reintroducing similar definitions, we assume that the same word replacements are made for the notion of $\vdash_R$ (definition 3.2.26 on page 40). The underlying notions of substitution and unification also apply to extended denials and general literals; only, after substitution with a constant or parameter, the existential quantifier of a variable is removed. For example, the extended denials $\phi =\!\leftarrow p(X, b) \wedge \neg\exists Z[q(Z, X)]$ and $\psi =\!\leftarrow p(a, Y) \wedge \neg q(c, a)$ unify with substitution $\{X/a, Y/b, Z/c\}$. By virtue of the similarity between denial bodies and NEE bodies, we extend the notion of optimization as follows.

Definition 4.1.14 *Given two sets of extended denials Δ and Γ, $\mathsf{Optimize}^{\Delta}_{\mathcal{L}_\mathsf{H}}(\Gamma)$ is the result of applying the following rewrite rules and the rules of definition 3.2.28 ($\mathsf{Optimize}_{\mathcal{L}_\mathsf{S}}$ on page 41) on Γ as long as possible. In the following, ϕ and ψ are NEEs, Γ' is a set of*

extended denials, and $\Phi(Arg)$ is an expression indicating the body of an extended denial in which Arg occurs.

$$\begin{array}{lcl}
\{\leftarrow \Phi(\phi)\} \sqcup \Gamma' & \Rightarrow & \{\leftarrow \Phi(true)\} \cup \Gamma' \text{ if } \phi^- = true \\
\{\leftarrow \Phi(\phi)\} \sqcup \Gamma' & \Rightarrow & \{\leftarrow \Phi(true)\} \cup \Gamma' \text{ if } (\Gamma' \cup \Delta) \vdash_R \phi \\
\{\leftarrow \Phi(\phi)\} \sqcup \Gamma' & \Rightarrow & \{\leftarrow \Phi(\phi^-)\} \cup \Gamma' \text{ if } \phi \neq \phi^- \neq true \\
\{\leftarrow \Phi(\phi)\} \sqcup \Gamma' & \Rightarrow & \{\leftarrow \Phi(\phi^-)\} \cup \Gamma' \text{ if } ((\{\phi\} \cup \{\leftarrow \Phi(\phi)\}) \sqcup \Gamma' \cup \Delta) \vdash_R \psi \\
 & & \text{and } \psi^- \text{ strictly extended-subsumes } \phi
\end{array}$$

Finally, the simplification procedure for $\mathcal{L}_{\mathsf{H}}$ is composed in terms of $\mathsf{After}_{\mathcal{L}_{\mathsf{H}}}$ and $\mathsf{Optimize}_{\mathcal{L}_{\mathsf{H}}}$.

Definition 4.1.15 *Consider a schema $S = \langle IDB, \Gamma \rangle \in \mathcal{L}_{\mathsf{H}}$ and an update U. Let $\Gamma' = \mathsf{Unfold}_{\mathcal{L}_{\mathsf{H}}}(S)$. We define*

$$\mathsf{Simp}^U_{\mathcal{L}_{\mathsf{H}}}(S) = \mathsf{Optimize}^{\Gamma'}_{\mathcal{L}_{\mathsf{H}}}(\mathsf{After}^U_{\mathcal{L}_{\mathsf{H}}}(S)).$$

Similarly to $\mathcal{L}_{\mathsf{S}}$, soundness of the optimization steps and the fact that After returns a WP entail the following.

Proposition 4.1.16 *Let $S \in \mathcal{L}_{\mathsf{H}}$ and U be an update. Then $\mathsf{Simp}^U_{\mathcal{L}_{\mathsf{H}}}(S)$ is a CWP of S with respect to U.*

4.1.1 Examples

We now discuss the most complex non-recursive examples that we found in the literature referenced in this thesis.

Example 4.1.17 *This example is taken from [114]. Consider a schema $S = \langle IDB, \Gamma \rangle$ with three extensional predicates a, b, c, two intensional predicates p, q, a constraint theory Γ and a set of trusted hypotheses Δ.*

$$\begin{array}{rl}
IDB = \{ & p(X,Y) \leftarrow a(X,Z) \wedge b(Z,Y), \\
 & q(X,Y) \leftarrow p(X,Z) \wedge c(Z,Y) \quad \} \\
\Gamma = \{ & \leftarrow p(X,X) \wedge \neg q(1,X) \ \} \\
\Delta = \{ & \leftarrow a(1,5) \ \}
\end{array}$$

This schema S is not in $\mathcal{L}_{\mathsf{S}}$ and the unfolding of S is as follows.

$$\mathsf{Unfold}_{\mathcal{L}_{\mathsf{H}}}(S) = \{\leftarrow a(X,Y) \wedge b(Y,X) \wedge \neg\exists W, Z(a(1,W) \wedge b(W,Z) \wedge c(Z,X))\}.$$

We want to verify that the update $U = \{b(X,Y) \Leftarrow b(X,Y) \wedge X \neq 5\}$ (the deletion of all b-tuples with the first argument different from 5) does not affect consistency. $\mathsf{After}^U_{\mathcal{L}_{\mathsf{H}}}(S)$ results in the following extended denial:

$$\leftarrow a(X,Y) \wedge b(Y,X) \wedge Y \neq 5 \wedge \neg\exists W, Z(a(1,W) \wedge b(W,Z) \wedge W \neq 5 \wedge c(Z,X)).$$

As previously described, during the optimization process, the last conjunct can be processed as a separate denial $\phi = \leftarrow a(1,W) \wedge b(W,Z) \wedge W \neq 5 \wedge c(Z,\mathbf{X})$, where $\mathbf{X}$ is a free variable

that can be treated as a parameter (and thus indicated in bold). With a resolution step with Δ, the literal $W \neq 5$ is proved to be redundant and can thus be removed from ϕ. The obtained formula is then subsumed by $\mathsf{Unfold}_{\mathcal{L}_\mathsf{H}}(S)$ and therefore $\mathsf{Optimize}^{\Delta}_{\mathcal{L}_\mathsf{H}}(\mathsf{Simp}^{U}(S)) = \emptyset$, i.e., the update cannot violate the integrity constraint, which is the same result that was found in [114].

Example 4.1.18 *The following schema S is the relevant part of an example described in [116] on page 24.*

$$
\begin{array}{rrcl}
S = \langle\{ & married\_to(X,Y) & \leftarrow & parent(X,Z) \wedge parent(Y,Z) \wedge \\
 & & & man(X) \wedge woman(Y), \\
 & married\_man(X) & \leftarrow & married\_to(X,Y), \\
 & married\_woman(X) & \leftarrow & married\_to(Y,X), \\
 & unmarried(X) & \leftarrow & man(X) \wedge \neg married\_man(X), \\
 & unmarried(X) & \leftarrow & woman(X) \wedge \neg married\_woman(X) \quad \}, \\
\{ & & \leftarrow & man(X) \wedge woman(X), \\
 & & \leftarrow & parent(X,Y) \wedge unmarried(X) \qquad \}\rangle
\end{array}
$$

If we reformulate the example using the shorthand notation introduced on page 42, the database is updated with $U = \{man(\mathbf{a})\}$, where $\mathbf{a}$ is a parameter. The unfolding given by $\mathsf{Unfold}_{\mathcal{L}_\mathsf{H}}(S)$ is as follows, where m, w, p respectively abbreviate man, woman, parent, which are the only extensional predicates.

$$
\begin{array}{ll}
\{ & \leftarrow m(X) \wedge w(X), \\
 & \leftarrow p(X,Y) \wedge m(X) \wedge \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge m(X) \wedge w(T)], \\
 & \leftarrow p(X,Y) \wedge w(X) \wedge \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge w(X) \wedge m(T)] \quad \}
\end{array}
$$

We start the simplification process by applying $\mathsf{After}_{\mathcal{L}_\mathsf{H}}$ to S wrt U.

$$
\begin{array}{rcl}
\mathsf{After}^{U}_{\mathcal{L}_\mathsf{H}}(S) \equiv \{ & \leftarrow & (m(X) \vee X = \mathbf{a}) \wedge w(X), \\
 & \leftarrow & p(X,Y) \wedge (m(X) \vee X = \mathbf{a}) \wedge \\
 & & \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge (m(X) \vee X = \mathbf{a}) \wedge w(T)], \\
 & \leftarrow & p(X,Y) \wedge w(X) \wedge \\
 & & \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge w(X) \wedge (m(T) \vee T = \mathbf{a})] \ \}
\end{array}
$$

After eliminating the disjunctions at level 0, $\mathsf{After}^{U}_{\mathcal{L}_\mathsf{H}}(S)$ is as follows:

$$
\begin{array}{l}
\{\leftarrow m(X) \wedge w(X), \\
\ \leftarrow X = \mathbf{a} \wedge w(X), \\
\ \leftarrow p(X,Y) \wedge m(X) \wedge \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge (m(X) \vee X = \mathbf{a}) \wedge w(T)], \\
\ \leftarrow p(X,Y) \wedge X = \mathbf{a} \wedge \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge (m(X) \vee X = \mathbf{a}) \wedge w(T)], \\
\ \leftarrow p(X,Y) \wedge w(X) \wedge \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge w(X) \wedge (m(T) \vee T = \mathbf{a})] \quad \}
\end{array}
$$

Now we can eliminate the disjunctions at level 1 *and obtain the following set.*

$$\begin{array}{lll}\{\leftarrow & m(X) \wedge w(X), & \\ \leftarrow & X = \mathbf{a} \wedge w(X), & \\ \leftarrow & p(X,Y) \wedge m(X) \wedge \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge m(X) \wedge w(T)] \wedge & \\ & \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge X = \mathbf{a} \wedge w(T)], & \\ \leftarrow & p(X,Y) \wedge X = \mathbf{a} \wedge \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge m(X) \wedge w(T)] \wedge & \\ & \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge X = \mathbf{a} \wedge w(T)], & \\ \leftarrow & p(X,Y) \wedge w(X) \wedge \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge w(X) \wedge m(T)] \wedge & \\ & \neg\exists(T,Z)[p(X,Z) \wedge p(T,Z) \wedge w(X) \wedge T = \mathbf{a}] & \} \end{array}$$

We can now proceed with the optimization of this set of extended denials by using the $\mathsf{Optimize}_{\mathcal{L}_\mathsf{H}}$ *transformation. Clearly, the first, the third and the fifth extended denial are extended-subsumed by the first, the second and, respectively, the third extended denial in* $\mathsf{Unfold}_{\mathcal{L}_\mathsf{H}}(S)$ *and are thus eliminated. The second denial reduces to* $\leftarrow w(\mathbf{a})$*. In the fourth denial the equality* $X = \mathbf{a}$ *at level* 0 *is eliminated, thus substituting* X *with* $\mathbf{a}$ *in the whole extended denial. We obtain the following.*

$$\begin{array}{lll}\{\leftarrow & w(\mathbf{a}), & \\ \leftarrow & p(\mathbf{a},Y) \wedge \neg\exists(T,Z)[p(\mathbf{a},Z) \wedge p(T,Z) \wedge m(\mathbf{a}) \wedge w(T)] \wedge & \\ & \neg\exists(T,Z)[p(\mathbf{a},Z) \wedge p(T,Z) \wedge \mathbf{a} = \mathbf{a} \wedge w(T)] & \} \end{array}$$

For the last extended denial, first we can eliminate the trivially succeeding equality $\mathbf{a} = \mathbf{a}$ *from the body of the second NEE. Then we can consider that*

$$\leftarrow \neg\exists(T,Z)[p(\mathbf{a},Z) \wedge p(T,Z) \wedge m(\mathbf{a}) \wedge w(T)]$$

extended-subsumes

$$\leftarrow p(\mathbf{a},Y) \wedge \neg\exists(T,Z)[p(\mathbf{a},Z) \wedge p(T,Z) \wedge w(T)]$$

so we can eliminate by subsumption factoring the subsuming part and leave the subsumed one. The simplification procedure for $\mathcal{L}_\mathsf{H}$ *applied to* S *and* U *returns the following result.*

$$\begin{array}{lll}\mathsf{Simp}^U_{\mathcal{L}_\mathsf{H}}(S) = \{ & \leftarrow w(\mathbf{a}), & \\ & \leftarrow p(\mathbf{a},Y) \wedge \neg\exists(T,Z)[p(\mathbf{a},Z) \wedge p(T,Z) \wedge w(T)] & \} \end{array}$$

This coincides with the result given in [116], rewritten with our notation, with the only difference that they do not assume disjointness of IDB and EDB, so, in the latter extended denial, they have the extra conjunct $\neg\exists V[married\_to(\mathbf{a},V)]$*.*

4.2 Aggregates and arithmetic

Aggregates and arithmetic operators are among the most used and widespread facilities in database utilization. To be of practical use, any simplification procedure must support such constructs. In this section we present a set of rewrite rules that allow the decomposition of aggregate expressions into simpler ones that can then be simplified by a constraint solver for arithmetic expressions. The practical significance of these rules is demonstrated with an extensive set of examples.

4.2.1 A syntax for aggregates and arithmetic

In order to extend our framework with arithmetic and aggregates (which are not part of DATALOG and are not first-order expressible), a suitable syntax is needed. For this purpose, we assume throughout this section that terms are typed, so as to recognize numeric arguments.

We also assume that the language includes built-in *arithmetic constraints* ($<$, $\leq$, $>$, $\geq$), whose arguments are arithmetic expressions. An *arithmetic formula* is a formula in which all the predicates are arithmetic constraints. An *arithmetic expression* is a numeric term, an aggregate term, or a legal combination of arithmetic expressions via the four operations ($+$, $-$, $\cdot$, $/$), indicated in infix notation.

An *aggregate term* is an expression of the form $A_{[X]}(\exists X_1, \ldots, X_n F)$, where A is an aggregate operator, F is a defining formula, $X_1, \ldots, X_n$ (the *local variables*) are F's distinguished variables, and the optional X, if present, is the variable, called *aggregate variable*, among the $X_1, \ldots, X_n$ on which the aggregate function is calculated. The non-local variables in F are called the *global variables*; the argument of the aggregate is called the *key formula*. To simplify notation, inside key formulas we underline the local variables and do not indicate the existential quantifications. We also assume that the local variables of a key formula do not occur outside that key formula.

The aggregate operators we consider are: Cnt (count), Sum, Max, Min, Avg (average).

Example 4.2.1 *Given the person relation $p(name, age)$, $\mathsf{Avg}_Y(p(\underline{X}, \underline{Y}))$ is an aggregate term that indicates the average age of all persons. Let $r(X)$ indicate that X is a reviewer and $sub(X, Y)$ indicate that submission X is assigned to reviewer Y. Then $\leftarrow r(X) \wedge \mathsf{Cnt}(sub(\underline{Y}, X)) > 3$ is a denial requiring that no reviewer is assigned more than 3 submissions.*

The extension of $\mathcal{L}_{\mathsf{s}}$ in which aggregates and arithmetic expressions can occur is denoted by $\mathcal{L}_{\mathsf{A}}$.

4.2.2 Set vs. bag semantics

The semantics of aggregates depends on the semantics underlying the representation of data. Two different semantics are of interest: set semantics, as traditionally used in logic, and bag semantics, typical of relational database systems. A state D contains, for every extensional predicate p, a relation p^D; under set semantics, as we saw in definition 2.2.19 on page 16, p^D is the extension of p and is thus a set of tuples; under bag semantics, p^D is a bag (or multiset) of tuples, i.e., a set of $\langle tuple, multiplicity \rangle$ pairs, where the multiplicity is a positive integer. Similarly, any defining formula F defines a new finite relation F^D. Under set semantics, the extension consists of all different answers that F produces over D; under bag semantics, the tuples are the same as for set semantics and the multiplicity of each tuple is the number of times it can be derived over D. A complete formal account on bag semantics is given in [52]; we show here how the multiplicity of a query is calculated.

Suppose that Q is a *conjunctive query*, i.e., a query of the form $\Leftarrow p_1(\vec{X}_1) \wedge \cdots \wedge p_n(\vec{X}_n)$, where the p_i's are extensional predicates. An *assignment mapping* of a conjunctive query Q into a database D is an assignment of constants in D to the variables of Q such that

every conjunct in Q is mapped to a tuple in D. For an assignment mapping θ, we denote by $\theta(\vec{X})$ the constants to which θ maps $\vec{X}$ and by m_i the multiplicity of the tuple to which $p_i(\vec{X}_i)$ is mapped. Then the result due to θ of Q over D is the tuple $\theta(\vec{X})$ with multiplicity $m = m_1 \cdot \ldots \cdot m_n$, where $\vec{X}$ are the distinguished variables of Q. The result of a query Q over a database D is given by $\uplus_\theta r_\theta$, where θ is any assignment mapping of Q into D and r_θ the result due to θ, i.e., the result is the bag union ($\uplus$) of all results due to the possible assignment mappings.

The notion of multiplicity trivially extends to defining formulas, which can be seen as a generalization of conjunctive queries. Besides positive occurrences of extensional predicates, defining formulas can contain built-in atoms, negative atoms, intensional predicates, and disjunctions. Consider a range restricted query Q of the form $\Leftarrow P \wedge R$, where P is the body of a conjunctive query and R is a conjunction of built-in or negated extensional atoms. An assignment mapping of Q into a database D is an assignment of constants in D to the variables of Q such that every conjunct in Q is mapped to a tuple in D *and every built-in atom in R is satisfied and, for every negative atom $\neg A$ in R, A is not a tuple in D*. Then the multiplicity of the result is defined as before. If the query contains intensional predicates, their multiplicity is defined as that of the corresponding queries. If the query is of the form $\Leftarrow P_1 \vee \cdots \vee P_n$, then the multiplicity of the result is the sum of the multiplicities of the results for $\Leftarrow P_1, \ldots, \Leftarrow P_n$. For simplicity, in the remainder of the chapter we disregard intensional predicates, as this does not affect the discourse to follow.

Global variables occurring in an aggregate are expected to be range bound for a denial in $\mathcal{L}_\mathsf{A}$ to be safe. Given an assignment mapping θ for its global variables $\vec{Y}$, the aggregate $\mathsf{Cnt}(\exists\vec{X}F)$ refers to the sum of the multiplicities of the answers to $\Leftarrow F\{\vec{Y}/\theta(\vec{Y})\}$. Similarly, given an assignment mapping θ for its global variables $\vec{Y}$, the aggregate $\mathsf{Sum}_X(\exists\vec{X}F)$ refers to the number

$$\sum\{(X\sigma)\cdot m | F\sigma \text{ is an answer to } \Leftarrow F\{\vec{Y}/\theta(\vec{Y})\} \text{ and } m \text{ its multiplicity}\}.$$

The other aggregates are defined in a similar way. We note that Max and Min are indifferent of the semantics; for the other aggregates, we use the notation Cnt, Sum, Avg for bag semantics and Cnt_D, Sum_D, Avg_D when referring to the set of *distinct* tuples.

Note that, according to the definition of update (2.2.20), in both set and bag semantics, when an update U determines the deletion of a tuple $p(\vec{c})$ from a relation p^D in a state D, then *all* of its occurrences in p^D are removed, i.e., $D^U \not\models p(\vec{c})$. Addition of a tuple $p(c)$, under bag semantics, means either adding 1 to the existing multiplicity or creating one with value 1.

4.2.3 Rewrite rules for aggregates and arithmetic

In $\mathcal{L}_\mathsf{A}$, we can still use $\mathsf{After}_{\mathcal{L}_\mathsf{S}}$ to generate weakest preconditions. This may introduce disjunctions inside key formulas of aggregates.

Example 4.2.2 *Let* $\Gamma = \{\leftarrow \mathsf{Cnt}(p(\underline{X})) < 10\}$, $U = \{p(\mathbf{a})\}$, *then:*

$$\mathsf{After}^U_{\mathcal{L}_\mathsf{S}}(\Gamma) = \{\leftarrow \mathsf{Cnt}(p(\underline{X}) \vee \underline{X} = \mathbf{a}) < 10\}.$$

The new Cnt expression returned by $\mathsf{After}_{\mathcal{L}_S}$ should indicate an increment by one with respect to the original expression. In order to determine this effect, we need to divide the expression into smaller pieces that can possibly be used during the optimization of weakest preconditions. To do that, we introduce further sound rewrite rules for aggregates. Care must be taken when applying transformations that preserve logical equivalence on aggregates with bag semantics. Consider, for instance, $\mathsf{Sum}_X(p(\underline{X}) \wedge \underline{X} = 1)$ and $\mathsf{Sum}_X((p(\underline{X}) \wedge \underline{X} = 1) \vee (p(\underline{X}) \wedge \underline{X} = 1))$. Their key formulas are logically equivalent, but in the latter, the tuple $p(1)$ has double multiplicity. In other words, the results may differ because the number of ways in which the key formula may succeed matters.

With regard to arithmetic constraints and expressions, we assume the presence of a standard constraint solver for arithmetic, such as, e.g., [39, 105], in order to determine whether the rewrite rules below (introduced in definition 4.2.5) are applicable. A constraint solver can be regarded as an algorithm whose input is a set of arithmetic constraints over terms and variables and whose output is a smaller, but logically equivalent, set of arithmetic constraints or a "no" if there was no solution for the given problem; the given input is solved when the output set is a set of variable/term assignments. We use the notation $\mathcal{A} \rightsquigarrow_{\mathcal{C}} \mathcal{A}'$ to indicate that a constraint solver $\mathcal{C}$ receiving in input an arithmetic formula $\mathcal{A}$ outputs, in finite time, a smaller arithmetic formula $\mathcal{A}'$; conventionally, $\mathcal{A}'$ is *false* when there is no solution. We now extend the notion of subsumption to denials containing arithmetic constraints.

Definition 4.2.3 *Let $\mathcal{C}$ be a constraint solver for arithmetic and D_1, D_2 be denials of the form $\leftarrow C_1 \wedge \mathcal{A}_1$ and $\leftarrow C_2 \wedge \mathcal{A}_2$, respectively, where C_1, C_2 are conjunctions of literals without arithmetic constraints and $\mathcal{A}_1, \mathcal{A}_2$ arithmetic formulas. Then D_1 $\mathcal{C}$-subsumes D_2 iff there is a substitution σ such that each literal in $C_1\sigma$ occurs in C_2 and $\mathcal{A}_2 \wedge \neg\mathcal{A}_1\sigma \rightsquigarrow_{\mathcal{C}}$ false.*

Example 4.2.4 *Consider the following integrity constraints:*

$$\begin{aligned} \phi = &\quad \leftarrow \mathsf{Cnt}(p(\underline{X}, Y)) < 10 \wedge q(Y) \\ \psi = &\quad \leftarrow \mathsf{Cnt}(p(\underline{X}, b)) < 9 \wedge q(b) \wedge r(Z). \end{aligned}$$

For any constraint solver $\mathcal{C}$ for which

$$E < 9 \wedge E \geq 10 \rightsquigarrow_{\mathcal{C}} \textit{false}$$

we have that ϕ $\mathcal{C}$-subsumes ψ. Note that the arithmetic variable E is used as a shorthand for $\mathsf{Cnt}(p(\underline{X}, b))$.

To make the definitions to follow more readable, we introduce *conditional expressions*, i.e., arithmetic expressions written $C \ ? \ E_1 : E_2$, whose value is given by the value of E_1 if condition C holds, or by the value of E_2 otherwise. Similarly, we introduce two binary arithmetic operators max and min (not to be confused with the aggregates Max and Min) and define them in terms of conditional expressions. Square brackets in the subscript of an aggregate indicate that the rule applies both with and without the subscripted aggregate variable. Note that the aggregate variable and the global variables are to be treated as parameters during reduction, as they are not quantified inside an aggregate.

Definition 4.2.5 *Given two constraint theories Δ and Γ in $\mathcal{L}_{\mathsf{A}}$ and a constraint solver for arithmetic $\mathcal{C}$, $\mathsf{Optimize}_{\mathcal{C}}{}^{\Delta}(\Gamma)$ is the result of applying the following rewrite rules and those of $\mathsf{Optimize}_{\mathcal{L}_{\mathsf{S}}}$ (definition 3.2.28 on page 41) on Γ as long as possible. Here, X is a local variable, Y a global one, A, B, C are conjunctions of literals such that $\leftarrow A$, $\leftarrow B$, $\leftarrow C$ are all range restricted (C with no local variables), E_1, E_2 arithmetic expressions, $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$ arithmetic formulas, t, s terms, c a constant or parameter, F a key formula, Γ' a constraint theory, σ a substitution, ρ a renaming, Agg any aggregate. $\Phi(Arg)$ and $\Psi(Arg)$ are expressions indicating a set of denials and, respectively, a conjunction of literals, in which Arg occurs. The $\bot$ symbol indicates the result of an aggregate applied to an empty bag of tuples (e.g., 0 for Cnt and Sum).*

Rules for all aggregates

$$\begin{array}{rcl}
\mathsf{Agg}_{[X]}(A \wedge Y = t) & \Rightarrow & Y = t \;?\; \mathsf{Agg}_{[X]}(A\{Y/t\}) : \bot^{1} \\
\mathsf{Agg}_{[X]}(A) & \Rightarrow & \bot \textit{ if } (\leftarrow A)^{-} = \textit{true} \\
\mathsf{Agg}_{[X]}(A) & \Rightarrow & \mathsf{Agg}_{[X]}(B) \textit{ if } (\leftarrow A)^{-} = \leftarrow B \textit{ and } A \neq B \\
\{\leftarrow \mathsf{Agg}(F) \neq c\} \sqcup \Phi(\mathsf{Agg}(F)\rho) & \Rightarrow & \{\leftarrow \mathsf{Agg}(F) \neq c\} \cup \Phi(c)
\end{array}$$

Rules for Cnt ***and*** $\mathsf{Cnt}_{\mathsf{D}}$

$$\begin{array}{rcl}
\mathsf{Cnt}(A \vee B) & \Rightarrow & \mathsf{Cnt}(A) + \mathsf{Cnt}(B) \\
\mathsf{Cnt}_{\mathsf{D}}(A \vee B) & \Rightarrow & \mathsf{Cnt}_{\mathsf{D}}(A) + \mathsf{Cnt}_{\mathsf{D}}(B) - \mathsf{Cnt}_{\mathsf{D}}(A \wedge B) \\
\mathsf{Cnt}_{[\mathsf{D}]}(A \wedge t \neq s) & \Rightarrow & \mathsf{Cnt}_{[\mathsf{D}]}(A) - \mathsf{Cnt}_{[\mathsf{D}]}(A \wedge t = s) \\
\mathsf{Cnt}(\textit{true}) & \Rightarrow & 1 \\
\mathsf{Cnt}_{\mathsf{D}}(C) & \Rightarrow & C \;?\; 1 : 0
\end{array}$$

Rules for Sum, Avg ***and*** $\mathsf{Sum}_{\mathsf{D}}$, $\mathsf{Avg}_{\mathsf{D}}$

$$\begin{array}{rcl}
\mathsf{Sum}_{X}(A \vee B) & \Rightarrow & \mathsf{Sum}_{X}(A) + \mathsf{Sum}_{X}(B) \\
\mathsf{Sum}_{\mathsf{D}\,X}(A \vee B) & \Rightarrow & \mathsf{Sum}_{\mathsf{D}\,X}(A) + \mathsf{Sum}_{\mathsf{D}\,X}(B) - \mathsf{Sum}_{\mathsf{D}\,X}(A \wedge B) \\
\mathsf{Sum}_{[\mathsf{D}]\,X}(A \wedge t \neq s) & \Rightarrow & \mathsf{Sum}_{[\mathsf{D}]\,X}(A) - \mathsf{Sum}_{[\mathsf{D}]\,X}(A \wedge t = s) \\
\mathsf{Sum}_{[\mathsf{D}]\,X}(A \wedge \underline{X} = c) & \Rightarrow & c \cdot \mathsf{Cnt}_{[\mathsf{D}]}(A\{\underline{X}/c\}) \\
\mathsf{Avg}_{[\mathsf{D}]\,X}(F) & \Rightarrow & \mathsf{Sum}_{[\mathsf{D}]\,X}(F)/\mathsf{Cnt}_{[\mathsf{D}]}(F)
\end{array}$$

Rules for Max ***and*** Min

$$\begin{array}{rcl}
\mathsf{Max}_{X}(A \vee B) & \Rightarrow & \mathsf{max}(\mathsf{Max}_{X}(A), \mathsf{Max}_{X}(B)) \\
\mathsf{Max}_{X}(A \wedge \underline{X} = c) & \Rightarrow & A\{X/c\} \;?\; c : \bot \\
\mathsf{Min}_{X}(A \vee B) & \Rightarrow & \mathsf{min}(\mathsf{Min}_{X}(A), \mathsf{Min}_{X}(B)) \\
\mathsf{Min}_{X}(A \wedge X = c) & \Rightarrow & A\{X/c\} \;?\; c : \bot
\end{array}$$

Rules for conditional expressions

$$\begin{array}{rcl}
\{\leftarrow \Psi(C \;?\; E_1 : E_2)\} & \Rightarrow & \{\leftarrow C \wedge \Psi(E_1),\ \leftarrow \neg C \wedge \Psi(E_2)\} \\
\mathsf{max}(E_1, E_2) & \Rightarrow & E_1 > E_2 \;?\; E_1 : E_2 \\
\mathsf{min}(E_1, E_2) & \Rightarrow & E_1 < E_2 \;?\; E_1 : E_2
\end{array}$$

[1] Provided that t is not a local variable.

Rules for the interaction with the arithmetic constraint solver

$$\begin{array}{rcl}
\{\phi\} \sqcup \Gamma' & \Rightarrow & \{\psi^-\} \cup \Gamma' \textit{ if } (\{\phi\} \sqcup \Gamma' \cup \Delta) \vdash_R \psi \\
 & & \textit{and } \psi^- \textit{ strictly } \mathcal{C}\textit{-subsumes } \phi \\
\{\leftarrow A \wedge \mathcal{A}_1\} & \Rightarrow & \{\leftarrow A \wedge \mathcal{A}_2\} \textit{ if } \mathcal{A}_1 \rightsquigarrow_{\mathcal{C}} \mathcal{A}_2 \\
\{\leftarrow A \wedge \mathcal{A}_1,\ \leftarrow A\sigma \wedge \mathcal{A}_2\} & \Rightarrow & \{\leftarrow A \wedge \mathcal{A}_1,\ \leftarrow A\sigma \wedge \mathcal{A}_3\} \textit{ if } \mathcal{A}_1\sigma \vee \mathcal{A}_2 \rightsquigarrow_{\mathcal{C}} \mathcal{A}_3{}^{2} \\
\{\leftarrow A \wedge X \gtrless t\} & \Rightarrow & \{\leftarrow A\} \textit{ if } \{t, X\} \cap \mathrm{vars}(A) = \emptyset \textit{ and } t \textit{ is not } X
\end{array}$$

As mentioned, (global) variables occurring in aggregates or arithmetic expressions must be range bound by other literals in a denial, for it to be safe. We observe that these rules preserve safeness, since they only modify the arithmetic part of denials.

The next example illustrates a few, simple rule applications.

Example 4.2.6 *Consider the aggregates* $A_1 = \mathsf{Cnt}(p(\underline{X}) \wedge \underline{X} \neq \mathbf{a})$ *and* $A_2 = \mathsf{Sum}_X(p(\underline{X}) \wedge \underline{X} \neq \mathbf{a})$, $\mathbf{a}$ *a numeric parameter. We have:*

$$\begin{array}{rcl}
A_1 & \Rightarrow & \mathsf{Cnt}(p(\underline{X})) - \mathsf{Cnt}(p(\underline{X}) \wedge \underline{X} = \mathbf{a}) \\
 & \Rightarrow & \mathsf{Cnt}(p(\underline{X})) - \mathsf{Cnt}(p(\mathbf{a})). \\
A_2 & \Rightarrow & \mathsf{Sum}_X(p(\underline{X})) - \mathsf{Sum}_X(p(\underline{X}) \wedge \underline{X} = \mathbf{a}) \\
 & \Rightarrow & \mathsf{Sum}_X(p(\underline{X})) - \mathbf{a} \cdot \mathsf{Cnt}(p(\mathbf{a})).
\end{array}$$

Note that the fourth Sum *rule in definition 4.2.5 indicates that, when the value of the aggregate variable is known, the sum will equal that value multiplied by the number of times the aggregate formula succeeds.*

The simplification procedure can now be extended with these new rules.

Definition 4.2.7 *Consider a schema* $S = \langle IDB, \Gamma \rangle \in \mathcal{L}_{\mathsf{A}}$ *and an update* U. *Let* $\mathcal{C}$ *be a constraint solver for arithmetic and let* $\mathsf{Unfold}_{\mathcal{L}_{\mathsf{S}}}(S) = \langle \emptyset, \Gamma' \rangle$. *We define*

$$\mathsf{Simp}_{\mathcal{C}}{}^{U}(S) = \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma'}(\mathsf{After}^{U}_{\mathcal{L}_{\mathsf{S}}}(S)).$$

Since the rules of $\mathsf{Optimize}_{\mathcal{C}}$ are equivalence preserving, the correctness of $\mathsf{Simp}_{\mathcal{C}}$ can be stated, as for Simp, in theorem 4.2.8, with the extra assumption that the constraint solver for arithmetic always terminates. Termination is then guaranteed, because it can easily be shown that the rules of $\mathsf{Optimize}_{\mathcal{C}}$ cannot go into a loop[3].

Theorem 4.2.8 *Given a constraint solver for arithmetic* $\mathcal{C}$ *that terminates on any input,* $\mathsf{Simp}_{\mathcal{C}}$ *terminates on any input and, for any schema* $S \in \mathcal{L}_{\mathsf{A}}$ *and update* U, $\mathsf{Simp}_{\mathcal{C}}{}^{U}(S)$ *is a CWP of* S *with respect to* U.

[2]Note that when σ is a renaming, the first produced denial is redundant and will be eliminated by the previous rule.

[3]The rules in the last subgroup follow the same termination principle as the rules in $\mathsf{Optimize}_{\mathcal{L}_{\mathsf{S}}}$, whereas the rules in the other subgroups remove equalities, non-equalities and disjunctions from aggregates, which are not reintroduced by the rules in the last subgroup.

4.2.4 Examples

We show now a series of examples that demonstrate the behavior of the rules and the simplification procedure in various cases; for readability, we leave out some of the trivial steps and only consider empty *IDB*'s.

Example 4.2.9 (4.2.2 continued) *Let* $\Gamma = \{\leftarrow \mathsf{Cnt}(p(\underline{X})) < 10\}$, $U = \{p(\mathbf{a})\}$, *then:*

$$\begin{array}{rcl}
\mathsf{Simp}_{\mathcal{C}}{}^{U}(\Gamma) & = & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(\{\leftarrow \mathsf{Cnt}(p(\underline{X}) \vee \underline{X} = \mathbf{a}) < 10\}) \\
& = & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(\{\leftarrow \mathsf{Cnt}(p(\underline{X})) + \mathsf{Cnt}(\underline{X} = \mathbf{a}) < 10\}) \\
& = & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(\{\leftarrow \mathsf{Cnt}(p(\underline{X})) + 1 < 10\}) \\
& = & \emptyset.
\end{array}$$

The update increments the count of p-tuples, which was known (Γ) *to be at least* 10 *before the update, so this increment cannot undermine the validity of* Γ *itself. The last step, obtained via* $\mathcal{C}$*-subsumption, allows one to conclude that no check is necessary to guarantee consistency of the updated database state.*

Example 4.2.10 *Let* $\Gamma = \{\leftarrow \mathsf{Cnt}_{\mathsf{D}}(p(\underline{X})) \neq 10\}$ *(there must be exactly* 10 distinct *p-tuples) and* $U = \{p(\mathbf{a})\}$. *With a set semantics, the increment of the count depends on the existence of the tuple* $p(\mathbf{a})$ *in the state:*

$$\begin{array}{rcll}
\mathsf{Simp}_{\mathcal{C}}{}^{U}(\Gamma) & = & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(& \{ \leftarrow \mathsf{Cnt}_{\mathsf{D}}(p(\underline{X}) \vee \underline{X} = \mathbf{a}) \neq 10\}) \\
& = & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(& \{ \leftarrow \mathsf{Cnt}_{\mathsf{D}}(p(\underline{X})) + 1 - \mathsf{Cnt}_{\mathsf{D}}(p(\mathbf{a})) \neq 10\}) \\
& = & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(& \{ \leftarrow 10 + 1 - p(\mathbf{a})\ ?\ 1 : 0 \neq 10\}) \\
& = & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(& \{ \leftarrow p(\mathbf{a}) \wedge 10 + 1 - 1 \neq 10, \\
& & & \leftarrow \neg p(\mathbf{a}) \wedge 10 + 1 - 0 \neq 10\}) \\
& = & \{\leftarrow \neg p(\mathbf{a})\}. &
\end{array}$$

The arithmetic constraint solver intervenes in the last step suggesting that $10+1-1 \neq 10$ *is a falsity and* $10 + 1 - 0 \neq 10$ *a tautology. In the second step,* $\mathsf{Cnt}_{\mathsf{D}}(p(X))$ *is replaced by* 10 *because of the hypothesis* Γ *by applying the last rule of the first subgroup of rules in definition 4.2.5. The result indicates that, if* $\mathbf{a}$ *is not among the values already in p, then the constraint will be violated.*

Example 4.2.11 *When global variables occur, conditional expressions are used to separate different cases. Suppose that* $p(X, Y)$ *indicates that patient* X *is hospitalized with illness* Y *and that* $q(Y)$ *means that illness* Y *requires quarantine. Let* $\Gamma = \{\leftarrow \mathsf{Cnt}(p(\underline{X}, Y)) > 10 \wedge q(Y)\}$ *be a hospital policy imposing that for each illness requiring quarantine there are no more than* 10 *hospitalized patients having that illness. Let* $U = \{p(\mathbf{a}, \mathbf{b})\}$ *be the addition of a patient-illness tuple. The calculation of* $\mathsf{Simp}_{\mathcal{C}}{}^{U}(\Gamma)$ *is as follows.*

$$\begin{array}{rcll}
 & & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(\mathsf{After}^{U}_{\mathcal{L}_{\mathsf{S}}}(\Gamma)) & \\
= & & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(& \{ \leftarrow \mathsf{Cnt}(p(\underline{X}, Y)) + \mathsf{Cnt}(\underline{X} = \mathbf{a} \wedge Y = \mathbf{b}) > 10 \wedge q(Y)\}) \\
= & & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(& \{ \leftarrow \mathsf{Cnt}(p(\underline{X}, Y)) + \mathsf{Cnt}(Y = \mathbf{b}) > 10 \wedge q(Y)\}) \\
= & & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(& \{ \leftarrow \mathsf{Cnt}(p(\underline{X}, Y)) + Y = \mathbf{b}\ ?\ 1 : 0 > 10 \wedge q(Y)\}) \\
= & & \mathsf{Optimize}_{\mathcal{C}}{}^{\Gamma}(& \{ \leftarrow Y = \mathbf{b} \wedge \mathsf{Cnt}(p(\underline{X}, Y)) + 1 > 10 \wedge q(Y), \\
 & & & \leftarrow Y \neq \mathbf{b} \wedge \mathsf{Cnt}(p(\underline{X}, Y)) + 0 > 10 \wedge q(Y)\}) \\
= & & \{\leftarrow \mathsf{Cnt}(p(\underline{X}, \mathbf{b})) > 9 \wedge q(\mathbf{b})\}. &
\end{array}$$

In the last step, the second constraint is $\mathcal{C}$-subsumed by Γ and thus eliminated. The result reads as follows: if illness **b** *requires quarantine, then there must not already be* 9 *hospitalized patients having that illness.*

Example 4.2.12 *We propose now an example with a complex update. Let $e(X, Y, Z)$ represent employees of a company, where X is the name, Y the years of service and Z the salary. The company's policy is expressed by*

$$\Gamma = \{ \quad \leftarrow e(X,Y,Z) \wedge Z = \mathsf{Max}_W(e(\underline{U},\underline{V},\underline{W})) \wedge Y < 5, \\ \leftarrow e(X,Y,Z) \wedge Z = \mathsf{Max}_W(e(\underline{U},\underline{V},\underline{W})) \wedge Y > 8 \quad \}$$

i.e., the seniority of the best paid employee must be between 5 *and* 8 *years, and*

$$U = \{e(X,Y,Z) \Leftarrow e(X,Y',Z) \wedge Y = Y' + 1\}$$

is the update transaction that is executed at the end of the year to increase the seniority of all employees. Note that the application of $\mathsf{After}^U_{\mathcal{L}_\mathsf{S}}$ to a literal of the form $e(X, Y, Z)$ generates

$$e(X,Y',Z) \wedge Y = Y' + 1.$$

The aggregate expression is transformed by $\mathsf{After}^U_{\mathcal{L}_\mathsf{S}}(\Gamma)$ into

$$\mathsf{Max}_W(e(\underline{U},\underline{V}',\underline{W}) \wedge \underline{V} = \underline{V}' + 1)$$

which is simplified by $\mathsf{Optimize}_\mathcal{C}$ into $\mathsf{Max}_W(e(\underline{U},\underline{V}',\underline{W}))$ and thus coincides, modulo renaming, with the original one in Γ. After the optimization steps described above, the result of $\mathsf{After}^U_{\mathcal{L}_\mathsf{S}}(\Gamma)$ is transformed into:

$$\{ \quad \leftarrow e(X,Y',Z) \wedge Y = Y' + 1 \wedge Z = \mathsf{Max}_W(e(\underline{U},\underline{V},\underline{W})) \wedge Y < 5, \\ \leftarrow e(X,Y',Z) \wedge Y = Y' + 1 \wedge Z = \mathsf{Max}_W(e(\underline{U},\underline{V},\underline{W})) \wedge Y > 8\}.$$

We assume that the arithmetic constraint solver can reduce $Y = Y' + 1 \wedge Y < 5$ into $Y' < 4 \wedge Y < 5$ for the first denial and $Y = Y' + 1 \wedge Y > 8$ into $Y' > 7 \wedge Y > 8$ for the second one. Then the first denial is subsumed by the first denial in Γ and we finally obtain:

$$\mathsf{Simp}_\mathcal{C}{}^U(\Gamma) = \{\leftarrow e(X,Y',Z) \wedge Z = \mathsf{Max}_W(e(U,V,W)) \wedge Y' > 7\},$$

i.e., the person with maximum salary must not exceed 7 *years of seniority when the update is applied.*

4.2.5 Discussion

The quality of the simplification produced by $\mathsf{Simp}_\mathcal{C}$ is highly dependent on the precision of the constraint solver for arithmetic, which might be unable to reduce particular combinations of arithmetic constraints. We also note that the interaction between the solver and the simplification procedure, characterized by the last subgroup of rules of definition 4.2.5, captures many interesting cases in which arithmetic-based simplifications are possible, even across different constraints in a theory (second rule in the last subgroup). These

rules are the result of several rounds of fine-tuning of the simplification process based on concrete cases and can fully solve all the examples presented in this section; however, we cannot exclude that more complex cases escape this definition. In this respect, further study is required to establish when minimality of the result can be guaranteed in the presence of aggregates and arithmetic expressions.

The constraint solver for finite domains described in [39] and available in current Prolog systems is able to handle the arithmetic part of most of the examples and rules described in this chapter (for integers). An implementation of the solver and the rules that characterize its interaction with the simplification procedure is also possible with the language of Constraint Handling Rules [85], which is an extremely versatile tool for constraint programming. For a survey on constraint solvers and constraint logic programming in general we refer to [105].

The simplification problem for integrity constraints containing aggregates seems, with few rare exceptions, to have been largely ignored. In [66], the most comprehensive simplification method handling aggregates we are aware of, Das extends the simplification method of [122] and applies it to aggregates based on an infinitary axiomatization of aggregates for DATALOG. However, the hypotheses about consistency of the database prior to the update is not fully exploited; consequently, the simplified test is a set of instances of the original constraints. In our example 4.2.9, for instance, Das' method would return the initial constraint theory, whereas we were able to conclude that no check was needed.

A definition of the semantics of SQL with aggregates is given in [166], where it is shown how to translate a subset of SQL into relational calculus and algebra and general strategies for query optimization are investigated for such cases. Further investigation on the semantics of aggregates is given in [109, 73, 169]. We also point out that query containment checking with aggregates is known to be more complex than in pure DATALOG. Recent works on this topic, providing several complexity results, are [62, 7].

4.3 Recursion

Simplification in the context of recursive databases is an extremely complicated issue for which, to our knowledge, no general and satisfactory solution has been proposed; in many cases, no essential simplification of the original integrity constraints is even possible. However, recursion is a central feature of deductive databases that allows the expression of complex query problems to be formulated within a declarative query language. To this end, we can mention flexible query answering based on taxonomies stored in the database, and various kinds of path-finding problems, such as network routing and travel planning. The introduction of recursion (since 1999 in the SQL standard [103] as *stratified linear* recursion based on fixpoint semantics) naturally raises a need for a satisfactory treatment of recursion in integrity constraints.

Many researchers have examined the expressive power of different classes of recursion and have described canonical problems for each class. Such investigations are highly relevant, as recursion makes deductive databases more expressive than first-order query languages, in that, thanks to fixpoint semantics, transitive closure of relations can be formulated. This gain in expressiveness and other comparisons between deductive database languages and first-order languages are discussed in [64].

The class of problems that can be expressed using linear recursion is very broad. An extensive discussion of the different kinds of linear recursion is found in [168], where recursive rules are rewritten as graphs and classified accordingly.

Furthermore, many non-linear problems have an equivalent linear formulation. In [6], it is explained how to produce a linear version of non-linear problems, such as *piecewise linear* DATALOG programs, and regular and pseudo-regular *chain queries*. In addition to that, there are problems that are formulated recursively but that are not inherently recursive, and therefore have a non-recursive counterpart [51][4]. However, there exist non-linearizable problems [5], but it is known that *bilinear* recursion is the hardest case [42], i.e., any database that is neither linear nor bilinear is equivalent to a database that is at most bilinear. As for *mutual* recursion, it has been shown that it can always be reduced to recursion over a single relation [49].

The task of efficiently evaluating a recursive query has been tackled by a large body of research and has given rise to many different methods using different combinations of strategies, approaches and implementations (top-down vs. bottom-up, compiled vs. interpreted, iterative vs. recursive, ...), as extensively described in [15]. Among these, *magic sets* and *counting* are probably the best known evolutions of the basic approach called *naive evaluation*.

In this section we describe a simplification procedure that applies to an important subclass of linear recursion. Discussion and comparison with related work is provided in section 4.3.3.

4.3.1 A simplification pattern for ordered linear recursion

Let $\mathcal{L}_{\mathsf{R}}$ be the database language which is defined as $\mathcal{L}_{\mathsf{S}}$ (definition 3.2.3 on page 30), but only requiring the database to be stratified, as opposed to the stronger requirement of having an acyclic (starred) dependency graph.

With the application of the After operator, occurrences of recursive predicates in integrity constraints are replaced by new recursive predicates defined in terms of new recursive views. Unfolding for $\mathcal{L}_{\mathsf{R}}$ is then redefined as in definition 3.2.10, but with the difference that, if a predicate is recursive, then it is not replaced by its defining formula (we will indicate it as $\mathsf{Unfold}_{\mathcal{L}_{\mathsf{R}}}$).

Definition 4.3.1 *Let $S = \langle IDB, \Gamma \rangle$ be a database schema in $\mathcal{L}_{\mathsf{R}}$. We define $\mathsf{Unfold}_{\mathcal{L}_{\mathsf{R}}}(S)$ as the schema $\langle \mathcal{R}, \Gamma' \rangle$, where $\mathcal{R}$ is the largest subset of IDB such that pred($\mathcal{R}$) is a set of recursive predicates and Γ' is the set of denials obtained by iterating the two following steps as long as possible:*

1. *replace, in Γ, each occurrence of an atom of the form $p(\vec{t})$ by $F^p\{\vec{X}/\vec{t}\}$, where $p \in (pred(IDB) \setminus pred(\mathcal{R}))$, F^p is p's defining formula and $\vec{X}$ its distinguished variables. If no replacement was made, then stop;*

2. *transform the resulting formula into a set of denials according to the following patterns:*

   - *$\leftarrow A \wedge (B_1 \vee B_2)$ is replaced by $\leftarrow A \wedge B_1$ and $\leftarrow A \wedge B_2$;*

[4] But determining whether a problem is inherently recursive is in general undecidable.

- $\leftarrow A \wedge \neg(B_1 \vee B_2)$ *is replaced by* $\leftarrow A \wedge \neg B_1 \wedge \neg B_2$*;*
- $\leftarrow A \wedge \neg(B_1 \wedge B_2)$ *is replaced by* $\leftarrow A \wedge \neg B_1$ *and* $\leftarrow A \wedge \neg B_2$*.*

Trivially, for any schema $S \in \mathcal{L}_{\mathsf{R}}$, we have $\mathsf{Unfold}_{\mathcal{L}_{\mathsf{R}}}(S) \equiv S$.

After unfolding, the simplification process could then proceed as in $\mathcal{L}_{\mathsf{S}}$, by ignoring the *IDB*, but then the resulting constraint theory would hardly be any simpler than the original one, at least if the update affects some recursively defined predicate.

In the following we describe an important class of linear recursion that embraces some of the most commonly used recursive patterns (such as left- and right-linear recursion[135]), known as *ordered linear recursion* (OLR) [154]. For these cases we are able to refine the simplification process, by possibly eliminating the introduction of new recursive views.

Definition 4.3.2 *A predicate r is an OLR predicate if it is defined as follows*

$$\begin{array}{ll} \{ & r(\vec{X},\vec{Y}) \leftarrow q(\vec{X},\vec{Y}) \\ & r(\vec{X},\vec{Y}) \leftarrow p(\vec{X},\vec{Z}) \wedge r(\vec{Z},\vec{Y}) \quad \}, \end{array} \tag{4.2}$$

where p and q are predicates on which r does not depend and $\vec{X},\vec{Y},\vec{Z}$ are disjoint sequences of distinct variables. The first rule is the exit rule*, while the other is the* recursive rule*.*

There may in principle be several exit rules and recursive rules for the same OLR predicate r; however, these can always be reduced to one single exit rule and recursive rule by introducing suitable new views. Note thus that p and q need not be extensional predicates.

We first transform the definition of r as to decompose it in two parts: a non-recursive definition (4.3) and a transitive closure definition r_p (4.4). If p and q are the same predicate, then no transformation is needed, as the definition of r is already the transitive closure of p. Otherwise we replace r's definition with the union of the two following sets of rules:

$$\begin{array}{ll} \{ & r(\vec{X},\vec{Y}) \leftarrow q(\vec{X},\vec{Y}) \\ & r(\vec{X},\vec{Y}) \leftarrow r_p(\vec{X},\vec{Z}) \wedge q(\vec{Z},\vec{Y}) \quad \}. \end{array} \tag{4.3}$$

$$\begin{array}{ll} \{ & r_p(\vec{X},\vec{Y}) \leftarrow p(\vec{X},\vec{Y}) \\ & r_p(\vec{X},\vec{Y}) \leftarrow p(\vec{X},\vec{Z}) \wedge r_p(\vec{Z},\vec{Y}) \quad \}. \end{array} \tag{4.4}$$

Note that the argument is perfectly symmetric when r's recursive rule is of the form

$$r(\vec{X},\vec{Y}) \leftarrow r(\vec{X},\vec{Z}) \wedge p(\vec{Z},\vec{Y}).$$

In this case the second rule in (4.3) becomes

$$r(\vec{X},\vec{Y}) \leftarrow q(\vec{X},\vec{Z}) \wedge r_p(\vec{Z},\vec{Y})$$

and r_p is defined exactly as in (4.4).

All occurrences of r in a constraint theory can now be unfolded with respect to the rules in (4.3), which introduce q and r_p, the latter being the transitive closure of p, defined in (4.4). Intuitively, it is easy to characterize the set of tuples that are added to r_p upon addition of a p-tuple, as r_p can be thought of as a representation of paths of a directed

graph of p-arcs. Suppose for the moment that update U is the addition of tuple $\langle \vec{\mathbf{a}}, \vec{\mathbf{b}} \rangle$ to p; then all added r_p paths are those that pass by the new p-arc and that were not there before the update. If we indicate as $\delta_U^+ r_p(\vec{X}, \vec{Y})$ the fact that there is a new path from $\vec{X}$ to $\vec{Y}$ after the update U, this circumstance can be expressed by the following rule:

$$\delta_U^+ r_p(\vec{X}, \vec{Y}) \leftarrow (r_p(\vec{X}, \vec{\mathbf{a}}) \vee \vec{X} = \vec{\mathbf{a}}) \wedge (r_p(\vec{\mathbf{b}}, \vec{Y}) \vee \vec{Y} = \vec{\mathbf{b}}) \wedge \neg r_p(\vec{X}, \vec{Y}).$$

However, in the general case U is not necessarily a single tuple update, so $\delta_U^+ r_p$ needs, in general, to be characterized in terms of "r_p in the updated state", as specified in definition 4.3.3.

Definition 4.3.3 *Let U be an update and r_p a recursive predicate that expresses the transitive closure of a non-recursive predicate p in a schema S in $\mathcal{L}_{\mathsf{R}}$; let r_p^U, p^U be the predicates replacing r_p, p in S to obtain $S' = \mathsf{After}^U(S)$, respectively, as in definition 3.2.5 on page 31. Let $OLR(r_p, S')$ be the following set of rules:*

$$\begin{array}{{}rl} \{ \quad r_p^U(\vec{X}, \vec{Y}) \leftarrow & (r_p(\vec{X}, \vec{Y}) \wedge \neg \delta_U^- r_p^U(\vec{X}, \vec{Y})) \vee \delta_U^+ r_p^U(\vec{X}, \vec{Y}), \\ \delta_U^+ r_p(\vec{X}, \vec{Y}) \leftarrow & (r_p^U(\vec{X}, \vec{W}_1) \vee \vec{X} = \vec{W}_1) \wedge (r_p^U(\vec{W}_2, \vec{Y}) \vee \vec{Y} = \vec{W}_2) \wedge \\ & \delta_U^+ p(\vec{W}_1, \vec{W}_2) \wedge \neg r_p(\vec{X}, \vec{Y}), \\ \delta_U^- r_p(\vec{X}, \vec{Y}) \leftarrow & (r_p(\vec{X}, \vec{W}_1) \vee \vec{X} = \vec{W}_1) \wedge (r_p(\vec{W}_2, \vec{Y}) \vee \vec{Y} = \vec{W}_2) \wedge \\ & \delta_U^- p(\vec{W}_1, \vec{W}_2) \wedge \neg r_p^U(\vec{X}, \vec{Y}), \\ \delta_U^+ p(\vec{X}) \leftarrow & p^U(\vec{X}) \wedge \neg p(\vec{X}), \\ \delta_U^- p(\vec{X}) \leftarrow & \neg p^U(\vec{X}) \wedge p(\vec{X}) \qquad \} \end{array}$$

We define $OLR(S')$ as the schema obtained from S' by replacing the clauses defining each transitive closure predicate r_p^U by $OLR(r_p^U, S')$.

Proposition 4.3.4 *Let S' be as in definition 4.3.3. Then $OLR(S') \equiv S'$.*

Proof *(Sketch) The constraint theories of $OLR(S')$ and S' are identical. To see that the replacement of r_p^U's definition does not affect its evaluation in any database, it suffices to consider that there is an isomorphism between r_p and the transitive closure of a graph of p-arcs [3] and that:*

- *a path is added (deleted) if it exists in the new (old) graph, it passes by an added (deleted) arc and it does not exist in the old (new) graph.*

The isomorphism is as follows: $p(\vec{X}, \vec{Y})$ indicates a (directed) arc between node $\vec{X}$ and node $\vec{Y}$ in the original graph, $p^U(\vec{X}, \vec{Y})$ an arc in the new graph, $r_p(\vec{X}, \vec{Y})$ a path between node $\vec{X}$ and node $\vec{Y}$ in the original graph, $r_p^U(\vec{X}, \vec{Y})$ a path in the new graph, $\delta_U^+ p(\vec{X})$ an added arc, $\delta_U^- p(\vec{X})$ a deleted arc, $\delta_U^+ r_p(\vec{X}, \vec{Y})$ an added path, $\delta_U^- r_p(\vec{X}, \vec{Y})$ a deleted path. □

The above isomorphism allows us also to state that if $\delta_U^+ p$ corresponds to the addition of 0 or 1 arcs, then a more convenient expression can be found for $\delta_U^+ r_p$, that only refers to r_p and not to r_p^U.

Proposition 4.3.5 *Consider the set of rules in definition 4.3.3. If* $\delta_U^+ p(\vec{W}_1, \vec{W}_2) \equiv \vec{W}_1 = \vec{c}_1 \wedge \vec{W}_2 = \vec{c}_2 \wedge A$*, where* A *is a conjunction of literals, and* $\vec{c}_1, \vec{c}_2$ *are vectors of constants or parameters, then the definition of* $\delta_U^+ r_p$ *can be written as follows.*

$$\begin{array}{rl} \delta_U^+ r_p(\vec{X}, \vec{Y}) \leftarrow & (r_p(\vec{X}, \vec{c}_1) \vee \vec{X} = \vec{c}_1) \wedge (r_p(\vec{c}_2, \vec{Y}) \vee \vec{Y} = \vec{c}_2) \wedge \\ & \vec{W}_1 = \vec{c}_1 \wedge \vec{W}_2 = \vec{c}_2 \wedge A \wedge \neg r_p(\vec{X}, \vec{Y}). \end{array} \tag{4.5}$$

The last literal in the body of (4.5) is also known as the *effectiveness test*.

A sufficient condition to test the applicability of proposition 4.3.5 is given as follows, using $\mathsf{After}_{\mathcal{L}_\mathsf{S}}$ and $\mathsf{Optimize}_{\mathcal{L}_\mathsf{S}}$. Let us first consider that, using the rules of definition 4.3.3, we have $\delta_U^+ p(\vec{X}, \vec{Y}) \equiv p^U(\vec{X}, \vec{Y}) \wedge \neg p(\vec{X}, \vec{Y})$. Then, writing all conjuncts as denials and using the definition of $\mathsf{After}_{\mathcal{L}_\mathsf{S}}$, we have $\delta_U^+ p(\vec{X}, \vec{Y}) \equiv \mathsf{After}^U_{\mathcal{L}_\mathsf{S}}(\{\leftarrow \neg p(\vec{X}, \vec{Y})\}) \cup \{\leftarrow p(\vec{X}, \vec{Y})\}$. Note that the variables in $\delta_U^+ p(\vec{X}, \vec{Y})$ are not quantified and therefore are to be treated as parameters. The expression $\mathsf{Optimize}^{\emptyset}_{\mathcal{L}_\mathsf{S}}(\mathsf{After}^U_{\mathcal{L}_\mathsf{S}}(\{\leftarrow \neg p(\vec{X}, \vec{Y})\}) \cup \{\leftarrow p(\vec{X}, \vec{Y})\})$ is then also equivalent to $\delta_U^+ p(\vec{X}, \vec{Y})$. Proposition 4.3.5 is applicable if the resulting constraint theory contains two constraints of the form $\leftarrow \vec{c}_1 \neq \vec{X}$ and $\leftarrow \vec{c}_2 \neq \vec{Y}$, i.e., if it matches the pattern given for $\delta_U^+ p$. This condition is, however, only sufficient, as the described expression may not reduce to the desired form in all cases.

Definition 4.3.6 *Let* U *be an update and* S *a schema in* $\mathcal{L}_\mathsf{R}$*. Let* S^* *be the same as* S *but in which, for all OLR predicate* r*, its definition (4.2) is replaced by (4.3) and (4.4). Then* $\mathsf{After}_{\mathcal{L}_\mathsf{R}}{}^U(S)$ *is defined as* $\mathsf{Unfold}_{\mathcal{L}_\mathsf{R}}(OLR(\mathsf{After}^U(S^*)))$*.*

Let $\mathcal{R}$ be a set of rules defining a transitive closure predicate r_p (as the rules in (4.4)). We denote with $den^*(\mathcal{R})$ the following set of denials:

$$\begin{array}{ll} \{ & \leftarrow \neg r_p(\vec{X}, \vec{Y}) \wedge p(\vec{X}, \vec{Y}), \\ & \leftarrow \neg r_p(\vec{X}, \vec{Y}) \wedge p(\vec{X}, \vec{Z}) \wedge r_p(\vec{Z}, \vec{Y}), \\ & \leftarrow \neg r_p(\vec{X}, \vec{Y}) \wedge r_p(\vec{X}, \vec{Z}) \wedge r_p(\vec{Z}, \vec{Y}) \quad \} \end{array}$$

The first two denials are merely a rewriting of the "if" part of the rules; the third denial captures the knowledge that the linear and the bilinear definition of the transitive closure are equivalent. Such denials can be used in the optimization phase and typically allow simplifying the effectiveness test. Let us denote with $den(A \leftarrow B)$ the denial $\leftarrow \neg A \wedge B$. Let $\mathcal{R} = \mathcal{R}' \cup \mathcal{R}_1 \cup \ldots \cup \mathcal{R}_n$ be a set of rules, where each $\mathcal{R}_i$ is a set of rules defining transitive closure predicate r_{p_i} and none of the predicates in $pred(\mathcal{R}')$ is a transitive closure predicate; then, with $den(\mathcal{R})$ we denote the set $\{den(C) \mid C \in \mathcal{R}'\} \cup den^*(\mathcal{R}_1) \cup \ldots \cup den^*(\mathcal{R}_n)$. We can now define the $\mathsf{Simp}_{\mathcal{L}_\mathsf{R}}$ operator by optimizing the result of $\mathsf{After}_{\mathcal{L}_\mathsf{R}}$ using the denial version of the recursive rules.

Definition 4.3.7 *Let* U *be an update and* $S = \langle IDB, \Gamma \rangle$ *a schema in* $\mathcal{L}_\mathsf{R}$*. Let* $\langle \mathcal{R}_\Delta, \Delta \rangle = \mathsf{Unfold}_{\mathcal{L}_\mathsf{R}}(S)$ *and* $\langle \mathcal{R}, \Gamma' \rangle = \mathsf{After}_{\mathcal{L}_\mathsf{R}}{}^U(S)$*, and let* $\Sigma = \mathsf{Optimize}^{\Delta \cup den(\mathcal{R})}_{\mathcal{L}_\mathsf{S}}(\Gamma')$*. We define* $\mathsf{Simp}_{\mathcal{L}_\mathsf{R}}{}^U(S)$ *as* $\langle \mathcal{R}^*, \Sigma \rangle$*, where* $\mathcal{R}^*$ *is the largest subset of* $\mathcal{R}$ *including only definitions of predicates on which* Σ *depends.*

The characterization of $\delta_U^- r_p$ given in definition 4.3.3 requires the evaluation of $\neg r_p^U$. However, we shall see that in many interesting cases $\delta_U^- r_p$ is going to be simplified away.

We also note that the new views introduced by $\mathsf{After}_{\mathcal{L}_\mathsf{R}}$ can be completely disregarded if r_p^U does not occur in the simplified constraints. If both the new and the old state are at disposal, as in some trigger implementations, r_p^U can be evaluated as "r_p in the new state". However, these are precisely the cases where the simplification was, to some extent, unsuccessful, as accessing or simulating the new state clearly requires extra work.

4.3.2 Examples

We now present an example that will also be used in section 4.3.3 to compare the present work with previous methods.

Example 4.3.8 *Consider the schema and update of example 3.2.8 on page 33 and the definitions in definition 4.3.3. In order to determine how to check whether the database is consistent after U, we calculate the simplification of S with respect to U. Γ unfolds to $\{\leftarrow p(X,X)\}$, which coincides with Γ itself. The constraint theory of $\mathsf{After}_{\mathcal{L}_\mathsf{R}}{}^U(S)$ is calculated as*

$$\{\leftarrow (p(X,X) \wedge \neg\delta_U^- p(X,X)) \vee \delta_U^+ p(X,X)\} \equiv \{ \begin{array}{l} \leftarrow p(X,X) \wedge \neg\delta_U^- p(X,X), \\ \leftarrow \delta_U^+ p(X,X) \end{array} \}.$$

and unfolded accordingly. When $\mathsf{Optimize}_{\mathcal{L}_\mathsf{S}}$ is applied to $\mathsf{After}_{\mathcal{L}_\mathsf{R}}{}^U(S)$, every unfolding of the first constraint will be removed, as it is subsumed by $\leftarrow p(X,X)$. Furthermore, $\delta_U^+ e(X,Y)$ bounds both X and Y:

$$\delta_U^+ e(X,Y) \equiv \neg e(\mathbf{a},\mathbf{b}) \wedge X = \mathbf{a} \wedge Y = \mathbf{b}.$$

Therefore we can replace $\delta_U^+ p$ as in (4.5)

$$\delta_U^+ p(X,Y) \leftarrow (p(X,\mathbf{a}) \vee X = \mathbf{a}) \wedge (p(\mathbf{b},Y) \vee Y = \mathbf{b}) \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(X,Y),$$

which unfolds in the remaining $\leftarrow \delta_U^+ p(X,X)$ expression as follows:

$$\{ \begin{array}{l} \leftarrow p(X,\mathbf{a}) \wedge p(\mathbf{b},X) \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(X,X), \\ \leftarrow p(\mathbf{b},\mathbf{a}) \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(\mathbf{b},\mathbf{b}), \\ \leftarrow p(\mathbf{b},\mathbf{a}) \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(\mathbf{b},\mathbf{b}), \\ \leftarrow \mathbf{a} = \mathbf{b} \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(\mathbf{a},\mathbf{a}) \end{array} \}.$$

The second and third constraints are identical, and therefore either can be removed. The $\neg p(-,-)$ literals are removed, in $\mathsf{Optimize}_{\mathcal{L}_\mathsf{S}}$, via resolution with the denial $\leftarrow p(X,X)$.

Since p is a transitive closure predicate, the hypotheses used in the optimization step include $\leftarrow \neg p(X,Y) \wedge e(X,Y)$, $\leftarrow \neg p(X,Y) \wedge e(X,Z) \wedge p(Z,Y)$ and $\leftarrow \neg p(X,Y) \wedge p(X,Z) \wedge p(Z,Y)$. Then, all $\neg e(\mathbf{a},\mathbf{b})$ literals can be removed in all denials but the first one. In the last denial, this is done by reduction and resolution via the intermediate $\vdash_R$-derivation of $\leftarrow e(X,X)$ (obtained via $\leftarrow \neg p(X,Y) \wedge e(X,Y)$ and $\leftarrow p(X,X)$). In the second (and third) constraint we use $\leftarrow e(X,Z) \wedge p(Z,X)$ in a similar way (obtained via $\leftarrow \neg p(X,Y) \wedge e(X,Z) \wedge p(Z,Y)$ and $\leftarrow p(X,X)$).

Finally, with a resolution step between $\leftarrow \neg p(X,Y) \wedge p(X,Z) \wedge p(Z,Y)$ and the obtained $\leftarrow p(\mathbf{b},\mathbf{a})$, we get $\leftarrow p(\mathbf{b},Z) \wedge p(Z,\mathbf{a})$, which subsumes the first constraint.

$$\mathsf{Simp}_{\mathcal{L}_\mathsf{R}}{}^U(S) = \{ \begin{array}{l} \leftarrow p(\mathbf{b},\mathbf{a}), \\ \leftarrow \mathbf{a} = \mathbf{b} \end{array} \}.$$

Note that $\mathsf{Simp}_{\mathcal{L}_\mathsf{R}}{}^{U}(S)$ *is a much simpler test than* Γ *as it basically requires to check whether there exists a path between two given nodes, whereas* Γ *implies testing the existence of a cyclic path for all the nodes in the graph.*

Another interesting and more complex example shows how important problems can be reduced to OLR and solved accordingly.

Example 4.3.9 *In [134], the following recursive predicate b is described:*

$$\begin{array}{ll}\{ & b(X,Y) \leftarrow k(X,Z) \wedge b(Z,Y) \wedge c(Y), \\ & b(X,Y) \leftarrow d(X,Y) \qquad\qquad \},\end{array}$$

where b stands for "buys", k for "knows", c for "cheap" and d for "definitely buys". These definitions can be rewritten [104] as[5]*:*

$$\begin{array}{ll}\{ & b'(X,Y) \leftarrow k(X,Z) \wedge b'(Z,Y), \\ & b'(X,Y) \leftarrow k(X,Z) \wedge d(Z,Y) \wedge c(Y), \\ & b(X,Y) \leftarrow b'(X,Y), \\ & b(X,Y) \leftarrow d(X,Y) \qquad\qquad \}.\end{array}$$

Replacing the body of b'*'s exit rule with a new view e, makes* b' *into an OLR predicate:*

$$\begin{array}{lll} IDB = \{ & b'(X,Y) \leftarrow k(X,Z) \wedge b'(Z,Y), & \\ & b'(X,Y) \leftarrow e(X,Y), & \\ & e(X,Y) \leftarrow k(X,Z) \wedge d(Z,Y) \wedge c(Y), & \\ & b(X,Y) \leftarrow b'(X,Y), & \\ & b(X,Y) \leftarrow d(X,Y) & \}.\end{array}$$

Consider now a schema $S = \langle IDB, \Gamma \rangle$ *and a scenario in which a given person p does not want to buy cheap products, expressed by* $\Gamma = \{\leftarrow b(p,X) \wedge c(X)\}$*. Suppose that a person meets another person who is definitely going to buy something. This event can be represented by the update* $U = \{k(\mathbf{a},\mathbf{b}), d(\mathbf{b},\mathbf{c})\}$*. The calculation of* $\mathsf{Simp}_{\mathcal{L}_\mathsf{R}}{}^{U}(S)$ *generates the following set of simplified constraints*[6]*:*

$$\begin{array}{ll}\{ & \leftarrow c(\mathbf{c}) \wedge [p = \mathbf{a} \vee p = \mathbf{b} \vee k'(p,\mathbf{a}) \vee k'(p,\mathbf{b})], \\ & \leftarrow c(X) \wedge [p = \mathbf{a} \vee k'(p,\mathbf{a})] \wedge \{d(\mathbf{b},X) \vee [k'(\mathbf{b},Z) \wedge d(Z,X)]\} \quad \},\end{array}$$

where k' *is the transitive closure of k. The result indicates that U introduces an inconsistency whenever:*

- $\mathbf{c}$ *is cheap, and p is or (in)directly knows* $\mathbf{a}$ *or* $\mathbf{b}$*, or*
- *p is or (in)directly knows* $\mathbf{a}$*, and* $\mathbf{b}$ *definitely buys (or (in)directly knows someone who does) something cheap.*

[5] The example transformation shown in the referenced article is incorrect. The correct version, which is shown here, was obtained after personal communication with the first author of the paper.

[6] For readability, the resulting formula is presented with disjunctions and rearranged via other trivial, equivalence-preserving steps.

4.3.3 Related work

With few exceptions, little attention has been devoted to the problem of checking the integrity of a database containing recursive views, although recursion is now part of the current SQL standard. Most methods have been explicitly designed for relational databases with no views or disallow recursion in integrity constraints; we refer to the survey [130] for further references falling under these categories. We conclude this section with a description of some of the methods that apply to recursive databases; we discuss their behavior when applied to the canonical case of linear recursion shown in example 4.3.8. We use constants a, b instead of parameters $\mathbf{a}$, $\mathbf{b}$ for compatibility with these methods.

The simplification technique for deductive databases described in [122] requires the calculation of two sets, P and N, that represent the positive and, respectively, negative potential updates generated by a given update on the database. A set Θ is then computed, which contains all the most general unifiers of the atoms in P and N with the atoms of corresponding sign in the integrity constraint. For example 4.3.8, this yields $P = \{e(a,b), p(X,Y)\}$, $N = \emptyset$ and $\Theta = \{Y/X\}$. The updated database is consistent iff every condition $\Gamma\theta$ holds in it, for all $\theta \in \Theta$, Γ being the original constraint theory. In this case, the obtained condition is identical to Γ and therefore there is no simplification, unlike the result shown for our method.

In [114], the authors determine low-cost pre-tests which are sufficient conditions that guarantee the integrity of the database. If the pre-tests fail, then integrity needs to be checked with a method, such as ours, that provides a necessary and sufficient condition. A set of literal/condition pairs, called *relevant set*, is calculated. If the update in question unifies with any of the literals in the relevant set and the attached condition succeeds, then the pre-test fails; otherwise we are sure that the update cannot falsify the integrity constraints. For example 4.3.8 the relevant set is as follows (the "N" subscript indicates that the predicate refers to the updated state):

$$\{p(X,X)/true, e(X,X)/true, e(X,Z)/true, p(Z,X)/e_N(X,Z)\}$$

The update $e(a,b)$ unifies with $e(X,Z)$, whose associated condition trivially succeeds, therefore the pre-test fails and an exact test needs to be executed.

In the previous chapter, we discussed a method based on partial evaluation that was described in [116]. In principle, such method *could* work for recursive databases, if a perfect partial evaluator were available. However, as the authors admit, a loop check needs to be included in the program to ensure termination. This does not partially evaluate satisfactorily, resulting in an explosion of (possibly unreachable) alternatives.

In [46], the authors proposed for the first time semi-automatic generation of triggers as a means for constraint maintenance. Their language allows recursively defined constraints as well as a number of other advanced features. However, the checking predicates are not semantically optimized: the generated active rules essentially embed the original constraints. The only semantic enhancement considered is when specific conditions are met that allow replacing a table by the set of new tuples in the update.

With the method described in [48], which is based on the notion of *partial subsumption*, database rules are annotated with *residues* to capture the relevant parts that are concerned by the integrity constraints. When doing semantic query optimization, such

parts can often allow faster query evaluation times. However, when it comes to integrity checking, the method typically leaves things unchanged in the presence of recursive rules. In example 4.3.8 we need to calculate the residue of the constraint in Γ associated with the extensional relation e. In this case the partial subsumption algorithm stops immediately, as no resolution step is possible, thus resulting in no simplification at all. As the evaluation of a recursive query involves the evaluation of non-recursive sub-queries inside a loop, the authors suggest the application of their method to these sub-queries.

In [154], the author proposes an approach based on incremental expressions for OLR. We have improved on his method in at least the following respects. Firstly, our rule-based update language is much more general, allowing compound updates, changes and any kind of bulk operation expressible with rules. Secondly, the simplified constraints produced by $\mathsf{Simp}_{\mathcal{L}_R}$ may only need to consult the present database state, whereas the method of [154] always requires the availability of both the old and the new state, even in the non-recursive case. For the treatment of recursion it imposes a number of restrictions on the language (e.g., no negation) that we do not need. Finally, if non-OLR recursive predicates occur in a constraint, the method fails to apply completely, whereas we can anyhow simplify the OLR and the non-recursive parts of the constraint; this may also produce further specialization of the non-OLR recursive predicates.

Recently, renewed attention arose in the field of update propagation building on previous investigations in the area of view maintenance [99]. Continuing the work of [98], the author of [20] regards integrity checking as an instance of update propagation. In the proposed method, integrity constraints are expressed as propositional predicates that must always be derivable. The database schema is then augmented with new rules that express the incremental evaluation of the new state with respect to a given update. The incrementality of the rules is maximized by rewriting them according to magic sets techniques [14]. After the rewriting, however, an originally stratified database may become non-stratified. A reference model (the so-called well-founded model [165]) for these databases exists, but its computation may be very complex. To this end, a *soft consequence* operator [19] is then introduced to compute the model of this augmented database. Instead of a symbolic simplification of the original constraints, as, e.g., in our method, this method rather provides an efficient way for evaluating the new state of the database. However, it can be used for an efficient evaluation of constraints that have been simplified by some other method; in this respect, it can be seen as orthogonal to our method, at least when $\mathsf{Simp}_{\mathcal{L}_R}$ does not eliminate references to the new state.

Chapter 5

Other Contexts and Applications

In this chapter we discuss the applicability of simplification techniques in three contexts in which integrity control is of interest. In section 5.1, we consider concurrency in the execution of update transactions and show how it may affect simplification. Section 5.2 presents an extension of the simplification procedure shown in the previous chapters that can be used for integrity checking in data integration systems. In section 5.3, the simplification procedure is reformulated in the context of databases consisting of collections of XML documents.

5.1 Simplification and concurrency

When an update transaction is executed on a database, it is important to ensure both that database consistency is preserved and that the transaction produces the desired result, i.e., its execution is not affected by the execution of other, possibly interleaved transactions. A common view in concurrent database systems is that a transaction executes correctly if it belongs to a schedule that is conflict serializable, i.e., equivalent to a schedule in which all the transactions are executed in series (not interleaved). Several strategies and protocols, such as *two-phase locking* (2PL) and *timestamp ordering*, have been established that can dynamically enforce conflict serializability. Locking is the most common practice for concurrency control; however, maintaining locks is expensive and may limit the throughput of the database, as locks actually reduce the concurrency of the accesses to the resources.

The literature is rich in methods aimed at the correctness of concurrently executed transactions, and other protocols than 2PL have been proposed, e.g., [149]. All these methods depend on the implicit assumption that *each single transaction preserves consistency when executed alone* [86]; this responsibility is, thus, entrusted to the designer of the transactions.

We present, instead, an integrated approach to automatically extend update transactions with locks and simplified consistency tests on the locked elements such that all

schedules produced in this way are conflict serializable and preserve consistency in an optimized way. Consequently, the designer only needs to concentrate on the declarative specification of integrity constraints. Furthermore, with our method, we also determine the minimal amount of database resources to be locked in order to guarantee correctness of all legal schedules. Indeed, the execution schedule must ensure that the different transactions can overlap in time without destroying the consistency requirements tested by other, concurrent transactions.

5.1.1 Transactions and serializability

We now briefly introduce basic notion related to concurrency; we refer to standard database books, such as [86], for further details.

The notion of transaction includes, in general, write as well as read operations on database elements. We identify a database element (or *resource*) with a ground atom of the Herbrand base, and a write operation adds or removes such an atom from the database state. A transaction is always concluded with either a commit or an abort; in the former case the executed write operations are finalized into the database state, in the latter they are cancelled. We omit, for simplicity, the indication of abort and commit operations in transactions and consider the execution of a transaction concluded and committed after its last operation.

Definition 5.1.1 (Transaction) *A* transaction T *of length* n *is a finite sequence of operations* $\langle T^1, \ldots, T^n \rangle$ *such that each step* T^i *is either a* $\mathsf{read}(e_i)$ *or a* $\mathsf{write}(e_i)$ *operation on a database element* e_i*.*

A schedule is a sequence of the operations of one or more transactions.

Definition 5.1.2 (Schedule) *A* schedule σ *over a set of transactions* $\mathcal{T}$ *is an ordering of the operations of all the transactions in* $\mathcal{T}$ *which preserves the ordering of the operations of each transaction. A schedule is* serial *if, for any two transactions* T' *and* T'' *in* $\mathcal{T}$*, either all operations in* T' *occur before all operations of* T'' *in* σ *or conversely. A schedule which is not serial, for* $|\mathcal{T}| > 1$*, is an* interleaved *schedule.*

A schedule executes correctly with respect to concurrency of transactions if it corresponds to the execution of some serial schedule, according to definition 5.1.3 below.

Definition 5.1.3 (Conflict serializability) *Two operations in a transaction are* conflicting *iff they refer to the same database element and at least one of them is a* write *operation. Two schedules are* conflict equivalent *iff they contain the same set of committed transactions and operations and every pair of conflicting operations is ordered in the same way in both schedules. A schedule is* conflict serializable *iff it is conflict equivalent to a serial schedule.*

Checking conflict serializability can be done in linear time by testing the acyclicity of the directed graph in which the nodes are the transactions in the schedule and the edges correspond to the order of conflicting operations in two different transactions.

5.1.2 Extended transactions

To explain the application of the framework in concurrent databases, we restrict to view-less schemata and limit updates to sets of tuple additions and deletions, i.e., to updates of the form (3.2) defined on page 45. In other words, the updates consist of write operations on database elements, where the elements are the tuples of the database relations. These operations are known and do not depend on previous read operations. Furthermore, an update does not contain conflicting operations. Therefore we can, for the moment, restrict our attention to write transactions, i.e., sequences of non-conflicting write operations. In order to be able to map back and forth between write transactions and updates, we also indicate for each write if it is an addition or a deletion (using a $\neg$ sign on the database element).

Definition 5.1.4 (Write transaction) *For a given update U, any sequence T, of minimal length, of* write *operations on all the literals in U is a* write transaction *on U. Given a write transaction T, we indicate the corresponding update as $\overline{T}$.*

Example 5.1.5 *The possible write transactions on an update $U = \{p(a), \neg q(a)\}$ are $T_1 = \langle \mathsf{write}(p(a)), \mathsf{write}(\neg q(a))\rangle$ and $T_2 = \langle \mathsf{write}(\neg q(a)), \mathsf{write}(p(a))\rangle$. Conversely, $\overline{T_1} = \overline{T_2} = U$.* □

In the following we shall indicate the i-th element of a sequence S with the notation S^i. For a given write transaction T of length n and a database state D, the notation D^T refers to the database state $(\ldots((D^{\overline{\langle T^1\rangle}})^{\overline{\langle T^2\rangle}})\cdots)^{\overline{\langle T^n\rangle}}$. Similarly, for a schedule σ over write transactions, the notation D^σ refers to the state $(\ldots((D^{\overline{\langle \sigma^1\rangle}})^{\overline{\langle \sigma^2\rangle}})\cdots)^{\overline{\langle \sigma^n\rangle}}$. Clearly, $D^T = D^{\overline{T}}$ for any write transaction T. Then the following proposition trivially follows from the definition of CWP (definition 3.1.2 on page 26). Note that, for a schedule σ, the notation $D^{\overline{\sigma}}$ is not allowed, as σ might contain conflicting operations, i.e., σ cannot always be mapped to an update $\overline{\sigma}$.

Proposition 5.1.6 *Let T be a write transaction, Γ a constraint theory, D a database state consistent with Γ and let Σ be a CWP of Γ with respect to $\overline{T}$. Then $D^T \models \Gamma$ iff $D \models \Sigma$.*

Proposition 5.1.6 indicates that a write transaction executes correctly if and only if the database state satisfies a corresponding CWP. This leads to the following notion of simplified write transaction.

Definition 5.1.7 (Simplified write transaction) *Let T be a write transaction, Γ a constraint theory and D a database state. The* simplified write transaction *of T with respect to Γ and D is*

- *T if $D \models \Sigma$, where Σ is a CWP of Γ with respect to $\overline{T}$.*
- *$\langle\rangle$ if $D \not\models \Sigma$, where $\langle\rangle$ is the empty sequence.*

Obviously, the execution of a simplified write transaction never violates the integrity constraints, as stated in the corollary below, that directly follows from proposition 5.1.6 and definition 5.1.7.

Corollary 5.1.8 *Let T be a write transaction and Γ a constraint theory. Then, for every database state D consistent with Γ, $D^{T_S} \models \Gamma$, where T_S is the simplified write transaction of T with respect to Γ and D.*

In order to determine a simplified write transaction, the database state must be accessed. We can therefore model the behavior of a scheduler that dynamically produces simplified write transactions by starting them with read operations corresponding to the database actions needed to evaluate the CWP. Checking a constraint theory Γ corresponds to reading all database elements that contribute to the evaluation of Γ, i.e., its resource set (see definition 3.4.1 and following text on page 50). The effort of evaluating a constraint theory can then be expressed by concatenating ($\circ$) the sequence of all needed read operations with the (simplified) write transaction. To simplify the notation, we indicate any minimal sequence of read operations on every element of a resource set $\mathcal{R}$ as Read($\mathcal{R}$); in section 5.1.3 we shall use a similar notation for lock (Lock($\mathcal{R}$)) and unlock (Unlock($\mathcal{R}$)) operations.

Definition 5.1.9 (Simplified read-write transaction) *For a constraint theory Γ, a write transaction T and a database state D, any transaction of the form*

$$T' = \text{Read}(\mathcal{R}(\Sigma)) \circ T_S$$

is a simplified read-write transaction *of T with respect to Γ and D, where T_S is the simplified write transaction of T with respect to Γ and D, and Σ is a CWP of Γ with respect to $\overline{T}$. The execution of T' is said to be* legal *if D is the state reached after* Read($\mathcal{R}(\Sigma)$). *The execution of a schedule over simplified read-write transactions is* legal *if the execution of all its transactions is.*

The notion of legal execution of a transaction or schedule relies on the presence of a scheduler that takes care of checking whether the CWP Σ holds in the state at which the write transaction T starts.

For a schedule σ containing read operations, we write D^σ as a shorthand for D^{σ_w}, where σ_w is the sequence that contains all the write operations as in σ and in the same order, but no read operation. Note that a legally executing schedule over simplified read-write transactions is not guaranteed to execute correctly.

Example 5.1.10 *[3.2.32 continued] Consider again $\Gamma = \{\leftarrow m(X,Y) \wedge m(X,Z) \wedge Y \neq Z\}$ disallowing bigamist husbands and the update $U = \{m(\mathbf{a},\mathbf{b})\}$. As shown, $\Sigma = \{\leftarrow \; m(\mathbf{a},y) \wedge y \neq \mathbf{b}\}$ is a CWP of $\{\phi\}$ with respect to U. Let us consider the transactions $T_1 = \langle \mathsf{write}(m(homer, marge))\rangle$ and $T_2 = \langle \mathsf{write}(m(homer, maude))\rangle$. Any schedule σ over T_1, T_2 yields an inconsistent database state, i.e., $D^\sigma \not\models \Gamma$ for any state D, because homer would end up having (at least) two different wives. We may try to remedy this situation by considering schedules over the simplified read-write transactions corresponding to T_1 and T_2 with respect to Γ and the state in which their write transaction is executed. Suitable CWPs of Γ with respect to $\overline{T}_1$, $\overline{T}_2$ are obtained by instantiating the parameters in Σ with the actual constants:*

$$\begin{aligned} \Sigma_1 = &\quad \{\leftarrow m(homer,y) \wedge y \neq marge\} \\ \Sigma_2 = &\quad \{\leftarrow m(homer,y) \wedge y \neq maude\}. \end{aligned}$$

In this way a schedule such as

$$\begin{aligned}\sigma_1 = \quad & \mathrm{Read}(\mathcal{R}(\Sigma_1)) \circ \langle \mathsf{write}(m(homer, marge))\rangle \circ \\ & \mathrm{Read}(\mathcal{R}(\Sigma_2)) \circ \langle \mathsf{write}(m(homer, maude))\rangle\end{aligned}$$

would not be a legally executing schedule over simplified read-write transactions, because in the database state after the last Read, Σ_2 *necessarily does not hold. However it is still possible to create legally executing schedules leading to an inconsistent final state. Consider, e.g., the following schedule:*

$$\begin{aligned}\sigma_2 = \quad & \mathrm{Read}(\mathcal{R}(\Sigma_1)) \circ \mathrm{Read}(\mathcal{R}(\Sigma_2)) \circ \\ & \langle \mathsf{write}(m(homer, marge)), \mathsf{write}(m(homer, maude))\rangle.\end{aligned}$$

At the end of the $\mathrm{Read}(\mathcal{R}(\Sigma_2))$ *sequence,* T_1 *has not yet performed the* write *operation on* m*, so* Σ_2 *may hold, but the final state is inconsistent.*

5.1.3 Locks

The problem pointed out in example 5.1.10 is due to the fact that some of the elements in the resource set of a CWP of a transaction T were modified by another transaction before T finished its execution. In order to prevent this situation we need to introduce locks. A lock is an operation of the form $\mathsf{lock}(e)$ where e is a database element; the lock on e is released with the dual operation $\mathsf{unlock}(e)$. A transaction containing lock and unlock operations is a *locked transaction*. A scheduler for locked transactions verifies that the transactions behave consistently with the locking policy, i.e., database elements are only accessed by transactions that have acquired the lock on them, no two transactions have a lock on the same element at the same time and all acquired locks are released. The 2PL protocol requires that in every transaction all lock operations precede all unlock operations; all 2PL transactions are conflict serializable.

We can now extend the notion of simplified read-write transaction with locks on the set of resources that are read at the beginning of the transaction and on the elements to be written. For an update U of the form (3.2), we write $\mathcal{R}(U)$ to indicate the set of atoms occurring in it.

Definition 5.1.11 (Simplified locked transaction) *For a constraint theory* Γ*, a write transaction* T *and a database state* D*, any transaction of the form*

$$T' = \mathrm{Lock}(\mathcal{R}(\Sigma)) \circ \mathrm{Read}(\mathcal{R}(\Sigma)) \circ \mathrm{Lock}(\mathcal{R}(\overline{T}_S) \setminus \mathcal{R}(\Sigma)) \circ T_S \circ \mathrm{Unlock}(\mathcal{R}(\overline{T}_S) \cup \mathcal{R}(\Sigma))$$

is a simplified locked transaction *of* T *with respect to* Γ *and* D*, where* T_S *is the simplified write transaction of* T *with respect to* Γ *and* D*, and* Σ *is a CWP of* Γ *with respect to* $\overline{T}$*. The execution of* T' *is* legal *iff* D *is the state reached before the execution of* T_S*. The execution of a schedule over simplified locked transactions is* legal *if the execution of all its transactions is.*

Example 5.1.12 *Consider* Γ*,* T_1*,* T_2*,* Σ_1*,* Σ_2 *and* U *from example 5.1.10. Using the policy of simplified locked transactions, there is no schedule that produces an inconsistent state. Consider, e.g., that the initial database state is empty. Suppose that the simplified*

locked transaction of T_1 executes first. Then the resource set of Γ_1 will be locked by it and the simplified locked transaction of T_2 will have to wait until $\mathcal{R}(\Gamma_1)$ is unlocked, i.e., after completion of T_1 (T_1 executes because Γ_1 is satisfied). On the contrary, the simplified transaction of T_2 will not execute T_2, since Γ_2 does not hold once it has the lock. The final state is $\{m(homer, marge)\}$ and is therefore consistent.

Any legally executing schedule over simplified locked transactions is guaranteed to be correct. For such a schedule σ, we write D^σ as a shorthand for D^{σ_w}, where σ_w is the sequence that contains all the write operations as in σ and in the same order, but no read, lock or unlock operation.

Theorem 5.1.13 *Let Γ be a constraint theory and D a database state such that $D \models \Gamma$. Given the write transactions $T_1, \ldots, T_n$, let T_i' be the simplified locked transaction of T_i with respect to Γ and the state D_i reached after the last* Lock *in T_i', for $1 \leq i \leq n$. Then any legally executing schedule σ over $\{T_1', \ldots, T_n'\}$ is conflict serializable and $D^\sigma \models \Gamma$.*

Proof *Conflict serializability follows from the fact that all simplified locked transactions are 2PL. If σ is serial, then the final state is consistent, because, by virtue of corollary 5.1.8, each T_i', when executed alone, maps a consistent state to a consistent state, and the initial state is consistent. Let us assume now that σ is not serial. Consistency of the final state is guaranteed by the level of isolation of any two transactions T_i' and T_j' in σ. Suppose that T_i' starts writing before T_j'. If any of the* writes *in T_i' was in $\mathcal{R}(\Sigma_j)$ or in $\mathcal{R}(\overline{T_j})$, or, vice versa, if any of the* writes *in T_j' was in $\mathcal{R}(\Sigma_i)$ or in $\mathcal{R}(\overline{T_i})$, then the* locks *would force T_j' to wait for completion of T_i'. In other words, if the behavior of T_i' affects the behavior of T_j' (or vice versa), the scheduler executes the* writes *of T_i' and T_j' serially. If they do not affect each other, their* writes *can be interleaved, but their execution does not depend on the other transaction. Therefore, in all cases their execution is the same as that of a serial schedule.* □

The result of theorem 5.1.13 describes a transaction system in which all possible schedules execute correctly. However, the database performance can vary dramatically, depending on the chosen schedule, on the database state and on the waits due to the locks. For this reason, we observe that if the CWPs in use in the simplified locked transactions are minimal according to the resource set criterion, then the amount of locked database resources is also minimized, which increases the database throughput. As we discussed in chapter 3, it is not possible to obtain a minimal simplification *in all cases*. Anyhow, any good approximation of an ideal simplification procedure, i.e., one which might occasionally contain some redundancy, such as $\mathsf{Simp}_{\mathcal{L}_S}$, will be useful as a component of an architecture for transaction management based on these principles.

5.1.4 Discussion

The proposed approach describes a schedule construction policy that guarantees conflict serializability and correctness, which are essential requirements in any concurrent database system. This approach distinguishes itself from previous ones in the following respects. Firstly, it complies with the semantics of deferred integrity checking (i.e., integrity constraints do *not* have to hold in intermediate transaction states), as opposed

to what transaction transformation techniques for view maintenance typically offer. Secondly, it features an early detection of inconsistency — before the execution of the update and without simulating the updated state — that allows one to avoid executions of illegal transactions and subsequent rollbacks. This requires the introduction of locks, whose amount is, however, minimized and, in the case of well designed transactions, often null. Consider, e.g., a referential integrity constraint and an update that inserts both the referencing and the referenced tuple: no lock and no check are needed with our approach, whereas an (immediate) view maintenance approach would require two checks.

The presented approach can be further refined in a number of ways. For example, we have implicitly assumed *exclusive* locks so far; however, a higher degree of concurrency is obtained, without affecting theorem 5.1.13, by using *shared* locks on $\mathcal{R}(\Sigma)$ and *update* locks on the other resources in definition 5.1.11. Besides, to achieve serializability, a so-called *predicate lock* needs to be used on the condition given by the simplified integrity constraints; as is well-known, predicate locks require a high computational effort and, implementation-wise, are approximated as *index locks*. To this end, performance is highly improved by adding an index for every argument position occupied by an update parameter in a database literal in the simplified integrity constraints. Finally, we note that, although the introduced locks may determine deadlocks, all standard deadlock detection and prevention techniques are still applicable. For instance, an arbitrary deadlocked transaction T can be restarted after unlocking its resources and granting them to the other deadlocked transactions.

5.2 Applications to data integration

Data integration has attracted much attention in recent years due to the explosion in online data sources and the whole aspect of globalization of society and business.

To integrate a set of different local data sources means to provide a common database schema, often called a *global* or *mediator schema*, and to describe a relationship between the different local schemata and the global one. Two common paradigms for defining such relationships are the so-called *local-as-view* (LaV, a.k.a. *source-centric*) and *global-as-view* (GaV, a.k.a. *global-centric*); see [164] for definitions and comparison.

The contribution of integrity constraints to data integration is usually confined to query reformulation and query optimization problems. In a GaV approach the global schema is expressed in terms of views over the sources, whereas in LaV the sources are formulated as views over the global schema. The former approach is usually considered simpler for query answering, as this typically amounts to unfolding a query with respect to the view definitions; on the other hand it is less flexible if new sources are to be added to (or removed from) the system. The latter has more involved query answering mechanisms, but enjoys good scalability, as changes at the sources do not require any modification in the global schema.

However the problem of checking and maintaining integrity in this context is also crucial, as even though each local database may satisfy its specific integrity constraints, the combined database may not have a good semantics. Consider, as an example, two databases registering marriages in two different countries. Both may satisfy, locally, an integrity constraint that disallows bigamy, but combining the two databases may violate

this.

As in a traditional database of nontrivial size, it is not practically feasible to check integrity constraints for the entire database in one operation: an incremental approach is needed so that only a small amount of work is required for each update. For the combined case, it is even more urgent to optimize integrity checking because of the presence of possible transition delays over network links.

We generalize the presented simplification technique to combined databases and apply it to two cases: when several sources are integrated and when an updated source notifies the mediator with a message about the update. When integrating a new source, we trust satisfaction of its local constraints. When a message about a local update is received, besides local consistency before the update, we assume that the update has been verified locally. As a natural result, we obtain that only the possible interference between the update and the other sources needs to be checked.

5.2.1 A framework for data integration

In the context of data integration, we have in general several databases (the sources and the mediator) and other operations than database updates need to be considered, such as database combination. We shall therefore generalize the notation in order to describe the relevant cases for data integration. In order to provide a unified view of the data residing at different sources, we need to indicate how the global predicates are expressed in terms of the source predicates. We shall therefore introduce the notion of mapping, which we assume to be *sound*, i.e., the information produced by the views over the sources contains only, but not necessarily all the data associated to the global predicates. *Complete* mappings and *exact* mappings are defined in a similar way (see, e.g., [33]), but we do not consider them here. For convenience, we present our formalization for schemata in $\mathcal{L}_{\mathsf{s}}$, but the methodology is general and does not rely on this assumption.

Definition 5.2.1 (Mapping) *Consider the disjoint schemata $S_0, \ldots, S_n$ in $\mathcal{L}_{\mathsf{s}}$. A* GaV-mapping *$M : (S_1, ..., S_n) \Rightarrow S_0$, is an update in $\mathcal{L}_{\mathsf{s}}$ for a schema $S = S_0 \cup \cdots \cup S_n$ such that the affected predicates are exactly the predicates in $pred(S_0)$ and the predicates in the bodies of the defining formulas of predicate updates are in $\cup_i pred(S_i)$ or are built-ins.*

We can now extend the After operator to handle source combination operations described by a mapping from the sources to the mediator. In particular, for a mapping $M : (S_1, ..., S_n) \Rightarrow S_0$, the notation $\mathsf{After}_{\mathsf{di}}{}^M(S_0)$ refers to $\mathsf{After}^M_{\mathcal{L}_{\mathsf{s}}}(\langle IDB_0 \cup \cdots \cup IDB_n, \Gamma_0 \rangle)$, where $S_i = \langle IDB_i, \Gamma_i \rangle$ for $0 \leq i \leq n$. We use the term *operation* to refer to either an update or the application of a GaV-mapping. For a database combination, $\mathsf{After}_{\mathsf{di}}$ moves from the state after the integration to the non-integrated state, i.e., from a theory concerning the mediator to one concerning the sources.

Examples of the application of these extended versions of the operators are shown in the next sections.

5.2.2 Integrity constraints under global-as-view

In a global-centric approach it is required that the global schema is expressed in terms of the sources. The global predicates must be associated with views over the sources: this

is exactly what the notion of GaV-mapping makes precise, as stated in definition 5.2.1.

We start our analysis by considering a borderline case of GaV-mapping consisting of the combination of two[1] databases having isomorphic schemata. Let S_1, S_2, S be three disjoint schemata that are identical up to consistent renaming of predicates and Γ_1, Γ_2 and Γ the constraint theories defined at the sources and the mediator, respectively. The global database state consists of the union of the local ones and the GaV-mapping M used for the combination is defined as a set containing, for every predicate p in $pred(S)$, the entries:

$$p(\vec{X}) \Leftarrow p_1(\vec{X}) \vee p_2(\vec{X}),$$

where $\vec{X}$ is a sequence of variables and p_1, p_2 are the predicates corresponding to p in S_1, S_2, respectively. A simplified test for checking that the combined database is consistent with Γ (and assuming that Γ holds in an empty database) is then given by $\mathsf{Simp}^{M}_{\Gamma_1 \wedge \Gamma_2}(\Gamma)$, where $\mathsf{Simp}^{U}_{\Delta}(\Sigma)$ is used in the following as syntactic sugar for $\mathsf{Optimize}^{\Sigma \cup \Delta}_{\mathcal{L}_S}(\mathsf{After}_{\mathsf{di}}{}^{U}(\Sigma))$.

Example 5.2.2 *Let us refer to example 3.2.15 on page 36 and consider the schemata S_0, S_1, S_2 with $S_i = \langle \emptyset, \Gamma_i \rangle$ and*

$$\Gamma_i = \{\leftarrow m_i(X,Y) \wedge m_i(X,Z) \wedge Y \neq Z\}$$

for $i \in \{0,1,2\}$. Suppose that S_1, S_2 are the schemata of two source databases and S_0 is the schema of a mediator defined by the GaV-mapping

$$M = \{m_0(X,Y) \leftarrow m_1(X,Y) \vee m_2(X,Y)\}$$

Assuming that the source databases are consistent, their combination at the mediator is consistent with Γ if and only if the condition given by $\mathsf{Simp}^{M}_{\Gamma_1 \wedge \Gamma_2}(\Gamma)$ holds.

$$\begin{array}{rlll} \mathsf{After}_{\mathsf{di}}{}^{M}(\Gamma) \equiv \{ & \leftarrow & (m_1(X,Y) \vee m_2(X,Y)) \wedge & \\ & & (m_1(X,Z) \vee m_2(X,Z)) \wedge Y \neq Z & \} \\ \equiv \{ & \leftarrow & m_1(X,Y) \wedge m_1(X,Z) \wedge Y \neq Z, & \\ & \leftarrow & m_1(X,Y) \wedge m_2(X,Z) \wedge Y \neq Z, & \\ & \leftarrow & m_2(X,Y) \wedge m_1(X,Z) \wedge Y \neq Z, & \\ & \leftarrow & m_2(X,Y) \wedge m_2(X,Z) \wedge Y \neq Z & \} \end{array}$$

The only check that is needed is, as expected, a cross-check between the two databases, as the other denials are removed by Optimize*:*

$$\mathsf{Simp}^{M}_{\Gamma_1 \wedge \Gamma_2}(\Gamma) = \{ \quad \leftarrow m_1(X,Y) \wedge m_2(X,Z) \wedge Y \neq Z \quad \}.$$

We can get even better results when extra knowledge concerning the combination of the sources is available. This simply amounts to adding the extra knowledge to the conditions in the subscript of Simp.

Example 5.2.3 *Consider example 5.2.2, where we now have the knowledge $\Gamma_{1,2}$ that the data concerning the husbands in the two databases are disjoint:*

$$\Gamma_{1,2} = \quad \{\leftarrow m_1(X,Y) \wedge m_2(X,Z)\}.$$

[1]The case with more than two sources is similar.

A much stronger simplification is obtained now, as

$$\mathsf{Simp}^{M}_{\Gamma_1 \wedge \Gamma_2 \wedge \Gamma_{1,2}}(\Gamma) = \emptyset.$$

The cross-check that was found in example 5.2.2 is subsumed by $\Gamma_{1,2}$ *and thus discarded, so no check is needed, as the combined state will anyhow be consistent.*

When the mapping is arbitrary, the method can be applied in a similar way.

Example 5.2.4 *We consider a data integration problem inspired by [117] but here extended with global and local integrity constraints. Suppose we have two sources containing information about movies. We use the variables* I, T, Y *and* R *for movie identifiers, titles, years and reviews respectively. The first source contains movies* $m(I,T,Y)$*, where* I *is key, whereas the second contains reviews* $r(I,R)$*. Furthermore, we know that the identifiers in* r *are a subset of the identifiers in* m*. The mediator assembles this information in a relation* $f(I,T,R)$ *(film), as defined by the following GaV-mapping:*

$$M = \{f(I,T,R) \leftarrow m(I,T,Y) \wedge r(I,R)\}.$$

The following conditions are therefore known to hold on the ensemble of sources:

$$\begin{array}{rll} \Gamma_1 & = \{ & \leftarrow m(I,T_1,Y_1) \wedge m(I,T_2,Y_2) \wedge T_1 \neq T_2, \\ & & \leftarrow m(I,T_1,Y_1) \wedge m(I,T_2,Y_2) \wedge Y_1 \neq Y_2 \quad \}, \\ \Gamma_{1,2} & = \{ & \leftarrow r(I,R) \wedge \neg m(I,T,Y) \qquad\qquad\qquad\quad \}. \end{array}$$

Let Γ *express the fact that* I *is a primary key for* f*:*

$$\begin{array}{rll} \Gamma & = \{ & \leftarrow f(I,T_1,R_1) \wedge f(I,T_2,R_2) \wedge T_1 \neq T_2, \\ & & \leftarrow f(I,T_1,R_1) \wedge f(I,T_2,R_2) \wedge R_1 \neq R_2 \quad \}. \end{array}$$

In order to check whether Γ *holds globally, we calculate* $\mathsf{After}_{\mathsf{di}}{}^{M}(\Gamma)$ *and obtain*

$$\begin{array}{ll} \{ & \leftarrow m(I,T_1,Y_1) \wedge r(I,R_1) \wedge m(I,T_2,Y_2) \wedge r(I,R_2) \wedge T_1 \neq T_2, \\ & \leftarrow m(I,T_1,Y_1) \wedge r(I,R_1) \wedge m(I,T_2,Y_2) \wedge r(I,R_2) \wedge R_1 \neq R_2 \quad \}. \end{array}$$

The first constraint is obviously subsumed by the first constraint in Γ_1 *and the second one can be simplified with* $\Gamma_{1,2}$*, which gives the following:*

$$\mathsf{Simp}^{M}_{\Gamma_1 \wedge \Gamma_{1,2}}(\Gamma) = \{\leftarrow r(I,R_1) \wedge r(I,R_2) \wedge R_1 \neq R_2\}.$$

This evidently corresponds to the requirement that I *must be a key for* r *as well.*

5.2.3 Integrity constraints under local-as-view

A LaV-mapping is usually understood as a set of views of source predicates over global predicates. From now on with the word *LaV-view* we will refer to a formula of the form $A \Rightarrow B$, where A is an atom (the antecedent) referring to a predicate at a source and B a conjunction of atoms (the consequents) referring to predicates at the mediator. A LaV-view $A \Rightarrow B$ is *safe* whenever all the variables in B appear in A as well. A set of

safe LaV-views is called a safe LaV-mapping. Given a safe LaV-mapping, and assuming it is sound as discussed in section 5.2.1, we can always rewrite it as an equivalent GaV-mapping. This is shown in the following examples and can be done by using the fact that, with safeness, $A \Rightarrow A_1 \wedge ... \wedge A_n$ is the same as $A_1 \Leftarrow A$ and ... and $A_n \Leftarrow A$, where $A, A_1, ..., A_n$ are atoms, and perhaps by adding some equalities in the bodies in order to have only distinct variables in the heads. Note that, without assuming safeness, a LaV-mapping cannot always be rewritten as a GaV-mapping in the sense of definition 5.2.1, in that existentially quantified variables might occur. The treatment of such cases may be handled with skolemization, but, for compatibility with the simplification framework, we will only focus on safe LaV-mappings.

Example 5.2.5 *Reconsider the scenario discussed in example 5.2.2. The global database combines now two sources where in the first one the husbands are Italian, and in the second one Danish, which is expressed by the LaV-mapping*

$$\begin{array}{ll} L = \{ & m_1(X,Y) \Rightarrow m(X,Y) \wedge n(X,it), \\ & m_2(X,Y) \Rightarrow m(X,Y) \wedge n(X,dk) \ \}, \end{array}$$

where m and n are global predicates. An equivalent GaV-mapping is as follows:

$$\begin{array}{ll} M_L = \{ & m(X,Y) \leftarrow m_1(X,Y) \vee m_2(X,Y), \\ & n(X,Z) \leftarrow (m_1(X,Y) \wedge Z = it) \vee (m_2(X,Y) \wedge Z = dk) \ \}. \end{array}$$

Consider a global constraint imposing uniqueness of nationality:

$$\phi = \leftarrow n(X,Y) \wedge n(X,Z) \wedge Y \neq Z.$$

Then, given the local assumptions Γ_1 and Γ_2, checking ϕ corresponds to verifying disjointness of m_1 and m_2:

$$\mathsf{Simp}^{M_L}_{\Gamma_1 \wedge \Gamma_2}(\{\phi\}) = \ \leftarrow m_1(X,Y) \wedge m_2(X,Z).$$

Example 5.2.6 *This example is also inspired by [117] and extended with global and local integrity constraints. The global database integrates three different sources that provide information about movies. We use the variables T, Y, D and G for movie titles, years, directors, and genres respectively. The global predicates are $m(T,Y,D,G)$, representing a given movie, and $d(D)$, $i(D)$, $a(D)$, ..., representing nationalities of directors, here Danish, Italian, American, etc. The following integrity constraints are assumed: a key constraint on m (T,Y is key), a domain constraint on film genre, and uniqueness of nationality. Underlines are used as anonymous variables* à la *Prolog for ease of notation.*

$$\begin{array}{ll} \Gamma = \{ & \leftarrow m(T,Y,D_1,\_) \wedge m(T,Y,D_2,\_) \wedge D_1 \neq D_2, \\ & \leftarrow m(T,Y,\_,G_1) \wedge m(T,Y,\_,G_2) \wedge G_1 \neq G_2, \\ & \leftarrow m(\_,\_,\_,G) \wedge G \neq comedy \wedge G \neq drama \wedge \cdots, \\ & \leftarrow d(D) \wedge i(D), \\ & \leftarrow d(D) \wedge a(D), \\ & \cdots \}. \end{array}$$

There are three source databases. The first one contains American comedies given as $m_1(T, Y, D)$ *with* T, Y *as key and a LaV-mapping as follows.*

$$\begin{array}{ll} \Gamma_1 = & \{\leftarrow m_1(T,Y,D_1) \wedge m_1(T,Y,D_2) \wedge D_1 \neq D_2\} \\ L_1 = & \{m_1(T,Y,D) \Rightarrow m(T,Y,D,comedy) \wedge a(D)\}. \end{array}$$

The second source contains Danish movies only, with a key constraint Γ_2 *on* T, Y *as usual and the following LaV-view.*

$$L_2 = \{m_2(T,Y,D,G) \Rightarrow m(T,Y,D,G) \wedge d(D)\}.$$

The third source is a general list of movies with predicates (m_3*,* d_3*,* i_3*,* a_3 *etc.) and integrity constraints* Γ_3 *identical to the global ones modulo consistent renaming of predicates. The LaV-views are specified as follows.*

$$\begin{array}{ll} L_3 = \{ & m_3(T,Y,D,G) \Rightarrow m(T,Y,D,G), \\ & d_3(D) \Rightarrow d(D), \\ & \cdots \} \end{array}$$

Since $L_1 \cup L_2 \cup L_3$ *is safe we can rewrite it as the following GaV-mapping.*[2]

$$\begin{array}{ll} M = \{ & m(T,Y,D,G) \leftarrow m_1(T,Y,D) \wedge G = comedy, \\ & m(T,Y,D,G) \leftarrow m_2(T,Y,D,G), \\ & m(T,Y,D,G) \leftarrow m_3(T,Y,D,G), \\ & a(D) \leftarrow m_1(\_,\_,D), \\ & d(D) \leftarrow m_2(\_,\_,D,\_), \\ & a(D) \leftarrow a_3(D), \\ & d(D) \leftarrow d_3(D), \\ & i(D) \leftarrow i_3(D), \\ & \cdots \} \end{array}$$

The simplified integrity constraints for the integration of the three databases, given as $\Sigma = \mathsf{Simp}^M_{\Gamma_1 \wedge \Gamma_2 \wedge \Gamma_3}(\Gamma)$*, are, as expected, simplified rules covering possible conflicts in cross combinations only, as local consistency is assumed.*

$$\begin{array}{ll} \Sigma = \{ & \leftarrow m_1(\_,\_,D) \wedge m_2(\_,\_,D,\_), \\ & \leftarrow m_1(\_,\_,D) \wedge \mathbf{p}_3(D), \quad (\mathbf{p}\ \textit{any nationality pred. different from a)} \\ & \leftarrow m_2(\_,\_,D,\_) \wedge \mathbf{p}_3(D), \quad (\mathbf{p}\ \textit{any nationality pred. different from d)} \\ & \leftarrow m_1(T,Y,\_) \wedge m_2(T,Y,\_,\_), \\ & \leftarrow m_1(T,Y,D_1) \wedge m_3(T,Y,D_2,\_) \wedge D_1 \neq D_2, \\ & \leftarrow m_2(T,Y,D_1,\_) \wedge m_3(T,Y,D_2,\_) \wedge D_1 \neq D_2, \\ & \leftarrow m_1(T,Y,\_) \wedge m_3(T,Y,\_,G) \wedge G \neq comedy, \\ & \leftarrow m_2(T,Y,\_,G_1) \wedge m_3(T,Y,\_,G_2) \wedge G_1 \neq G_2, \\ & \leftarrow m_2(\_,\_,\_,G) \wedge G \neq comedy \wedge G \neq drama \wedge \cdots \}. \end{array}$$

The example can be modified in several ways, e.g., by assuming that the third source is an unchecked database to which enthusiastic amateurs can add arbitrary information. In that case, the simplified constraints would also include a copy of the full set of global constraints with predicate names m_3*,* a_3*, etc.*

[2] If, say, in m_1 the genre was left unspecified, skolemization would be needed.

5.2.4 Absorption of local updates

A data integration system needs to be able to adjust itself dynamically as sources are updated over time. We assume that source databases maintain their own consistency and that reports are available at the global level about which updates have been performed; this may be supplied, e.g., by a process monitoring the sources. Consistency needs then to be checked globally. This problem, that we refer to as *absorption* of local updates, can be handled as follows.

Assume a GaV-mapping M and a set of conditions Δ that hold at the sources and a constraint theory Γ that is maintained at the mediator; this means that the condition $\Sigma = \mathsf{Simp}^M_\Delta(\Gamma)$ is known to hold. Suppose that a source database can receive an update U of the form (3.2), as defined on page 45. Since U may affect Σ and maintenance of Σ can be performed only after execution of U, we need a post-test of Σ with respect to U. For such updates, we conjectured that $\mathsf{Simp}_{\mathcal{L}_\mathsf{S}}$ can also be used as a post-test. Therefore, if the claim of conjecture 3.2.41 (on page 45) holds (which it does in all cases presented in this section), Γ is then still satisfied at the mediator if and only if $\Phi = \mathsf{Simp}^U_{\mathcal{L}_\mathsf{S}}(\Sigma)$ holds.

If the claim does not hold, one can always derive a post-test of Σ wrt. U from $\mathsf{Simp}^U_{\mathcal{L}_\mathsf{S}}(\Sigma)$ in the following way. Let us indicate as $\neg U$ the set that contains the same atoms as U but with the negation signs interchanged. We say that a database D is *revertible* with respect to U if $D = (D^U)^{\neg U}$, i.e., if D does not already contain any addition in U and D contains all deletions in U. We can assume here that an update U is only executed on a database that is revertible wrt. U, since the source will actually know which single updates were performed and which ones were not. Then a post-test of Σ wrt. U is given by

$$\Psi = \mathsf{Optimize}^{\emptyset}_{\mathcal{L}_\mathsf{S}}(\mathsf{After}^{\neg U}_{\mathcal{L}_\mathsf{S}}(\Phi)).$$

To see this, consider that $D \models \Psi$ iff $D^{\neg U} \models \Phi$ for any D, and, thus, $D^U \models \Psi$ iff $(D^U)^{\neg U} \models \Phi$ for any D. Therefore, $D^U \models \Psi$ iff $D \models \Phi$ for any D revertible wrt. U. Besides, since Φ is a CWP of Σ wrt. U, $D \models \Phi$ iff $D^U \models \Sigma$ for any D consistent with Σ. Finally, by transitivity, $D^U \models \Psi$ iff $D^U \models \Sigma$ for any D consistent with Σ and revertible wrt. U.

Example 5.2.7 *In example 5.2.2, the following integrity constraint was generated for the integration of two sources referred to by predicates m_1 and m_2:*

$$\Sigma = \leftarrow m_1(X, Y) \wedge m_2(X, Z) \wedge Y \neq Z.$$

If the update $U = \{m_1(a, b)\}$ is reported, the optimal way to check the global consistency is to test $\mathsf{Simp}^U_{\mathcal{L}_\mathsf{S}}(\Sigma) = \leftarrow m_2(a, Z) \wedge b \neq Z$ at the updated sources.

Example 5.2.8 *Consider Σ from the LaV integrated movie database of example 5.2.6 and assume that the second source reports the addition of a new movie*

$$U = \{m_2(\mathit{dogville}, 2003, \mathit{vonTrier}, \mathit{drama})\}.$$

The following tests, calculated as $\mathsf{Simp}^U_{\mathcal{L}_\mathsf{S}}(\Sigma)$, remain:

$$\begin{array}{ll} \{ & \leftarrow m_1(\_, \_, \mathit{vonTrier}), \\ & \leftarrow \mathbf{p}_3(\mathit{vonTrier}), \quad (\mathbf{p} \textit{ any nationality pred. different from } d) \\ & \leftarrow m_1(\mathit{dogville}, 2003, \_), \\ & \leftarrow m_3(\mathit{dogville}, 2003, d, \_) \wedge d \neq \mathit{vonTrier}, \\ & \leftarrow m_3(\mathit{dogville}, 2003, \_, g) \wedge g \neq \mathit{drama}\}. \end{array}$$

5.2.5 Related work

In [95] the problem of answering queries using views (*query folding*) under LaV is addressed with a technique based on resolution, and several cases, including integrity constraints, negation and recursion, are dealt with.

A short survey on the role of integrity constraints in data integration is given in [35]. They are regarded as means to extract more information from incomplete sources as well as components that raise the issue of dealing with possibly inconsistent global databases. Several GaV typologies are studied that include the treatment of key and foreign key constraints and both sound and exact mappings. In [33] the same authors develop these ideas to show that, in the presence of integrity constraints, query answering in GaV becomes as difficult as in LaV, as the problem of incomplete information implicitly arises. Further discussion on the expressive power of the two approaches, in terms of query-preserving transformations, is given in [34].

In [164], Ullman relates the problem of constructing answers to queries using views to query containment algorithms and compares two implemented systems in these terms.

Levy [117] applies techniques from artificial intelligence to the problem of data integration and shows examples of both GaV and LaV query reformulation with integrity constraints, including particular data access patterns.

Li [119] considers the use of integrity constraints in LaV query processing and optimization and distinguishes between local and global constraints and, for these, between general global constraints and source-derived global constraints.

Others, e.g., [16, 17, 120], have approached the global consistency problem by introducing *disjunctive databases*. In addition to data-related inconsistencies, the authors of [120] also consider the problem of merging sources whose compatibility is affected by the presence of synonyms, homonyms or type conflicts in the schemata.

Another direction of research concerns automatic identification of a common global schema with mappings from different source schemata. Various techniques, such as linguistic and ontological similarity between relation and attribute names and type structures, can be used; see [144] for an overview.

Paraconsistent logics can be used to model the possible inconsistencies coming from database integration and update. In [68] it is shown how inconsistent information can be stored and managed in this way.

5.3 Simplified integrity checking for XML

Consistency requirements for XML data are as important as those holding for relational data, and constraint definition and enforcement are expected to become fundamental aspects of XML data management. However, expressing, verifying and automatically enforcing integrity in the XML context requires additional effort with respect to the relational setting.

A first, evident reason is that there is no standard means of specifying generic constraints over XML document collections. In current XML specifications, fixed-format structural integrity constraints can already be defined by using XML Schema definitions; they are concerned with type definitions, occurrence cardinalities, unique constraints, and

referential integrity. However, a generic constraint definition language for XML, with expressive power comparable to assertions and checks of SQL, is still not present in the XML Schema specification [167]. We deem this a crucial issue, as this lack of expressiveness does not allow one to specify business rules to be directly included in the schema.

Secondly, new difficulties are inherent in XML's hierarchical data model. The infeasibility of a brute force approach to integrity checking, i.e., verifying the whole database each time data are updated, is even more evident when the underlying data management technology is as young and unripe as in the case of XML.

A suitable formalism for the declarative specification of integrity constraints over XML data is therefore required in order to apply simplification techniques. More specifically, we adopt for this purpose a formalism called XPathLog, a logical language inspired to DATALOG and XPath and introduced in [129]. In our approach, the tree structure of XPathLog constraints is mapped to a relational representation. Then, simplification takes place as usual, and, finally, the simplified constraints are translated into XQuery expressions that can be matched against the XML document so as to check that the update does not introduce any constraint violation.

In principle it would be possible to apply simplification techniques directly to XPathLog constraints, by appropriately extending the Simp operator. However, XML data is most often represented and stored with an underlying relational format. In such cases, the relational representation is already available, and the updates are eventually reverted into modifications of relational tuples. Still, as native XML repositories become more popular and reliable, it is desirable to express simplified constraints in XQuery. To this end, we have introduced the above mentioned final translation step from simplified DATALOG constraints to XQuery.

An experimental evaluation of the method is going to be provided in the next chapter, in section 6.3.

We assume, throughout this section, a basic knowledge of standard XML notions, such as element and path, and the corresponding syntax [2].

5.3.1 General constraints over semi-structured data

In [129], the XPath language is extended with variable bindings and embedded into first-order logic to form XPath-Logic; XPathLog is then the Horn fragment of XPath-Logic. Thanks to its logic-based nature, XPathLog is well-suited for querying XML data and providing declarative specifications of integrity constraints. It uses an *edge-labeled graph* model in which subelements are ordered and attributes are unordered. Path expressions have the form root/axisStep/.../axisStep, where root specifies the starting point of the expressions (such as the root of a document or a variable bound to a node) and every axisStep has the form axis::nodetest[qualifier]*. An axis defines a navigation direction in the XML tree: child, attribute, parent, ancestor, descendant, preceding-sibling and following-sibling. All elements satisfying nodetest along the chosen axis are selected, then the qualifier(s) are applied to the selection to further filter it. Axes are abbreviated as usual, e.g., path/nodetest stands for path/child::nodetest and path/@nodetest for path/attribute::nodetest.

XPath-Logic formulas are built as follows. An infinite set of variables is assumed along with a signature of element names, attribute names, function names, constant symbols

and predicate names. A *reference expression* is a path expression that may be extended to bind selected nodes to variables with the construct "$\rightarrow Var$". XPath-Logic predicates are predicates over reference expressions and atoms and literals are defined as usual. Formulas are thus obtained by combining atoms with connectives ($\wedge$, $\neg$, and sometimes $\vee$ as syntactic sugar) and variables are assumed to be universally quantified. Clauses are written in the form $Head \leftsquigarrow Body$ where the head, if present, is an atom and the body a conjunction of literals. In particular, a *denial* is a headless clause; integrity constraints will be written as denials, which indicates that there must be no variable binding satisfying the condition in the denial body for the data to be consistent.

Example 5.3.1 *Consider the following two documents:* `pub.xml` *containing a collection of published articles and* `rev.xml` *containing information on reviewer/paper assignment for all tracks of a given conference. We expect the former to be a huge collection of articles, such as DBLP [118], whereas the latter is a smaller document maintained locally by a conference administrator. The DTDs are as follows.*

```
<!ELEMENT dblp (pub)*>              <!-- pub.xml -->
<!ELEMENT pub  (title,(aut)+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT aut (name)>
<!ELEMENT name (#PCDATA)>

<!ELEMENT review (track)+>          <!-- rev.xml -->
<!ELEMENT track (name,(rev)+)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT rev (name,(sub)+)>
<!ELEMENT sub (title,(aut_s)+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT aut_s (name)>
```

Consider the following integrity constraint, which imposes the absence of conflict of interests *in the submission review process (i.e., no one can review papers written by a former coauthor or by him/herself):*

$$\begin{aligned} \leftsquigarrow\ & //rev[name/text()\rightarrow R]/sub/aut_s/name/text()\rightarrow A \\ & \wedge(A = R \vee //pub[aut/name/text() \rightarrow A \wedge \\ & \qquad\qquad aut/name/text() \rightarrow R]) \end{aligned}$$

The `text()` *function refers to the text content of the enclosing element. The condition in the body of this constraint indicates that there is a reviewer named R who is assigned a submission whose author has name A and, in turn, either A and R are the same or two authors of a same publication have names A and R, respectively*[3]*. Integrity is kept when this condition does not hold.*

This example shows that typical business rules can be expressed naturally with XPathLog. It would not have been possible to express the same constraint with currently available specification paradigms, such as XML Schema or DTDs; the same task would have been much more difficult using any procedural approach.

[3]Note that, for simplicity, we have assumed here that a reviewer's `name` is unique; with a more realistic design, we could have added, say, a SSN to each reviewer and expressed the constraint on the SSN instead of the name. Also note that the disjunct $A = R$ is not redundant w.r.t. the other disjunct (in which $A \neq R$ is not stated), because it catches the (yet unlikely) case of a reviewer without any publication.

5.3.2 Mapping to the relational data model

In order to apply our simplification framework to XML constraints, the update patterns, and the constraints themselves need to be mapped from the XML domain to the relational model.

Mapping of the schema

There exist several approaches to the problem of representing semi-structured data in relations [157, 25, 74, 83]. A survey can be found in [112]. Our approach is targeted to deductive databases: each node type, as defined in the DTD, is mapped to a corresponding predicate; in some cases further optimization is possible. The shape of the predicates is fixed for the first three positions, while the remaining positions depend on the schema. More precisely, the first three attributes of all predicates respectively represent, for each XML item:

- Its (unique) node identifier.
- Its position among the children of its parent node.
- The node identifier of its parent node.

It is worth noting that the second attribute is crucial, as the XML data model is ordered, while the relational data model is unordered. Therefore, the positions within the document must be explicitly represented for each XML item in order to preserve structural constraints.

Whenever a parent-child relationship within a DTD is a one-to-one correspondence (or an optional inclusion), a more compact form is possible, because a new predicate for the child node is not necessary: the attributes of the child may be equivalently represented within the predicate that corresponds to the parent (possibly allowing null values in case of optional child nodes).

According to these mapping criteria, the two documents of example 5.3.1 map to the following relational schema,

$$\begin{array}{l} pub(Id, Pos, IdParent_{dblp}, Title) \\ aut(Id, Pos, IdParent_{pub}, Name) \\ \\ track(Id, Pos, IdParent_{review}, Name) \\ rev(Id, Pos, IdParent_{track}, Name) \\ sub(Id, Pos, IdParent_{rev}, Title) \\ aut_s(Id, Pos, IdParent_{sub}, Name) \end{array} \tag{5.1}$$

where Id, Pos and $IdParent_{tagname}$ preserve the hierarchy of the documents and where the PCDATA content of the `name` and `title` node types is systematically embedded into the container nodes, so as to reduce the number of predicates.

As already mentioned, mapping a hierarchical ordered structure to a flat unordered data model forces the exposition of information that is typically hidden within XML

repositories, such as the order of the sub-nodes of a given node and unique node identifiers[4]. Such identifiers and order information are necessary in order to reconstruct the hierarchy by means of join conditions over the parent-child containment relationship.

The root nodes of the documents (`dblp` and `review`) are not represented as predicates, as they have no local attributes but only subelements; however, such nodes are referenced in the database as values for the IdParent_{dblp} and IdParent_{review} attributes respectively, within the representation of their child nodes. Publications map to the *pub* predicate, authors in `pub.xml` map to *aut*, while authors in `rev.xml` map to aut_s, and so on, with predicates corresponding to tagnames. Last, `name`s and `title`s map to attributes within the predicates corresponding to their containers.

This representation is convenient for our simplification purposes, in that it embeds containment relationships in different predicates depending on the element types. If we had chosen to adopt a more weakly structured representation using, say, a relation $cont(Parent, Child)$ to indicate parent-child containment independently of the element type, a potentially high number of instances of the *cont* relation would be required to represent non-trivial XML constraint and updates. This would result in explosive behavior and intractability of the simplification process.

Mapping of update statements

In order to show the mapping criteria for XML updates, we refer to the XUpdate language [113], but any other formalism that allows the specification of insertions of data fragments would apply. Consider the following XUpdate statement:

```
<xupdate:modifications version="1.0" xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/review/track[2]/rev[5]/sub[6]" >
    <xupdate:element name="sub">
      <title> Taming Web Services </title>
      <aut_s> <name> Jack </name> </aut_s>
    </xupdate:element>
  </xupdate:insert-after>
</xupdate:modifications>
```

In the relational model, this update statement corresponds to adding to the database the following set of facts:

$\{\ sub(id_s, 7, id_r, \text{“Taming Web Services”}),\ \ aut_s(id_a, 2, id_s, \text{“Jack”})\ \}$

where id_a and id_s represent the identifiers that are to be associated to the new nodes and id_r is the identifier associated to the target `rev` element. Their value is immaterial to the semantics of the update, provided that a mechanism to impose their uniqueness is available. In other words, it is guaranteed that the insertion of a new node never overwrites an existing node, as a new reserved identifier is supposed to be always available, and uniqueness of identifiers is guaranteed and maintained by construction.

[4]Identifiers are unique and characterize the nodes from a structural viewpoint, but they are not logical keys for the nodes they are attached to. Moreover, such identifiers are typically opaque to the XPath/XQuery programmer, and we will refer to them as immaterial values without further investigation, as we can assume that they are implicitly available whenever a document collection is stored into an XML repository. As an example, consider that the *same* reviewer can occur more than once in the document for reviewing different papers; such multiple entries occur of course with the same name (which is the logical key in the running example) but with different node identifiers.

On the other hand, the actual value of id_r depends on the dataset and needs to be retrieved by interpreting the `select` clause of the XUpdate statement. Namely, id_r is the identifier for the fifth (`reviewer`) child of the second (`track`) node, in turn contained into the root (`review`) node of the document `rev.xml`. Positions (7 and 2 in the second argument of both predicates) are also derived by parsing the update statement: 7 is determined as the successor of 6, according to the `insert-after` semantics of the update; 2 is due to the ordering, since the `aut_s` element comes after the `title` element.

For simplicity we have shown an example in which node identifiers were easily derived from the candidate location of the inserted fragment. Should the location specification be more complex, this would result in a heavier pre-processing phase so as to identify the positions at which the update is to occur. Generality of the approach, however, would not be affected.

Finally, note that the *same* value id_s occurs both as the first argument of $sub()$ and the third argument of $aut_s()$, since the latter represents a subelement of the former.

Mapping of integrity constraints

The last step in the mapping from XML to the framework of deductive databases is to compile XPathLog denials into corresponding DATALOG denials. Input to this phase are the two schemata (the XML and its relational counterpart, generated according to the principles stated above) and an XPathLog denial.

All path expressions in an XPathLog statement generate a chain of conditions over the predicates corresponding to the node types traversed by the path expression. The containment in terms of parent-child relationship translates to correspondences between variables in the first position of the container and in the third position of the contained item. Quite straightforwardly, the XPathLog denial

$$\leftsquigarrow //pub[title = \text{“Duckburg tales”}]/aut/name \rightarrow N \wedge\ N = \text{“Goofy”}$$

which expresses the fact that the author of the “Duckburg tales” cannot be Goofy, maps into

$$\leftarrow pub(I_p, \_, \_, \text{“Duckburg tales”}) \wedge aut(\_, \_, I_p, N) \wedge\ N = \text{“Goofy”}.$$

In general, all variables used in the XPathLog denial may also occur in the DATALOG denial. In the above example, however, variable N is not strictly necessary to formulate the constraint (it would disappear, e.g., after a reduction step).

Longer path expressions would map to longer chains of predicates. Difficulties may arise if the path expressions contain wildcards or the ‘//’ step. In both cases the predicate chain may not be uniquely determined; a schema like (5.1) allows the inference of the possible steps in the chain; alternative chains map to disjunctions within denials (i.e., different denials). If the schema allowed circular references between node types, the mapping would then require the introduction of recursive views that express the transitive closure of the containment relationship between all possible combinations of compatible node types. We focus here on acyclic schemata only.

All other XPathLog predicates are transcribed in the DATALOG translation, as we assume that the set of built-in predicates is the same.

References to the `position()` function or position filters in the XPathLog denial are handled by associating a variable to the second argument in the relational predicate and then matching it against the corresponding expression.

Example 5.3.2 *The XPathLog constraint of example 5.3.1, here repeated for the reader's convenience,*

$$\begin{aligned} \leftsquigarrow \quad & //rev[name/text() \rightarrow R]/sub/aut_s/name/text() \rightarrow A \\ & \wedge (A = R \vee //pub[aut/name/text() \rightarrow A \wedge \\ & \qquad\qquad aut/name/text() \rightarrow R]) \end{aligned}$$

is translated into the following set of DATALOG *denials.*

$$\begin{aligned} \Gamma = \{\leftarrow \quad & rev(I_r, \_, \_, R) \wedge sub(I_s, \_, I_r, \_) \wedge aut_s(\_, \_, I_s, R), \\ \leftarrow \quad & rev(I_r, \_, \_, R) \wedge sub(I_s, \_, I_r, \_) \wedge aut_s(\_, \_, I_s, A) \\ & \wedge aut(\_, \_, I_p, R) \wedge aut(\_, \_, I_p, A)\} \end{aligned}$$

5.3.3 Simplification of XML constraints

The mappings described in the previous section allow us to express XML schemata, updates and constraints in the relational model. Therefore, we can apply the simplification procedure shown in the previous chapters to simplify the relational counterparts of constraints and updates.

Example 5.3.3 *Let us consider constraint* Γ *from example 5.3.2. An update of interest is, e.g., the insertion of a new submission to the attention of a reviewer, as considered in section 5.3.2. For instance, a submission with a single author complies with the pattern* $U = \{sub(\mathbf{i}_s, \mathbf{p}_s, \mathbf{i}_r, \mathbf{t}), aut_s(\mathbf{i}_a, \mathbf{p}_a, \mathbf{i}_s, \mathbf{n})\}$*, where the parameter referring to the submission id (*$\mathbf{i}_s$*) is the same in both added tuples. The fact that* $\mathbf{i}_s$ *and* $\mathbf{i}_a$ *are fresh new node identifiers can be expressed as a set of extra hypotheses to be exploited in the constraint simplification process:*

$$\Delta = \{\leftarrow sub(\mathbf{i}_s, \_, \_, \_), \leftarrow aut_s(\_, \_, \mathbf{i}_s, \_), \leftarrow aut_s(\mathbf{i}_a, \_, \_, \_)\}.$$

Uniqueness of each such parameter can be imposed both as an element identifier and as a parent identifier, for all element types. Here, Δ *only includes such uniqueness conditions for the relevant element types: as an identifier for all elements of the same type, and as a parent identifier for all children of that element (note that* aut_s *has no children in the relational model).*

The simplified integrity check with respect to update U *and constraint theory* Γ *is given by* $\mathsf{Simp}^U_\Delta(\Gamma)$*:*

$$\begin{aligned} \{\leftarrow \quad & rev(\mathbf{i}_r, \_, \_, \mathbf{n}), \\ \leftarrow \quad & rev(\mathbf{i}_r, \_, \_, R) \wedge aut(\_, \_, I_p, \mathbf{n}) \wedge aut(\_, \_, I_p, R)\}. \end{aligned}$$

The first denial requires that the added author of the submission ($\mathbf{n}$*) is not the same person as the assigned reviewer (*$\mathbf{i}_r$*). The second denial imposes that the assigned reviewer is not a coauthor of the added author* $\mathbf{n}$*. These conditions are clearly much cheaper to evaluate*

than the original constraints Γ*, as they are instantiated to specific values and involve fewer relations. Note that, without the first hypothesis in* Δ*, one would also need to check that the reviewer is not (already) an author of submission* $\mathbf{i}_s$*, nor a coauthor of an (existing) author of* $\mathbf{i}_s$*. Without the second hypothesis in* Δ*, one should also check that* $\mathbf{n}$ *is not a reviewer to which* $\mathbf{i}_s$ *has already been assigned nor the coauthor of such a reviewer. The last hypothesis in* Δ *is not used in this case.*

5.3.4 Translation into XQuery

The obtained simplified constraints must be checked before the corresponding update statement, so as to prevent the execution of statements that would violate integrity. Under the hypothesis that the dataset is stored into an XML repository capable of executing XQuery statements, the simplified constraints need to be translated into suitable equivalent XQuery expressions in order to be checked. This section discusses the translation of DATALOG denials into XQuery.

We exemplify the translation process using the (non-simplified) set of constraints Γ defined in example 5.3.2. For brevity, we only show the translation of the second denial.

We first expand the atoms of the DATALOG denial, except for the first and the third positions, which refer to element and parent identifiers and thus keep information on the parent-child relationships of the XML nodes. Here the expansion is:

$$\begin{aligned} \leftarrow \quad & rev(I_r, B, C, R) \wedge sub(I_s, D, I_r, E) \wedge aut_s(F, G, I_s, A) \\ & \wedge aut(H, I, I_p, J) \wedge aut(K, L, I_p, M) \wedge J = R \wedge M = A \end{aligned}$$

The atoms in the denial must be sorted so that, if a variable referring to the parent of a node also occurs as the identifier of another node, then the occurrence as an identifier comes first. Here, no such rearrangement is needed. Then, for each atom $p(Id, Pos, Par, D_1, \ldots, D_n)$ where D_1, ..., D_n are the values of tags `d1`, ..., `dn`, respectively, we do as follows. If the definition of `$Par` has not yet been created, then we generate the following XQuery variable definitions

```
$Id in //p,    $Par in $Id/..
```

Otherwise we just generate

```
$Id in $Par/p
```

This is followed by

```
$Pos in $Id/position(),   $D1 in $Id/d1/text(),   ...,   $Dn in $Id/dn/text()
```

After the generation of all variable definitions, we build an XQuery boolean expression (returning `true` in case of violation) by prefixing the definitions with the `some` keyword and by suffixing them with the `satisfies` keyword followed by all the remaining conditions in the denial separated by `and`s. This is a well-formed XQuery expression. Here we have:

```
some $Ir in //rev,  $C  in $Ir/..,    $B in $Ir/position(),  $R in $Ir/name/text(),
                    $Is in $Ir/sub,   $D in $Is/position(),  $E in $Is/title/text(),
                    $F  in $Is/auts,  $G in $F/position(),   $A in $F/name/text(),
     $H  in //aut,  $Ip in $H/..,     $I in $H/position(),   $J in $H/name/text(),
                    $K  in $Ip/aut,   $L in $K/position(),   $M in $K/name/text()
satisfies $J=$R and $M=$A
```

Such expression can be optimized by eliminating definitions of variables which are never used, unless they refer to node identifiers. Such variables are to be retained because they express an existential condition on the element they are bound to. Variables referring to the position of an element are to be retained only if used in other parts of the denial. In the example, we can therefore eliminate the definitions of variables `$B`, `$C`, `$D`, `$E`, `$G`, `$I`, `$L`. If a variable is used only once outside its definition, its occurrence is replaced with its definition. Here, e.g., the definition of `$Is` is removed and `$Is` is replaced by `$Ir/sub` in the definition of `$F`, obtaining `$F in $Ir/sub/auts`.

Variables occurring in the `satisfies` part are replaced by their definition. Here we obtain the following query.

```
some      $Ir in //rev, $H in //aut
satisfies $H/name/text()=$Ir/name/text() and
          $H/../aut/name/text()=$Ir/sub/auts/name/text()
```

The translation of simplified version $\mathsf{Simp}^U_\Delta(\Gamma)$ is made along the same lines. Again, we only consider the simplified version of the second constraint (the denial $\leftarrow\ rev(\mathbf{i}_r, \_, \_, R) \wedge aut(\_, \_, I_p, \mathbf{n}) \wedge aut(\_, \_, I_p, R)$). Now, parameters can occur in the simplified denial. If a parameter occurs in the first or third position of an atom, it must be replaced by a suitable representation of the element it refers to. Here we obtain:

```
some      $D in //aut
satisfies $D/name/text()=%n and
          $D/../aut/name/text() = /review/track[%i]/rev[%j]/name/text()
```

where `/review/track[%i]/rev[%j]` conveniently represents $\mathbf{i}_r$, as was done for the update statement of section 5.3.2. Similarly, `%n` corresponds to $\mathbf{n}$. The placeholders `%i`, `%j` and `%n` will be known at update time and replaced in the query.

5.3.5 Related work

An attempt to adapt view maintenance techniques to the semi-structured data model has been made in [159] and, recently, in [150], which however only considers additions and deletions of leaf nodes and only allows predicate tests that refer to nodes in the subtree of the node being tested. Several recent papers address the problem of incrementally validating XML documents with respect to DTD or XML Schema definitions. In particular, [141] provides an algorithm for validation upon insertions and deletions of leaf nodes, whereas [18] also considers insertions of subtrees. The complexity of validating an XML document with respect to a DTD, including structural as well as ID/IDREF constraints, has been shown to be low [18]: in $\mathcal{O}(n \log n)$ time and linear space. Incremental validation with respect to document updates has been studied for specific classes of DTDs (*1,2-CF DTDs*) and shown [18] to be substantially simpler than complete revalidation, with a logarithmic time complexity. Validation of schema, key and foreign key constraints is also considered in [4], where the validator is represented by a bottom-up tree transducer; as in our approach, an update is performed only if it is first accepted by the validator. An attempt to simplification of general integrity constraints for XML has been made in [21], where, however, constraints are specified in a procedural fashion with an extension of XML Schema that includes loops with embedded `forbidden` assertions.

We are not aware of other works that address the validation problem with respect to general constraints for XML. However, integrity constraint simplification can be reduced to query containment as long as the constraints can be viewed as queries. To this end, relevant works on containment for various fragments of XPath that would foster a direct simplification approach for XML are [151, 136].

There are several proposals and studies of constraint specification languages for XML by now. In [79], a unified constraint model (UCM) for XML is proposed that captures in a single framework the main features of object-oriented schemata and XML DTDs. Special attention has been paid to key constraints for XML in [31], where the notion of *relative* key (i.e., key relative to a fragment) is introduced and the implication problem is shown to be decidable in polynomial time; further complexity and axiomatization results are given in [81] and [32]. Satisfiability of DTD specifications has been studied in [80] and shown to be NP-complete.

The XUpdate language that was used for the experimental evaluation is described in [113]. A discussion on update languages for XML is found in [161]; an XQuery-based implemented prototype of XML update language is described in [160], whereas a rule-based framework is presented in [129]. The output of the simplification process described in this paper is a query expressed in XQuery, a query language based on XPath. The complexity of XPath query evaluation has been studied in [91, 92].

Chapter 6

Experimental Evaluation

In order to demonstrate the effectiveness of the simplification procedure, we have tested it on more complex examples and show here our experimental results, both for the recursive and the non-recursive case, in sections 6.1 and 6.2 respectively.

All the tests were run on a machine with a 2.4 GHz processor, 1 GB of RAM and 80 GB of hard disk. The random data sets used for the tests were generated beforehand, so that the different procedures under analysis could run on exactly the same data and thus be compared fairly. All tests were repeated 20 times, so as to have an average measure of the execution time.

The symbolic simplifications shown in this chapter were obtained with an experimental implementation of the simplification procedure which is available on the world wide web [126]. The time needed to obtain the simplifications shown in this chapter is negligible with respect to the execution times of the tests and is therefore not reported.

Finally, in section 6.3 we show a performance analysis for XML databases.

6.1 Experiments for non-recursive databases

We first consider the tests presented in [153], which were initially proposed in [67]; the method of the so-called *inconsistency indicators* of [153] was shown to run more efficiently than previous methods, namely [147, 122, 69] and naive constraint checking (i.e., with no simplification). We show that, on their examples, we obtain a much better performance (all obtained simplifications are indeed ideal). For compatibility with the compared method, the tests are run under a Prolog system[1].

[1] All experiments, including the replication of results from [153] and [155], were run under SICStus Prolog 3.11.

Let S_1 be the schema with the following intensional database

$$\begin{array}{rcl}
mother(X,Y) & \leftarrow & husband(Z,X) \wedge father(Z,Y) \\
parent(X,Y) & \leftarrow & father(X,Y) \\
parent(X,Y) & \leftarrow & mother(X,Y) \\
wife(X,Y) & \leftarrow & husband(Y,X) \\
married(X,Y) & \leftarrow & husband(X,Y) \\
married(X,Y) & \leftarrow & wife(X,Y) \\
employed(X) & \leftarrow & occup(X,serv) \\
student(X) & \leftarrow & occup(X,stud) \\
dependent(X,Y) & \leftarrow & parent(Y,X) \wedge employed(Y) \wedge student(X) \\
dependent(X,Y) & \leftarrow & married(Y,X) \wedge employed(Y) \wedge \neg employed(X) \\
self(X) & \leftarrow & married(Y,X) \wedge \neg employed(Y) \\
guardian(X,Y) & \leftarrow & dependent(Y,X)
\end{array}$$

and the following integrity constraints:

$$\begin{array}{l}
\leftarrow guardian(X,Y) \wedge \neg sponsor(X,Y) \\
\leftarrow married(X_1,Y_1) \wedge student(X_1) \\
\leftarrow occup(X_2,Y_2) \wedge occup(X_2,Z) \wedge Z \neq Y_2
\end{array}$$

The distribution of facts in the initial database considered in [153] is as follows: 177 $father$ facts, 229 *husband* facts, 620 *occup* facts and 59 *sponsor* facts. We considered additions of tuples to the $father$ and *husband* relations. To test whether an update $U_1 = \{father(\mathbf{a},\mathbf{b})\}$ leads to inconsistency, the method of the inconsistency indicators proposes the following tests (rewritten with our notation):

$$\begin{array}{ll}
\{ & \leftarrow \neg sponsor(\mathbf{a},\mathbf{b}) \wedge guardian(\mathbf{a},\mathbf{b}), \\
 & \leftarrow guardian(X,\mathbf{b}) \wedge \neg sponsor(X,\mathbf{b}) \quad \}.
\end{array}$$

These can be checked by asserting the update as a Prolog fact `father(a,b)` and calling the Prolog query `inconsistent(father(a,b))` on the following Prolog program:

```
inconsistent(father(X,Y)) :- \+ sponsor(X,Y), guardian(X,Y).
inconsistent(father(Z,Y)) :- guardian(X,Y), \+ sponsor(X,Y).
```

Their checking strategy is therefore: assert the update, then retract if inconsistency was detected.

The simplification given by Simp is more specialized and refers only to the extensional predicates:

$$\begin{array}{l}
\mathsf{Simp}^{U_1}(S_1) = \\
\quad \begin{array}{ll}
\{ & \leftarrow occup(\mathbf{a},serv) \wedge occup(\mathbf{b},stud) \wedge \neg sponsor(\mathbf{a},\mathbf{b}), \\
 & \leftarrow husband(\mathbf{a},X) \wedge occup(X,serv) \wedge occup(\mathbf{b},stud) \wedge \neg sponsor(X,\mathbf{b})\}.
\end{array}
\end{array}$$

Our strategy is: first test, then assert the update if inconsistency was not detected.

To see whether the approaches "scale", we ran our tests on databases that are bigger than the initial one by a given factor. Figures 6.1 and 6.2 report this factor on the X-axis and the measured average execution times (in milliseconds) for the additions of 177

father facts and 229 *husband* facts, respectively, with both the inconsistency indicators (II) approach and our simplification (Simp) approach.

In both cases, the performance worsens very quickly with the inconsistency indicators method, whereas it basically remains constant with our approach, with times under $10ms$.

The last example of [153] considers a schema S_2 consisting of the following integrity constraints and intensional predicates[2].

$$\leftarrow civilst(X, Y_1, Z_1, t_1) \wedge civilst(X, Y_2, Z_2, t_2) \wedge \neg(Y_1 = Y_2 \wedge Z_1 \neq Z_2 \wedge t_1 \neq t_2)$$
$$\leftarrow father(X_1, Y) \wedge father(X_2, Y) \wedge X_1 \neq X_2$$
$$\leftarrow husband(X_1, Y) \wedge husband(X_2, Y) \wedge X_1 \neq X_2$$
$$\leftarrow husband(X, Y_1) \wedge husband(X, Y_2) \wedge Y_1 \neq Y_2$$
$$\leftarrow civilst(X, Y, Z, Tax) \wedge (\neg(X > 0 \wedge X < 100000 \wedge Y > 0 \wedge Y < 125)$$
$$\vee(Z \neq m \wedge Z \neq f) \vee (Tax \neq stud \wedge Tax \neq ret \wedge Tax \neq biz \wedge Tax \neq serv))$$
$$\leftarrow civilst(X, Y, Z, stud) \wedge \neg(Y < 25)$$
$$\leftarrow civilst(X, Y, Z, ret) \wedge \neg(Y > 60)$$
$$\leftarrow father(X, Y) \wedge civilst(X, P, S, Q) \wedge S \neq m$$
$$\leftarrow father(X, Y) \wedge civilst(Y, P, S, Q) \wedge S \neq m$$
$$\leftarrow husband(X, Y) \wedge civilst(X, P, S, Q) \wedge S \neq m$$
$$\leftarrow husband(X, Y) \wedge civilst(Y, P, S, Q) \wedge S \neq f$$
$$\leftarrow husband(X, Y) \wedge age(X, P) \wedge age(Y, Q) \wedge (P < 20 \vee Q < 20)$$
$$\leftarrow civilst(X, Y, Z, Tax) \wedge Y < 20 \wedge Tax \neq stud$$
$$\leftarrow dependent(X, Y) \wedge \neg tax(Y, X)$$

$$\begin{aligned}
parent(X, Y) &\leftarrow father(X, Y)\\
parent(X, Y) &\leftarrow mother(X, Y)\\
mother(X, Y) &\leftarrow father(Z, Y) \wedge husband(Z, X)\\
age(X, Y) &\leftarrow civilst(X, Y, P, Q)\\
dependent(X, Y) &\leftarrow parent(Y, X) \wedge occup(Y, serv) \wedge occup(X, stud)\\
occup(X, Y) &\leftarrow civilst(X, P, Q, Y)
\end{aligned}$$

The update in question is a transaction of the form:

$$\begin{aligned}
U_2 = \{ \quad & civilst(\mathbf{a}, \mathbf{p}_a, m, \mathbf{o}_a),\\
& civilst(\mathbf{b}, \mathbf{p}_b, f, \mathbf{o}_b),\\
& civilst(\mathbf{c}, \mathbf{p}_c, \mathbf{s}_c, stud),\\
& husband(\mathbf{a}, \mathbf{b}), father(\mathbf{a}, \mathbf{c}), tax(\mathbf{a}, \mathbf{c})\}
\end{aligned}$$

We observe that in the example it is explicitly assumed that the added family facts were not already in the database; let us indicate this extra hypothesis as Δ. The simplification given by the II method consists of several sets of simplified constraints: one set for every single update in U_2. Instead, the simplification with respect to the whole transaction given by $\mathsf{Optimize}_{\Delta}(\mathsf{Simp}^{U_2}(S_2))$ returns $\langle\emptyset, \emptyset\rangle$. The results of [153] have execution times that vary roughly linearly with respect to the size of the database. Our simplified theory ($\emptyset$) is clearly a great improvement over these results, since it executes in virtually no time and guarantees, without further checking, that this transaction pattern cannot affect integrity. This example was also used in [116], where the authors, unfortunately, only

[2]For compactness, the notation uses disjunctions, but is otherwise compatible with $\mathcal{L}_{\mathsf{S}}$.

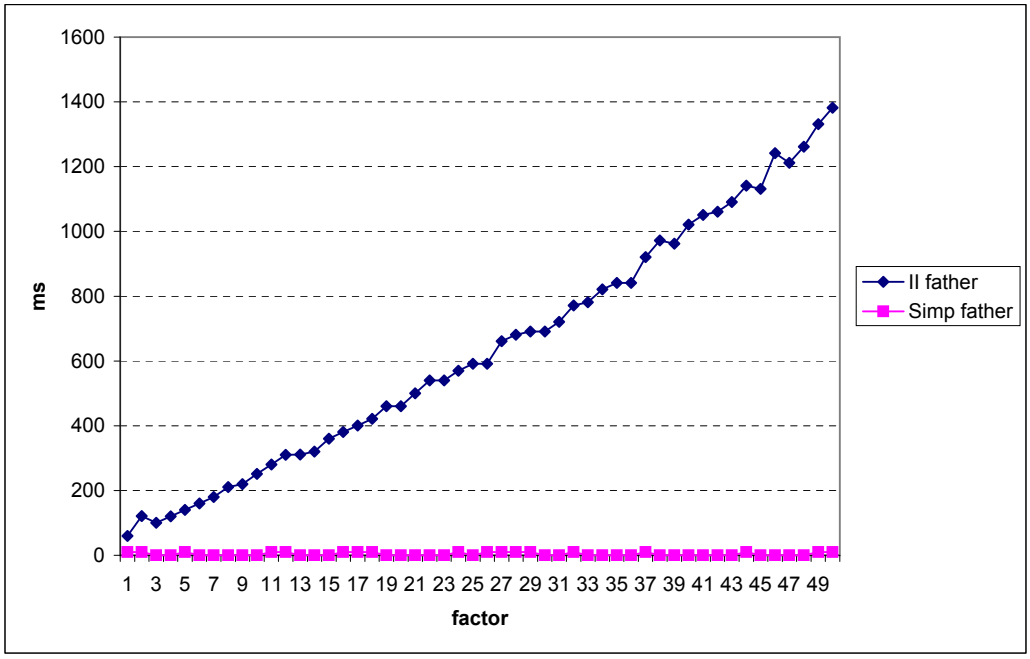

Figure 6.1: Simp vs. Inconsistency Indicators: $father$

compare their method to [122], but not to [153]. However, our transactional simplification is clearly unbeatable.

The redundancies of [153] were reconsidered by the same author in [155]. For the extended example discussed in [155], the schema is as follows.

$$\begin{aligned} S_3 = \langle\ \{\ & mother(X,Y) \leftarrow husband(Z,X) \wedge father(Z,Y), \\ & parent(X,Y) \leftarrow father(X,Y), \\ & parent(X,Y) \leftarrow mother(X,Y), \\ & agediff(X,Y,n) \leftarrow age(X,n_1) \wedge age(Y,n_2) \wedge minus(n,n_1,n_2)\}, \\ \{\ & \leftarrow parent(X,Y) \wedge agediff(X,Y,n) \wedge n < 15\ \}\ \rangle \end{aligned}$$

The results for their revised inconsistency indicators method (RII) and Simp on the addition of $father$ facts on a distribution similar to that considered for S_1 are reported in figure 6.3. In this case our simplifications basically correspond to the unfolding of their so-called revised inconsistency indicators, so there is almost no observable difference in the execution times of the two methods. We stress, however, that the method of [155] has a much more restricted expressive power, in that the updates are limited to singleton insertions and no negations are allowed in the database. Furthermore, in this case the update was simple, so the computational effort required for assertion and retraction of facts was little; however, our approach based on early recognition of inconsistency proves yet more efficient for cases in which updates lead to illegal states (dramatically, if the

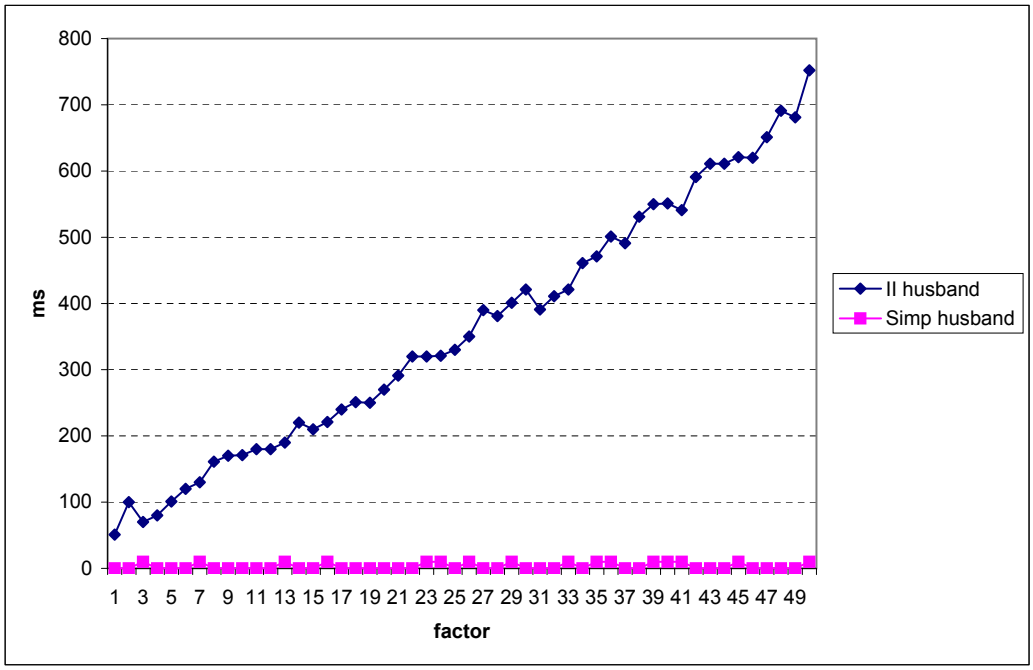

Figure 6.2: Simp vs. Inconsistency Indicators: *husband*

transactions are complex). To see this effect we updated a small database (2 $father$ facts and 2 age facts) with schema S_3 with an illegal $father$ insertion and measured, with the RII method, an answer time approximately four times bigger than with the method based on Simp. This behavior is amplified as the database grows (and it is thus more expensive to add facts for the DBMS): attempting 10000 times the insertion of an illegal $father$ fact on a database with approximately 5000 $father$ facts took about $1s$ with the RII method, but only $70ms$ with Simp. This reflects the fact that with our strategy, upon an illegal update, we just perform a test, whereas the RII method requires to execute the update, perform a consistency test and then roll back the update.

6.2 Experiments for recursive databases

In order to demonstrate the effectiveness of the simplification procedure, we have tested it on random update sets for example 4.3.8 on page 82. Our tests were run using DES 1.1 [148], which is a DATALOG system featuring full recursive evaluation and stratified negation. DES is implemented on top of Prolog; we could therefore program our tests in Prolog and simulate insertions by means of `assert` and deletions by `retract`. The DES query engine is optimized with memoization techniques for answering queries based on previous answers. In this case, we always pose the same query $\leftarrow p(X, X)$ to check whether the graph is acyclic, and therefore answers can be reused for subsequent queries. Our method greatly improves performance even in the presence of an already optimized

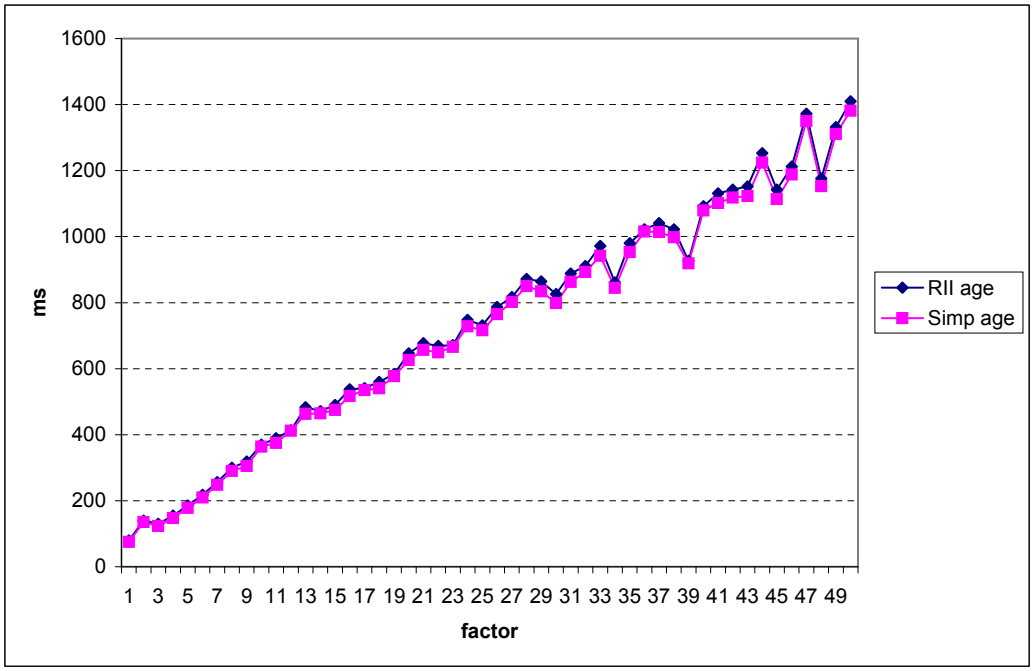

Figure 6.3: Simp vs. Revised Inconsistency Indicators: *age*

system.

Average execution times are indicated in milliseconds (within a time frame of 50 seconds) and the number of attempted insertions of edge facts is indicated on the X-axis. Each figure reports the execution times needed to update the database and check its consistency according to:

- The un-optimized integrity constraint (diamonds).
- The II produced by Seljée's method [154] (crosses).
- The formula $\leftarrow p(b, a)$ (II*), produced by manually removing redundancies from the II (squares).
- The simplification obtained with Simp (triangles). Note that in this case consistency is checked before the update.

The third curve (a "perfect" post-test, which was produced by hand) was included for comparison with the test-before-update strategy. In particular, in figure 6.4 we randomly generated 1500 arcs between 1000 different nodes, whereas in figure 6.5 we only used 50 different nodes. In the former case the formation of cycles is less likely and the times are generally better. In the latter, however, updates are much more likely to be rejected (44% of the updates were rejected in total, while only 12% in the former case); Simp in this case performs significantly better, with improvements around 20% even with respect

to the manually produced formula. The interpretation of these results is in accordance with the following observations:

- The comparison between the performance of the optimized and non-optimized checks shows that the optimized version is always more efficient than the original one.
- In both the un-optimized and II methods the updated state (i.e., many more paths) needs to be computed before the consistency test, which is a possibly expensive operation in the presence of recursive views.
- The gain of early detection of inconsistency, which is a distinctive feature of our approach, is unquestionable in the case of illegal updates. In such a case, with our optimized strategy, the simplified constraint immediately reports an integrity violation with respect to the proposed update, which is therefore *not* executed. On the other hand, the other methods require to execute the update, perform a consistency test and then roll back the update.

Note that the extra burden due to the execution and subsequent rollback of an illegal update is even more evident for compound updates, such as those of example 4.3.9 on page 83; in these cases the benefits of a pre-test with respect to a post-test are even greater. We observe that the above comparisons did not take into account the time spent to produce the optimized constraints, as these can be generated at schema design time and thus do not interfere with run time performance.

6.3 Experiments for XML databases

We now present some experiments conducted on a series of XML datasets matching the DTD of example 5.3.1 on page 102, varying in dimension from 32 to 256 MB. Figure 6.6 refers to the integrity constraints of example 5.3.1. The data were generated by re-mapping data from the DBLP repository [118] into the schema of our running example. Our tests were run using eXist [132] as XQuery engine.

Execution times are indicated in milliseconds and represent the average of the measured times of 20 attempts for each experiment (plus 5 additional operations that were used as a "warm-up" procedure and thus not measured). The size of the documents is indicated in MB on the x-axis. The figure reports three curves representing respectively the time needed to:

- Verify the original constraint (diamonds).
- Verify the optimized constraint (squares).
- Execute an update, verify the original constraint and undo the update (triangles).

Again, we do not have to take into account the time spent to produce the optimized constraints, as these are generated at schema design time and thus do not interfere with run time performance.

The curves with diamonds and squares are used to compare integrity checking in the non-simplified and, resp., simplified case, when the update is legal. The execution time

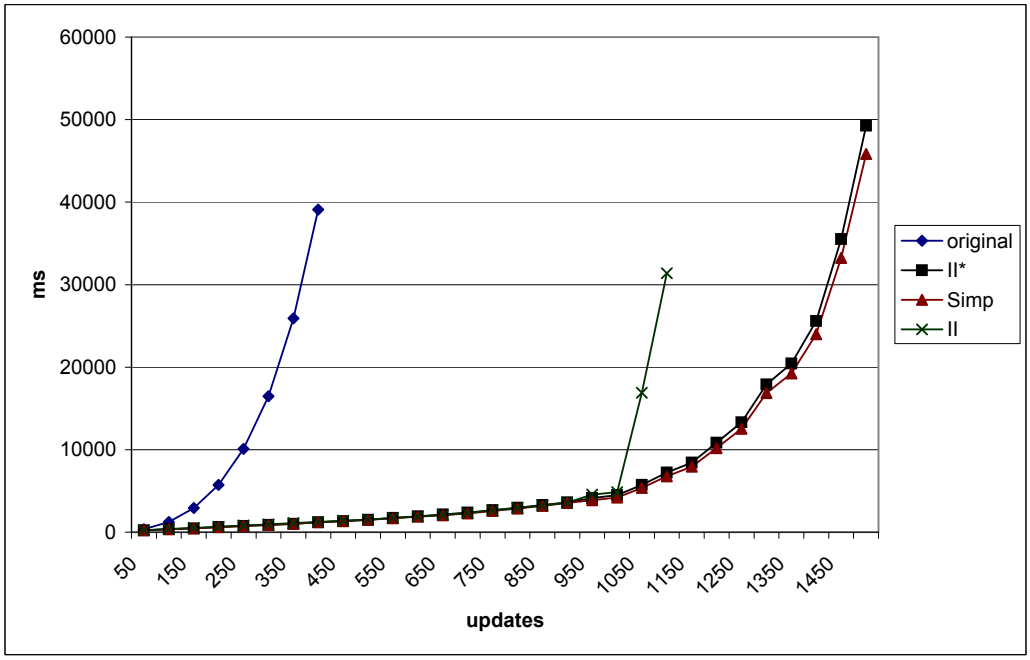

Figure 6.4: Tests on recursive databases: sparse data

needed to perform the update is not included, as this is identical (and unavoidable) in both the optimized and un-optimized case. The curve with triangles includes both the update execution time and the time needed to rollback the update, which is necessary when the update is illegal; when the update is illegal, we then compare the curve with triangles to the curve with squares. Rollbacks were simulated by performing a compensating action to re-construct the state prior to the update. The interpretation of these results is twofold, as we must consider two possible scenarios:

- *The update is legal*: in the un-optimized framework the update is executed first and the full constraint is then checked against the updated database (showing that the update is legal); on the other hand, with our optimized strategy, the simplified constraint is checked first and the update is performed afterwards.
- *The update is illegal*: in the un-optimized framework execution is as in the previous case, but this time the check shows that there is some inconsistency and, finally, a compensative action is performed. On the contrary, with our optimized strategy, the simplified constraint is checked first, which reports an integrity violation with respect to the proposed update; therefore the update statement is *not* executed.

From the experimental results shown in figure 6.6 it is possible to observe two important features.

- Again, the optimized version is always more efficient than the original one. In some cases, as shown in figure 6.6, the difference is remarkable, since the simplified

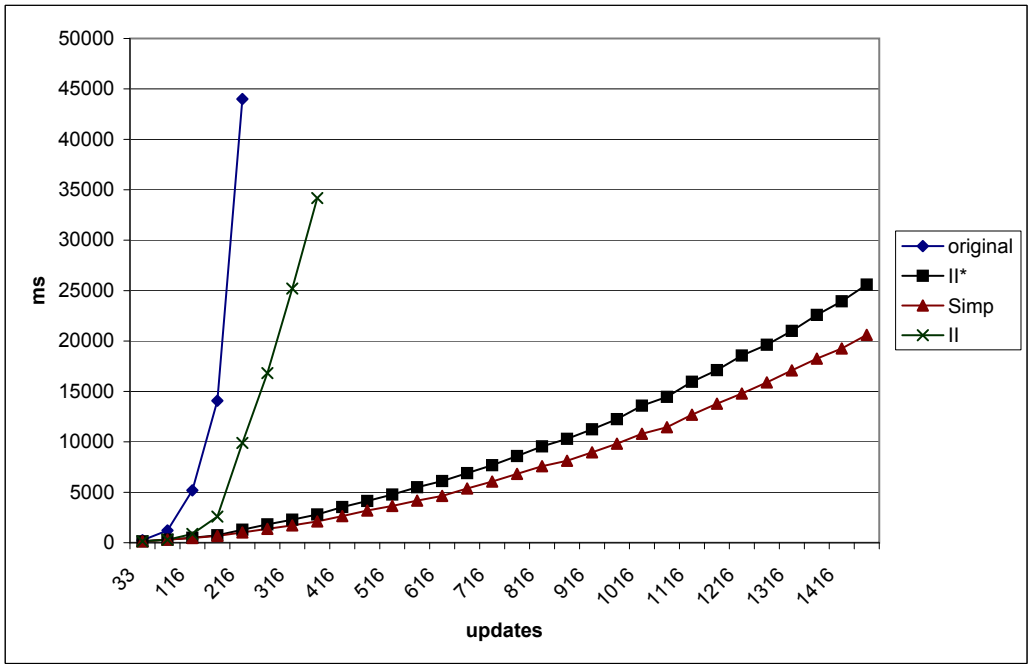

Figure 6.5: Tests on recursive databases: dense data

version contains specific values coming from the concrete update statement which allow one to filter the values on which complex computations are applied. Further improvement is due to the elimination of a join condition in the optimized query.

- Early detection of inconsistency pays off in the case of illegal updates. This is prominently apparent in the case considered in figure 6.6, since, as is well-known, the modification of XML documents is an expensive task.

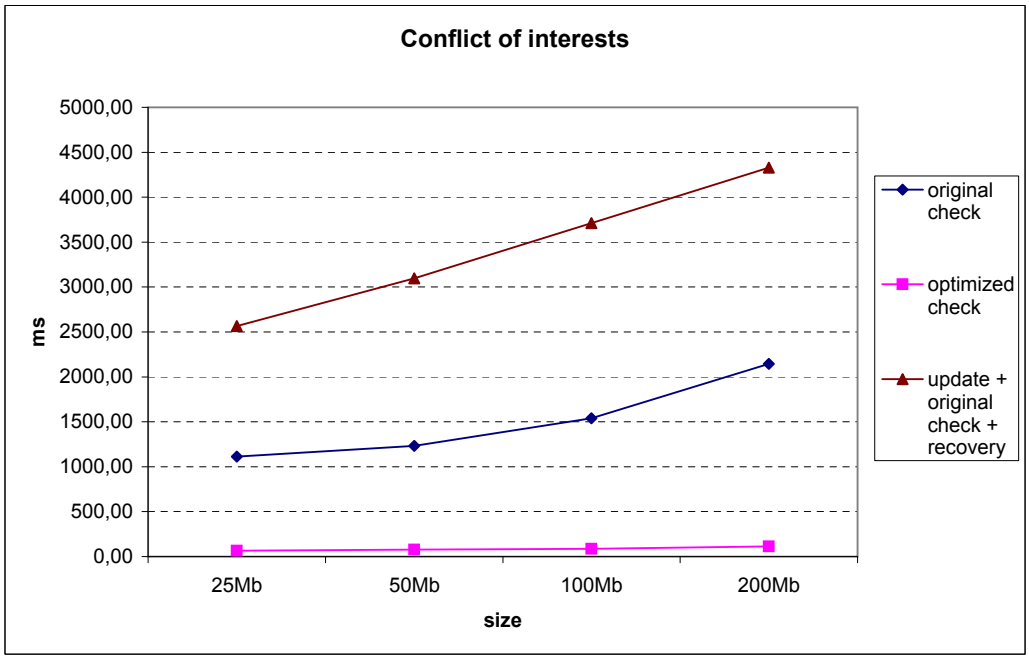

Figure 6.6: Conflict of interests

Chapter 7

Conclusion

We applied program transformation operators to the generation of simplified integrity constraints. A procedure was constructed that makes use of these transformations and produces the simplification searched for according to a minimality criterion. An important contribution of this thesis is the definition of the notion of ideality of a simplification procedure, its connection with the query containment problem, and the analysis of different minimality criteria that can be used to characterize an ideal procedure. In particular, we showed that, for any sensible ordering in which an empty constraint theory represents a minimal element, ideality of simplification corresponds to decidability of query containment.

For specific database languages, we described the implementation of these operators in terms of rewrite rules based on resolution, subsumption and replacement of specific patterns. The versatility of the transformation operators, together with the ability of producing a necessary and sufficient condition for checking integrity before a database update, constitutes the main advantage of our method with respect to earlier approaches. Importantly, we have shown practically relevant cases for which the simplification process is guaranteed to return a minimal result and we have given evidence of the feasibility of the approach with an extensive set of experiments.

We have also indicated how this method can be extended so as to handle existential quantification, recursion, arithmetic and aggregates and how it integrates with locking strategies for databases with concurrent transactions. Cases including all these extensions at the same time could be trivially handled by a rule set comprising all rewrite rules defined in chapter 4, in that such rules are not mutually exclusive. However, it should be interesting to study to which extent this is possible and whether further improvements can be obtained by exploiting the interaction between these rules. Investigation on sublanguages for which some form of minimality can be guaranteed in these cases should also be part of future research.

We believe that present database practice can benefit by the described method. An immediate application is, e.g., the implementation of a database driver that, upon requests from an application, communicates with the database and transparently carries out the required simplified integrity checking operations. This would result in major assets in terms of efficiency, all with a compiled approach, since simplifications are generated at

design time. Practical work based on the foundations provided in this thesis still is in its initial phase, although a Prolog implementation of the simplification procedure is already available [126]. Our results form a nucleus that may inspire the extension of commercial systems based on these principles.

Although we used a logical notation throughout the thesis, standard ways of translating integrity constraints into SQL exist, although further investigation is needed in order to handle additional language concepts of SQL like null values. In [70], Decker showed how to implement integrity constraint checking by translating first-order logic specifications into SQL triggers. The result of our transformations can be combined with similar translation techniques and thus integrated in an active database system, according to the idea of embedding integrity control in triggers, as was indicated (albeit without semantic optimization) by Ceri and Widom [46, 44, 43]. In this way the advantages of declarativity are combined with the efficiency of execution. Along these lines, it should be interesting to study the feasibility of a trigger-based integrity maintenance approach for XML that would combine active behavior with constraint simplification.

Further lines of investigation include integration of visual query specification tools, such as [28], to allow the intuitive specification of XML constraints; in this way domain experts lacking specific competencies in logic would be provided with the ability to design constraints that can be further processed with our approach. Initial analysis of the topic was given in [29]. In this respect, a challenging research topic would be a reformulation of the notion of simplification described in this thesis by means of transformation rules into a set of graph transformation rules expressed in terms of graph grammars for the manipulation of visual constraints.

Other possible enhancements of the proposed framework may be developed using statistical measures on the data, such as estimated relation sizes and cost measurements for performing join and union operations, which are often known by database administrators. Work in this area is closely related to methods for dynamic query processing, e.g., [63, 156]. Finally, we mention that the problem of integrity checking has been or could be addressed also for other paradigms, which were not dealt with in this thesis, such as object-oriented database languages [23, 22] and data streams [13, 12].

Bibliography

[1] S. Abdennadher and H. Christiansen. An experimental CLP platform for integrity constraints and abduction. In H. L. Larsen, J. Kacprzyk, S. Zadrozny, T. Andreasen, and H. Christiansen, editors, *Flexible Query-Answering Systems (FQAS 2000)*, pages 141–152, 2000.

[2] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, 1999.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[4] M. A. Abrão, B. Bouchou, M. H. F. Alves, D. Laurent, and M. A. Musicante. Incremental Constraint Checking for XML Documents. In Z. Bellahsene, T. Milo, M. Rys, D. Suciu, and R. Unland, editors, *Database and XML Technologies, Second International XML Database Symposium, XSym 2004, Toronto, Canada, August 29-30, 2004, Proceedings*, volume 3186 of *Lecture Notes in Computer Science*, pages 112–127. Springer, 2004.

[5] F. Afrati and S. S. Cosmadakis. Expressiveness of restricted recursive queries. In D. S. Johnson, editor, *ACM Symposium on Theory of Computing*, pages 113–126. ACM Press, 1989.

[6] F. Afrati, M. Gergatsoulis, and F. Toni. Linearisability on Datalog programs. *Theoretical Computer Science*, 308(1-3):199–226, 2003.

[7] F. N. Afrati, C. Li, and P. Mitra. On containment of conjunctive queries with arithmetic comparisons. In E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, and E. Ferrari, editors, *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*, pages 459–476. Springer, 2004.

[8] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, Los Altos, CA, 1988.

[9] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19/20:9–71, 1994.

[10] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 68–79. ACM Press, 1999.

[11] O. Arieli, M. Denecker, B. V. Nuffelen, and M. Bruynooghe. Database repair by signed formulae. In D. Seipel and J. M. T. Torres, editors, *Foundations of Information and Knowledge Systems, Third International Symposium (FoIKS 2004)*, volume 2942 of *Lecture Notes in Computer Science*, pages 14–30. Springer, 2004.

[12] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In L. Popa, editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 1–16. ACM, 2002.

[13] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems (TODS)*, 29(3):545–580, 2004.

[14] F. Bancilhon. Naive evaluation of recursively defined relations. In M. L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems (Book resulting from the Islamorada Workshop 1985)*, Topics in Information Systems, pages 165–178. Springer, 1986.

[15] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In C. Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 28-30, 1986*, pages 16–52. ACM Press, 1986.

[16] C. Baral, S. Kraus, and J. Minker. Combining multiple knowledge bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(2):208–220, 1991.

[17] C. Baral, S. Kraus, J. Minker, and V. S. Subrahmanian. Combining knowledge bases consisting of first-order analysis. *Computational Intelligence*, 8:45–71, 1992.

[18] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient Incremental Validation of XML Documents. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, pages 671–682. IEEE Computer Society, 2004.

[19] A. Behrend. Soft stratification for magic set based query evaluation in deductive databases. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–110. ACM Press, 2003.

[20] A. Behrend. *Soft Stratification for Transformation-Based Approaches to Deductive Databases*. PhD thesis, University of Bonn, 2004.

[21] M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng. Automated Update Management for XML Integrity Constraints. In *Informal Proceedings of PLAN-X Workshop*, 2002.

[22] V. Benzaken and A. Doucet. Thémis: A database programming language handling integrity constraints. *VLDB Journal*, 4(3):493–517, 1995.

[23] V. Benzaken and X. Schaefer. Static management of integrity in object-oriented databases: Design and implementation. In H.-J. Schek, F. Saltor, I. Ramos, and G. Alonso, editors, *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27, 1998, Proceedings*, volume 1377 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 1998.

[24] P. A. Bernstein and B. T. Blaustein. Fast methods for testing quantified relational calculus assertions. In M. Schkolnick, editor, *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data, Orlando, Florida, June 2-4, 1982*, pages 39–50. ACM Press, 1982.

[25] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*, pages 64–75. IEEE Computer Society, 2002.

[26] R. N. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1):25–46, 1993.

[27] P. A. Bonatti. On the decidability of containment of recursive Datalog queries - preliminary report. In A. Deutsch, editor, *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 297–306. ACM, 2004.

[28] D. Braga, A. Campi, and S. Ceri. XQBE (XQuery By Example): a visual interface to the standard XML query language. *ACM Transactions on Database Systems (TODS)*, 30(2):398–443, June 2005.

[29] D. Braga, A. Campi, D. Martinenghi, A. Raffio, and D. Salvi. XQBE: the Swiss Army Knife for Semi-structured Data. In *Proceedings of the Thirteenth Italian Symposium on Advanced Database Systems, SEBD 2005, Brizen-Bressanone, Italy, June 19-22, 2005*, pages 284–291, 2005.

[30] F. Bry and R. Manthey. Checking consistency of database constraints: a logical basis. In W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 13–20. Morgan Kaufmann, 1986.

[31] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Reasoning about keys for XML. *Information Systems*, 28(8):1037–1063, 2003.

[32] P. Buneman, W. Fan, and S. Weinstein. Interaction between path and type constraints. *ACM Transactions on Computational Logic (TOCL)*, 4(4):530–577, 2003.

[33] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. Data integration under integrity constraints. In A. B. Pidduck, J. Mylopoulos, C. C. Woo, and M. T. Özsu, editors, *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings*, volume 2348 of *Lecture Notes in Computer Science*, pages 262–279. Springer, 2002.

[34] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. On the expressive power of data integration systems. In S. Spaccapietra, S. T. March, and Y. Kambayashi, editors, *Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling, Tampere, Finland, October 7-11, 2002, Proceedings*, volume 2503 of *Lecture Notes in Computer Science*, pages 338–350. Springer, 2002.

[35] A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini. On the role of integrity constraints in data integration. *IEEE Data Engineering Bulletin*, 25(3):39–45, 2002.

[36] A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 260–271, New York, NY, USA, 2003. ACM Press.

[37] D. Calvanese, G. De Giacomo, and M. Lenzerini. On the decidability of query containment under constraints. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 149–158. ACM Press, 1998.

[38] D. Calvanese, G. De Giacomo, and M. Vardi. Decidable containment of recursive queries. In D. Calvanese, M. Lenzerini, and R. Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2003.

[39] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 1997.

[40] J. Carmo, R. Demolombe, and A. J. I. Jones. An application of deontic logic to information system constraints. *Fundamenta Informaticae*, 48(2-3):165–181, 2001.

[41] S. Carrico, B. Ewbank, T. G. Griffin, J. Meale, and H. Trickey. A tool for developing safe and efficient database transactions. In *XV International Switching Symposium of the World Telecommunications Congress. April, 1995*, pages 173–177, 1995.

[42] T. Catarci and I. F. Cruz. On expressing stratified datalog. In *2nd ICLP Workshop on Deductive Databases and Logic Programming, June 1994, Italy*, pages 85–100, 1994.

[43] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 254–262. Morgan Kaufmann, 2000.

[44] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Transactions on Database Systems (TODS)*, 19(3):367–422, 1994.

[45] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases.* Springer-Verlag New York, Inc., 1990.

[46] S. Ceri and J. Widom. Deriving production rules for constraint maintainance. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 566–577. Morgan Kaufmann, 1990.

[47] U. S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273, Los Altos, CA, 1988. Morgan Kaufmann.

[48] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems (TODS)*, 15(2):162–207, 1990.

[49] A. K. Chandra and D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 1(2):158–163, 1985.

[50] C. L. Chang and R. C. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, 1973.

[51] S. Chaudhuri and M. Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California*, pages 55–66. ACM Press, 1992.

[52] S. Chaudhuri and M. Y. Vardi. Optimization of *real* conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC*, pages 59–70. ACM Press, 1993.

[53] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems (TODS)*, 20(2):149–186, 1995.

[54] H. Christiansen. Automated reasoning with a constraint-based metainterpreter. *Journal of Logic Programming*, 37(1-3):213–254, 1998.

[55] H. Christiansen. Integrity constraints and constraint logic programming. In *INAP'99; Proceedings of the 12th International Conference on Application of Prolog*, pages 5–12. Science University of Tokyo, Japan, 1999. Invited talk.

[56] H. Christiansen and V. Dahl. Assumptions and abduction in Prolog. In S. Muñoz-Hernández, J. M. Gómez-Perez, and P. Hofstedt, editors, *Proceedings of WLPE 2004: 14th Workshop on Logic Programming Environments and MultiCPL 2004: Third Workshop on Multiparadigm Constraint Programming Languages Workshop Proceedings*, pages 87–101, 2004.

[57] H. Christiansen and D. Martinenghi. Symbolic constraints for meta-logic programming. *Applied Artificial Intelligence*, 14(4):345–367, 2000.

[58] H. Christiansen and D. Martinenghi. Simplification of database integrity constraints revisited: A transformational approach. In M. Bruynooghe, editor, *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*, volume 3018 of *Lecture Notes in Computer Science*, pages 178–197. Springer, 2004.

[59] H. Christiansen and D. Martinenghi. Simplification of integrity constraints for data integration. In D. Seipel and J. M. T. Torres, editors, *Foundations of Information and Knowledge Systems, Third International Symposium (FoIKS 2004)*, volume 2942 of *Lecture Notes in Computer Science*, pages 31–48. Springer, 2004.

[60] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, 1977. Advances in Data Base Theory*, pages 293–322. Plenum Press, New York, 1978.

[61] E. F. Codd. Further normalization of the database relational model. In R. Rustin, editor, *Courant Computer Science Symposium 6: Data Base Systems*, pages 33–64. Prentice-Hall, Englewood Cliffs, N.J., 1972.

[62] S. Cohen, W. Nutt, and Y. Sagiv. Containment of aggregate queries. In D. Calvanese, M. Lenzerini, and R. Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2003.

[63] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 150–160. ACM Press, 1994.

[64] S. S. Cosmadakis. On the first-order expressibility of recursive queries. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 311–323. ACM Press, 1989.

[65] W. Cowley and D. Plexousakis. Temporal integrity constraints with indeterminacy. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter,

and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 441–450. Morgan Kaufmann, 2000.

[66] S. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.

[67] S. K. Das and M. H. Williams. Integrity checking methods in deductive databases: a comparative evaluation. In *BNCOD 7: Proceedings of the seventh British national conference on Databases*, pages 85–116. Cambridge University Press, 1989.

[68] S. de Amo, W. A. Carnielli, and J. Marcos. A logical framework for integrating inconsistent information in multiple databases. In T. Eiter and K.-D. Schewe, editors, *Foundations of Information and Knowledge Systems, Second International Symposium, FoIKS 2002 Salzau Castle, Germany, February 20-23, 2002, Proceedings*, volume 2284 of *Lecture Notes in Computer Science*, pages 67–84. Springer, 2002.

[69] H. Decker. Integrity enforcement on deductive databases. In L. Kerschberg, editor, *Expert Database Conference, Proceedings From the First International Conference, Charleston, South Carolina, USA, April 1-4, 1986*, pages 381–395. Benjamin Cummings, 1987.

[70] H. Decker. Translating advanced integrity checking technology to SQL. In J. H. Doorn and L. C. Rivero, editors, *Database integrity: challenges and solutions*, pages 203–249. Idea Group Publishing, 2002.

[71] H. Decker and M. Celma. A slick procedure for integrity checking in deductive databases. In P. Van Hentenryck, editor, *Logic Programming: Proc. of the 11th International Conference on Logic Programming*, pages 456–469. MIT Press, Cambridge, MA, 1994.

[72] H. Decker, J. Villadsen, and T. Waragai, editors. *Paraconsistent Computational Logic, This proceedings volume contains the papers presented at the ICLP 2002 workshop Paraconsistent Computational Logic, on July 27, in Copenhagen, Denmark, as part of the Federated Logic Conference (FLoC)*, volume 95 of *Datalogiske Skrifter*. Roskilde University, Roskilde, Denmark, 2002.

[73] M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In P. Codognet, editor, *Logic Programming, 17th International Conference, ICLP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2237 of *Lecture Notes in Computer Science*, pages 212–226. Springer, 2001.

[74] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 431–442. ACM Press, 1999.

[75] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[76] G. Dong and J. Su. Incremental Maintenance of Recursive Views Using Relational Calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.

[77] N. Eisinger and H. J. Ohlbach. Deduction systems based on resolution. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming - Vol 1: Logical Foundations*, pages 183–271. Clarendon Press, Oxford, 1993.

[78] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems (TODS)*, 2(3):262–278, 1977.

[79] W. Fan, G. M. Kuper, and J. Siméon. A unified constraint model for XML. *Computer Networks*, 39(5):489–505, 2002.

[80] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *Journal of the ACM*, 49(3):368–406, 2002.

[81] W. Fan and J. Siméon. Integrity constraints for XML. *Journal of Computer and System Sciences*, 66(1):254–291, 2003.

[82] C. G. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1791–1849. Elsevier and MIT Press, 2001.

[83] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[84] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS '98. Proceedings of the ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, June 1–3, 1998, Seattle, Washington*, pages 139–148, New York, NY 10036, USA, 1998. ACM Press.

[85] T. W. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

[86] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems. The complete book*. Prentice-Hall, 2002.

[87] M. Gelfond and V. Lifschitz. Minimal model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth Logic Programming Symposium*, pages 1070–1080. MIT Press, 1988.

[88] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems (the book grow out of the Dagstuhl Seminar 9529: Role of Logics in Information Systems, 1995)*, pages 265–306. Kluwer, 1998.

[89] P. Godfrey, J. Gryz, and C. Zuzarte. Exploiting constraint-like data characterizations in query optimization. In W. G. Aref, editor, *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 582–592. ACM Press, 2001.

[90] G. Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109–111, 1987.

[91] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the Twenty-Second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 9-12, 2003, San Diego, CA, USA*, pages 179–190. ACM, 2003.

[92] G. Gottlob, C. Koch, and K. Schulz. Conjunctive Queries over Trees. In A. Deutsch, editor, *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 189–200, 2004.

[93] J. Grant and J. Minker. Integrity constraints in knowledge based systems. In H. Adeli, editor, *Knowledge Engineering Vol II, Applications*, pages 1–25. McGraw-Hill, 1990.

[94] J. Grant and J. Minker. The impact of logic programming on databases. *Communications of the ACM (CACM)*, 35(3):66–81, 1992.

[95] J. Grant and J. Minker. A logic-based approach to data integration. *Theory and Practice of Logic Programming (TPLP)*, 2(3):323–368, 2002.

[96] S. Greco, C. Sirangelo, I. Trubitsyna, and E. Zumpano. Preferred repairs for inconsistent databases. In *7th International Database Engineering and Applications Symposium (IDEAS 2003), 16-18 July 2003, Hong Kong, China*, pages 202–211. IEEE Computer Society, 2003.

[97] P. W. P. J. Grefen. Combining theory and practice in integrity control: A declarative approach to the specification of a transaction modification subsystem. In R. Agrawal, S. Baker, and D. A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 581–591. Morgan Kaufmann, 1993.

[98] U. Griefahn. *A Uniform Approach to the Implementation of Deductive Databases.* PhD thesis, University of Bonn, 1997.

[99] A. Gupta and I. S. Mumick, editors. *Materialized views: techniques, implementations, and applications.* MIT Press, 1999.

[100] L. Henschen, W. McCune, and S. Naqvi. Compiling constraint-checking programs from first-order formulas. In H. Gallaire, J. Minker, and J.-M. Nicolas, editors, *Advances In Database Theory, February 1-5, 1988, Los Angeles, California, USA*, volume 2, pages 145–169. Plenum Press, New York, 1984.

[101] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[102] A. Hsu and T. Imielinski. Integrity checking for multiple updates. In S. B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on*

Management of Data, Austin, Texas, May 28-31, 1985, pages 152–168. ACM Press, 1985.

[103] INCITS. *Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation) - INCITS/ISO/IEC 9075-2-1999*, 1999.

[104] Y. E. Ioannidis and E. Wong. Towards an algebraic theory of recursion. *Journal of the ACM*, 38(2):329–381, 1991.

[105] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[106] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[107] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2:719–770, 1992.

[108] D. Kapur and P. Narendran. NP-Completeness of the Set Unification and Matching Problems. In J. H. Siekmann, editor, *8th International Conference on Automated Deduction, Oxford, England, July 27 - August 1, 1986, Proceedings*, volume 230 of *Lecture Notes in Computer Science*, pages 489–495. Springer, 1986.

[109] D. B. Kemp and P. J. Stuckey. Semantics of logic programs with aggregates. In V. A. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium (ISLP), San Diego, California, USA, Oct. 28 - Nov 1, 1991*, pages 387–401. MIT Press, 1991.

[110] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebras*, pages 263–297. Pergamon Press, 1970.

[111] P. Kolaitis. The expressive power of stratified logic programs. *Information and Computation*, 90(1):50–66, january 1991.

[112] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL query translation literature: The state of the art and open problems. In *Proceedings of the International XML Database Symposium (XSym)*, pages 1–18, 2003.

[113] A. Laux and L. Matin. *XUpdate working draft*. `http://www.xmldb.org/xupdate`, October 2000.

[114] S. Y. Lee and T. W. Ling. Further improvements on integrity constraint checking for stratifiable deductive databases. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 495–505. Morgan Kaufmann, 1996.

[115] A. Leitsch. Resolution theorem proving: A logical point of view. In M. Baaz, editor, *Logic Colloquium '01: Proceedings Of The Annual European Summer Meeting Of The Association For Symbolic Logic, Held In Vienna, Austria, August 6-11, 2001*, pages 3–42. A. K. Peters Ltd, 2001.

[116] M. Leuschel and D. de Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *Journal of Logic Programming*, 36(2):149–193, 1998.

[117] A. Y. Levy. Combining artificial intelligence and databases for data integration. In M. Wooldridge and M. M. Veloso, editors, *Artificial Intelligence Today: Recent Trends and Developments*, volume 1600 of *Lecture Notes in Computer Science*, pages 249–268. Springer, 1999.

[118] M. Ley. Digital Bibliography & Library Project. `http://dblp.uni-trier.de/`.

[119] C. Li. Describing and utilizing constraints to answer queries in data-integration systems. In S. Kambhampati and C. A. Knoblock, editors, *Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), August 9-10, 2003, Acapulco, Mexico*, pages 163–168, 2003.

[120] J. Lin and A. O. Mendelzon. Merging databases under constraints. *International Journal of Cooperative Information Systems*, 7(1):55–76, 1998.

[121] J. Lloyd. *Foundations of Logic Programming (2nd Edition)*. Springer, Berlin, 1987.

[122] J. W. Lloyd, L. Sonenberg, and R. W. Topor. Integrity constraint checking in stratified databases. *Journal of Logic Programming*, 4(4):331–343, 1987.

[123] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 3:225–240, 1984.

[124] J. W. Lloyd and R. W. Topor. A basis for deductive database systems II. *Journal of Logic Programming*, 30(1):55–67, 1986.

[125] D. Martinenghi. Simplification of integrity constraints with aggregates and arithmetic built-ins. In H. Christiansen, M.-S. Hacid, T. Andreasen, and H. L. Larsen, editors, *Flexible Query Answering Systems, 6th International Conference, FQAS 2004, Lyon, France, June 24-26, 2004, Proceedings*, volume 3055 of *Lecture Notes in Computer Science*, pages 348–361. Springer, 2004.

[126] D. Martinenghi. A simplification procedure for integrity constraints. `http://www.dat.ruc.dk/~dm/spic/index.html`, 2004.

[127] D. Martinenghi and H. Christiansen. Efficient integrity checking for databases with recursive views. In J. Eder, H.-M. Haav, A. Kalja, and J. Penjam, editors, *Ninth East-European Conference on Advances in Databases and Information Systems (ADBIS 05), September 12-15, 2005, Tallinn, Estonia*, volume 3631 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2005.

[128] D. Martinenghi and H. Christiansen. Transaction management with integrity checking. In K. V. Andersen, J. Debenham, and R. R. Wagner, editors, *Database and Expert Systems Applications, 16th International Conference, DEXA 2004 Copenhagen, Denmark, August 22-26, 2005, Proceedings*, volume 3588 of *Lecture Notes in Computer Science*, pages 606–615. Springer, 2005.

[129] W. May. XPath-Logic and XPathLog: a logic-programming-style XML data manipulation language. *Theory and Practice of Logic Programming (TPLP)*, 4(3):239–287, 2004.

[130] E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In P. P. Chen, D. W. Embley, J. Kouloumdjian, S. W. Liddle, and J. F. Roddick, editors, *Advances in Conceptual Modeling: ER '99 Workshops, Paris, France, November 15-18, 1999, Proceedings*, volume 1727 of *Lecture Notes in Computer Science*, pages 62–73. Springer, 1999.

[131] J. D. McCharen, R. A. Overbeek, and L. Wos. Problems and experiments for and with automated theorem-proving programs. *IEEE Transactions on Computers*, 25(8):773–782, 1976.

[132] W. Meier. eXist: An Open Source Native XML Database. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 169–183, London, UK, 2003. Springer-Verlag.

[133] A. Motro. Integrity = validity + completeness. *ACM Transactions on Database Systems (TODS)*, 14(4):480–502, 1989.

[134] J. F. Naughton. Minimizing function-free recursive inference rules. *Journal of the ACM*, 36(1):69–91, 1989.

[135] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 235–242. ACM Press, 1989.

[136] F. Neven and T. Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In D. Calvanese, M. Lenzerini, and R. Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2003.

[137] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.

[138] S.-H. Nienhuys-Cheng and R. de Wolf. The equivalence of the subsumption theorem and the refutation-completeness for unconstrained resolution. In K. Kanchanasut and J.-J. Lévy, editors, *Algorithms, Concurrency and Knowledge: 1995 Asian Computing Science Conference, ACSC '95, Pathumthani, Tailand, December 11-13, 1995, Proceedings*, volume 1023 of *Lecture Notes in Computer Science*, pages 269–285. Springer, 1995.

[139] U. Nilsson and J. Małuzyński. *Logic, Programming and Prolog (2nd ed.)*. John Wiley & Sons Ltd, 1995.

[140] Ocelot Computer Services Inc. The Ocelot SQL DBMS. `http://www.ocelot.ca/`.

[141] Y. Papakonstantinou and V. Vianu. Incremental Validation of XML Documents. In D. Calvanese, M. Lenzerini, and R. Motwani, editors, *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*, pages 47–63. Springer, 2003.

[142] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, Los Altos, CA, 1988. Morgan Kaufmann.

[143] X. Qian. An effective method for integrity constraint simplification. In *Proceedings of the Fourth International Conference on Data Engineering, February 1-5, 1988, Los Angeles, California, USA*, pages 338–345. IEEE Computer Society, 1988.

[144] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[145] R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 56–76. Plenum Press, New York, 1978.

[146] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[147] F. Sadri and R. Kowalski. A theorem-proving approach to database integrity. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 313–362. Morgan Kaufmann, Los Altos, CA, 1988.

[148] F. Sáenz-Pérez. Datalog Educational System V1.1. User's Manual. Technical Report 139-04, Faculty of Computer Science, UCM, 2004. Available from http://www.fdi.ucm.es/profesor/fernan/DES/.

[149] K. Salem, H. Garcia-Molina, and J. Shands. Altruistic locking. *ACM Transactions on Database Systems (TODS)*, 19(1):117–165, 1994.

[150] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental maintenance of path expression views. In *SIGMOD Conference, 2005: Baltimore, Maryland, USA*, 2005.

[151] T. Schwentick. XPath query containment. *SIGMOD Record*, 33(1):101–109, 2004.

[152] M. Sebag and C. Rouveirol. Any-time relational reasoning: Resource-bounded induction and deduction through stochastic matching. *Machine Learning*, 38(1-2):41–62, 2000.

[153] R. Seljée. A new method for integrity constraint checking in deductive database. *Data & Knowledge Engineering*, 15(1):63–102, 1995.

[154] R. Seljée. *A Fact Integrity Constraint Checking System for the Validation of Semantic Integrity Constraints after Updating Consistent Deductive Databases*. PhD thesis, Tilburg University, 1997.

[155] R. Seljée and H. C. M. de Swart. Three types of redundancy in integrity checking: An optimal solution. *Data & Knowledge Engineering*, 30(2):135–151, 1999.

[156] P. Seshadri, J. M. Hellerstein, H. Pirahesh, T. Y. C. Leung, R. Ramakrishnan, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 435–446. ACM Press, 1996.

[157] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 302–314. Morgan Kaufmann, 1999.

[158] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 237–249. ACM Press, 1987.

[159] D. Suciu. Query Decomposition and View Maintenance for Query Languages for Unstructured Data. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 227–238. Morgan Kaufmann, 1996.

[160] G. Sur, J. Hammer, and J. Siméon. UpdateX - An XQuery-Based Language for Processing Updates in XML. In *PLAN-X 04*, pages 40–53, 2004.

[161] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In W. G. Aref, editor, *ACM SIGMOD Conference 2001: Santa Barbara, CA, USA*, 2001.

[162] R. W. Topor. Domain-independent formulas and databases. *Theoretical Computer Science*, 52:281–306, 1987.

[163] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I & II.* Computer Science Press, 1988/89.

[164] J. D. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.

[165] A. van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 88)*, pages 221–230. ACM Press, 1988.

[166] G. von Bültzingsloewen. Translating and Optimizing SQL Queries Having Aggregates. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 235–243. Morgan Kaufmann, 1987.

[167] W3C. XML Schema 1.0. `http://www.w3.org/XML/Schema`, May 2001.

[168] C. Youn, L. J. Henschen, and J. Han. Classification of recursive formulas in deductive databases. In H. Boral and P.-Å. Larson, editors, *Proceedings of the 1988 ACM SIGMOD international conference on Management of data, Chicago, Illinois, June 1-3, 1988*, pages 320–328. ACM Press, 1988.

[169] C. Zaniolo. Key constraints and monotonic aggregates in deductive databases. In A. C. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408 of *Lecture Notes in Artificial Intelligence*, pages 109–134. Springer, 2002.

Index