Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

TU WIEN
FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

New Representations of Uncertain Databases for Efficient Updates

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Andreas Hubmer, BSc
Matrikelnummer 0525780

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.-Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Projektass. Vadim Savenkov, MSc
Mitwirkung: Projektass. Dipl.-Ing. Sebastian Skritek

Wien, 31.08.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Andreas Hubmer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 31. August 2011

____

(Unterschrift)

Kurzfassung

Ungewisse Daten entstehen in vielen Anwendungen, beispielsweise in Sensornetzwerken, bei der Informationsextraktion und bei der Datenintegration. Während gewöhnliche relationale Datenbanken keine Möglichkeit bieten diese Daten zu verarbeiten, verwalten ungewisse Datenbanken die Ungewissheit auf geeignete Weise. Sie modellieren eine Vielzahl an möglichen Welten. Die Herausforderung besteht darin, ungewisse Datenbanken möglichst Platz sparend darzustellen, und gleichzeitig eine effiziente Abfragebeantwortung zu ermöglichen. U-Relationen sind ein ein äußerst Platz sparendes Darstellungssystem für ungewisse Datenbanken. Abfragen in positiver relationaler Algebra, erweitert um einen Operator für mögliche Antworten, können in polynomieller Zeit (Datenkomplexität) bearbeitet werden.

In dieser Arbeit gehen wir der Frage nach, wie U-Relationen aktualisiert werden können. Wir zeigen, dass positive relationale Algebra nicht ausreicht, um beliebige Updates durchzuführen, und reduzieren das Aktualisierungsproblem auf das Problem der Mengendifferenz zwischen zwei U-Relationen. Wir stellen zwei Methoden dar: Dekompression und Negation. Während die Dekompression eine hohe Komplexität aufweist, berücksichtigt die Negation auch die Struktur der U-Relationen und erzielt dadurch bessere Ergebnisse. Zusätzlich erweitern wir U-Relationen und beschreiben zwei neue Darstellungssysteme für ungewisse Datenbanken: U^i-Relationen und U^{int}-Relationen. Sie behalten die Vorteile von U-Relationen bei und reduzieren die Komplexität für die Mengendifferenz um einen exponentiellen Faktor.

Nach einer Aktualisierung werden U-Relationen möglicherweise nicht optimal dargestellt. Daher untersuchen wir das diesbezügliche Minimierungsproblem. Wir beweisen, dass das Minimierungsproblem NP-schwer ist und erläutern zwei Optimierungsheuristiken. Eine der beiden kann in die Berechnung der Mengendifferenz integriert werden, wodurch wir eine bessere obere Schranke für die Komplexität der Mengendifferenz zeigen können.

Um die praktische Anwendbarkeit der beschriebenen theoretischen Konzepte zu belegen, haben wir einen Prototyp entwickelt. Unsere Experimente zeigen, dass unsere Lösungen gut mit der Größe der Datenbanken skalieren. Außerdem zeigt ein Vergleich, dass sowohl U^i-Relationen als auch U^{int}-Relationen deutlich den U-Relationen überlegen sind.

Abstract

Uncertain data arises in many applications, for example from sensor networks, information extraction and data integration. While ordinary relational databases do not have the capabilities to process uncertain data, uncertain databases conveniently manage the uncertainty. They model a multitude of possible worlds. The challenge is to represent uncertain databases succinctly, and at the same time allow for efficient query evaluation. U-relations are a very succinct representation system for uncertain databases, which have polynomial time data complexity for positive relational algebra queries extended by an operator for computing possible answers.

In this work we study the problem of updating U-relations. We show that positive relational algebra does not suffice to model arbitrary updates and reduce the problem of updating U-relations to computing set difference on U-relations. We present two approaches: decompression and negation. While decompression has a rather high complexity, negation considers the structure of a U-relation and thus gives much better results. Additionally we extend U-relations and present two new representation systems for uncertain databases: U^{i}-relations and U^{int}-relations. They keep the advantages of U-relations and lower the complexity of set difference exponentially.

As updating U-relations can lead to a suboptimal representation of the uncertain database, we investigate the problem of optimizing U-relations. We prove that finding the minimal representation is intractable and present two optimization heuristics. We integrate one of them into the computation of set difference, which gives a better bound for the complexity of set difference.

To show the practicability of our theoretical concepts we have created a prototype. Our experiments show that it scales well with increasing database sizes, and a comparison makes clear that U^{i}-relations and U^{int}-relations are an improvement over U-relations.

Acknowledgments

First I want to thank my supervisor Reinhard Pichler, who invited me to work in his group, for his expertise, his support and his helpful guidance. Just as well I acknowledge the numerous pieces of advice which I received from Vadim Savenkov and Sebastian Skritek. They provided valuable feedback in many occasions.

I want to thank Martin, Ruslan, Jussi and Ali for the time we spent together. With them I could discuss the life and problems of a young researcher, and a shared lunch break (with German lessons) could relieve one from hard thoughts.

When finishing a period of life it is also time to thank those who made it possible and who supported one the most. For the lifelong support I want to thank my parents. For the daily support and for the unbounded confidence in the success of my work I feel grateful towards Cathy.

This work was supported by the Vienna Science and Technology Fund (WWTF) under grant ICT08-032.

Contents

Kurzfassung ii

Abstract iii

Acknowledgments iv

Contents v

List of Figures vii

List of Algorithms viii

1 Introduction 1

2 Preliminaries 4

3 U-relations 6
- 3.1 Running Example . . . 7
- 3.2 Definition of U-relations . . . 8
- 3.3 Probabilistic U-relations . . . 13
- 3.4 Query Language . . . 13

4 Updates and Set Difference on U-relations 18
- 4.1 Motivation . . . 18
- 4.2 Updates . . . 19
- 4.3 Delete Statements . . . 23
- 4.4 Set Difference . . . 24
- 4.5 Inverse of a Ws-set . . . 26
- 4.6 Summary . . . 31

5 Novel Descriptors for World Sets 32
- 5.1 Descriptors with Inequalities . . . 34
- 5.2 Iws-descriptors outpace Ws-descriptors . . . 37
- 5.3 Descriptors with Intervals . . . 48
- 5.4 Intws-descriptors outpace Ws-descriptors . . . 51
- 5.5 Summary . . . 52

6 Optimization **53**
6.1 Optimizing Descriptors of World Sets 54
6.2 Tractable Optimization 57
6.3 Optimizing Intervals 61
6.4 Optimizing Set Difference 65
6.5 Summary 73

7 Implementation **74**
7.1 Data Type Wsdescriptor 75
7.2 Architecture 75
7.3 Repair-Key 76
7.4 Except 78
7.5 Details 79
7.6 Correctness 80
7.7 Summary 81

8 Experimental Results **82**
8.1 Test System and Test Data 82
8.2 Updates 84
8.3 Set Difference 89
8.4 Repair-Key 92
8.5 Summary 93

9 Conclusion **95**

Bibliography **97**

List of Figures

3.1 A U-relational database representing product ratings. 7
3.2 A possible world of the product ratings database. 11
3.3 A probabilistic U-relational database. 12
3.4 Concept of query evaluation on the representation level. 14
3.5 Translation of positive relational algebra queries with *poss* and *merge* into queries on U-relations (based on [6]). 16
3.6 Exemplary queries and their results. 17

4.1 Updated U-relation which had to be decompressed. 19
4.2 Expressing updates by queries. 20
4.3 Positive relational algebra cannot model all updates - a counterexample. . . 21
4.4 The semi-join update fully decompresses the updated relation. 23
4.5 Expressing delete statements by queries. 23
4.6 Translation of set difference, based on the inverse of ws-sets. 26
4.7 The inverse of a ws-set, using negation. 29

5.1 A U-relational database representing stolen car investigations. 33
5.2 The U-relation *Cars* after applying the update. 33
5.3 The U-relation *Cars* after applying the update, using inequalities. 33
5.4 The U-relation *Cars* after applying the update, using intervals. 34
5.5 Consistency check for iws-descriptors. 37
5.6 Negation for iws-sets. 37
5.7 Sketch for the proof of Claim 2. 44
5.8 Consistency check for intws-descriptors. 50
5.9 Negation for intws-sets. 50

6.1 A U-relation that can be optimized. 53
6.2 From GAP OPT to CBM. 63
6.3 Except for level j, all levels below the i-th are skipped. 72

7.1 Query processing in PostgreSQL. 75
7.2 The relation Person before and after being repaired. 76
7.3 Code Structure. 79

8.1 TPC-H Schema [48]. 83

8.2 Test case 1: Update queries. . . . 85
8.3 Influence of the scale factor on updates. . . . 86
8.4 Influence of the uncertainty ratio on updates. . . . 87
8.5 Influence of the domain size of the variables on updates. . . . 88
8.6 Test case 2: Queries with set difference. . . . 90
8.7 Comparison for query Q_6. . . . 90
8.8 Comparison for query Q_7. . . . 91
8.9 Comparison for query Q_8. . . . 91
8.10 Comparison for query Q_9. . . . 91
8.11 Runtime comparison for repair-key. . . . 92

List of Algorithms

1 CONSISTENT(d_1, d_2) . . . 17
2 INV-DECOMPRESS(S) . . . 27
3 INV-NEGATE$(S, \textit{level}, \textit{tmp}, \textit{result})$ . . . 30
4 FEASIBLE(d) . . . 36
5 CONTAINS(d_1, d_2) . . . 59
6 CAN-SKIP$(S, \textit{level}, \textit{tmp})$ . . . 69
7 INV-NEGATE-SKIP$(S, \textit{level}, \textit{tmp}, \textit{result})$ . . . 69
8 REPAIR-KEY(R, K) . . . 77
9 EXCEPT(R, S) . . . 78

1. Introduction

The amount of uncertain data rises continually. Sources of uncertain data are for example sensor networks [13], information extraction and data integration. More and more devices get equipped with different sensors, just think of smartphones. They gather a lot of highly uncertain location data, which is used by many applications. When performing information extraction, ambiguous information on the web leads to uncertain data. And in data integration conflicting sources introduce uncertainty. Ordinary relational databases do not provide the necessary abilities to process uncertain data. This explains the increasing need for uncertain databases that manage the uncertainty conveniently.

Incomplete and uncertain databases were introduced in the 1980's [26, 2]. Over the past years, research on uncertain databases has increased steadily. For a fairly recent overview, see for example [4, 47, 17, 3]. Mostly the possible worlds semantics [2] is used to manage uncertain data. In certain databases there is one world, consisting of all the tuples in the database. Uncertain databases consist of a multitude of possible worlds, of which one is supposed to be the true one. A special case of uncertain databases are probabilistic databases, which have received a lot of attention by the research community recently (e.g. [32, 44, 27, 5, 43]). In the probabilistic case each world is assigned a probability value. Conceptually, queries on uncertain databases are evaluated on every world individually. In practice this cannot be done as the number of worlds easily reaches astronomic numbers [9]. The challenge is to represent uncertain databases succinctly, and at the same time allow for efficient query evaluation.

Several representation systems for uncertain databases have been described and are available for use [11, 51, 52, 45, 12]. Antova et al. introduced *U-relations* [6], a very competitive representation system for uncertain databases. They have polynomial time data complexity for positive relational algebra queries extended by an operator for computing possible answers. Compared to other representation systems of uncertain databases, U-relations can be exponentially more succinct. So-called *ws-sets* describe in which worlds a tuple actually exists. Ws-sets are formulas in disjunctive normal form over variables with finite domains. U-relations can be used in practice through the MayBMS project [28] which implements U-relations on top of a relational database management system.

As in the case of certain databases, it is also natural to update uncertain databases. In [7] an API for uncertain databases, which covers updates, is presented. The paper mentions that for updates decompression of the succinct representation may be necessary. However, it leaves open the details and also the question whether there are any

better methods possible. Another open problem is the evaluation of unrestricted relational algebra queries on uncertain databases. The problem of evaluating positive relational algebra queries (queries without set difference) on uncertain databases has been studied extensively (e.g. [16, 8, 15, 19]). There has been some work beyond positive relational algebra, like queries with *having* clauses [41] and queries with one level of *not-exists* [50]. Fink et al. [20] describe an approach for unrestricted relational algebra queries, but in a formalism that does not exhibit polynomial time data complexity for computing possible answers.

In this thesis, we investigate the problem of updating uncertain and probabilistic databases (in the form of U-relations). We have identified four principal research questions:

- *Updates*. How can updates be modeled on U-relations? It is quite easy to see that set difference can be used to model updates. But set difference is expensive. Are there any restricted cases for which positive relational algebra suffices? The advantage of positive relational algebra is that it has polynomial time data complexity.

- *Set difference*. How can set difference be modeled on U-relations? In [7] the use of decompression for updates is mentioned. In a similar way decompression can be used for set difference. What is the complexity of decompression? Are there other approaches that have a lower complexity?

- *Optimizing U-relations*. Different U-relations can represent the same uncertain database. After updating a U-relation the representation might not be optimal. How hard is it to minimize a U-relation such that it still represents the same information? In case the problem is intractable it is of interest to find tractable optimization heuristics.

- *Experimental evaluation*. How do the developed concepts and algorithms behave in practice? A prototypical implementation can serve as a proof of concept.

Summary of Results

We study the problem of updating U-relations. In summary, we define the set difference operator for U-relations and show how to use it to model updates. We work out algorithms and study their complexity, and we prove the benefit of two new representation systems and of various optimization measures. The main contributions are:

- *Formal definition*. We are the first to define relational set difference on U-relations such that the result is again a U-relation. We reduce the problem to computing inverses of ws-sets. To this end we propose two approaches: Decompression and

negation. Whereas decompression is a naive method that always depends exponentially on the number of involved variables, negation considers the structure of the ws-set to reduce the cardinality of the result. On top of relational set difference we define a comprehensive update language.

- *New representations*. We introduce U^i-relations, which are a generalization of U-relations in that they additionally allow inequalities. We prove that by using U^i-relations the complexity of query answering can drop exponentially but never increase. Because of certain disadvantages of U^i-relations, we introduce another generalization: U^{int}-relations. They replace equalities by intervals over the variables. For both generalizations we define how relational algebra queries extended by the possible operator can be evaluated.

- *Optimization of world sets*. Updating uncertain databases can lead to a non-optimal representation of uncertain relations, which increases the size of the database and the time needed to answer queries. We prove that finding the minimal representation of a U-relation, U^i-relation or a U^{int}-relation is intractable. In consequence we propose two polynomial-time optimization heuristics that reduce the size of the representation. We integrate one of them into the computation of inverses, which indeed results in a better performance. Additionally we can thus prove a significantly better worst-case complexity. On U^{int}-relations another optimization problem arises which we prove to be intractable too.

- *Prototype*. We have extended MayBMS to support set difference and arbitrary updates. Experimentally we show the practicability and also the limits of our approach. A comparison makes clear that U^i-relations and U^{int}-relations are an improvement over U-relations. With them the query evaluation time and the size of the updated databases is lowered considerably.

Organization

The following Chapter 2 shortly defines the basic notions used in this work. Then in Chapter 3 we formally introduce U-relations and show how positive relational queries are translated into queries on U-relations. Our main theoretical results are presented in the Chapters 4–6. In Chapter 4 we trace updates back to computing the inverse of a ws-set. To reduce the complexity of computing inverses we describe U^i-relations and U^{int}-relations in Chapter 5. In Chapter 6 we study the optimization of U-relational databases. Chapter 7 describes our implementation of the developed concepts and in Chapter 8 we discuss the experimental results. Finally, Chapter 9 concludes and gives an outlook to future work.

2. Preliminaries

In this chapter we shortly introduce the basic notions used throughout this thesis. For more details on database theory we refer to comprehensive textbooks [1, 21, 40, 18].

Definition 2.0.1 (schema)**.** A schema Σ is a finite set of relation symbols $\{R_1, \ldots, R_n\}$ which have a sequence of attributes $[A_{i,1}, \ldots, A_{i,l_i}]$.

We write *sch*(R) to refer to the attributes of a relation symbol R and $R[A, B, C]$ to denote a relation symbol R with attributes $[A, B, C]$.

Definition 2.0.2 (database)**.** A database D over a schema Σ associates with each relation symbol R_i in Σ a relation R_i^D.

Elements of a relation are called *tuples*. Given a tuple t, we use the notation $t.A$ to reference the value of the attribute A of t and $t.\bar{A}$ to reference the values of the set $\bar{A}$ of attributes of t.

Definition 2.0.3 (uncertain database)**.** An uncertain database $\mathcal{D}$ over a schema Σ is a finite set of databases over Σ. Each database $D \in \mathcal{D}$ is called a *possible world*. Uncertain databases are also called *world sets* because they are sets of possible worlds.

A *probabilistic database* is an uncertain database where each possible world is assigned a probability. We call a formalism for representing uncertain databases a *representation system*.

Definition 2.0.4 (complete representation system)**.** A representation system for uncertain databases is complete if it can represent every uncertain database.

Relational algebra is a variable-free query language over a schema, consisting of the operators select (σ), project (π), cross product ($\times$), rename (ρ), union ($\cup$) and set difference ($\setminus$). *Positive relational algebra* is relational algebra without set difference.

SQL is a data manipulation and query language based on relational algebra. We use it additionally to relational algebra because it allows us to define functions and custom data types.

We use formulas over Boolean variables and over variables with finite domains. In the case of Boolean variables an *atom* is just a variable, otherwise an atom is an equality between a variable v and a constant in the domain of v. A *literal* is an atom or a negated atom. A *term* is a conjunction of literals and a formula in *disjunctive normal form* (DNF) is a disjunction of terms. A *clause* is a disjunction of literals and a formula in

conjunctive normal form (CNF) is a conjunction of clauses. For variables we use letters at the end of the alphabet, whereas for constants letters at the beginning of the alphabet.

We assume that the reader is familiar with the basic concepts of complexity theory and refer to [10, 37, 22, 24] for further details. We just recall two complexity classes which occur later in this work. NP is the class of problems that can be decided in polynomial time by a nondeterministic Turing machine. NP-hard problems are considered to be intractable which means that presumably there does not exist any efficient algorithm to solve them. Σ_2^P is the class of decision problems solvable in nondeterministic polynomial time with access to a coNP-oracle and part of the polynomial hierarchy. Σ_2^P contains NP and is supposed to be different from NP. The intractability of problems in Σ_2^P follows from the intractability of problems in NP.

Given a function $f : X \rightarrow Y$ we call X the *domain* of f and Y the *codomain* of f. The *image* of f is a set $Y' \subseteq Y$ such that $Y' = \{f(x) \mid x \in X\}$. A function $f : X \rightarrow Y$ is *surjective* if for each $y \in Y$ there exists an $x \in X$ such that $f(x) = y$. If $f : X \rightarrow Y$ is surjective then $|X| \geq |Y|$. Given a function f we write f^{-1} to denote its *inverse*. The inverse of a function is not necessarily unique. In such a case we mean with $f^{-1}(y)$ the set of elements x for which $f(x) = y$.

When using an unordered structure like for example a set we sometimes assume an arbitrary order on it to be able to address the elements of the structure. Given an ordered structure A, we write $A[i]$ to refer to the i-th element of A. Note that we use zero-based indexes.

3. U-relations

The simplest representative of uncertain data in a relational database are Codd's "*null*" values [14] which are in use in SQL. Imieliński and Lipski [26] showed that, when using null values, only selection and projection are supported in a semantically meaningful way. They proposed conditional tables (c-tables) to represent uncertain data such that all operators of relational algebra are supported in a semantically meaningful way. In c-tables each tuple is annotated with a formula over equalities $x = c$ between variables and constants, using $\wedge$, $\vee$ and $\neg$. The formula describes in which of the possible worlds the tuple actually exists. While c-tables are a great theoretical concept they have less practical value, because they allow formulas with negation which are hard to evaluate.

According to [6], a representation system for uncertain databases should satisfy the following needs:

- It should be complete, i.e. it should be able to represent every uncertain database.
- It should be succinct, i.e. large sets of possible worlds should be representable using only little space.
- It should allow for efficient query evaluation.

Antova, Jansen, Koch and Olteanu introduced U-relations [6] as a succinct representation system for uncertain databases. U-relations are restricted c-tables [30] and mainly satisfy the described needs. They are a complete representation system for uncertain databases and more succinct than other representation systems. Due to the restrictions positive relational algebra queries, extended by an operator for possible answers, can be evaluated in polynomial time (data complexity). For positive relational algebra queries practically feasible algorithms have been shown, but not yet for set difference. In Chapter 4 we will close this gap.

U-relations use the closed world assumption [42]. This means that every fact which is not represented by a database is assumed to be false. World-set algebra [8] (WSA) is a query language for uncertain databases based on relational algebra. Additionally to the operators of relational algebra WSA provides operators to introduce uncertainty (*repair-key*) and to reduce uncertainty (*poss*). Its full power – it captures second-order logic over finite structures – is shown in [29]. The query language we use for U-relations is a subset of WSA.

First we introduce U-relations by giving an example, then the subsequent sections define U-relations formally and how the operators of positive relational algebra queries

W	Var	Val
	x	1
	x	2
	y	1
	y	2
	z	1
	z	2

U_P	D	Tid	Product
	$()$	1	Mozart-CD
	$()$	2	Key ring
	$(y = 1)$	3	Globe
	$(y = 2)$	3	Snow globe

U_C	D	Tid	Country
	$(x = 1)$	1	Austria
	$(x = 2)$	1	France
	$()$	2	Germany
	$()$	3	Switzerland

U_S	D	Tid	Stars
	$()$	1	⋆⋆⋆⋆⋆
	$()$	2	⋆⋆⋆
	$(z = 1)$	3	⋆⋆⋆⋆
	$(z = 2)$	3	⋆⋆⋆⋆⋆

Figure 3.1: A U-relational database representing product ratings.

can be evaluated on them. We also describe what probabilistic U-relations are. Compared to the original definition we sometimes use different notations that fit better for our purpose.

3.1 Running Example

On the Internet there are lots of sites where users can rate products. For example Amazon customers can write reviews and rate products by using one to five stars. Assume that we maintain a database to collect such user ratings. The ratings we get are often ambiguous, for example because of different rating schemes on different web sites, or because of uncertainty introduced by the information extraction tool we use. We want to store for each rating the product name, the country of origin of the reviewer and the rating itself. To do that we use an uncertain database of schema $\textit{Rating}[\textit{Product}, \textit{Country}, \textit{Stars}]$.

Suppose there are three user ratings we want to store in the database. The first one is about a CD by Mozart and gives five stars. The country of the reviewer could be either Austria or France because the website where the review was posted is Austrian but the language of the review French. To model this ambiguity we introduce a binary variable x. The valuation $x = 1$ means we believe that Austria is the correct country and $x = 2$ means we believe that France is the correct country. We use three U-relations to represent the three fields of the relation *Rating*. They vertically partition the relation. Figure 3.1 shows how this rating is encoded in the three U-relations U_P (for the product name), U_C (for the country) and U_S (for the star rating). The D columns (descriptor columns) restrict in which cases we believe an attribute to be the true one and the *Tid*

columns (tuple ids) are used to join the vertical partitions.

The tuple with id 1 has the product name "Mozart-CD". This attribute is certain, hence the empty D column. The two possibilities for the country are encoded by the first two rows of U_C, where the D column differentiates between the two valuations of the variable x. The rating with 5 stars is certain again, hence the empty D column in U_S for the tuple with id 1. Beside this first rating the database in Figure 3.1 holds two more ratings. A certain one with id 2 which is about a key ring in form of the St. Stephen's Cathedral which gets 3 stars by a German reviewer. The third rating is by a Swiss tourist. The data extraction tool we use cannot determine whether it is about a globe or a snow globe. We use the new variable y to distinguish these two cases. $y = 1$ means that the third rating is about a globe, $y = 2$ that it is about a snow globe. The rating by the Swiss is "good", which could be 4 or 5 stars in our system. The variable z distinguishes the two cases, $z = 1$ meaning 4 stars, $z = 2$ meaning 5 stars. Altogether this gives 4 possibilities for the third rating.

The relation W is the so-called world table and defines the possible values the variables can have. The three variables all have the domain $\{1, 2\}$, hence we have $2*2*2 = 8$ possible worlds in total. We will use the described U-relational database as a running example in this work.

3.2 Definition of U-relations

In this section we formally define U-relations and their semantics. We use a set of variables over finite integer domains to define the set of possible worlds. The variables and domains are represented by a *world-table*.

Definition 3.2.1 (world table)**.** A world table is a relation of schema $W[\mathit{Var}, \mathit{Val}]$. It defines variables (attribute *Var*) and their domains (attribute *Val*).

Definition 3.2.2 (variables defined by a world table)**.** Given a world table W, $\mathit{vars}(W) = \{x \mid (x, y) \in W\}$ is the set of variables defined by W.

Definition 3.2.3 (domain of a variable)**.** Given a world-table W and a variable x defined by W, $\mathit{dom}_W(x) = \{y \mid (x, y) \in W\}$ is the set of domain values of x.

Note that by the context it is mostly clear which world table is meant. In these cases we just write $\mathit{dom}(x)$ instead of $\mathit{dom}_W(x)$. Given a world-table W and a variable x in W, we use the function $\mathit{max}(x) = \mathit{max}(\mathit{dom}(x))$ to refer to the maximum domain value of x.

Definition 3.2.4 (normalized world-table)**.** A world table W is normalized if the domains of all variables defined by W are consecutive natural numbers starting with 1, i.e. $\forall x \in \mathit{vars}(W) : \mathit{dom}(x) = \{1, \ldots, max(x)\}$.

From now on we consider only normalized world tables that define variables with a domain size ≥ 2. This does not restrict expressiveness.

Given a world table W each total valuation of variables defined in W identifies a possible world. It follows that the total number of possible worlds is $\Pi_{x \in vars(W)} max(x)$.

Definition 3.2.5 (ws-descriptor)**.** Given a world table W, a world-set descriptor (ws-descriptor) over W is a (not necessarily total) valuation of variables defined by W.

The name refers to the semantics and will become clear in the next section. If a ws-descriptor is a total valuation then we call it a *total ws-descriptor*. In case it is clear which world table (that defines the variables and domains) is meant we omit it and speak just of ws-descriptors. We use the logical "*and*" symbol in our notation as a separator, and to denote valuations we use the "*equality*" symbol. For example we write $(x = 1 \wedge y = 2 \wedge z = 5)$. Given a ws-descriptor d we write $|d|$ do denote the number of equalities d consists of (its *length*). We assume an arbitrary order on the equalities in a ws-descriptor d, to be able to address them. We write $d[i].var$ to refer to the variable on the left hand side of its i-th equality and $d[i].val$ to refer to the constant on the right hand side of its i-th equality. Given a ws-descriptor d, we use $vars(d)$ to refer to the set of variables occurring on the left hand side of the equalities in d.

Definition 3.2.6 (consistent ws-descriptors)**.** Two ws-descriptors are *consistent* if they do not assign different values to the same variable.

For example $(x = 1 \wedge y = 2)$ and $(x = 1 \wedge z = 3)$ are consistent, whereas $(x = 1 \wedge y = 2)$ and $(x = 2)$ are inconsistent.

Definition 3.2.7 (ws-set)**.** A ws-set over a world table W is a set of ws-descriptors over W.

Definition 3.2.8 (consistent ws-sets)**.** Two ws-sets S_1 and S_2 are consistent if there exist ws-descriptors $d_1 \in S_1, d_2 \in S_2$ such that d_1 and d_2 are consistent.

We write $vars(S)$ to denote the set of variables occurring in the ws-descriptors of a ws-set S. Similar to ws-descriptors we assume an arbitrary order on ws-sets so that we can address their elements by index. Given a ws-set S we write $S[i]$ to denote the i-th ws-descriptor in S.

Following Antova et al. [6] we define U-relations.

Definition 3.2.9 (U-relational database)**.** A U-relational database over schema $\Sigma = (R_1[\overline{A}_1], \ldots, R_k[\overline{A}_k])$ is a tuple $(U_{1,1}, \ldots, U_{1,m_1}, \ldots, U_{k,1}, \ldots, U_{k,m_k}, W)$, where W is a world-table and each relation $U_{i,j}$ has schema $U_{i,j}[D_{i,j}; \overline{T}_{R_i}; \overline{B}_{i,j}]$ such that $D_{i,j}$ defines ws-descriptors over W, $\overline{T}_{R_i}$ defines tuple ids (positive integers) and $\overline{B}_{i,1} \cup \cdots \cup \overline{B}_{i,m_i} = \overline{A}_i$.

We call the relations $U_{1,1}, \ldots, U_{k,m_k}$ that represent the actual relations $R_1, \ldots, R_k$ *U-relations*. The *data fields* $\overline{B}_{i,j}$ are the attributes of a U-relation that represent the actual data, whereas the ws-descriptors and the tuple ids are meta-data. U-relations partition the actual relations R_i vertically and the tuple ids are used to reconstruct the entire relations. The vertical partitions are used to concisely represent uncertainty at the attribute-level. Different attributes of the same tuple can independently have different values. Uncertainty at the attribute-level allows a more succinct representation than uncertainty at the tuple-level.

Example 3.2.10. To fill the presented terms with more life we relate them to our running example (Figure 3.1). It represents a U-relational database over a schema consisting of one relation symbol *Rating*, hence $k = 1$ in the example. The three U-relations U_P, U_C and U_S represent the actual relation *Rating*, hence $m_1 = 3$. The data fields are the attributes *Product*, *Country* and *Stars*.

The same attribute can occur in more than one of the vertical partitions of a U-relational database. This overlap can be useful to speed up querying. However, in this thesis we assume that there is no overlap, i.e. $\overline{B}_{i,j} \cap \overline{B}_{i,l} = \emptyset$ for $1 \leq i \leq k$ and $1 \leq j, l \leq m_i, j \neq l$.

We make one difference to the original definition by Antova et al. in that we use an appropriate data type *ws-descriptor* to encode ws-descriptors into a single column of the U-relations $U_{i,j}$. Antova et al. encode ws-descriptors into several columns such that each column represents one variable/value combination. This difference is necessary to be able to carry out difference and update operations, as we will see later in Chapter 4.

Semantics

Consider a U-relational database $U = (U_{1,1}, \ldots, U_{1,m_1}, \ldots, U_{k,1}, \ldots, U_{k,m_k}, W)$ over schema $\Sigma = (R_1[\overline{A}_1], \ldots, R_k[\overline{A}_k])$. Its world table W defines the set of variables and their domains. Each total valuation of the variables identifies a possible world.

The function ω defines the meaning of ws-descriptors.

Definition 3.2.11 (ω for ws-descriptors)**.** Given a ws-descriptor d over a world table W, $\omega(d)$ is the set of total valuations one gets by extending d to total valuations over the variables defined in W.

For example consider the ws-descriptor $d = (x = 1 \wedge y = 3)$ and assume that the world table defines three variables x, y and z such that $dom(x) = dom(y) = dom(z) = \{1, 2, 3\}$. Then $\omega(d)$ is the set of valuations where $x = 1$, $y = 3$ and z is either 1, 2 or 3.

Total valuations identify possible worlds. Thus we can say that ws-descriptors describe sets of worlds, which finally explains the term *world-set descriptor*. The empty

Rating	Product	Country	Stars
	Mozart-CD	Austria	⋆⋆⋆⋆⋆
	Key ring	Germany	⋆⋆⋆
	Snow globe	Switzerland	⋆⋆⋆⋆

Figure 3.2: A possible world of the product ratings database.

ws-descriptor () describes the set of all worlds. In contrast total ws-descriptors describe exactly one world. If and only if two ws-descriptors d_1 and d_2 are consistent, $\omega(d_1) \cap \omega(d_2) \neq \emptyset$.

We define the relations for each possible world separately. Consider an arbitrary world identified by a total valuation w. For each relation R_i ($1 \leq i \leq k$) we consider all the vertical partitions $U_{i,j}$ ($1 \leq j \leq m_i$). Let $sch(U_{i,j}) = [D, \bar{T}, \bar{A}]$. We consider all tuples in $U_{i,j}$. Let $(d, \bar{t}, \bar{a})$ be a tuple in $U_{i,j}$. d is the ws-descriptor of the tuple, $\bar{t}$ are the tuple ids and $\bar{a}$ the data values. If $w \in \omega(d)$ then we insert the values $\bar{a}$ into the $\bar{A}$-attributes of the tuple with identifier $\bar{t}$ in relation R_i.

In general, some tuples in R_i may be left partial, i.e. there are tuples where not all attributes are defined. These tuples are removed from R_i. We consider only U-relational databases where no tuple is left partial. Antova et al. call such U-relational databases *reduced*.

Example 3.2.12. We return to the running example about product ratings (see Figure 3.1). With the total valuation $w = (x = 1 \wedge y = 2 \wedge z = 1)$ we choose a possible world. Let us see what tuples are part of the relation *Rating* in this world. We consider first the vertical partition U_P. It holds that $w \in omega(())$, therefore “Mozart-CD” is the *Product* attribute of a tuple with id 1, and “Key ring” is the *Product* attribute of a tuple with id 2. We have that $w \notin \omega((y = 1))$, hence we skip the “Globe”. In contrast, $w \in \omega((y = 2))$ and so “Snow globe” is the *Product* attribute of a tuple with id 3. We do the same with the tuples in the vertical partitions U_C and U_S, which fills for each of the tuples the *Country* and the *Stars* attribute. Figure 3.2 pictures the result, i.e. the relation *Rating* in the possible world identified by w.

The U-relational database of the example would not be reduced, if – for example – the vertical partition U_C did not contain the tuple where the *Country* attribute is “Austria”. Then the *Country* attribute of the tuple with id 1 would not be defined, while the other two attributes would be defined. It follows that the tuple would be left partial in the world identified by w, and thus the U-relational database would not be reduced. Note that a U-relational database can still be reduced, if none of the attributes of a tuple is defined in some world. This just means that this tuple does not exist in that world.

The following definition expresses that valid U-relational databases do not provide contradictory values for the same tuple attribute in the same world.

W	Var	Val	Prob
	x	1	0.4
	x	2	0.5
	x	3	0.1
	y	1	0.2
	y	2	0.8

R	D	T	A
	$(x = 2 \wedge y = 2)$	1	a_1
	$(x = 3)$	2	a_2
	$(y = 1)$	2	a_2

Figure 3.3: A probabilistic U-relational database.

Definition 3.2.13 (valid U-relational database)**.** A U-relational database $(U_1, \ldots, U_n, W)$ is called valid if for all U-relations $U_i[D_i, \overline{T}, \overline{A}_i]$ and $U_j[D_j, \overline{T}, \overline{A}_j]$ $(1 \leq i, j \leq n)$ that are vertical partitions of the same relation the following holds. For all tuples $t_1 \in U_i$ and for all tuples $t_2 \in U_j$ it holds that if $t_1.\overline{T} = t_2.\overline{T}$ and $t_1.D_i$ is consistent with $t_2.D_j$, then for all $A \in (\overline{A}_i \cap \overline{A}_j), t_1.A = t_2.A$.

Note that this includes the case $i = j$. If $i \neq j$ then $\overline{A}_i \cap \overline{A}_j = \emptyset$ because we assumed that there is no overlap. We consider only valid U-relational databases.

We write *rep*(U) to refer to the set of partial relations a U-relation U represents, and we write *rep*$($*UDB*$)$ to refer to the uncertain database a U-relational database *UDB* represents.

Note that a U-relation can contain tuples t_1 and t_2 that have equivalent tuple ids and data values but different ws-descriptors, i.e. $t_1 = (d_1, \bar{t}, \overline{a})$ and $t_2 = (d_2, \bar{t}, \overline{a})$ and $d_1 \neq d_2$. This means that the values $\overline{a}$ are part of the worlds described by d_1 or d_2. To find out in which worlds a data value $\overline{a}$ exists we have to consider all ws-descriptors of tuples that have the same data value $\overline{a}$. This is why we need to deal with sets of ws-descriptors, ws-sets. The semantics of ws-sets follows from the semantics of U-relations. We extend the function ω to ws-sets.

Definition 3.2.14 (ω for ws-sets)**.** Given a ws-set S over a world table W, $\omega(S) = \bigcup_{d \in S} \omega(d)$.

If and only if two ws-sets S_1 and S_2 are consistent, $\omega(S_1) \cap \omega(S_2) \neq \emptyset$. A ws-set S over a world table W can be seen as a formula ϕ in DNF over the variables defined in W, where the atoms are equalities between variables and constants from their domains. Then S describes exactly the worlds that are identified by total valuations which satisfy ϕ. For example, consider the ws-set $\{(x = 1 \wedge y = 1), (z = 1)\}$. It corresponds to the formula $(x = 1 \wedge y = 1) \vee z = 1$.

3.3 Probabilistic U-relations

A probabilistic database is an uncertain database where each possible world is assigned a probability, such that the sum of probabilities over all worlds is 1. In U-relations, we add to each domain value of a variable a probability value. Then the world-table has the schema $W[\mathit{Var}, \mathit{Val}, \mathit{Prob}]$. Figure 3.3 gives an example of a probabilistic U-relational database. Considering the world table, the worlds described by the ws-descriptor $(x = 2 \wedge y = 2)$ of the first tuple in R have the probability $0.5 * 0.8 = 0.4$. The probability of a world described by a total ws-descriptor d is the product of the probabilities assigned to the valuations in d as defined in W. For each variable/value pair the probability has to be unique, i.e. $(\mathit{Var}, \mathit{Val})$ is a key in W, and the sum of probability values for the different valuations of a variable has to be 1. It follows that the sum of probabilities over all worlds is 1, as defined.

When using probabilistic databases we are interested in the probabilities of tuples, i.e. the probability that a tuple is actually part of the relation. For example, we want to know the probability that there is a tuple with $A = a_2$ (see Figure 3.3). a_2 occurs twice, therefore we need to compute the probability of the worlds described by the ws-set $\{(x = 3), (y = 1)\}$. The two ws-descriptors in the ws-set are independent, hence the probability for a_2 is $1 - (1 - 0.1) \times (1 - 0.2) = 0.28$. The problem of computing the exact probability of a ws-set is #P-hard [16]. In [30] an algorithm is presented to compute the exact probability of a ws-set. There also exists an approximation algorithm for probability computation of ws-sets [36].

3.4 Query Language

The query language we employ consists of the operators of relational algebra, the possible operator (*poss*) and the *merge* operator. The merge operator is used to reconstruct relations from the vertical partitions which U-relations are. It works by joining two partitions on the common tuple id attributes and selecting only those combinations that have consistent ws-descriptors, i.e. that actually exist in some world. The possible operator closes the possible worlds semantics and returns the tuples that are part of any world.

Conceptually we want to evaluate a query Q on each possible world of an uncertain database. This is infeasible in practice because of the enormous number of possible worlds. Therefore we translate Q into a query $\hat{Q}$ which we can evaluate on the U-relational encoding of the uncertain database, such that the result of $\hat{Q}$ represents the result of Q. This is pictured in Figure 3.4. We start with a U-relational database U and a query Q. U represents an uncertain database D. By applying Q to each of the worlds in D individually we get an uncertain database D'. We want that the translated query $\hat{Q}$ produces a U-relational database U' that represents exactly D'.

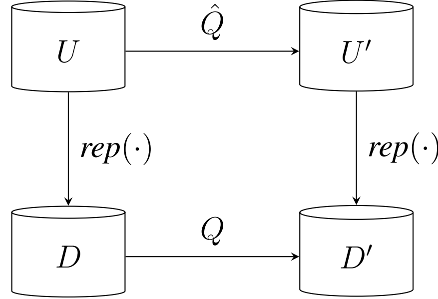

Figure 3.4: Concept of query evaluation on the representation level.

Figure 3.5 shows the function $[\![\cdot]\!]$ that translates positive relational algebra queries with *poss* and *merge* operators into SQL queries on U-relations. Compared to the original definition (in [6]) we translate to SQL because it supports user-defined data types. Instead of giving a translation for the join operator we just give a translation for the cross product because the join operator is equivalent to a cross product followed by a selection. Q_1 and Q_2 are subqueries whose results are represented by the U-relations U_1 and U_2. The D_1 respectively D_2 column holds ws-descriptors and describes the set of worlds in which a tuple is present. The $\overline{T}_1$ respectively $\overline{T}_2$ columns hold the tuple ids and the $\overline{A}_1$ respectively $\overline{A}_2$ columns the actual data values. The condition α is used by the merge operator and the condition ψ by all operators that join tuples (cross product and merge).

Selection works as on a certain database, with the restriction that the select predicate must not refer to the ws-descriptor nor to the tuple id. The *projection* operator keeps the ws-descriptor and the id of a tuple additionally to the selected attributes. *Renaming* is straightforward, with the constraint that only the data attributes are renamed. The *union* operator adds "empty" columns $\bar{T}_2$ to U_1 and $\bar{T}_1$ to U_2 to bring U_1 and U_2 to the same schema. The *poss* operator projects out the ws-descriptors and the tuple ids, and keeps only the data values with an empty ws-descriptor (which represents all worlds). SQL essentially works on multisets, so we use the *distinct* keyword to keep only pairwise different tuples. The *cross product* uses the function $\psi(\cdot)$ to filter out combinations of tuples that do not exist in any world by checking the consistency of the two ws-descriptors. If the two ws-descriptors are consistent then they are combined by the function *concat*$(\cdot)$ that appends one ws-descriptor to another. Given two ws-descriptors d_1 and d_2, $\omega(\mathit{concat}(d_1, d_2)) = \omega(d_1) \cap \omega(d_2)$. The condition α guarantees that only tuples which have the same tuple id are joined by the *merge* operator, while ψ filters again the consistent combinations. Note that union, cross product and merge rely on the following equalities (S_1, S_2 are ws-sets):

$$\omega(S_1) \cup \omega(S_2) = \omega(S_1 \cup S_2)$$
$$\omega(S_1) \cap \omega(S_2) = \omega(\{\mathit{concat}(d_1, d_2) \mid d_1 \in S_1, d_2 \in S_2, \psi(d_1, d_2)\})$$

From the translation $[\![\cdot]\!]$ if follows that

Theorem 3.4.1 (see [6]). *Positive relational algebra queries extended with poss and merge can be evaluated on U-relational databases in polynomial time data complexity.*

In Section 4.4 we will complete the translation of relational algebra for U-relations and explain how the set difference operator can be translated.

Examples

Consider again the U-relational database of Figure 3.1. In Figure 3.6 two queries on this database are presented. Query Q_1 asks for all information about ratings from Swiss persons. The selection can be pushed inside and the query rewritten into the equivalent query Q_1', to reduce the number of tuples in the join. The result U_{Q_1} contains all possible four combinations of the tuple with id 3 which is selected by the condition *Country* $=$ *Switzerland*. Due to the vertical partitioning the database does not have to explicitly store all possible combinations. They are generated only when needed.

Query Q_2 is a Boolean query that asks whether there are any ratings from Switzerland. It does not need the vertical partitions U_P and U_S and therefore they do not have to be merged, which gives a succinct result relation U_{Q_2}.

Linear Time Consistency Check

The naive implementation of the consistency check ψ (see Figure 3.5) needs quadratic time to decide whether the intersection $\omega(d_1) \cap \omega(d_2)$ of two ws-descriptors d_1 and d_2 is empty or not. By keeping the equalities of a ws-descriptor sorted by variable name (using an arbitrary but fixed order) we can check consistency of two ws-descriptors in linear time, as shown by Algorithm 1. The equalities of the two ws-descriptors are iterated quasi in parallel.

It remains to keep the ws-descriptors sorted. To achieve that the *concat* function must not append one ws-descriptor to the other, but merge them. This needs linear time, as the plain concatenation.

$$
\begin{aligned}
\text{Let } \overline{A}_1 &:= \{a_1, \ldots, a_n\}, \\
U_1 &:= [\![Q_1]\!] \text{ with schema } [D_1, \overline{T}_1, \overline{A}_1], \\
U_2 &:= [\![Q_2]\!] \text{ with schema } [D_2, \overline{T}_2, \overline{A}_2], \\
\alpha &:= \bigwedge_{T \in \overline{T}_1 \cap \overline{T}_2} (U_1.T = U_2.T), \\
\psi(d_1, d_2) &:= \bigwedge_{i<|d_1|, j<|d_2|} (d_1[i].\mathit{var} = d_2[j].\mathit{var} \rightarrow d_1[i].\mathit{val} = d_2[j].\mathit{val}), \\
\mathit{concat}(d_1, d_2) &:= (d_1 \wedge d_2).
\end{aligned}
$$

$$
\begin{aligned}
[\![\sigma_\phi(Q_1)]\!] &:= \textbf{select } * \textbf{ from } U_1 \textbf{ where } \phi; \\
&\quad \text{Constraint: } \phi \text{ is a condition on } A_1. \\
[\![\pi_{\bar{X}}(Q_1)]\!] &:= \textbf{select } D_1, \bar{T}_1, \bar{X} \textbf{ from } U_1; \\
&\quad \text{Constraint: } \bar{X} \subseteq \bar{A}_1. \\
[\![\rho_{b \leftarrow a_k}(Q_1)]\!] &:= \textbf{select } D_1, \bar{T}_1, a_1, \ldots, a_{k-1}, a_k \textbf{ as } b, a_{k+1}, \ldots, a_n \textbf{ from } U_1; \\
[\![Q_1 \cup Q_2]\!] &:= \textbf{select } D_1, \bar{T}_1, \bar{0} \textbf{ as } \bar{T}_2, \bar{A}_1 \textbf{ from } U_1 \textbf{ union} \\
&\qquad \textbf{select } D_2, \bar{0} \textbf{ as } \bar{T}_1, \bar{T}_2, \bar{A}_2 \textbf{ from } U_2; \\
&\quad \text{Constraint: } \bar{A}_1 = \bar{A}_2, \bar{T}_1 \cap \bar{T}_2 = \emptyset. \\
[\![\mathit{poss}(Q_1)]\!] &:= \textbf{select distinct } (), \bar{A}_1 \textbf{ from } U_1; \\
[\![Q_1 \times Q_2]\!] &:= \textbf{select } \mathit{concat}(D_1, D_2), \overline{T}_1, \overline{T}_2, \overline{A}_1, \overline{A}_2 \textbf{ from } U_1, U_2 \\
&\qquad\qquad \textbf{where } \psi(D_1, D_2); \\
&\quad \text{Constraint: } \overline{T}_1 \cap \overline{T}_2 = \emptyset. \\
[\![\mathit{merge}(Q_1, Q_2)]\!] &:= \textbf{select } \mathit{concat}(D_1, D_2), \overline{T}_1 \cup \overline{T}_2, \overline{A}_1, \overline{A}_2 \textbf{ from } U_1, U_2 \\
&\qquad\qquad \textbf{where } \alpha \textbf{ and } \psi(D_1, D_2);
\end{aligned}
$$

Figure 3.5: Translation of positive relational algebra queries with *poss* and *merge* into queries on U-relations (based on [6]).

$Q_1: \sigma_{Country=Switzerland}(merge(U_P, merge(U_C, U_S)))$
$Q'_1: merge(U_P, merge(\sigma_{Country=Switzerland}(U_C), U_S))$

U_{Q_1}	D	Tid	Product	Country	Stars
	$(y = 1 \wedge z = 1)$	3	Globe	Switzerland	⋆⋆⋆⋆
	$(y = 1 \wedge z = 2)$	3	Globe	Switzerland	⋆⋆⋆⋆⋆
	$(y = 2 \wedge z = 1)$	3	Snow globe	Switzerland	⋆⋆⋆⋆
	$(y = 2 \wedge z = 2)$	3	Snow globe	Switzerland	⋆⋆⋆⋆⋆

$Q_2: \pi_{true}(\sigma_{Country=Switzerland}(U_C))$

U_{Q_2}	D	Tid	
	()	3	true

Figure 3.6: Exemplary queries and their results.

Algorithm 1: CONSISTENT(d_1, d_2)

Require: sorted ws-descriptors d_1, d_2
1: $i = j = 0$
2: **while** $i < |d_1|$ and $j < |d_2|$ **do**
3: **if** $d_1[i].var = d_2[j].var$ and $d_1[i].val \neq d_2[j].val$ **then**
4: **return** false
5: **end if**
6: **if** $d_1[i].var < d_2[j].var$ **then**
7: i++
8: **else**
9: j++
10: **end if**
11: **end while**
12: **return** true

4. Updates and Set Difference on U-relations

We consider updates that manipulate existing worlds. Other forms of updates introduce new worlds or remove possible worlds. The repair-key operator introduces new worlds to repair violated key constraints. In [38] U-relations are proposed for collaborative data management. Instead of updating data items new versions of the data items are inserted. A rating system allows users to democratically define the best version of a data item, which is then used in query answering. In [30] the problem of conditioning a probabilistic database is investigated. Conditioning an uncertain database means to delete worlds that do not satisfy some constraint.

In this chapter we will show how update and delete statements that manipulate existing worlds can be performed on U-relations. We show that positive relational algebra is not sufficient to express arbitrary updates. We use set difference to model updates and explain how set difference can be computed on U-relations. The main ingredient is the computation of inverses of ws-sets, for which we propose two methods: decompression and negation. By providing support for set difference we support full relational algebra on U-relations and close U-relations under the application of relational algebra queries.

4.1 Motivation

We consider again the example about product ratings as introduced in Section 3.1. Assume we found out that there are only snow globes and no globes, i.e. every globe is in fact a snow globe. Therefore we issue the following update:

```
update Rating set Product='Snow globe'
where Product='Globe';
```

This update can be directly executed on the U-relation U_P, because none of the other vertical partitions is involved. But what if one wants to update the stars given by a reviewer based on the reviewed product? For example

```
update Rating set Stars=⋆ where Product='Globe';
```

Due to the vertical partitioning the *Product* attribute and the *Stars* attribute are not in the same U-relation. The where condition on the Product attribute holds for the tuple with id 3 in the worlds described by $(y = 1)$. But in U_S the tuple with id 3 differentiates only between the domain values of the variable z. To encode the result of the update

U_S	D	Tid	Stars
	()	1	⋆⋆⋆⋆⋆
	()	2	⋆⋆⋆
	$(y = 1 \wedge z = 1)$	3	⋆
	$(y = 2 \wedge z = 1)$	3	⋆⋆⋆⋆
	$(y = 1 \wedge z = 2)$	3	⋆
	$(y = 2 \wedge z = 2)$	3	⋆⋆⋆⋆⋆

Figure 4.1: Updated U-relation which had to be decompressed.

the representation of this tuple has to be decompressed such that it differentiates also between the values of y. Figure 4.1 shows a possible representation of the result of the update. In it all four combinations of valuations of the variables y and z are considered, which we call a full decompression. Actually it would not be necessary because the two cases where $Stars = \star$ can be shortly described by the ws-descriptor $(y = 1)$ and can be merged. In practice we want to avoid full decompression. Just imagine an update that involves 10 variables with each of them having a domain of size 10. Altogether there are 10^{10} combinations which we do not want to enumerate. The necessity for decompression is shortly mentioned in [7], but not worked out in depth.

The questions are: How can we model updates on U-relations? Can we avoid full decompression?

4.2 Updates

To find out how updates can be computed on U-relations we look at what SQL updates essentially are. An update is in fact a special command that runs a query and replaces parts of the original relation. We can express the result of every SQL update by a query. Consider the following update:

Δ: **update** R **set** B=*Expr* **where** *Cond*;

We assume that *sch*$(R) = [\bar{A}, B]$ and that Δ does not contain subqueries. We can translate Δ into a query Q that is the union of the updated tuples and the non-updated tuples:

Q: **select** $\bar{A}$, *Expr* **as** B **from** R **where** *Cond*
union
select $*$ **from** R **where not** *Cond*;

Obviously, the result of Q is exactly the table R updated by Δ.

Now assume that R is an uncertain relation and that R is represented by U-relations $U_1, \ldots, U_n$. Let the updated attribute B be part of the vertical partition U_i. If the

```
GU: update R_1 set B=Expr from R_2, ..., R_n
    where JoinCond and Cond;

Q_GU: select Ā, Expr as B from R_1, ..., R_n
      where JoinCond and Cond
    union
      select R_1.* from R_1
      except
      select R_1.* from R_1, ..., R_n
      where JoinCond and Cond;
```

Figure 4.2: Expressing updates by queries.

expression *Expr* and the condition *Cond* only refer to attributes of U_i then the update can be directly executed on the partition U_i. This is the case in the first example of the previous section. But if any attribute of another vertical partition U_j $(j \neq i)$ is used in *Expr* or *Cond*, this simple method is not possible anymore. To decide which attributes of U_i should be updated we need to somehow join U_i with U_j along the tuple ids.

As we already need to join tables let us consider a more general form of updates, which is also practically relevant. In practice it is often necessary to update a table based on information in another table. For example we have a product relation and want to lower the price of a product if there are bad reviews. We consider the general case where we update an attribute B of a relation $R_1(\bar{A}, B)$ and allow to join R_1 with other relations $R_2, \ldots, R_n$. Query *GU* (like General Update) in Figure 4.2 shows how such a general update looks like in the syntax of PostgreSQL. Other relational database management systems support similar constructs. The join condition *JoinCond* describes how the relations $R_1, \ldots, R_n$ are joined together and the condition *Cond* selects the tuples that should be updated. *Expr* and *Cond* can refer to any attribute in the relations $R_1, \ldots, R_n$. It is important that the update is unambiguous, i.e. if a tuple in R_1 has more than one join partner in $R_2, \ldots, R_n$ then *Expr* must return the same value for all join partners.

Theorem 4.2.1. *Consider the update statement GU defined as follows:*

```
GU: update R_1 set B=Expr from R_2, ..., R_n
    where JoinCond and Cond;
```

Positive relational algebra is not sufficient to compute the result of GU on U-relational databases.

Proof. We provide a simple counterexample. Consider the U-relational database $\mathbf{U} = (U_1, U_2, W)$ as given in Figure 4.3 and the following update Δ:

```
Δ: update U_1 set A=a_2 from U_2 where B=b_1;
```

W	Var	Val
	x	1
	x	2

U_1	D	Tid	A
	()	1	a_1

U_2	D	Tid	B
	$(x = 1)$	1	b_1

U_1'	D	Tid	a
	$(x = 1)$	1	a_2
	$(x = 2)$	1	a_1

Figure 4.3: Positive relational algebra cannot model all updates - a counterexample.

A correct representation of the result of Δ would be the U-relation U_1' as shown in Figure 4.3. It distinguishes between $x = 1$ and $x = 2$. Obviously, any correct representation of the result of Δ needs to distinguish between $x = 1$ and $x = 2$.

The equality $x = 2$ does neither occur in U_1 nor in U_2. As the translation $[\![\cdot]\!]$ shows, positive relational algebra queries on U-relations generate new ws-descriptors only by concatenating existing ws-descriptors. It follows that using only positive relational algebra it is not possible to get the valuation $x = 2$. Therefore positive relational algebra does not suffice to compute the result of the update Δ. □

Positive relational algebra does not suffice to model updates. But we can express the update *GU* as query Q_{GU}, using set difference (the keyword *except* in SQL). Again, we use the union of the updated tuples and the non-updated tuples, as shown in Figure 4.2. Instead of a plain selection with a negated condition *Cond* we use the except statement (set difference) to subtract from all the tuples in R_1 the tuples that are updated.

We have shown that arbitrary updates cannot be computed by using positive relational algebra only. But we can model arbitrary updates by using set difference. Before going into details on set difference we argue that our update language is comprehensive and have a look at a special case.

Expressiveness

The kind of updates that can be done with an update query following the syntax of *GU* (see Figure 4.2) includes update queries that use subqueries. Consider the following update query:

update R_1 **set** B=(**select** *Expr* **from** $R_2, \ldots, R_n$ **where** $Cond_1$) **where** $Cond_2$;

It is equivalent to the following update that does not need subqueries:

update R_1 **set** B=*Expr* **from** $R_2, \ldots, R_n$ **where** $Cond_1$ **and** $Cond_2$;

Similarly the syntax of *GU* includes update queries that use subqueries in the condition. For this consider the following update query:

```
update R_1 set B=Expr_1 where Cond_1 and
    C=(select Expr_2 from R_2,...,R_n where Cond_2);
```

It is equivalent to the following update that does not need subqueries:

```
update R_1 set B=Expr_1 from R_2,...,R_n where
    Cond_1 and Cond_2 and C=Expr_2;
```

In this way we can unnest subqueries in the expression that defines the value of an updated item, as well as we can unnest subqueries in the *where* condition. Obviously arbitrary levels of subqueries can be unnested. It follows that using *GU* we can model arbitrary updates with subqueries.

Special case: Semi-join updates

If every tuple in R_1 has exactly one join partner in $R_2, \ldots, R_n$ then we can express *GU* using a negated condition instead of set difference, as Q'_{GU} does:

```
Q'_GU:   select Ā, Expr as B from R_1,...,R_n
         where JoinCond and Cond
       union
         select R_1.* from R_1 from R_1,...,R_n
         where JoinCond and not Cond;
```

We call this kind of update the *semi-join update*. The precondition of each tuple in R_1 having exactly one join partner in $R_2, \ldots, R_n$ is fulfilled for instance by joins on attributes that fulfill a foreign key constraint. The precondition is necessary for the following reasons.

- If a tuple in R_1 does not have any join partner in $R_2, \ldots, R_n$ then it gets lost in the update, because it is neither part of the first operand for the union, nor part of the second operand).
- If a tuple in R_1 has more than one join partner in $R_2, \ldots, R_n$ then the condition *Cond* could hold for one join partner but not for another. So the tuple can possibly be updated and not updated at the same time, which is definitely not desired.

Using $[\![\cdot]\!]$ we can translate Q'_{GU} into a query on U-relations and perform the update. To apply semi-join updates to U-relations the precondition has to hold in every world, i.e. in every world a tuple in R_1 has to have exactly one join partner in $R_2, \ldots, R_n$. The advantage of semi-join updates on U-relations is that they do not need set difference. Their disadvantage is that all the tuples in the result relation get fully decompressed. As an example consider the following update query on our well-known product ratings database:

```
update Rating set Stars=⋆
where Product='Ski' and Country='Belgium';
```

U'_S	D	Tid	Stars
	$(x = 1)$	1	⋆⋆⋆⋆⋆
	$(x = 2)$	1	⋆⋆⋆⋆⋆
	$()$	2	⋆⋆⋆
	$(z = 1 \wedge y = 1)$	3	⋆⋆⋆⋆
	$(z = 2 \wedge y = 1)$	3	⋆⋆⋆⋆⋆
	$(z = 1 \wedge y = 2)$	3	⋆⋆⋆⋆
	$(z = 2 \wedge y = 2)$	3	⋆⋆⋆⋆⋆

Figure 4.4: The semi-join update fully decompresses the updated relation.

```
GD:  delete from R1 using R2, ..., Rn
     where JoinCond and Cond;

QGD: select R1.* from R1
     except
     select R1.* from R1, ..., Rn
     where JoinCond and Cond;
```

Figure 4.5: Expressing delete statements by queries.

Note that we assume an implicit merge (along the tuple ids) of the three partitions the relation *Rating* consists of. The query updates tuples in the partition U_S in case the conditions on the other two partitions U_P and U_C are satisfied. When using the semi-join update the result is a relation U'_S as shown in Figure 4.4. The update conditions apply nowhere and therefore the result U'_S represents the same information as U_S. Nevertheless, with a semi-join update all the tuples in U_S get decompressed and therefore U'_S consists of much more tuples than U_S. This can be avoided by using set difference for updates, as we will show in the next section.

4.3 Delete Statements

We consider the case that tuples are deleted in specific worlds. Delete statements are quite similar to update statements with the difference that the tuples satisfying the condition are not updated but deleted. Figure 4.5 shows how a general delete statement *GD*, that allows us to join other relations, is expressed as a query Q_{GD}. Using the function $[\![\cdot]\!]$ we can translate it into a query on U-relations.

But one subtlety remains: U-relational databases use vertical partitioning and we said that we want them to always be *reduced*. Consider an arbitrary relation R repre-

sented by U-relations $U_1, \ldots, U_n$. When deleting a tuple with a tuple id *tid* in the worlds described by a ws-set S in U_1, then we also have to delete the tuple *tid* in the worlds described by S in $U_2, \ldots, U_n$. Otherwise the U-relational database would not be reduced anymore.

We cannot apply Q_{GD} to one partition after another because the conditions *JoinCond* and *Cond* possibly refer to tuples that are being deleted. So we first select the tuples that should be deleted into a temporary table. Given a delete statement *GD* (as in Figure 4.5) we create a temporary table R_{del} as follows:

```
create table R_del as
select R_1.* from R_1, ..., R_n
where JoinCond and Cond;
```

Let U_{del} be the U-relation that represents the temporary table R_{del}. Given U_{del} we delete the respective tuples in every partition U_i of R_1 as follows:

```
select U_i.* from U_i
except
select U_i.* from merge(U_i, U_del);
```

We use *merge* to join the tuples in U_i with the tuples in U_{del} which are the tuples we want to delete. Using **except** we subtract them from the U-relation U_i. We have thus shown how to perform delete statements on U-relations.

4.4 Set Difference

We have explained how set difference can be used to perform arbitrary updates. Besides that set difference is valuable on its own, for example to select all products that have never been rated badly. Set difference introduces nonmonotonicity to relational algebra as the operators of positive relational algebra are all monotone. By defining set difference on U-relations we complete relational algebra on U-relations. For all we know, so far there have been only the following publications that consider set difference on uncertain databases. Wang et al. [50] describe how one level of not-exists subqueries can be supported on tuple-independent databases. This limited approach cannot be followed to do sequences of updates because their formalism is not closed under their definition of not-exists. In [20] Fink, Olteanu and Rath investigate the problem of evaluating unrestricted relational algebra queries (including set difference) on probabilistic U-relations (limited to Boolean variables). Their goal is to answer queries, i.e. to compute the probabilities of tuples in the query result. But the query results are not U-relations anymore. Instead of ws-descriptors they generate arbitrarily nested formulas using $\wedge$, $\vee$ and $\neg$. Following their definition of set difference U-relations are not closed under the application of relational algebra anymore. This does not satisfy our needs because for

updates we need a definition of set difference such that U-relations are closed under the application of relational algebra.

We show how set difference can be computed such that the result is again a U-relation. Thus we stay within the formalism. As explained in Section 3.4 we have to define set difference such that it behaves as if it was executed in every possible world separately. Given two relations R and S, a tuple t is in $R \setminus S$ if $t \in R$ and $t \notin S$. In U-relations ws-sets describe in which worlds a tuple is part of a relation. If a tuple t is part of a relation R in the worlds described by a ws-set W_1, and t is part of a relation S in the worlds described by a ws-set W_2, then t is part of $R \setminus S$ in the worlds identified by $\omega(W_1) \setminus \omega(W_2)$. How can we compute a ws-set W_3 such that $\omega(W_3) = \omega(W_1) \setminus \omega(W_2)$? Using the definition of $\omega(\cdot)$ for ws-sets we can reduce the problem to computing the difference between a ws-descriptor and a ws-set because

$$\omega(W_1) \setminus \omega(W_2) = \left(\bigcup_{d_1 \in W_1} \omega(d_1)\right) \setminus \omega(W_2) = \bigcup_{d_1 \in W_1} \left(\omega(d_1) \setminus \omega(W_2)\right)$$

We express the difference between a ws-descriptor d and a ws-set W using the inverse of W. To this end let us define the inverse of a ws-set. $\omega(())$ is the set of all valuations (which identify all worlds), hence $\omega(()) \setminus \omega(W)$ is the inverse of $\omega(W)$.

Definition 4.4.1 (inverse of a ws-set)**.** A ws-set W^i is the inverse of a ws-set W if and only if $\omega(W^i) = \omega(()) \setminus \omega(W)$.

We will define a function *negate*(W) in the next section that computes the inverse of a ws-set W. Using this function, it follows that

$$\omega(d) \setminus \omega(W) = \omega(d) \cap \omega(\textit{negate}(W)) \tag{4.1}$$

Union and intersection of ws-sets are done using union and concatenation (see Section 3.4). In Figure 4.6 we define the difference *diff* of a ws-descriptor and a ws-set. Recall that ψ (see Section 3.4) checks the consistency of two ws-descriptors. Thus it filters out inconsistent ws-descriptors which would describe only empty sets of worlds. From Equation 4.1 it follows that *diff* is correct, i.e.

$$\omega(\textit{diff}(d, W)) = \omega(d) \setminus \omega(W).$$

Now we are ready to define the translation of relational set difference, using *diff*. Figure 4.6 gives the details. For each tuple t in U_1 we select the ws-descriptors of tuples in U_2 with the same data values, using the subquery W. So W is the ws-set we have to subtract from the ws-descriptor $t.D_1$, which we do with *diff*. *diff* returns a set of ws-descriptors, let us call it W'. Each of the ws-descriptors in W' forms together with $\overline{T_1}$ and $\overline{A_1}$ a tuple in the result. We are aware that the SQL standard does not support functions that return sets, like *diff*. But there is no way in the SQL standard to "de-aggregate" the result of *diff*. The syntax we use is supported by PostgreSQL and we

$$\text{Let } U_1 := [\![Q_1]\!] \text{ with schema } [D_1, \overline{T_1}, \overline{A_1}],$$
$$U_2 := [\![Q_2]\!] \text{ with schema } [D_2, \overline{T_2}, \overline{A_2}],$$
$$\textit{diff}(d, W) := \{\textit{concat}(d, d') \mid d' \in \textit{negate}(W), \psi(d, d')\}.$$

$$[\![Q_1 \setminus Q_2]\!] := \textbf{select } \textit{diff}(D_1, W), \overline{T_1}, \overline{A_1} \textbf{ from } U_1;$$
$$\text{where } W := \textbf{select } D_2 \textbf{ from } U_2 \textbf{ where } \overline{A_2} = \overline{A_1};$$
$$\text{Constraint: } \overline{A_1} = \overline{A_2}.$$

Figure 4.6: Translation of set difference, based on the inverse of ws-sets.

think that this is a quite natural approach. Another approach would be to introduce a new data type *ws-set* and to use it instead of ws-descriptors. Note that the tuple ids of the second U-relation U_2 do not play a role in set difference.

4.5 Inverse of a Ws-set

In the previous sections we explained how to use the inverse of a ws-set to compute relational set difference and updates in a U-relational database. It remains to give an algorithm for computing the inverse of a ws-set. We propose two methods: full decompression and negation. In general, an exponential blowup cannot be avoid, as we will see. We prove that negation can give exponentially smaller results then decompression.

Full decompression

Consider that we want to compute the inverse of a ws-set S. The idea of full decompression is to generate all total ws-descriptors, relative to the set of variables that occur in S. Variables that do not occur in S do not have to be considered because they are not relevant for the inverse of S. The details are shown in Algorithm 2. Given the list V of variables that occur in the input ws-set S, a ws-set Γ is generated. Γ contains all total ws-descriptors relative to the variables in V. They are pairwise inconsistent and together they represent all worlds, i.e. $\omega(\Gamma) = \omega(())$. Note that in case $S = \emptyset$ or $S = \{()\}$ (when the number of variables is 0) we have that $\Gamma = \{()\}$. The inverse I is built of all the ws-descriptors d in Γ that are not consistent with any of the ws-descriptors in S. The lengths of the ws-descriptors in the result I do not depend on the length of the ws-descriptors in S but on the number of variables occurring in S.

Given a ws-set S with variables which have a maximum domain size of d, we show with the following two lemmas that $d^{|\textit{vars}(S)|}$ is a sharp upper bound for the size of the

Algorithm 2: INV-DECOMPRESS(S)

Require: a ws-set S
1: $V = \{v_1, \ldots, v_n\} = \mathit{vars}(S)$
2: $\Gamma = \{(v_1 = c_1 \wedge \cdots \wedge v_n = c_n) \mid c_1 \in \mathit{dom}(v_1), \ldots, c_n \in \mathit{dom}(v_n)\}$
3: $I = \emptyset$
4: **for** $d \in \Gamma$ **do**
5:     **if** not *consistent*(d, S) **then**
6:         $I = I \cup \{d\}$
7:     **end if**
8: **end for**
9: **return** I

inverses produced by INV-DECOMPRESS.

Lemma 4.5.1. *Given an arbitrary ws-set S with variables which have a maximum domain size of d,* $|\text{INV-DECOMPRESS}(S)| \leq d^{|\mathit{vars}(S)|}$.

Proof. Consider an arbitrary ws-set S. Let d be the maximum domain size of the variables occurring in S. The cardinality of the inverse I - as defined in Algorithm 2 - is obviously bounded by the cardinality of Γ. $|\Gamma| = \Pi_{v \in \mathit{vars}(S)} |\mathit{dom}(v)| \leq d^{|\mathit{vars}(S)|}$. It follows that $|I| \leq d^{|\mathit{vars}(S)|}$. □

Lemma 4.5.2. *There are ws-sets S such that* $|\text{INV-DECOMPRESS}(S)| = \Theta(d^{|\mathit{vars}(S)|})$, *where d is the maximum domain size of variables occurring in S.*

Proof. We consider a ws-set S of cardinality 1. Let S consist of one ws-descriptor of length n and let the domain size of each variable be d. It follows that $|\mathit{vars}(S)| = n$. Without loss of generality the constant used in the equalities is always 1, i.e. :

$$S = \{(x_1 = 1 \wedge \cdots \wedge x_n = 1)\}.$$

When applying INV-DECOMPRESS to S, we get a ws-set Γ in line 2 such that $|\Gamma| = d^n$. Of all the ws-descriptors in Γ only one, namely $(x_1 = 1 \wedge \cdots \wedge x_n = 1)$ is consistent with S. All the other $d^n - 1$ are inconsistent with S and therefore part of the result I. It follows that $|\text{INV-DECOMPRESS}(S)| = d^n - 1 = d^{|vars(S)|} - 1$. □

The complexity of INV-DECOMPRESS is in $\Theta(|S| d^{|vars(S)|} |vars(S)|)$. Each of the $d^{|vars(S)|}$ ws-descriptors in Γ has to be compared to each ws-descriptor in S, which takes linear time (in $|vars(S)|$), when using sorted ws-descriptors.

Negation

Decompression depends exponentially on the number of involved variables. The complexity is exponential in the number of the involved variables and the size of the resulting inverse possibly too. To avoid this we present a second method, negation, to compute the inverse. A ws-set is essentially a formula over finite variables in DNF. We negate it (which results in a formula in CNF) and turn it back into a formula in DNF. The following example shall clarify the idea.

Example 4.5.3. Consider the ws-set

$$S = \{(x = 3), (x = 1 \land y = 2)\} \text{ and } \mathit{dom}(x) = \mathit{dom}(y) = \{1, 2, 3\}.$$

We view S as a formula $(x = 3) \lor (x = 1 \land x = 2)$. Negating it and pushing the negation inside results in

$$\big(\neg(x = 3)\big) \land \big(\neg(x = 1) \lor \neg(y = 2)\big).$$

To turn this formula into a ws-set we have to turn it into DNF, resolve the negated equalities and remove inconsistent terms. First we turn it into DNF and get

$$\big(\neg(x = 3) \land \neg(x = 1)\big) \lor \big(\neg(x = 3) \land \neg(y = 2)\big).$$

Then we resolve the negated equalities by substituting them with equalities on all other domain values and get

$$\begin{array}{c}(x = 1 \land x = 2) \lor (x = 1 \land x = 3) \lor \\ (x = 2 \land x = 2) \lor (x = 2 \land x = 3) \lor \\ (x = 1 \land y = 1) \lor (x = 1 \land y = 3) \lor \\ (x = 2 \land y = 1) \lor (x = 2 \land y = 3).\end{array}$$

Finally we keep only the consistent terms, thus removing the first, the second and the fourth term. In addition we merge equivalent equalities in a term. The result is a ws-set S^i which is the inverse of S:

$$S^i = \{(x = 2), (x = 1 \land y = 1), (x = 1 \land y = 3), (x = 2 \land y = 1), (x = 2 \land y = 3)\}.$$

Note that the first ws-descriptor $(x = 2)$ in S^i subsumes the last two ws-descriptors, which means that we can safely remove them. We will return to this point in Chapter 6.

In Figure 4.7 we define *negate*$(\cdot)$ for the inverse of a ws-set S consisting of n ws-descriptors. The inverse consists of all combinations of inverses of equalities in the ws-descriptors $d_1, \ldots, d_n$. The result is again a ws-set. To build its ws-descriptors, from each of the n ws-descriptors in S an equality is chosen. From the i-th ws-descriptor we choose the equality j_i and use its variable $d_i[j_i].\mathit{var}$. Its value ($d_i[j_i].\mathit{val}$) is substituted by

$$
\begin{aligned}
&\text{Let } S = \{d_1, \ldots, d_n\}.\\
\textit{negate}(S) := \big\{ &(d_1[j_1].\textit{var} = c_{1,j_1} \wedge \cdots \wedge d_n[j_n].\textit{var} = c_{n,j_n}) \;\big|\\
&0 \leq j_i < |d_i|, 1 \leq i \leq n,\\
&c_{i,j_i} \in \textit{dom}(d_i[j_i].\textit{var}), c_{i,j_i} \neq d_i[j_i].\textit{val},\\
&1 \leq k \leq n,\\
&d_i[j_i].\textit{var} = d_k[j_k].\textit{var} \rightarrow c_{i,j_i} = c_{k,j_k} \big\}
\end{aligned}
$$

Figure 4.7: The inverse of a ws-set, using negation.

another, different value (c_{i,j_i}) in the domain of the variable. The last condition performs a consistency check in the same manner as condition ψ (see Figure 3.5). It filters out inconsistent ws-descriptors. Note that *negate*$(\emptyset) = \{()\}$ and *negate*$(\{()\}) = \emptyset$.

Theorem 4.5.4. negate$(\cdot)$ *is correct, i.e. given a ws-set* S*,* $\omega(\textit{negate}(S)) = \omega(())\backslash\omega(S)$*.*

Proof. The proof goes by induction. The base case is a ws-set S of cardinality 1. Without loss of generality, let $S = \{(x_1 = 1 \wedge \cdots \wedge x_m = 1)\}$. Obviously

$$
\textit{negate}(S) = \{(x_1 = 2), \ldots, (x_1 = \textit{max}(x_1)), \ldots, (x_m = 2), \ldots, (x_m = \textit{max}(x_m))\}
$$

is a correct inverse of S.

In the inductive step we assume that *negate*$(\cdot)$ is correct for ws-sets of cardinality at most n. We consider a ws-set of cardinality $n + 1$. Let $S = \{d_1, \ldots, d_{n+1}\}$. Let $S_1 = \{d_1, \ldots, d_n\}$ and $S_2 = \{d_{n+1}\}$. By assumption $I = \textit{negate}(S_1)$ and $I' = \textit{negate}(S_2)$ are correct inverses. *negate*(S) is equivalent to $S^i = \{\textit{concat}(d_1, d_2) \mid d_1 \in I, d_2 \in I', \psi(d_1, d_2)\}$, for which it holds that $\omega(S^i) = \omega(I) \cap \omega(I')$. By the correctness of I and I' it follows that

$$
\begin{aligned}
\omega(I) \cap \omega(I') &= \big(\omega(()) \setminus \omega(S_2)\big) \cap \big(\omega(()) \setminus \omega(S_2)\big)\\
&= \omega(()) \setminus \big(\omega(S_1) \cup \omega(S_2)\big)\\
&= \omega(()) \setminus \omega(S).
\end{aligned}
$$

□

Negation can be computed using the recursive algorithm INV-NEGATE as shown in Algorithm 3. On the i-th level of the recursion it builds the inverse of the i-th ws-descriptor of the input ws-set S and extends the temporary ws-descriptor *tmp*. It does this by iterating the equalities of the i-th ws-descriptor and the domain values of their variables. The consistency of *tmp* is always ensured. If *tmp* gets inconsistent the recursion is not continued. In the end the ws-set *result* holds the inverse of S.

Algorithm 3: INV-NEGATE(S, *level*, *tmp*, *result*)

Recursive function: first call is INV-NEGATE($S, 0, (), \{\}$)
Require: ws-set S, integer *level*, ws-descriptor *tmp*, ws-set *result*

```
1:  if level ≥ |S| then
2:     return result ∪ {tmp}
3:  else
4:     d = S[level]
5:     for e = 0 to |d| − 1 do
6:        v = d[e].var
7:        for i = 1 to max(v), i ≠ d[e].val do
8:           if consistent((v = i), tmp) then
9:              result = INV-NEGATE(S, level + 1, concat(tmp, (v = i)), result)
10:          end if
11:       end for
12:    end for
13:    return result
14: end if
```

We analyze the complexity of INV-NEGATE. Let n be the number of ws-descriptors the input ws-set consists of, m the maximum length of the ws-descriptors and d the maximum domain size of the occurring variables. At each level there are $O(md)$ possibilities (branches) and the n levels can be combined arbitrarily. For the consistency check (line 8 in INV-NEGATE) linear time in n is needed at each level, giving n^2 for all n levels. Altogether the complexity of INV-NEGATE is in $O(m^n d^n n^2)$. In Chapter 6 we improve INV-NEGATE such that its complexity can be bounded by an expression exponential in $|vars(S)|$, like the complexity of INV-DECOMPRESS.

Let us analyze the cardinality of the ws-sets generated by INV-NEGATE. Consider n, d and m as before. A ws-set generated by INV-NEGATE is maximal when the consistency check (line 8 in INV-NEGATE) always returns *true*. This is the case when in the input ws-set no variable occurs more than once. Then the cardinality of the inverse ws-set is in $O(m^n d^n)$.

Comparison

We compare the cardinality of the inverses produced by decompression and negation. The following theorem states that, given a ws-set S, $|$INV-NEGATE$(S)|$ can be exponentially smaller than $|$INV-DECOMPRESS$(S)|$.

Theorem 4.5.5. *Let S be a ws-set such that $|S| = 1$ and $\forall v \in vars(W) : dom(v) = \{1, \ldots, d\}$. Then $|$INV-NEGATE$(S)| = (d-1)n$ and $|$INV-DECOMPRESS$(S)| = d^n - 1$.*

Proof. Let S be an arbitrary ws-set of cardinality 1 such that $\forall v \in \mathit{vars}(W) : \mathit{dom}(v) = \{1, \ldots, d\}$, i.e. the domain size of all variables occurring in S is d. The ws-set S consists of one ws-descriptor. Without loss of generality the constant used in the equalities is always 1, i.e. :

$$S = \{(x_1 = 1 \wedge \cdots \wedge x_n = 1)\}.$$

The proof for Lemma 4.5.2 shows that $|\text{INV-DECOMPRESS}(S)| = d^n - 1$. It is easy to see that $|\text{INV-NEGATE}(S)| = (d-1)n$ because

$$\text{INV-NEGATE}(S) = \{(x_1 = 2), \ldots, (x_1 = d), \ldots, (x_n = 2), \ldots, (x_n = d)\}.$$

□

The complexity of INV-DECOMPRESS is exponential in the number of variables occurring in a ws-set, whereas the complexity of INV-NEGATE is exponential in the cardinality of the ws-set, based on the length of the ws-descriptors and the domain size of the used variables. In Chapter 5 we show how new descriptors make the complexity of negation independent of the domains of the used variables. In Chapter 6 we improve INV-NEGATE such that its complexity can drop to polynomial time.

4.6 Summary

At this point we want to briefly summarize the results of this chapter and point out new questions that have emerged. The update language we have presented is comprehensive and includes updates with subqueries. In case an update fulfills specific conditions we can apply the semi-join update method which requires only positive relational algebra. This means that it can be processed in polynomial time.

We have described how set difference can be used to model arbitrary update and delete statements on U-relations, and we have shown how to use the inverse of ws-sets to compute set difference. To compute the inverse of a ws-set we have presented two algorithms, one based on decompression and the other one based on negation. We have shown that negation can give exponentially smaller results. But negation also has an exponential worst-case complexity. Can we improve the complexity of negation? We tackle this question in the next chapters by extending U-relations and by extending the algorithm for negation.

5. Novel Descriptors for World Sets

In the previous chapter we have defined set difference on U-relations and shown how to compute the inverse of a ws-set using negation (the algorithm INV-NEGATE). Negation includes an exponential blowup in the cardinality of the input ws-set. There are two sources for the exponential blowup: the length of the ws-descriptors and the domain size of the variables occurring in the ws-descriptors. In this chapter we introduce new descriptors for world sets that allow us to get rid of the dependence on the domain size of the variables when computing the inverse. In this way the complexity of computing inverses can be reduced exponentially.

Example 5.0.1. Suppose the police wants to keep track of stolen vehicles and observations of possibly stolen ones. After gathering data about a stolen car the police waits for evidence before doing further investigations. For the sake of simplicity we abstract from car models, colors and number plates and simply assign unique ids to cars. For the same reason we also ignore location and time of an observation and only mention car ids. We model this by a U-relational database $(\textit{Cars}, \textit{Obs}, W)$ consisting of two U-relations of schema $\textit{Cars}[D, \textit{CId}, \textit{Invest}]$ and $\textit{Obs}[D, \textit{OId}, \textit{Car}]$. The D columns hold the ws-descriptors. For simplicity we omit columns for tuple ids in this example because we do not vertically partition the relations. *Cars* holds ids of stolen cars (attribute *CId*) and defines whether the theft should be investigated (attribute *Invest*). *Obs* holds observation ids (attribute *OId*) and ids of the observed cars (attribute *Car*). A simple scenario is shown in Figure 5.1. A car with id 1 is registered as stolen and the police has decided not to further investigate the theft. Two observations are reported and in both cases the observer was not sure which car he has seen and mentions four possibilities. In both cases the observed car possibly is the car with id 1 (in the first case if $x = 1$ and in the second case if $y = 2$). To update the status of investigation the following update statement is issued:

```
update Cars set Invest='yes' from Obs
where Car = CId;
```

There are two observations of car 1, and due to them the *where* condition of the update is satisfied in the worlds identified by valuations where $x = 1$ or $y = 2$. Using negation we get the result of the update as shown in Figure 5.2. In the worlds where $x = 1$ or $y = 2$ the car with id 1 was observed and the theft should be further investigated. In all the other worlds not. The value of the *Invest* attribute is nine times “no”. In

W	Var	Rng
	x	1
	x	2
	x	3
	x	4
	y	1
	y	2
	y	3
	y	4

Cars	D	CId	Invest
	$()$	1	no

Obs	D	OId	Car
	$(x = 1)$	1	1
	$(x = 2)$	1	2
	$(x = 3)$	1	3
	$(x = 4)$	1	4
	$(y = 1)$	2	5
	$(y = 2)$	2	1
	$(y = 3)$	2	6
	$(y = 4)$	2	7

Figure 5.1: A U-relational database representing stolen car investigations.

Cars	D	CId	Invest
	$(x = 1)$	1	yes
	$(y = 2)$	1	yes
	$(x = 2 \wedge y = 1)$	1	no
	$(x = 2 \wedge y = 3)$	1	no
	$(x = 2 \wedge y = 4)$	1	no
	$(x = 3 \wedge y = 1)$	1	no
	$(x = 3 \wedge y = 3)$	1	no
	$(x = 3 \wedge y = 4)$	1	no
	$(x = 4 \wedge y = 1)$	1	no
	$(x = 4 \wedge y = 3)$	1	no
	$(x = 4 \wedge y = 4)$	1	no

Figure 5.2: The U-relation *Cars* after applying the update.

Cars	D	CId	Invest
	$(x = 1)$	1	yes
	$(y = 2)$	1	yes
	$(x \neq 1 \wedge y \neq 2)$	1	no

Figure 5.3: The U-relation *Cars* after applying the update, using inequalities.

nine worlds it is the same and these nine worlds cannot be described by less than nine ws-descriptors.

Assume we can use inequalities to describe world sets. Then the result of the update can be shortly represented by the relation depicted in Figure 5.3. Instead of nine tuples only one is necessary to represent that the value of the *Invest* attribute is “no” in the nine worlds. Or assume we can use intervals to describe world sets. Then the result of

Cars	D	CId	Invest
	$(x = 1)$	1	yes
	$(y = 2)$	1	yes
	$(2 \leq x \leq 4 \wedge 1 \leq y \leq 1)$	1	no
	$(2 \leq x \leq 4 \wedge 3 \leq y \leq 4)$	1	no

Figure 5.4: The U-relation *Cars* after applying the update, using intervals.

the update can be shortly represented by the relation depicted in Figure 5.4. By using inequalities or intervals the result gets smaller and fewer descriptors are needed. In the next sections we present two extensions of U-relations, one additionally allowing inequalities and the other one additionally allowing intervals. While both extensions preserve polynomial time data complexity for positive relational algebra queries extended by the possible operator, they bring an exponential advantage for set difference.

5.1 Descriptors with Inequalities

We extend U-relations and present descriptors for world sets that consist not only of equalities but also of inequalities.

Definition 5.1.1 (iws-descriptor)**.** Given a world table W, an inequality-ws-descriptor (iws-descriptor) over W is a pair (E, I), where E and I are sets of pairs (v, c) where $v \in \mathit{vars}(W)$ and $c \in \mathit{dom}_W(v)$, such that a variable $v \in \mathit{vars}(W)$ does not occur more than once in E and such that $E \cap I = \emptyset$.

The set E represents equalities and is actually a ws-descriptor. The set I represents inequalities. The meaning of $(v, c) \in I$ is $v \neq c$. To emphasize the conjunctive nature of iws-descriptors we make use of the logical "*and*" symbol in our notation and combine the two sets E and I. We use equalities for the elements in E and inequalities for the elements in I. For example we write

$$(x = 1 \wedge y \neq 1 \wedge y \neq 3)$$

for the iws-descriptor $(\{(x, 1)\}, \{(y, 1), (y, 3)\})$. The condition that a variable must not occur more than once in E serves to exclude inconsistent cases like $(x = 1 \wedge x = 2)$. The condition $E \cap I = \emptyset$ serves to exclude inconsistent cases like $(x = 1 \wedge x \neq 1)$. Given an iws-descriptor $d = (E, I)$ we use $|d| = |E| + |I|$ do denote the *length* of d (the number of equalities and inequalities d consists of). We use $\mathit{vars}(d)$ to refer to the set of variables occurring on the left hand side of the (in-)equalities in d.

Analogously to ws-sets we call sets of iws-descriptors *iws-sets*, and the inverse of an iws-set S is an iws-set S^i such that $\omega(S^i) = \omega(()) \setminus \omega(S)$.

We define the semantics of iws-descriptors by defining ω for iws-descriptors.

Definition 5.1.2 (ω for iws-descriptors)**.** Given an iws-descriptor $d = (E, I)$ over a world table W, $\omega(d)$ is the set of total valuations f of variables defined by W such that for every $(v, c) \in E : f(v) = c$ and for every $(v, c) \in I : f(v) \neq c$.

Suppose for example the iws-descriptor $d = (x \neq 1 \wedge x \neq 2 \wedge y = 3)$ and $dom(x) = \{1, \ldots, 5\}$. Then $\omega(d)$ is the set of valuations where x is 3, 4 or 5, and y is 3. The semantics of iws-sets is defined by further extending the function ω to iws-sets. Given an iws-set S, $\omega(S) = \bigcup_{d \in S} \omega(d)$. Two iws-descriptors d_1 and d_2 are *consistent* if $\omega(d_1) \cap \omega(d_2) \neq \emptyset$. On the syntactic level consistency of two iws-descriptors d_1 and d_2 can be checked by ensuring the following constraints:

- $\forall_{(v_1,c_1) \in d_1.E} \forall_{(v_2,c_2) \in d_2.E} (v_1 = v_2 \rightarrow c_1 = c_2)$
- $\forall_{(v_1,c_1) \in d_1.E} \forall_{(v_2,c_2) \in d_2.I} (v_1 = v_2 \rightarrow c_1 \neq c_2)$
- $\forall_{(v_2,c_2) \in d_2.E} \forall_{(v_1,c_1) \in d_1.I} (v_1 = v_2 \rightarrow c_1 \neq c_2)$

The first one equals the consistency check for ws-descriptors (the condition ψ). In contrast to ws-descriptors an iws-descriptor can be consistent but not satisfiable by any valuation at the same time. Assume a variable x and and $dom(x) = \{1, 2, 3\}$. The iws-descriptor $(x \neq 1 \wedge x \neq 2 \wedge x \neq 3)$ is consistent, its equalities and inequalities do not contradict each other. But regarding the domain of x it is not satisfied by any valuation because all the domain values of x occur in an inequality with x. No domain value is left that could be assigned to x.

Definition 5.1.3 (Infeasible iws-descriptor)**.** An iws-descriptor d for which $\omega(d) = \emptyset$ is called *infeasible*.

To decide feasibility of iws-descriptors we present FEASIBLE in Algorithm 4. For all the variables in an iws-descriptor that occur in inequalities it looks up their domain in the world table and checks whether all domain values occur in inequalities with that variable. If this is the case for at least one variable then the iws-descriptor is infeasible. By ordering the inequalities in I by variable name the check for feasibility is possible in linear time in $|I|$.

Definition 5.1.4. U^i-relations are U-relations based on feasible iws-descriptors instead of ws-descriptors.

The semantics of U^i-relations follows from the definition of ω for iws-descriptors. Iws-descriptors extend ws-descriptors, therefore U^i-relations are also a *complete representation system* for uncertain databases, i.e. every uncertain database can be represented using U^i-relations. As iws-descriptors extend ws-descriptors we can use a U-relation every time we expect a U^i-relation.

Algorithm 4: FEASIBLE(d)

Require: iws-descriptor $d = (E, I)$
1: *vars* $= \{v \mid (v, c) \in I\}$
2: **for** v in *vars* **do**
3: *values* $= \{c \mid (v, c) \in I\}$
4: **if** $|values| \geq max(v)$ **then**
5: **return** false
6: **end if**
7: **end for**
8: **return** true

Encoding Iws-descriptors

We use an encoding for iws-descriptors that is more practical and gives rise to an easy definition of the inverse via negation. Remember that we assumed a normalized world table and therefore the domains of the variables are positive integers. We encode an iws-descriptor $d = (E, I)$ as one set $E \cup \{(x, \text{-}c) \mid (x, c) \in I\}$. So in this encoding an iws-descriptor over a world table W is a set of pairs (v, c) where $v \in vars(W)$ and $c \in dom_W(v)$ or $\text{-}c \in dom_W(v)$. Positive constants denote equalities and negative constants denote inequalities. For example the iws-descriptor $(x = 1 \wedge y \neq 1 \wedge y \neq 3)$ is encoded by the set $\{(x, 1), (y, \text{-}1), (y, \text{-}3)\}$. We assume an arbitrary order on the set and write $d[i].var$ to refer to the variable in the i-th element of d and $d[i].val$ to refer to the constant in the i-th element.

Queries on U^i-relations

In this section we show how to translate relational algebra queries with *poss* and *merge* into queries on U^i-relations. We use a translation function $[\![\cdot]\!]'$ to translate relational algebra with *poss* and *merge*. Let $[\![\cdot]\!]'$ be equivalent to $[\![\cdot]\!]$ (see Figure 3.5 and 4.6) except for the consistency check and the inverse. In Figure 5.5 we define the function ψ' that checks the consistency and feasibility of two iws-descriptors. It replaces ψ in $[\![\cdot]\!]'$. The function *consistent*$(\cdot)$ uses the beforehand described encoding of iws-descriptors. In Figure 5.6 we define the function *negate*$'(\cdot)$ for the inverse of an iws-set. It replaces *negate*$(\cdot)$ in the translation $[\![\cdot]\!]'$. The function *negate*$'(\cdot)$ is quite similar to *negate*$(\cdot)$ for ws-descriptors but does not depend on the domains of the occurring variables. From each of the iws-descriptors d_i an (in-)equality $d_i[j_i]$ is chosen and the value negated to turn an equality into an inequality and vice versa. The function ψ' is used to check that the generated iws-descriptors are consistent and feasible. Analogously to INV-NEGATE we can easily turn *negate*$'(\cdot)$ into a recursive algorithm.

Let us analyze the cardinality of the iws-sets generated by *negate*$'(\cdot)$. Let n be the

$$consistent(d_1, d_2) := \bigwedge_{i<|d_1|, j<|d_2|} \big(d_1[i].var = d_2[j].var \rightarrow d_1[i].val \neq \text{-}d_2[j].val \wedge$$
$$(d_1[i].val = d_2[j].val \vee d_1[i].val < 0 \vee d_2[j].val < 0)\big)$$

$$\psi'(d_1, d_2) := consistent(d_1, d_2) \wedge feasible(d_1 \wedge d_2)$$

Figure 5.5: Consistency check for iws-descriptors.

$$\text{Let } S = \{d_1, \ldots, d_n\}.$$
$$negate'(S) := \big\{d = (d_1[j_1].var = \text{-}d_1[j_1].val \wedge \cdots \wedge d_n[j_n].var = \text{-}d_n[j_n].val) \;\big|\; 0 \leq j_i < |d_i|, \psi'(d, d)\big\}$$

Figure 5.6: Negation for iws-sets.

number of iws-descriptors the input iws-set consists of and m the maximum length of the iws-descriptors. An iws-set generated by *negate*$'(\cdot)$ is maximal when the consistency check ψ' always returns *true*. This is the case when in the input iws-set no variable occurs more than once. Then the cardinality of the inverse iws-set is in $O(m^n)$.

From the translation $[\![\cdot]\!]'$ it follows that

Theorem 5.1.5. *Positive relational algebra queries extended with poss and merge can be evaluated on U^i-relational databases in polynomial time data complexity.*

Proof. By Theorem 3.4.1 positive relational algebra queries extended with *poss* and *merge* can be evaluated on U-relational databases in polynomial time data complexity. On U^i-relations the translation of positive relational algebra queries extended with *poss* and *merge* differs only in the consistency check ψ'. The function ψ' needs only polynomial time, as the function ψ. Hence positive relational algebra queries extended with *poss* and *merge* can be evaluated on U^i-relational databases also in polynomial time data complexity. □

5.2 Iws-descriptors outpace Ws-descriptors

In this section we compare U-relations to U^i-relations. We show that U^i-relations can be exponentially more succinct and that they never get less succinct than U-relations.

The following theorem states that the result of set difference (and thus of updates) can be exponentially more succinct when using iws-descriptors, compared to using ws-descriptors.

Theorem 5.2.1. *There are ws-sets S such that $|negate'(S)|$ is exponentially smaller than $|negate(S)|$.*

Proof. Consider the ws-set $S = \{(x_1 = 1), \dots, (x_n = 1)\}$ and let $max(x_i) = d$ for $1 \le i \le n$. Then $|negate(S)| = (d-1)^n$ because

$$\begin{aligned} negate(S) = \{&(x_1 = 2 \land x_2 = 2 \land \cdots \land x_n = 2), \\ &(x_1 = 3 \land x_2 = 2 \land \cdots \land x_n = 2), \\ &(x_1 = 2 \land x_2 = 3 \land \cdots \land x_n = 2), \\ &\vdots \\ &(x_1 = d \land x_2 = d \land \cdots \land x_n = d)\}. \end{aligned}$$

In contrast $|negate'(S)| = 1$ because $negate'(S) = \{(x_1 \neq 1 \land \cdots \land x_n \neq 1)\}$. □

Now we will show that with U^i-relations we never get less succinct than with U-relations. Let us start with some preliminary observations and a definition which we will use in the coming proof.

Remark 5.2.2. Given a ws-descriptor d_1 and a feasible iws-descriptor d_2, there are exactly two possibilities that they are inconsistent, i.e. that $\omega(d_1) \cap \omega(d_2) = \emptyset$:

- $(x = c) \in d_1 \land (x = c') \in d_2, c \neq c'$
- $(x = c) \in d_1 \land (x \neq c) \in d_2$

Definition 5.2.3. An instantiation of an iws-descriptor d is a ws-descriptor d_{inst} such that $vars(d_{inst}) = vars(d)$ and $\omega(d_{inst}) \cap \omega(d) \neq \emptyset$.

So an iws-descriptor is consistent with its instantiations. We can see the inequalities in an iws-descriptor as templates that allow different instantiations. For example: Assume a general domain $\{1, 2, 3\}$, then the iws-descriptor $(x \neq 1, y \neq 1, z = 1)$ has four instantiations, namely $(x = 2 \land y = 2 \land z = 1)$, $(x = 2 \land y = 3 \land z = 1)$, $(x = 3 \land y = 2 \land z = 1)$ and $(x = 3 \land y = 3 \land z = 1)$. Let d_{inst} be an instantiation of an iws-descriptor d. Then $\omega(d_{inst}) \subseteq \omega(d)$.

Remark 5.2.4. Let S be the set of all ws-descriptors that are instantiations of an iws-descriptor d. Then $\omega(d) = \omega(S)$.

The following theorem states that the result of a query is never less succinct when using U^i-relations, compared to using U-relations. Beforehand, we want to point out that its proof is rather long and complex.

Theorem 5.2.5. *Given a query Q in relational algebra extended by poss and merge, and a U-relational database $\mathbf{A}$, the number of tuples in $[\![Q(\mathbf{A})]\!]'$ is never more than the number of tuples in $[\![Q(\mathbf{A})]\!]$.*

Proof. The proof proceeds by structural induction on all the operators a query can consist of. The main idea is to show that there is a surjective function that maps tuples in the U-relation $T = [\![Q(\mathbf{A})]\!]$ to tuples in the U^i-relation $T' = [\![Q(\mathbf{A})]\!]'$. Surjectiveness of the function implies that $|T| \geq |T'|$. We will rely on two conditions on the function. They help to conduct the inductive step and imply surjectiveness.

We first define the two conditions and describe their characteristics. Let $f(\cdot)$ be a mapping between the tuples of a U-relation R and a U^i-relation R', both with schema $[D, \bar{T}, \bar{A}]$, $f : R \to R'$. R uses ws-descriptors, whereas R' uses iws-descriptors. As usual, the D attribute represents the (i)ws-descriptors, the $\bar{T}$ attributes represent the tuple ids and the $\bar{A}$ attributes represent the actual data values. Given a tuple $t \in R$ and a tuple $t' \in R'$ we say that t and t' are *corresponding* tuples if $f(t) = t'$.

Condition *C1*:

$$\begin{aligned} &\forall t \in R, \forall t' \in R' : \\ &f(t) = t' \Rightarrow \omega(t.D) \subseteq \omega(t'.D) \wedge t.\bar{T} = t'.\bar{T} \wedge t.\bar{A} = t'.\bar{A} \end{aligned}$$

Condition *C2*:

$$\forall t' \in R' : \Big(\bigcup_{t:f(t)=t'} \omega(t.D) \Big) \supseteq \omega(t'.D)$$

Intuitively *C1* is about the completeness of R' and *C2* about the soundness of R'. *C1* ensures that $f(\cdot)$ is a mapping between tuples that have the same tuple ids and the same data values, and that the ws-descriptor $t.D$ describes a subset of the worlds the iws-descriptor $t'.D$ describes. *C2* ensures that an iws-descriptor does not represent more worlds than the corresponding ws-descriptors together. $\omega(t'.D)$ is never the empty set of worlds because only consistent and feasible iws-descriptors are allowed in U^i-relations. Hence *C2* also implies that $f(\cdot)$ is a surjective function.

When we combine the two conditions we have that

$$\forall t' \in R' : \bigcup_{t:f(t)=t'} \omega(t.D) = \omega(t'.D)$$

It follows that *rep*(R) = *rep*(R'). The two conditions assert that R and R' represent the same set of relations and that a tuple t' in R' has at least one counterpart in R, such that all counterparts together represent the same information as t'.

Now we are ready to start with the actual proof. Let $\mathbf{A}$ be an arbitrary U-relational database and Q an arbitrary query on $\mathbf{A}$. By structural induction we show that there is a function mapping the tuples in $[\![Q(\mathbf{A})]\!]$ to the tuples in $[\![Q(\mathbf{A})]\!]'$ that satisfies the conditions *C1* and *C2*.

The *base case* is a query without operators that just selects an arbitrary relation, i.e. $Q = R$. Using ws-descriptors the result is a relation $T = R$ and using iws-descriptor the result is an identical relation $T' = R$. We define the function $f_T : T \to T'$, that maps tuples in T to tuples in T', as the identity function. It obviously satisfies the two conditions.

Inductive step: Let $\circ$ be any of the operators of relational algebra, merge and poss. Let $Q = \circ Q_1$ (in case $\circ$ is unary) or $Q = Q_1 \circ Q_2$ (in case $\circ$ is binary). Let $R = [\![Q_1(\mathbf{A})]\!]$ and $R' = [\![Q_1(\mathbf{A})]\!]'$, and in case of a binary operator $S = [\![Q_2(\mathbf{A})]\!]$ and $S' = [\![Q_2(\mathbf{A})]\!]'$. R and S are U-relations and R' and S' are U^i-relations. R and R' as well as S and S' are of the same schema. Let $sch(R) = sch(R') = [D_1, \bar{T}_1, \bar{A}_1]$ and $sch(S) = sch(S') = [D_2, \bar{T}_2, \bar{A}_2]$. Our induction hypothesis is that there are functions $f_R : R \to R'$ and $f_S : S \to S'$ that satisfy the two conditions *C1* and *C2*. In case $\circ$ is a unary operator, let $T = [\![\circ Q_1(\mathbf{A})]\!]$ and $T' = [\![\circ Q_1(\mathbf{A})]\!]'$. Similarly, in case $\circ$ is a binary operator, let $T = [\![Q_1(\mathbf{A}) \circ Q_2(\mathbf{A})]\!]$ and $T' = [\![Q_1(\mathbf{A}) \circ Q_2(\mathbf{A})]\!]'$. Let $sch(T) = sch(T') = [D_3, \bar{T}_3, \bar{A}_3]$. We show for each operator separately that there is a function $f_T : T \to T'$, that satisfies *C1* and *C2*.

Selection: Selection allows only conditions on the data attributes $\bar{A}_1$ of R. We consider also conditions on the tuple ids $\bar{T}_1$ so that we can reuse this part of the proof for the merge operator later. Given a select predicate ϕ on $\bar{A}_1$ or $\bar{T}_1$, we get relations

```
T  = select * from R  where φ;
T' = select * from R' where φ;
```

We define $f_T(t) = f_R(t)$, where the domain of f_T is $T \subseteq R$ and the codomain is $T' \subseteq R'$. By the inductive hypothesis $f_R(\cdot)$ satisfies *C1* and therefore corresponding tuples in R and R' have the same tuple ids and data values. Thus a tuple $r' \in R'$ satisfies ϕ if and only if all corresponding tuples $\{r_1, \ldots, r_l\} = f_R^{-1}(r)$ satisfy ϕ. Either $r' \in T'$ and all $r_1, \ldots, r_l \in T$ or $r' \notin T'$ and none of $r_1, \ldots, r_l \in T$. It follows that $f_T(\cdot)$ satisfies *C1* and *C2*.

Rename: Renaming does not change the tuples, but only the attribute names. We simply define $f_T(t) = f_R(t)$.

Projection: Projection can be used to reorder the attributes, in which case the induction hypothesis is trivially satisfied by the resulting relations. Besides that projection is used to project out attributes. Without loss of generality we project out only one attribute, because projecting out more than one attribute can be done one by one in separate projection steps.

Suppose we project out an arbitrary attribute $A \in \bar{A}_1$, i.e.

```
T  = select D1, T̄1, Ā1\{A} from R;
T' = select D1, T̄1, Ā1\{A} from R';
```

Let $p(\cdot)$ be the function that projects out the attribute A from a tuple. For every tuple $t \in T$ there exists a tuple $r \in R$ such that $t = p(r)$.

Its inverse $p^{-1}(\cdot)$ is well-defined (which is not trivial). To see this assume to the contrary that there are two tuples $r_1 = (d_1, \bar{t}_1, \bar{a}_1)$ and $r_2 = (d_2, \bar{t}_2, \bar{a}_2)$ in R such that $r_1 \neq r_2$ and $p(r_1) = p(r_2)$. As $p(r_1) = p(r_2)$ it follows that $d_1 = d_2$, $\bar{t}_1 = \bar{t}_2$ and that $\bar{a}_1 \neq \bar{a}_2$. This is a contradiction to the definition of a valid U-relation, because it means that in the worlds described by d_1 the tuple with id $\bar{t}_1$ gets contradictory attribute values $\bar{a}_1$ and $\bar{a}_2$. So the assumption was wrong and $p^{-1}(\cdot)$ is well-defined.

We define $f_T(t) := p(f_R(p^{-1}(t)))$. Clearly f_T satisfies the two conditions.

Union: A precondition for the union of R (R') and S (S') is that $\bar{T}_1 \cap \bar{T}_2 = \emptyset$. Hence the union is always a union of two disjoint sets. Let

$$\begin{aligned} f_T(t) &:= f_R(t) \text{ if } t \in R \\ &:= f_S(t) \text{ if } t \in S \end{aligned}$$

By the inductive hypothesis $f_R(\cdot)$ and $f_S(\cdot)$ satisfy *C1* and *C2* and therefore $f_T(\cdot)$ satisfies the two conditions too.

Cross product: We build the cross product of R and S, respectively R' and S':

T = **select** *concat*$(D_1, D_2), \bar{T}_1, \bar{T}_2, \bar{A}_1, \bar{A}_2$ **from** R, S **where** $\psi(D_1, D_2)$;
T' = **select** *concat*$(D_1, D_2), \bar{T}_1, \bar{T}_2, \bar{A}_1, \bar{A}_2$ **from** R', S' **where** $\psi'(D_1, D_2)$;

Consider an arbitrary tuple $t = ((d_1 \wedge d_2), \bar{t}_1, \bar{t}_2, \bar{a}_1, \bar{a}_2)$ in T. d_1 are the equalities in the ws-descriptor of t that originate in R and d_2 are the equalities in the ws-descriptor of t that originate in S. By definition of the cross product $r = (d_1, \bar{t}_1, \bar{a}_1) \in R$ and $s = (d_2, \bar{t}_2, \bar{a}_2) \in S$. Let $r' = f_R(r)$ and $s' = f_S(s)$. We define

$$f_T(t) := ((r'.D_1 \wedge s'.D_2), \bar{t}_1, \bar{t}_2, \bar{a}_1, \bar{a}_2)$$

We will first show that $f_T(t) \in T'$ and that $f_T(\cdot)$ satisfies *C1*. Then we will show that $f_T(\cdot)$ also satisfies *C2*.

Due to the join condition ψ the intersection $\omega(d_1) \cap \omega(d_2) \neq \emptyset$, otherwise t would not be part of T. As $f_R(\cdot)$ and $f_S(\cdot)$ satisfy *C1*, $\omega(r'.D_1) \supseteq \omega(r.D_1)$ and likewise $\omega(s'.D_2) \supseteq \omega(s.D_2)$. Given this and the fact that $\omega(r.D_1) \cap \omega(s.D_2) \neq \emptyset$ it follows that $\omega(r'.D_1) \cap \omega(s'.D_2) \neq \emptyset$. Therefore $r'.D_1$ and $s'.D_2$ satisfy the join condition ψ' and $f_T(t) \in T'$. By definition of f_T we have that t and $f_T(t)$ have equivalent tuple ids and data values, and obviously $\omega(r'.D_1 \wedge s'.D_2) \supseteq \omega(r.D_1 \wedge s.D_2)$. This means that f_T satisfies *C1*.

To see that $f_T(\cdot)$ also satisfies *C2*, consider an arbitrary tuple

$$t' = ((d_1 \wedge d_2), \bar{t}_1, \bar{t}_2, \bar{a}_1, \bar{a}_2) \in T'.$$

d_1 are the (in-)equalities in the iws-descriptor of t' that originate in R' and d_2 are the (in-)equalities in the iws-descriptor of t' that originate in S'. By the definition of the cross product it follows that $r' = (d_1, \bar{t}_1, \bar{a}_1) \in R'$ and $s' = (d_2, \bar{t}_2, \bar{a}_2) \in S'$. Let

$\{r_1, \ldots, r_k\} = f_R^{-1}(r')$ and $\{s_1, \ldots, s_l\} = f_S^{-1}(s')$. The following sequence of equations shows that $f_T(\cdot)$ satisfies *C2*.

$$\begin{aligned}
\omega(t'.D_3) &= \omega(d_1) \cap \omega(d_2) = \omega(r'.D_1) \cap \omega(s'.D_2) \\
&\subseteq \bigcup_{1 \leq i \leq k} \omega(r_i.D_1) \cap \bigcup_{1 \leq i \leq l} \omega(s_i.D_2) \\
&\subseteq \bigcup_{1 \leq i \leq k, 1 \leq j \leq l} (\omega(r_i.D_1) \cap \omega(s_j.D_2)) \\
&\subseteq \bigcup_{1 \leq i \leq k, 1 \leq j \leq l, \omega(r_i.D_1) \cap \omega(s_j.D_2) \neq \emptyset} (\omega(r_i.D_1) \cap \omega(s_j.D_2)) \\
&\subseteq \bigcup_{t : f(t) = t'} \omega(t.D_3)
\end{aligned}$$

f_R and f_S satisfy *C2* which enables the step from the first to the second line. In line 3 we apply distributivity of set intersection and union. In line 4, omitting empty sets (the inconsistent ws-descriptors) in the union does not change the result of the union. These are exactly the ws-descriptors of the tuples in T that are mapped to t' by $f_T(\cdot)$: All combinations of tuples r_i and s_j where $\omega(r_i.D_1 \wedge s_j.D_2) \neq \emptyset$ (condition ψ). Altogether we have that $f_T(\cdot)$ satisfies *C2*.

Merge: The merge operator is in fact a cross product followed by a selection on the tuple id attributes (the condition α) and a projection that merges the common tuple id attributes. We have shown that for the cross product and the selection on the tuple id attributes a function $f_T(\cdot)$ satisfying *C1* and *C2* exists. Due to condition α the values of the common tuple id attributes are identical and thus in the projection only duplicate data is projected out, which means that the projection does not make tuples equivalent. It follows that there is a function $f_T : T \rightarrow T'$ that satisfies *C1* and *C2*.

Possible: The possible operator projects out the tuple ids and replaces the (i)ws-descriptors with empty ws-descriptors. As f_R satisfies *C1* and *C2*, $\mathit{rep}(R) = \mathit{rep}(R')$. It follows that $T = T'$. We define $f_T(t) := t$, which clearly satisfies *C1* and *C2*.

Set Difference: The core of set difference is the inverse of a ws-set. We split the proof for set difference into two parts: one for the inverse (negation), and one for set difference (building on the first part).

For the *inverse* we show that the resulting ws-sets/iws-sets satisfy conditions similar to the two defined conditions. We transform the two conditions *C1* and *C2* to two conditions on ws-sets and iws-sets by omitting the data values and tuple ids. Let $g : W \rightarrow W'$ be a function from ws-descriptors in a ws-set W to iws-descriptors in an iws-set W'.

Condition *C1'*:

$$\begin{aligned}
&\forall d \in W, \forall d' \in W' : \\
&g(d) = d' \Rightarrow \omega(d) \subseteq \omega(d')
\end{aligned}$$

Condition *C2'*:

$$\forall d' \in W' : \Big(\bigcup_{d:g(d)=d'} \omega(d) \Big) \supseteq \omega(d')$$

Claim 1. If $g(\cdot)$ satisfies *C1'* and $g(d) = d'$ for an arbitrary ws-descriptor d, then we can use the following conclusions:

- $(x = c) \in d' \Rightarrow (x = c) \in d$
- $(x \neq c) \in d' \Rightarrow (x = c') \in d, c' \neq c$

The correctness of Claim 1 is easy to see: by assuming the opposite we get a contradiction with *C1'*.

Claim 2. Given a ws-set W and an iws-set W' such that there is a function $g : W \to W'$ that satisfies *C1'* and *C2'*, there is a function $g_{neg} : negate(W) \to negate'(W')$ that satisfies *C1'* and *C2'*.

The proof for Claim 2 is more involved. First we will define the function g_{neg} : $negate(W) \to negate'(W')$. Then we show that $g_{neg}(\cdot)$ satisfies the two conditions *C1'* and *C2'*, one after another.

Consider an arbitrary ws-set W and an arbitrary iws-set W' such that there exists a function $g : W \to W'$ that satisfies *C1'* and *C2'*. Let $W' = \{w'_1, \ldots, w'_n\}$ and

$$G_i = \{w_{i,1}, \ldots, w_{i,l_i}\} = g^{-1}(w'_i) \text{ for } 1 \leq i \leq n.$$

$G_i \subseteq W$ and G_i is the set of ws-descriptors in W that map to w'_i. The union of all n sets is the ws-set W and the n groups are pairwise disjoint, i.e. $W = \bigcup_{1 \leq i \leq n} G_i$ and $G_i \cap G_j = \emptyset$ for $i \neq j$. So the ws-sets G_i partition the ws-set W and

$$W = \{w_{1,1}, \ldots, w_{1,l_1}, \ldots, w_{n,1}, \ldots, w_{n,l_i}\}.$$

As g satisfies *C1'* and *C2'* we conclude that $\omega(G_i) = \omega(w'_i)$. G_i and w'_i describe the same set of worlds. Figure 5.7 depicts the setting.

To define $g_{neg}(\cdot)$ we consider a ws-descriptor $d \in negate(W)$ and show how to construct $d' = g_{neg}(d)$. Note that by the definition of $negate(\cdot)$ the ws-descriptor d contains one equality for each ws-descriptor in W. So let

$$d = (x_{1,1} = c_{1,1} \wedge \cdots \wedge x_{1,l_1} = c_{1,l_1} \wedge \cdots \wedge x_{n,1} = c_{n,1} \wedge \cdots \wedge x_{n,l_n} = c_{n,l_n})$$

be an arbitrary ws-descriptor in $negate(W)$. Note that the $x_{i,j}$'s are meta-variables and the $c_{i,j}$'s meta-constants. For example it could hold that $x_{1,1}$ and $x_{3,1}$ represent the same variable or that $c_{5,2}$ and $c_{6,3}$ represent the same constant. For ease of notation let $d_i = (x_{i,1} = c_{i,1} \wedge \cdots \wedge x_{i,l_i} = c_{i,l_i})$, so that $d = (d_1 \wedge \cdots \wedge d_n)$. We say that d_i is the i-th group of equalities that d consists of.

By the definition of $negate(\cdot)$ we know that every ws-descriptor $w_{i,j} \in W$ ($1 \leq i \leq n$, $1 \leq j \leq l_i$) contains an equality $x_{i,j} = \hat{c}_{i,j}$, such that $\hat{c}_{i,j} \neq c_{i,j}$ for $1 \leq j \leq l_i$ (the

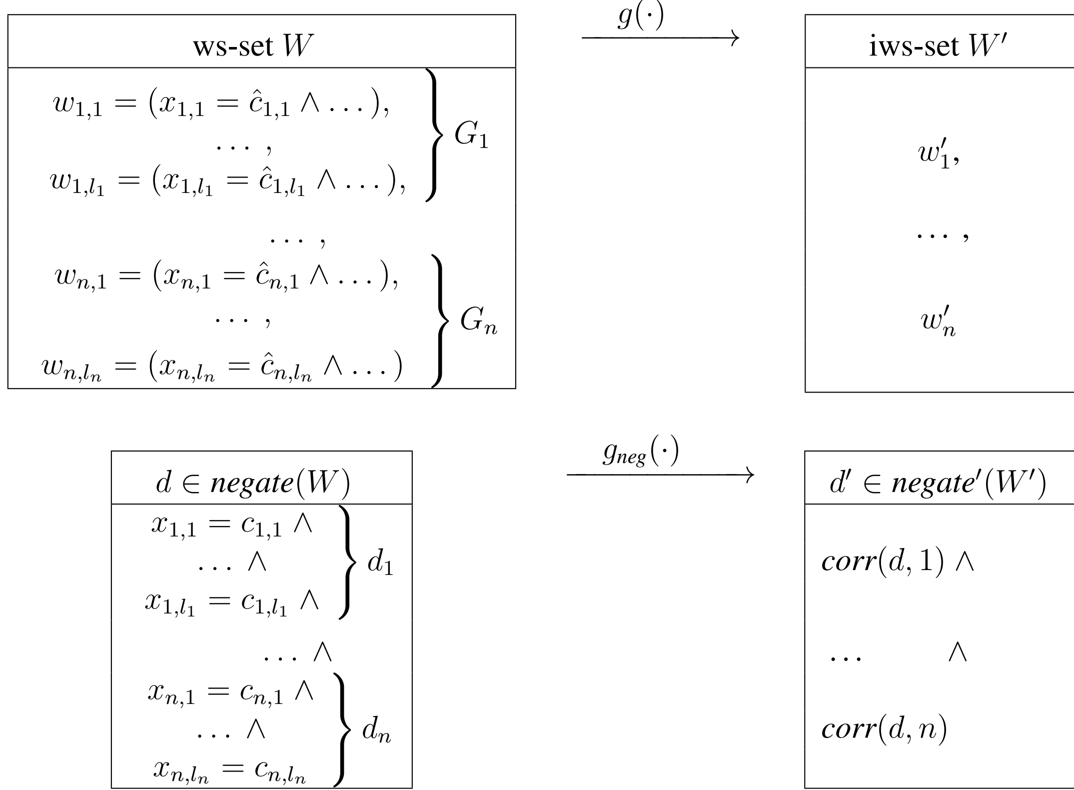

Figure 5.7: Sketch for the proof of Claim 2.

same variables as in d, but a different constant). It follows that $\omega(d_i) \cap \omega(G_i) = \emptyset$. As G_i and w'_i describe the same set of worlds we conclude that

$$\omega(d_i) \cap \omega(w'_i) = \emptyset \tag{5.1}$$

We define a correspondence function *corr*$(\cdot, \cdot)$ that assigns an equality or inequality to each d_i. We will use *corr*$(\cdot)$ later to define $g_{neg}(\cdot)$. Remember that $x_{i,j} = \hat{c}_{i,j}$ is part of every $w_{i,j}$. We define *corr*(d, i) for the ws-descriptor d in *negate*(W) and its i-th group of equalities as follows:

Case 1: There is a j such that $x_{i,j} = \hat{c}_{i,j}$ is part of w'_i. This means that both w'_i and $w_{i,j}$ contain the equality $x_{i,j} = \hat{c}_{i,j}$. If there is more than one j we arbitrarily choose one. We define *corr*$(d, i) := (x_{i,j} \neq \hat{c}_{i,j})$.

Case 2: There is no j such that $x_{i,j} = \hat{c}_{i,j}$ is part of w'_i. This means that the variables $x_{i,1}, \dots, x_{i,l_i}$ can only occur in an inequality in w'_i. Otherwise, if any of the variables occurs in an equality with another constant, $\omega(w_{i,j}) \nsubseteq \omega(w'_i)$ and g would violate *C1'*. We will show that there is a j such that the inequality $x_{i,j} \neq c_{i,j}$ is part of w'_i. If there

is more than one j we arbitrarily choose one. For case 2 we define $\mathit{corr}(d, i) := (x_{i,j} = c_{i,j})$.

Assume to the contrary that there is no j such that $x_{i,j} \neq c_{i,j}$ is part of w'_i. This means that none of the equalities in d_i occurs negated in w'_i. In summary, d_i only consists of equalities with variables that do not occur in w'_i and of equalities $x_{i,j} = c_{i,j}$ for which the following holds:

$x_{i,j}$ is not part of an equality in w'_i due to case 2 and
$x_{i,j} \neq c_{i,j}$ does not occur in w'_i as assumed.

By Remark 5.2.2 it follows that d_i is consistent with w'_i, i.e. $\omega(d_i) \cap \omega(w'_i) \neq \emptyset$. This is a contradiction to Equation 5.1. We can conclude that the assumption is wrong and there is a j such that $x_{i,j} \neq c_{i,j}$ is part of w'_i.

We are ready to define

$$g_{\mathit{neg}}(d) = (\mathit{corr}(d, 1) \wedge \cdots \wedge \mathit{corr}(d, n)).$$

We need to show that the image of $g_{\mathit{neg}}(\cdot)$ is actually contained in $\mathit{negate}(W')$, i.e. that $g_{\mathit{neg}}(d) \in \mathit{negate}'(W')$. Let $d' = g_{\mathit{neg}}(d)$ for our arbitrary tuple d.

Each $\mathit{corr}(d, i)$ is the negation of an (in-)equality in w'_i. To show that d' is part of $\mathit{negate}'(W')$ we need to show that d' satisfies ψ' (that it is consistent and feasible).

d' is consistent because its equalities are all part of d, which is consistent, and inequalities cannot lead to inconsistency either, as we show now. Assume to the contrary that d' contains the equality $x = c$ and the inequality $x \neq c$ for an arbitrary variable x and an arbitrary constant c. By the construction of d' via g_{neg} we can conclude that d contains $x = c$ as well as $x = c'$ for a $c' \neq c$. This would mean that d is inconsistent, which is not possible. Therefore we know that d' is consistent as well.

Similarly we can see that d' is feasible. Assume to the contrary that d' is infeasible, which means that there is a variable x such that d' contains inequalities $x \neq c$ for all $c \in \mathit{dom}(x)$. By the construction of d' there is an equality $x = c'$ in d for each inequality $x \neq c$ in d' and $c' \neq c$. As d' contains inequalities $x \neq c$ for all $c \in \mathit{dom}(x)$, the equalities in d cannot be all the same, which is a contradiction to the consistency of d. Hence the assumption must be wrong and d' is feasible.

We have shown that $d' \in \mathit{negate}(W')$. By the definition of g_{neg} the iws-descriptor d' consists of equalities that likewise occur in d and of inequalities for which corresponding equalities with the same variables but different constant values occur in d. Hence $\omega(d) \subseteq \omega(d')$. It follows that $g_{\mathit{neg}}(\cdot)$ fulfills condition *C1'*.

It remains to show that $g_{\mathit{neg}}(\cdot)$ satisfies condition *C2'*. For this consider an arbitrary iws-descriptor $d' \in \mathit{negate}'(W')$. Remember that $W' = \{w'_1, \ldots, w'_n\}$. So $|d'| = n$. Without loss of generality we assume that the first k conjuncts are inequalities and the last $n - k$ conjuncts are equalities, i.e.

$$d' = (x_1 \neq c_1 \wedge \cdots \wedge x_k \neq c_k \wedge x_{k+1} = c_{k+1} \wedge \cdots \wedge x_n = c_n).$$

By the definition of *negate*$'$ we know that $x_i = c_i$ is part of w'_i for $i \in \{1, \ldots, k\}$ and $x_i \neq c_i$ is part of w'_i for $i \in \{k+1, \ldots, n\}$. As before, let $G_i = \{w_{i,1}, \ldots, w_{i,l_i}\} = g^{-1}(w'_i)$.

By assumption $g(\cdot)$ satisfies condition *C1'* and so by Claim 1 the equality $x_i = c_i$ is part of all $w_{i,j}$, for $i \in \{1, \ldots, k\}, j \in \{1, \ldots, l_i\}$. For the same reason for $i \in \{k+1, \ldots, n\}$ the ws-descriptors $w_{i,j}$ contain equalities $x_i = c_{i,j}$, where $c_{i,j} \neq c_i$.

Assume an arbitrary instantiation d'_{inst} of d':

$$d'_{inst} = (x_1 = \hat{c}_1 \wedge \cdots \wedge x_k = \hat{c}_k \wedge x_{k+1} = c_{k+1} \wedge \cdots \wedge x_n = c_n),$$

where $\hat{c}_i \neq c_i$ for $1 \leq i \leq k$. We show that there exists a ws-descriptor $d \in$ *negate*(W), such that $\omega(d) = \omega(d'_{inst})$. Let

$$d = ((x_1 = \hat{c}_1)^{l_1} \wedge \cdots \wedge (x_k = \hat{c}_k)^{l_k} \wedge (x_{k+1} = c_{k+1})^{l_{k+1}} \wedge \cdots \wedge (x_n = c_n)^{l_n}),$$

in which exponentiation denotes multiple occurrence of an equality, connected by a logical conjunction. Obviously $\omega(d) = \omega(d'_{inst})$.

We argue that d is generated as a part of *negate*(W). The first k groups of equalities in d are generated because as argued before $x_i = c_i$ is part of all $w_{i,j}$ (for $1 \leq i \leq k$) and $\hat{c}_i \neq c_i$ (for $1 \leq i \leq k$). The last $n-k$ groups of equalities in d are generated because each $c_{i,j} \neq c_i$ (for $k+1 \leq i \leq n$) as shown before. So d is indeed part of *negate*(W).

Let D be the set of all ws-descriptors that are instantiations of d'. By Remark 5.2.4 we conclude $\omega(D) = \omega(d')$. For each $d \in D$ it holds that $g_{neg}(d) = d'$, therefore $D \subseteq \{d \mid g_{neg}(d) = d'\}$. It follows that $\omega(d') \subseteq \bigcup_{d:g_{neg}(d)=d'} \omega(d)$. We have chosen d' arbitrarily, therefore

$$\forall d' \in negate'(W') : \omega(d') \subseteq \bigcup_{d:g_{neg}(d)=d'} \omega(d)$$

So $g_{neg}(\cdot)$ satisfies condition *C2'*. We have shown that $g_{neg}(\cdot)$ satisfies both *C1'* and *C2'*, which concludes the proof of Claim 2.

Now we are ready to show that *set difference* preserves *C1* and *C2*. Recall that

T = **select** *diff*(D_1, W) , $\overline{T_1}$, $\overline{A_1}$ **from** R;
where W := **select** D_2 **from** S **where** $\overline{A_2} = \overline{A_1}$;
and *diff*$(d, W) := \{$*concat*$(d, d') \mid d' \in$ *negate*$(W), \psi(d, d')\}$.

Similarly T' is computed from R', S' and *negate*$'$. Consider an arbitrary tuple $t \in T$ and let $t.D_3 = (d_1 \wedge d_2)$. d_1 is the part of $t.D_3$ that originates in R and d_2 the part that originates in *negate*$(\cdot)$ (as defined by *diff*). It follows that there exists a tuple $r = (d_1, t.\bar{T}_3, t.\bar{A}_3)$ in R. Let W be the set of ws-descriptors of tuples in S that agree with r on the data attributes, i.e. the result of the subquery `select` D_2 `from` S `where` $\bar{A}_2 = \bar{A}_1$ relative to r. Let $r' = f_R(r)$. As f_R satisfies *C1* it follows that $r'.\bar{A}_1 = r.\bar{A}_1$. Let W' be the set of iws-descriptors of tuples in S' that agree with r'

on the data attributes, i.e. the result of the subquery `select` D_2 `from` S' `where` $\bar{A}_2 = \bar{A}_1$ relative to r'.

f_S satisfies *C1* and *C2*, the subqueries select by the same values $r'.\bar{A}_1 = r.\bar{A}_1$, hence there is a function $g : W \to W'$ that satisfies *C1'* and *C2'*. Using Claim 2 we conclude that there is a function $g_{neg} : \mathit{negate}(W) \to \mathit{negate}'(W')$ that satisfies *C1'* and *C2'*.

By the definition of *diff* we know that $d_2 \in \mathit{negate}(W)$. We define f_T as follows:

$$\text{Let } d_3 = (f_R(r).D_1 \wedge g_{neg}(d_2)).$$
$$f_T(t) := (d_3, t.\bar{T}_3, t.\bar{A}_3).$$

First we show that f_T satisfies *C1*. By definition of f_T it follows that the data attributes and the tuple ids of t and $f_T(t)$ are equivalent. Recall that $r.D_1 = d_1$. It holds that $\omega(t.D_3) \subseteq \omega(f_T(t).D_3)$ as

$$\omega(t.D_3) = \omega(d_1 \wedge d_2) \subseteq \omega(f_R(r).D_1 \wedge g_{neg}(d_2)) = \omega(f_T(t).D_3)$$

because f_R satisfies *C1* and g_{neg} satisfies *C1'*.

To see that f_T satisfies *C2* consider a tuple $t' \in T'$. Let $t'.D_3 = (d_1 \wedge d_2)$. d_1 is the set of (in-)equalities in $t'.D_3$ that originate in R' and d_2 is the set of (in-)equalities in $t'.D_3$ that originate in $\mathit{negate}'(W')$. By the definition of set difference $r' = (d_1, t'.\bar{T}_3, t'.\bar{A}_3) \in R'$ and $d_2 \in W'$. Let $\{r_1, \ldots, r_k\} = f_R^{-1}(r')$ and $\{w_1, \ldots, w_l\} = g_{neg}^{-1}(d_2)$. The following sequence of equations shows that f_T satisfies *C2*.

$$\begin{aligned}
\omega(d_1 \wedge d_2) &= \omega(r'.D_1) \cap \omega(d_2) \\
&\subseteq \bigcup_{1 \le i \le k} \omega(r_i.D_1) \cap \bigcup_{1 \le i \le l} \omega(w_i) \\
&\subseteq \bigcup_{1 \le i \le k, 1 \le j \le l} (\omega(r_i.D_1) \cap \omega(w_j)) \\
&\subseteq \bigcup_{1 \le i \le k, 1 \le j \le l, \omega(r_i.D_1) \cap \omega(w_j) \neq \emptyset} (\omega(r_i.D_1) \cap \omega(w_j)) \\
&\subseteq \bigcup_{t: f(t) = t'} \omega(t.D_3)
\end{aligned}$$

f_R satisfies *C2* and g_{neg} satisfies *C2'* which enables the step from the first to the second line. Then we apply distributivity of set intersection and set union. In line 4, omitting empty sets (the inconsistent ws-descriptors) in the union does not change the result of the union. These are exactly the ws-descriptors of the tuples in T that are mapped to t' by $f_T(\cdot)$: All combinations of tuples r_i and ws-descriptors w_j, where $\omega(r_i.D_1 \wedge w_j) \neq \emptyset$ (the condition ψ). So $f_T(\cdot)$ satisfies *C2*. We have shown that $f_T(\cdot)$ satisfies both conditions *C1* and *C2*. Set difference preserves the two conditions.

For every operator we have shown that there exists a function $f_T : T \rightarrow T'$, that satisfies the two conditions *C1* and *C2*. This completes the inductive step. As discussed in the beginning, *C2* implies surjectiveness. Hence $[\![Q(\mathbf{A})]\!]'$ consists of not more tuples than $[\![Q(\mathbf{A})]\!]$. $\square$

With this proof we have shown that by using iws-descriptors we never get larger query results than by using ws-descriptors. On the other hand we have shown that by using iws-descriptors the query results can be exponentially smaller. Together this means that iws-descriptors outpace ws-descriptors.

5.3 Descriptors with Intervals

Iws-descriptors can be exponentially more succinct than ws-descriptors when a difference operator or an update operation is involved. But there is a disadvantage: Whenever two iws-descriptors are joined, they have to be checked for feasibility. This is expensive to do, because a lookup in the world table is needed to check the domain. Iws-descriptors are so succinct because the inequalities allow to bypass the strictly conjunctive nature of ws-descriptors, when we regard an inequality as a shortcut for a disjunction over all the other domain values of a variable. The feasibility check is needed because the "allowed" domain values are described only indirectly by an inequality.

Avoiding the expensive feasibility check, but at the same time trying to keep the benefit of iws-descriptors, we introduce another type of world set descriptor building on intervals: the interval-ws-descriptor. Instead of equalities and inequalities an interval-ws-descriptor consists of intervals on variables.

Definition 5.3.1 (intws-descriptor)**.** Given a world table W, an interval-ws-descriptor (intws-descriptor) over W is a set of triples (v, lo, hi) that consist of a variable $v \in vars(W)$ with a lower bound *lo* and an upper bound *hi*, such that $lo \leq hi$, $lo, hi \in dom_W(v)$ and such that a variable occurs at most once in the set.

The intended meaning for intws-descriptors is that they define lower and upper bounds for variables. To emphasize this notationally we write for example

$$d = \big((1 \leq x \leq 3) \wedge (6 \leq y \leq 6)\big)$$

for the intws-descriptor $\{(x, 1, 3), (y, 6, 6)\}$. We say that d defines *intervals* on x ($[1, 3]$) and y ($[6, 6]$). Given an intws-descriptor d we write $|d|$ do denote the number of intervals d defines (its length). We assume an arbitrary order on the intervals an intws-descriptor d consists of to be able to address them. We write $d[i].var$ to refer to the variable of its i-th interval, $d[i].lo$ to refer to the lower bound of its i-th interval and $d[i].hi$ to refer to the upper bound of its i-th interval.

Two intws-descriptors are *consistent* if their intervals on the variables they have in common intersect. This is equivalent to the fact that none of all the pairwise intersections of intervals on the same variable is empty. The function ψ'' in Figure 5.8 formally defines consistency of two intws-descriptors. For example the intws-descriptors $((1 \leq x \leq 3))$ and $((2 \leq x \leq 4))$ are consistent. $((1 \leq z \leq 3))$ and $((5 \leq z \leq 7))$ are an example of two inconsistent intws-descriptors.

We define the semantics of intws-descriptors by extending the function ω to intws-descriptors. Given an intws-descriptor d, $\omega(d)$ is the set of total valuations that obey the lower and upper bounds defined in d.

Definition 5.3.2 (ω for intws-descriptors)**.** Given an intws-descriptor d over a world table W, $\omega(d)$ is the set of total valuations f over variables in W for which it holds that $\forall_{i<|d|}\big(f(d[i].\mathit{var}) \geq d[i].\mathit{lo} \wedge f(d[i].\mathit{var}) \leq d[i].\mathit{hi}\big)$.

For example: The intws-descriptor $\big((1 \leq x \leq 3) \wedge (6 \leq y \leq 6)\big)$ represents the set of valuations where $1 \leq x \leq 3$ and $y = 6$. It can be seen as a shortcut for the ws-set $\{(x = 1 \wedge y = 6), (x = 2 \wedge y = 6), (x = 3 \wedge y = 6)\}$. Analog to ws-sets we call sets of intws-descriptors *intws-sets*. The semantics of intws-sets is defined by further extending the function ω to intws-sets. Given an intws-set S, $\omega(S) = \bigcup_{d \in S} \omega(d)$.

Intws-descriptors make especially sense when the domains of variables do not contain gaps. This is the case because we assume a normalized world table. As for ws-descriptors, and in contrast to iws-descriptors, consistency of two intws-descriptors d_1 and d_2 already implies that $\omega(d_1) \cap \omega(d_2) \neq \emptyset$. No separate check for feasibility is necessary.

Definition 5.3.3. U^{int}-relations are U-relations based on intws-descriptors instead of ws-descriptors.

The semantics of U^{int}-relations follows from the definition of ω for intws-descriptors. Ws-descriptors can be seen as a special case of intws-descriptors, where the lower and upper bounds are equivalent. It follows that U^{int}-relations are a *complete representation system* for uncertain databases, because intws-descriptors extend the expressiveness of ws-descriptors. This means that every uncertain database can be represented using U^{int}-relations.

Queries on U^{int}-relations

In this section we show how to translate relational algebra queries with *poss* and *merge* into queries on U^{int}-relations. We use a translation function $[\![\cdot]\!]''$ to translate relational algebra with *poss* and *merge*. Let $[\![\cdot]\!]''$ be equivalent to $[\![\cdot]\!]$ (see Figure 3.5 and 4.6) except for the consistency check and the inverse. In Figure 5.8 we define the function ψ'' that checks the consistency of two intws-descriptors. It replaces ψ in $[\![\cdot]\!]''$. ψ'' checks

$$\psi''(d_1, d_2) := \bigwedge_{i<|d_1|, j<|d_2|} (d_1[i].\mathit{var} = d_2[j].\mathit{var} \rightarrow$$
$$d_1[i].\mathit{lo} \leq d_2[j].\mathit{hi} \wedge d_1[i].\mathit{hi} \geq d_2[j].\mathit{lo}).$$

Figure 5.8: Consistency check for intws-descriptors.

$$\begin{aligned}
&\text{Let } S = \{d_1, \ldots, d_n\}.\\
\mathit{negate}''(S) := \big\{ d = &((\mathit{lo}_{1,j_1} \leq d_1[j_1].\mathit{var} \leq \mathit{hi}_{1,j_1}) \wedge \cdots \wedge\\
&(\mathit{lo}_{n,j_n} \leq d_n[j_n].\mathit{var} \leq \mathit{hi}_{n,j_n})) \;\big|\\
&0 \leq j_i < |d_i|, 1 \leq i \leq n,\\
&(\mathit{lo}_{i,j_i} = 1 \text{ and } \mathit{hi}_{i,j_i} = d_i[j_1].\mathit{lo} - 1) \text{ or}\\
&(\mathit{lo}_{i,j_i} = d_i[j_1].\mathit{hi} + 1 \text{ and } \mathit{hi}_{i,j_i} = \mathit{max}(d_i[j_i].\mathit{var})),\\
&\psi''(d, d)\big\}
\end{aligned}$$

Figure 5.9: Negation for intws-sets.

consistency of two intws-descriptors by requiring that each intersection of intervals on the same variable is nonempty. In Figure 5.9 we define *negate*$''(\cdot)$ for the inverse of an intws-set. It replaces *negate*$(\cdot)$ in $[\![\cdot]\!]''$. *negate*$''(\cdot)$ is quite similar to *negate*$(\cdot)$ for ws-descriptors but the size of the inverses it produces does not depend on the domains of the occurring variables. Given a normalized world table, the inverse of an interval $(lo \leq x \leq hi)$ for a variable x are the two intervals $(1 \leq x \leq lo - 1)$ and $(hi + 1 \leq x \leq max(x))$. Both can be empty (in case $lo = 1$ or $hi = max(x)$). *negate*$''(\cdot)$ defines the inverse of an intws-set S that consists of n intws-descriptors. The inverse consists of all combinations of inverses of intervals in the intws-descriptors $d_1, \ldots, d_n$. The result is again an intws-set. To build its intws-descriptors, from each of the n ws-descriptors in S an interval is chosen and substituted by one of the two inverse intervals. ψ'' filters out inconsistent ws-descriptors.

Let us analyze the cardinality of the intws-sets generated by *negate*$''(\cdot)$. Let n be the number of intws-descriptors the input intws-set consists of and m the maximum length of the intws-descriptors. An intws-set generated by *negate*$''(\cdot)$ is maximal when the consistency check ψ'' always returns *true*. This is the case when in the input intws-set no variable occurs more than once. Then the cardinality of the inverse intws-set is in $O((2m)^n)$.

From the translation $[\![\cdot]\!]''$ it follows that

Theorem 5.3.4. *Positive relational algebra queries extended with poss and merge can be evaluated on U^{int}-relational databases in polynomial time data complexity.*

Proof. By Theorem 3.4.1 positive relational algebra queries extended with *poss* and *merge* can be evaluated on U-relational databases in polynomial time data complexity. On U^{int}-relations the translation of positive relational algebra queries extended with *poss* and *merge* differs only in the consistency check ψ''. The function ψ'' needs only polynomial time, as the function ψ. Hence positive relational algebra queries extended with *poss* and *merge* can be evaluated on U^{int}-relational databases also in polynomial time data complexity. □

When we concatenate two intws-descriptors then we can merge intervals on the same variable. For example consider the intws-descriptors $d_1 = (1 \leq x \leq 5)$ and $d_2 = (3 \leq x \leq 8)$. $concat(d_1, d_2) = (1 \leq x \leq 5 \wedge 3 \leq x \leq 8)$ describes the same worlds like the intws-descriptor $(3 \leq x \leq 5)$ where the two intervals on the variable x are intersected. By ordering the intervals of an intws-descriptor by the variable names we can check the consistency of two intws-descriptors in linear time, as we have done it for ws-descriptors.

5.4 Intws-descriptors outpace Ws-descriptors

In this section we compare U-relations to U^{int}-relations. We show that U^{int}-relations can be exponentially more succinct. The following theorem states that there are ws-sets S such that $|negate''(S)|$ is exponentially smaller than $|negate(S)|$.

Theorem 5.4.1. *There are ws-sets S such that $|negate''(S)| = 1$ and $|negate(S)| = (d-1)^{|S|}$, where d is the maximum domain size of the variables in S.*

Proof. Consider the ws-set $S = \{(x_1 = 1), \ldots, (x_n = 1)\}$ and let $max(x_i) = d$ for $1 \leq i \leq n$. It holds that $|negate(S)| = (d-1)^n$ because

$$\begin{aligned} negate(S) = \{&(x_1 = 2 \wedge x_2 = 2 \wedge \cdots \wedge x_n = 2),\\ &(x_1 = 3 \wedge x_2 = 2 \wedge \cdots \wedge x_n = 2),\\ &(x_1 = 2 \wedge x_2 = 3 \wedge \cdots \wedge x_n = 2),\\ &\qquad\vdots\\ &(x_1 = d \wedge x_2 = d \wedge \cdots \wedge x_n = d)\}. \end{aligned}$$

In contrast $|negate''(S)| = 1$ because
$negate''(S) = \{((2 \leq x_1 \leq d) \wedge \cdots \wedge (2 \leq x_n \leq d))\}$. □

It follows that using U^{int}-relations the result of a query with set difference can be exponentially more succinct than the result of the same query when using U-relations.

We believe that analogously to Theorem 5.2.5 it can be proven that the result of a query is never less succinct when using U^{int}-relations, compared to using U-relations. The proof has to be adopted for intws-descriptors which seems to be possible but not trivial. We skip it because it would go beyond the scope of this thesis.

5.5 Summary

I this chapter we have presented two new representation systems for uncertain databases: U^{i}-relations and U^{int}-relations. They extend U-relations by relying on iws- respectively intws-descriptors instead of ws-descriptors. We have described how relational algebra queries can be evaluated on U^{i}-relations and on U^{int}-relations.

Both of the new representation systems preserve the advantages of U-relations, namely polynomial time data complexity for positive relational algebra queries extended by the possible operator. We have shown that with both U^{i}-relations and U^{int}-relations the worst-case complexity for computing the inverse of a ws-set (negation) drops exponentially. This directly improves the computation of set difference and of updates. Furthermore we have shown, with a rather complex proof, that by using U^{i}-relations one never gets larger query results than by using U-relations.

While U^{i}-relations and U^{int}-relations are a big improvement, there is still a potential for further optimization, as the next chapter shows.

6. Optimization

In this chapter we describe different optimizations. We show how to optimize U-relational databases and we show how to improve set difference on U-relations. In both cases we consider U-relations, U^i-relations and U^{int}-relations.

With optimizing a U-relational database we mean to reduce the size, i.e. given a U-relational database *UDB* we look for a smaller U-relational database *UDB*$'$ such that $rep(UDB) = rep(UDB')$. To motivate why this is necessary we return to the recurring example about product ratings as introduced in Section 3.1. As in Section 4.1, assume there are only snow globes and no globes and therefore the following update was issued:

```
update U_P set Product='Snow globe' where Product='Globe';
```

The result of the update is the table U'_P:

U'_P	D	Tid	Product
	$()$	1	Mozart-CD
	$()$	2	Key ring
	$(y = 1)$	3	Snow globe
	$(y = 2)$	3	Snow globe

Figure 6.1: A U-relation that can be optimized.

It contains twice the value “Snow globe” for the tuple with id 3, with different ws-descriptors $(y = 1)$ and $(y = 2)$. Let us call the two tuples t_1 and t_2. Knowing that the domain of y is $\{1, 2\}$ we can optimize U'_P by replacing t_1 and t_2 with a tuple $s = ((), 3, \text{'Snow globe'})$. The tuple s is equivalent to the replaced tuples except for the ws-descriptor. But the ws-descriptors of t_1 and t_2 together represent the same worlds as the empty ws-descriptor in s, i.e. $\omega(t_1.D) \cup \omega(t_2.D) = \omega(s.D)$. Therefore this replacement is correct. In general we have n tuples that are equivalent except for the ws-descriptor. Their ws-descriptors make up a ws-set of cardinality n, which we want to optimize.

In the first section of this chapter we formally define the problem of minimizing ws/iws/intws-sets and prove its intractability. In Section 6.2 we describe two tractable optimization procedures. In Section 6.3 we present a special optimization problem for intws-descriptors and discuss its complexity. In Section 6.4 we show how the complexity of set difference can be reduced by integrating one of the tractable optimization procedures into the algorithm INV-NEGATE.

6.1 Optimizing Descriptors of World Sets

In this section we first focus on iws-sets. In the end we discuss that all the results also hold for ws-sets and intws-sets. As example, we consider an iws-set

$$S_1 = \{(x = 1), (x = 2), (x = 3)\}$$

and let $dom(x) = \{1, 2, 3\}$. The iws-set $S_2 = \{()\}$, consisting of one empty iws-descriptor, represents the same set of worlds as S_1, i.e. $\omega(S_2) = \omega(S_1)$. Using S_2 instead of S_1 to describe in which worlds a tuple exists is a clear optimization. Consider again the iws-set S_1, but now let $dom(x) = \{1, 2, 3, 4\}$. The iws-set $S_3 = \{(x \neq 4)\}$ is a shorter way to represent the same set of worlds. Only one iws-descriptor of length 1 is needed instead of three iws-descriptors of length 1. Now let $dom(x) = \{1, \ldots, 7\}$. Another way to represent $\omega(S_1)$ is the following iws-set

$$S_4 = \{(x \neq 4 \wedge x \neq 5 \wedge x \neq 6 \wedge x \neq 7)\}.$$

It consists of one iws-descriptor of length 4. Is it better to use S_1 or to use S_4? S_1 minimizes the total number of equalities and inequalities while S_4 minimizes the cardinality of the iws-set. We argue that the former is preferable.

When using positive queries iws-descriptors are checked for consistency/feasibility and are merged. The time needed to merge two iws-descriptors depends linearly on their length. The number of comparisons needed for the consistency/feasibility check also depends directly on the number of equalities and inequalities the two iws-descriptors consist of. Hence we should minimize the total number of equalities and inequalities an iws-set contains.

Definition 6.1.1 (Size of an iws-set)**.** The size of an iws-set is the total number of equalities and inequalities its iws-descriptors consist of.

Example: The size of the iws-set $S = \{(x = 1 \wedge y \neq 2), (z = 1), (x = 1 \wedge y = 3)\}$ is $2 + 1 + 2 = 5$. We are now ready to define the optimization problem:

IWS-SET MINIMIZATION [IWS MIN]

Instance: A world table W, an iws-set S over W and an integer k.

Question: Is there an iws-set S' over W of size at most k such that $\omega(S') = \omega(S)$?

We will show that IWS MIN is Σ_2^P-complete. To show Σ_2^P-hardness we reduce the Minimum Equivalent DNF problem (MIN DNF) [46], which is Σ_2^P-complete [49], to

IWS MIN. MIN DNF is a famous problem inspired by the minimization of Boolean functions and defined as follows.

> MINIMUM EQUIVALENT DNF [MIN DNF]
>
> *Instance*: A DNF formula ϕ over Boolean variables and an integer k.
>
> *Question*: Is there a DNF formula equivalent to ϕ that contains k or fewer occurrences of literals?

Lemma 6.1.2. *There is a polynomial time reduction from* MIN DNF *to* IWS MIN.

Proof. Consider an arbitrary instance $I_D = (\phi, k)$ of MIN DNF. Let $V = \{x_1, \ldots, x_n\}$ be the set of variables occurring in ϕ. ϕ is a disjunction of conjunctions of literals, i.e.

$$\phi = (a_{1,1} \wedge \cdots \wedge a_{1,l_1}) \vee \cdots \vee (a_{m,1} \wedge \cdots \wedge a_{m,l_m}).$$

The $a_{i,j}$'s are positive or negative literals over V. We construct an instance $I_S = (W, S, k)$ of IWS MIN. We define the world table

$$W := \{(x_1, 1), (x_1, 2), \ldots, (x_n, 1), (x_n, 2)\}.$$

It contains the same variables as V and each variable has the domain $\{1, 2\}$. Let S be a set of m iws-descriptors $d_1, \ldots, d_m$. To every term in ϕ we define a corresponding iws-descriptor d_i:

$$d_i := (\mathit{var}(a_{i,1}) = \mathit{val}(a_{i,1}) \wedge \cdots \wedge \mathit{var}(a_{i,l_i}) = \mathit{val}(a_{i,l_i})),$$

where $\mathit{var}(x)$ indicates the atom used in the literal x and $\mathit{val}(x)$ is 1 in case x is a positive literal and 2 otherwise.

Before showing the correctness of the reduction we want to discuss an important property. Each assignment of truth values to the variables in V corresponds to a total valuation of the variables in V to $\{1, 2\}$ (where *true* corresponds to 1 and *false* to 2). Let f be a function from truth assignments to valuations that turns truth assignments into total valuations by mapping *true* to 1 and *false* to 2. The inverse of f, f^{-1}, is well-defined. Due to the construction of S we conclude that

$$\text{a truth assignment } I \text{ satisfies } \phi \Leftrightarrow f(I) \in \omega(S), \text{ and}$$
$$T \in \omega(S) \Leftrightarrow f^{-1}(T) \text{ satisfies } \phi$$

We show the correctness of the reduction in two steps.

- Assuming that I_D is a positive instance of MIN DNF, we know that there is a formula ϕ' equivalent to ϕ that contains at most k literals. Equivalent means that ϕ and ϕ' are satisfied by the same truth assignments. In the same manner as we

constructed S from ϕ we construct an iws-set S' from ϕ'. Obviously the size of S' is at most k. We show that $\omega(S') = \omega(S)$. First consider an arbitrary valuation $T \in \omega(S)$. As argued above, $f^{-1}(T)$ satisfies ϕ. As ϕ and ϕ' are equivalent, $f^{-1}(T)$ also satisfies ϕ'. By construction of S' analogously to S, $f(f^{-1}(T)) = T \in \omega(S')$. Secondly we consider an arbitrary valuation $T \notin \omega(S)$. Then $f^{-1}(T)$ does neither satisfies ϕ nor ϕ'. $f(f^{-1}(T)) \notin \omega(S')$, therefore $T \notin \omega(S')$. It follows that $\omega(S') = \omega(S)$.

- Now assume I_S is a positive instance of IWS MIN. Then there is an iws-set S' of size $k' \leq k$, such that $\omega(S) = \omega(S')$. Without loss of generality S' consists of equalities only. Because of the binary domain every inequality can be replaced by an equality without changing the size of S'. From S' we construct a formula ϕ'. For every iws-descriptor $d_i = (a_{i,1} = c_{i,1} \wedge \cdots \wedge a_{i,l_i} = c_{i,l_i})$ in S', $1 \leq i \leq k'$, we define a corresponding term t_i:

$$t_i := \mathit{lit}(a_{i,1}, c_{i,1}) \wedge \cdots \wedge \mathit{lit}(a_{i,l_i}, c_{i,l_i}),$$

where $\mathit{lit}(x, c)$ is x in case $c = 1$ and $\neg x$ otherwise. Let $\phi' := t_1 \vee \cdots \vee t_{k'}$. It consists of $k' \leq k$ occurrences of literals. By the construction of ϕ' we conclude that

$$T \in \omega(S') \Leftrightarrow f^{-1}(T) \text{ satisfies } \phi'.$$

We show that ϕ and ϕ' are equivalent, i.e. that they are satisfied by the same truth assignments. First consider an arbitrary truth assignment I that satisfies ϕ. It follows that $f(I) \in \omega(S)$ and also $f(I) \in \omega(S')$ (because $\omega(S) = \omega(S')$). As we concluded $f^{-1}(f(I))$ satisfies ϕ'. $f^{-1}(f(I)) = I$, hence I satisfies ϕ'. Secondly we consider an arbitrary truth assignment I that does not satisfy ϕ. It follows that $f(I) \notin \omega(S)$ and also $f(I) \notin \omega(S')$. As we concluded $f^{-1}(f(I))$ does not satisfy ϕ'. $f^{-1}(f(I)) = I$, hence I does not satisfy ϕ'. It follows that ϕ and ϕ' are equivalent.

□

Lemma 6.1.3. IWS MIN *is in* Σ_2^P.

Proof. Consider an arbitrary instance (W, S, k) of IWS MIN. We guess an iws-set S' and check in polynomial time whether it contains at most k equalities and inequalities. With a coNP oracle we check that there does not exist a valuation w such that $w \in \omega(S)$ and $w \notin \omega(S')$ or vice versa. The check itself is feasible in polynomial time. Given w as a total ws-descriptor we just have to check that w is consistent with both S and S' or with neither of them. □

By taking into account that MIN DNF is Σ_2^P-complete and combining the two lemmas we get:

Theorem 6.1.4. IWS MIN *is* Σ_2^P*-complete.*

We define the analogous optimization problem for ws-sets and intws-sets. The size of a ws-set (intws-set) is the total number of equalities (intervals) its ws-descriptors (intws-descriptors) consist of.

WS-SET MINIMIZATION [WS MIN]

Instance: A world table W, a ws-set S over W and an integer k.

Question: Is there a ws-set S' over W of size at most k such that $\omega(S') = \omega(S)$?

INTWS-SET MINIMIZATION [INTWS MIN]

Instance: A world table W, an intws-set S over W and an integer k.

Question: Is there an intws-set S' over W of size at most k such that $\omega(S') = \omega(S)$?

As one can easily see, the reduction from MIN DNF also works for WS MIN and INTWS MIN. Due to the binary domain that we get by reducing an arbitrary instance of MIN DNF, any inequality can be represented by an equality. Similarly, intervals are not more powerful than equalities on a binary domain. To show that WS MIN and INTWS MIN are in Σ_2^P, the same guess-and-check algorithm can be used.

Corollary 6.1.5. WS MIN *and* INTWS MIN *are* Σ_2^P*-complete.*

The complexity class Σ_2^P contains NP and is assumed to be differend from NP. Therefore WS MIN, IWS MIN and INTWS MIN are presumably harder to solve than any problem in NP. Unless the polynomial hierarchy collapses, there does not exist any efficient algorithm that solves WS MIN, IWS MIN or INTWS MIN.

6.2 Tractable Optimization

In the previous section we have shown that deciding if there is a smaller ws/iws/intws-set that is equivalent to a given ws/iws/intws-set is intractable. It immediately follows

that finding the smallest ws/iws/intws-set which is equivalent to a given ws/iws/intws-set S is intractable. It is natural to ask for polynomial time algorithms that reduce the size of S, even if the result is not minimal. We assume that the equalities/inequalities/intervals of the descriptors in S are ordered by variable names as described in Section 3.4. For the complexity analysis of the proposed algorithms we let $n = |S|$ and $m = max\{|d| \mid d \in S\}$.

Subset Elimination

So let S be a ws-, iws- or intws-set. Assume there are two ws-descriptors $d_1, d_2 \in S$ such that d_2 represents a subset of worlds of d_1, i.e. $\omega(d_1) \supseteq \omega(d_2)$. Then we can safely use a ws-set $S' = S \setminus \{d_2\}$ instead of S. because $\omega(S) = \omega(S')$ by the definition of $\omega(\cdot)$ and $\omega(d_1) \supseteq \omega(d_2)$. We show how all ws-descriptors that describe subsets of worlds can be detected.

Detecting that $\omega(d_1) \supseteq \omega(d_2)$ is a matter of comparing d_1 and d_2. If every valuation in d_1 also occurs in d_2 then $\omega(d_1) \supseteq \omega(d_2)$. The following lemma states the relationship between the equalities two ws-descriptors d_1 and d_2 consist of and d_1 describing a subset of the worlds of d_2.

Lemma 6.2.1. *Consider two ws-descriptors d_1 and d_2 and regard them as sets. Then $d_1 \subseteq d_2 \Leftrightarrow \omega(d_1) \supseteq \omega(d_2)$.*

Proof. We proof the two directions separately.

$\Rightarrow$: Assume that $d_1 \subseteq d_2$. Then $d_2 = (d_1 \wedge E)$, where E is some set of equalities. By definition of ω it holds that $\omega(d_2) = \omega(d_1 \wedge E) = \omega(d_1) \cap \omega(E)$. It follows that $\omega(d_1) \supseteq \omega(d_2)$.

$\Leftarrow$: Assume that $d_1 \not\subseteq d_2$. Then there exists an equality $x = c$ in d_1 which does not occur in d_2. Either there exists an equality $x = c'$ with $c' \neq c$ in d_2, or the variable x does not occur in d_2. In the latter case we choose an arbitrary $c' \in dom(x), c' \neq c$. There is always such a c' because we do not have variables with a domain of size 1. It holds that $\omega(d_2 \wedge x = c') \subseteq \omega(d_2)$, but obviously not that $\omega(d_2 \wedge x = c') \subseteq \omega(d_1)$. Hence $\omega(d_1) \not\supseteq \omega(d_2)$. Note that $\omega(d_2 \wedge x = c') \neq \emptyset$ because d_2 and $x = c'$ are obviously consistent. □

Using sorted ws-descriptors this property can be checked in linear time (in m), as shown in Algorithm 5. If d_1 and d_2 are not consistent, we can stop. If d_1 and d_2 are consistent, we know that they assign the same value to variables that occur in both of them. The equalities in both ws-descriptors are iterated in parallel. In case d_1 contains a variable which does not occur in d_2 *false* is returned. The condition in the last line checks whether all equalities of d_1 were consumed.

Algorithm 5: CONTAINS(d_1, d_2)

Require: sorted ws-descriptors d_1, d_2
1: **if** not consistent(d_1, d_2) **then**
2: **return** false
3: **end if**
4: $i = j = 0$
5: **while** $i < |d_1|$ and $j < |d_2|$ **do**
6: **if** $d_1[i].var = d_2[j].var$ **then**
7: i++, j++
8: **else if** $d_1[i].var < d_2[j].var$ **then**
9: **return** false
10: **else**
11: j++
12: **end if**
13: **end while**
14: **return** $i == |d_1|$

We can compare all ws-descriptors in S pairwise, which results in $O(n^2m)$ time complexity for complete subset elimination on ws-sets. Note that subset elimination also eliminates duplicate ws-descriptors.

Subset elimination is also possible on intws-sets and iws-sets, with the same complexity. In case we use intervals subset elimination works slightly differently. When comparing the variables (line 6 in Algorithm 5) we additionally have to check whether the i-th interval of d_1 contains the j-th interval of d_2, i.e. whether $d_1[i].lo \leq d_2[j].lo$ and $d_1[i].hi \geq d_2[j].hi$.

When using iws-sets we have to consider the following facts. We always assume that the two iws-descriptors are consistent and that the constants are pairwise distinct.

- $\omega((x \neq b_1, \ldots, x \neq b_l, x \neq c)) \subseteq \omega((x \neq b_1, \ldots, x \neq b_l))$
- $\omega((x = a)) \subseteq \omega((x \neq b_1, \ldots, x \neq b_l))$
- $\omega((x = a)) \supseteq \omega((x \neq b_1, \ldots, x \neq b_l))$ if *dom*$(x) = \{a, b_1, \ldots, b_l\}$

Using these facts we can decide in linear time whether an iws-descriptor represents a subset of worlds of another iws-descriptor.

Merge of Similar Descriptors

When using iws-sets we can merge iws-descriptors that differ only in the assignment of values to one variable. For example: Assume *dom*$(x) = \{1, 2, 3, 4\}$ and

$$S = \{(x = 1 \wedge y = 1), (x = 2 \wedge y = 1)\}.$$

We can optimize S and merge its two descriptors into one, producing an iws-set $S' = \{(x \neq 3, x \neq 4, y = 1)\}$. It holds that $\omega(S) = \omega(S')$.

We describe the general case. Assume two iws-descriptors $d_1, d_2 \in S$ and suppose $\mathit{vars}(d_1) = \mathit{vars}(d_2) = \{x_1, \ldots, x_l\}$. Further suppose d_1 and d_2 have the same equalities and inequalities for $\{x_2, \ldots, x_l\}$. Then we can build a ws-descriptor d_3 such that $\omega(d_3) = \omega(d_1) \cup \omega(d_2)$ and continue with a ws-set $S' = S \setminus \{d_1, d_2\} \cup \{d_3\}$. Depending on whether x_1 occurs in an equality or inequality in d_1 and d_2, we differentiate between four cases.

- $d_1 = (x_1 = c_{a_1} \wedge E), d_2 = (x_1 = c_{a_2} \wedge E)$:
  We construct $d_3 := (x_1 \neq c_{b_3} \wedge \cdots \wedge x_1 \neq c_{b_k} \wedge E)$,
  where $\{c_{b_3}, \ldots, c_{b_k}\} = \mathit{dom}(x_1) \setminus \{c_{a_1}, c_{a_2}\}$.

- $d_1 = (x_1 = c_{a_1} \wedge E), d_2 = (x_1 \neq c_{b_1} \wedge \cdots \wedge x_1 \neq c_{b_l} \wedge E)$
  We construct $d_3 := (x_1 \neq c_{d_1} \wedge \cdots \wedge x_1 \neq c_{d_{l'}} \wedge E)$,
  where $\{c_{d_1}, \ldots, c_{d_{l'}}\} = \{c_{b_1}, \ldots, c_{b_l}\} \setminus \{c_{a_1}\}$.

- $d_1 = (x_1 \neq c_{b_2} \wedge \cdots \wedge x_1 \neq c_{b_l} \wedge E), d_2 = (x_1 = c_{b_1} \wedge E)$:
  This case is symmetric to case 2.

- $d_1 = (x_1 \neq c_{a_1} \wedge \cdots \wedge x_1 \neq c_{a_l} \wedge E), d_2 = (x_1 \neq c_{b_1} \wedge \cdots \wedge x_1 \neq c_{b_{l'}} \wedge E)$:
  We construct $d_3 := (x_1 \neq c_{d_1} \wedge \cdots \wedge x_1 \neq c_{d_{l''}} \wedge E)$,
  where $\{c_{d_1}, \ldots, c_{d_{l''}}\} = \{c_{a_1}, \ldots, c_{a_l}\} \cap \{c_{b_1}, \ldots, c_{b_{l'}}\}$.

Each time we can apply one of the cases we get a ws-set of smaller cardinality. Only in the first case the total number of equalities and inequalities can increase, i.e. $|d_3| > |d_1| + |d_2|$. We accept this because we possibly can merge the result d_3 with another ws-descriptor in the ws-set. Note that in all cases it can happen that we do not need the variable x_1 anymore, i.e. that $d_3 = (E)$.

Given sorted iws-descriptors d_1 and d_2, the initial checks, the case distinction and the construction of d_3 are possible in linear time in m. As in the subset elimination we can try to merge all pairs of iws-descriptors in S. Additionally, after merging two iws-descriptors into an iws-descriptor d_3 it can become possible to merge d_3 with another iws-descriptor in S. This can happen at most $n - 1$ times because after $n - 1$ merges only one iws-descriptor, which can be the empty iws-descriptor, is left. So altogether the time complexity of this merge optimization is bounded by $O(n^3 m)$.

When using intws-sets a similar merge optimization is possible. Consider

$$S = \{(1 \leq x \leq 5 \wedge 1 \leq y \leq 2), (3 \leq x \leq 7 \wedge 1 \leq y \leq 2)\}.$$

We can optimize S and merge its two descriptors into one, producing an intws-set

$$S' = \{(1 \leq x \leq 7 \wedge 1 \leq y \leq 2)\}.$$

It holds that $\omega(S) = \omega(S')$.

Let us consider the general case. Given an arbitrary intws-set S, we take two intws-descriptors $d_1, d_2 \in S$. Suppose $vars(d_1) = vars(d_2) = \{x_1, \ldots, x_l\}$. When d_1 and d_2 differ only in the interval on one variable x_1 and the two intervals on x_1 intersect or are consecutive, i.e. when

$$d_1 = (lo_1 \leq x_1 \leq hi_1 \wedge E), d_2 = (lo_2 \leq x_1 \leq hi_2 \wedge E) \text{ and}$$
$$lo_1 \leq hi_2 + 1 \wedge lo_2 \leq hi_1 + 1.$$

Then we can construct an intws-descriptor

$$d_3 = (min\{lo_1, lo_2\} \leq x_1 \leq max\{hi_1, hi_2\} \wedge E),$$

for which it holds that $\omega(d_3) = \omega(d_1) \cup \omega(d_2)$, and continue with a ws-set $S' = S \setminus \{d_1, d_2\} \cup \{d_3\}$. The time complexity of merging intws-descriptors until no more merge is possible is the same as for iws-descriptors, namely $O(n^3 m)$.

6.3 Optimizing Intervals

In the previous section we have described how intws-descriptors can be merged. This is only possible if the two intervals intersect or are consecutive. If there is a gap between the intervals they cannot be merged, which gives rise to another optimization problem. Assume the world table of a U-relational database contains a variable x and $dom(x) = \{1, \ldots, 10\}$. Using intws-descriptors we exploit that there is a linear ordering of the domain values: $\langle 1, 2, 3, \ldots, 10 \rangle$. To state that a tuple exists in the worlds where $x = 1$ or $x = 2$, one intws-descriptor, $(1 \leq x \leq 2)$, suffices. But now assume that a tuple t exists in the worlds described by $x = 1$ or $x = 4$. To represent t we need two intws-descriptors: $(1 \leq x \leq 1)$ and $(4 \leq x \leq 4)$. There is a gap, namely $\langle 2, 3 \rangle$. Without this gap, only one intws-descriptor would be needed.

The ordering of the domain values of a variable is only used to work with intervals. It carries no additional meaning, because the domain values are only used to identify a world, but do not describe any properties of the world. Therefore we can optimize the described case by defining a new linear ordering for the domain values of x, such that 1 and 4 are consecutive. So let us define a new linear ordering for the variable x: $\langle 1, 4, 2, 3, 5, \ldots, 10 \rangle$ (ignoring the - now puzzling - ordering of natural numbers). Given this new ordering, we can merge the two intws-descriptors of t into one intws-descriptor $(1 \leq x \leq 4)$. In practice it makes more sense to see the new ordering as a permutation and rename once all the occurrences according to the permutation. The

new linear ordering in the example is equivalent to the permutation σ:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 4 & 2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \end{pmatrix}$$

This way we stay with the ordering of the natural numbers, the new intws-descriptor for t is $(1 \leq x \leq 2)$. If there are occurrences of x in any other intws-descriptor, they have to be adapted too.

By applying a permutation to the domain values of a variable we can reduce the number of intws-descriptors needed to represent tuples. Of course, this can also make things worse. Maybe it decreases the number of intws-descriptors needed for one tuple and increases the number of intws-descriptors needed for another tuple. The first question is, whether there is an optimal permutation for a variable x such that there are no more gaps. The second question is whether we can minimize the number of gaps for a variable x and how complex this minimization is.

Let us formally define what a gap is.

Definition 6.3.1 (Gap)**.** Given a U-relational database D with intws-descriptors, there is a gap on a variable x of its world table if there is a relation R in D that contains tuples t_1 and t_2 such that the data attributes and the tuple ids of t_1 and t_2 are equivalent, the intws-descriptors of t_1 and t_2 are equivalent except for the intervals on x and the intervals on x neither do overlap nor are adjacent.

This view is complicated and therefore we abstract of U-relations and formulate the problem on sets. What we have is a variable x and sets $S_1, \ldots, S_m$ of tuples that coincide in all attributes except that the intervals on a variable x are different, i.e. the tuples in each of the m sets are equivalent (same table, same attribute values, almost same intws-descriptor), only the interval on the variable x can be different. We demand that m is minimal to avoid that each tuple gets its own set, i.e. tuples that only differ in the interval on the variable x end up in the same set.

For every set S_i of tuples we construct a set of domain values T_i. Let T_i be the set of domain values of x that are covered by intervals of the intws-descriptors in S_i, for $1 \leq i \leq m$. Note that $T_1, \ldots, T_m$ are all subsets of *dom*(x).

Now let us formally define the problem:

GAP OPTIMIZATION [GAP OPT]

Instance: An integer n defining a set $D = \{1, \ldots, n\}$, a family of non-empty subsets $T_1, \ldots, T_m \subseteq D$ and an integer k.

Question: Is there a permutation σ of D such that the total number of intervals needed to represent each of $\sigma(T_i)$, for $1 \leq i \leq m$, is at most $k + m$?

R	D	A
	$1 \leq x \leq 1$	a_1
	$3 \leq x \leq 4$	a_1
	$1 \leq x \leq 1$	a_2
	$4 \leq x \leq 4$	a_2
	$6 \leq x \leq 6$	a_2
	$5 \leq x \leq 8$	a_3

M	1	2	3	4	5	6	7	8
S_1	1	0	1	1	0	0	0	0
S_2	1	0	0	1	0	1	0	0
S_3	0	0	0	0	1	1	1	1

Figure 6.2: From GAP OPT to CBM.

The set D represents the domain of the variable x, the subsets represent the intervals that should be optimized and k is the maximum number of allowed gaps. In the optimal case every subset can be represented by one interval and the total number of intervals is m (and $k = 0$). The instance defined by the relation R in Figure 6.2 consists of three subsets: $\{1, 3, 4\}, \{1, 4, 6\}$ and $\{5, 6, 7, 8\}$. They can be represented by the matrix M.

We show that GAP OPT is equivalent to another, well-known problem about matrices.

CONSECUTIVE BLOCK MINIMIZATION [CBM] [31]

Instance: A binary matrix A of size $m \times n$ and an integer l.

Question: Is there a permutation π of the columns of A, such that the total number of blocks of consecutive 1's in the rows of $\pi(A)$ is at most l?

Lemma 6.3.2. *There is a polynomial time reduction from* GAP OPT *to* CBM.

Proof. Consider an arbitrary instance $I_G = (n, \{T_1, \dots, T_m\}, k)$ of GAP OPT. n defines the domain $D = \{1, \dots, n\}$. We construct an instance $I_C = (A, l)$ of CBM as follows. Let A be a matrix of size $m \times n$ and $a_{i,j} = 1$ iff $j \in T_i$ (otherwise 0). Set $l = k + m$.

Note that the rows of A are the subsets, the columns the domain values D and a 1 (0) at $a_{i,j}$ signifies that the i-th subset contains (does not contain) the value j. A block of 1's in the i-th row of A corresponds to a subset of T_i that is an interval. Further, a permutation of the columns of A corresponds to a permutation of D.

If I_G is a positive instance then there exists a permutation σ fulfilling the properties in question. At most $k + m$ intervals are needed to represent the by σ permuted subsets. As intervals correspond to blocks of 1's in the rows of A, $\sigma(A)$ has at most $k + m = l$ blocks of consecutive 1's.

Vice versa, if I_C is a positive instance then there exists a permutation π such that the total number of blocks of consecutive 1's in the rows of $\pi(A)$ is at most l. As blocks of

1's in the rows of A correspond to intervals, in summary at most $l = k + m$ intervals are needed to represent each of the permuted subsets $\sigma(T_i)$. □

Lemma 6.3.3. *There is a polynomial time reduction from* CBM *to* GAP OPT.

Proof. Consider an arbitrary instance $I_C = (A, l)$ of CBM, where A is a matrix of size $m \times n$. Let A' be the matrix that we get by removing all the rows of A that contain only zeros. A' is of size $m' \times n$ ($m' \leq m$). We construct an instance $I_G = (n, \{T_1, \dots, T_{m'}\}, k)$ of GAP OPT. Let $k = l - m'$, $T_i \subseteq \{1, \dots, n\}$ and $j \in T_i$ iff $a_{i,j} = 1$.

We use the restricted matrix A' so that the subsets $T_1, \dots, T_{m'}$ are non-empty. Rows that contain only zeros can be ignored because they never contain any blocks of 1's, no matter how they are permuted. As in the proof above a block of 1's in the i-th row of A' corresponds to a subset of T_i that is an interval, and a permutation of the columns of A' corresponds to a permutation of D. Again, a permutation σ is a witness of I_C iff it also is a witness of I_G. □

CBM is NP-complete [31]. Due to the two lemmas above it follows immediately that GAP OPT is NP-complete too. Unless P=NP there does not exist any efficient algorithm that solves GAP OPT. Recently a polynomial-time approximation algorithm was presented [23] that generates solutions for CBM that do not differ from the optimal solutions by more than 50%. It can be used to improve bad orderings of domain values. A special case remains: $k = 0$. Is there a permutation such that all the subsets can be represented by a single interval? On binary matrices this is known as the *consecutive ones property* [34] and can be solved in linear time [25]. Due to the first lemma this means that only linear time is needed to either find a satisfying permutation of the domain values of a variable x or to find out that there is none. To actually achieve linear time the subsets must not be represented by a matrix but for instance by arrays.

Let us summarize the results of this section:

Theorem 6.3.4. GAP OPT *is NP-complete and there exists a polynomial time 1.5 approximation algorithm. The special case $k = 0$ is solvable in linear time.*

6.4 Optimizing Set Difference

This section is about optimizing the set difference operator. With optimizing we mean to reduce the complexity of computing set difference and to reduce the size of the result. The source of complexity is the inverse of a ws-set. First we show how we can reduce the size of the ws-set we have to compute the inverse of. Then we show how the algorithm INV-NEGATE can be improved to generate equivalent, but smaller results in less time.

Difference of Ws-Sets

We describe two ways to reduce the size of a ws-set we have to compute the inverse of. Remember that ψ checks the consistency of two ws-descriptors and that in Section 4.4 we defined the difference between a ws-descriptor and a ws-set as follows:

$$\textit{diff}(d, W) := \{\textit{concat}(d, d') \mid d' \in \textit{negate}(W), \psi(d, d')\}.$$

We can ignore ws-descriptors in W that are inconsistent with d, i.e.

Lemma 6.4.1. $\omega(d) \setminus \omega(W) = \omega(d) \setminus \omega(W')$*, where* $W' = \{d' \in W \mid \psi(d, d')\}$.

Proof. The consistency check filters out ws-descriptors that are inconsistent with d. Let W'' be the set of ws-descriptors that are filtered out, i.e. $W = W' \cup W''$, $W' \cap W'' = \emptyset$ and $\omega(d) \cap \omega(W'') = \emptyset$. We have that

$$\omega(d) \setminus \omega(W) = \omega(d) \setminus (\omega(W') \cup \omega(W'')) = (\omega(d) \setminus \omega(W'')) \setminus \omega(W')$$

As stated $\omega(d) \cap \omega(W'') = \emptyset$, hence $\omega(d) \setminus \omega(W'') = \omega(d)$. It follows that $\omega(d) \setminus \omega(W) = \omega(d) \setminus \omega(W')$. □

We consider now the difference between two ws-descriptors d_1 and d_2. We can ignore equalities in d_2 that also occur in d_1. For example it holds that

$$\omega((x = 1)) \setminus \omega((x = 1 \land y = 2)) = \omega((x = 1)) \setminus \omega((y = 2)).$$

Removing the equality $x = 1$ from the second ws-descriptor does not change the result because $x = 1$ also occurs in the first ws-descriptor.

Definition 6.4.2 (remove)**.** Consider two ws-descriptors d_1 and d_2 and regard them as sets of equalities. We define *remove*$(d_1, d_2) := d_1 \setminus d_2$.

For example: *remove*$\big((x = 1 \land y = 2), (x = 1)\big) = (y = 2)$. The following lemma states that we can use *remove* to possibly reduce the length of the ws-descriptors in a ws-set that we want to subtract from a ws-descriptor.

Lemma 6.4.3. *$\omega(d) \setminus \omega(W) = \omega(d) \setminus \omega(W')$, where $W' = \{\textit{remove}(d', d) \mid d' \in W\}$.*

Proof. Let $W = \{d_1, \ldots, d_n\}$. Then $\omega(d) \setminus \omega(W) = \omega(d) \setminus \omega(d_1) \setminus \cdots \setminus \omega(d_n)$. Therefore it suffices to show that

$$\omega(d) \setminus \omega(d_i) = \omega(d) \setminus \omega(\textit{remove}(d_i, d)) \tag{6.1}$$

If d and d_i do not have any equality in common, then *remove*$(d_i, d) = d_i$ and equation 6.1 trivially holds. Assume d and d_i have exactly one equality $x = c$ in common, i.e. $d = (A \wedge x = c)$ and $d_i = (B \wedge x = c)$, where A and B are disjoint sets of equalities. Due to Lemma 6.4.1 it holds that

$$\omega(d) \setminus \omega(d_i) = \omega(d) \setminus \big(\omega(d_i) \cup \bigcup_{c' \in \textit{dom}(x), c' \neq c} \omega(B \wedge x = c')\big)$$

The ws-descriptor d_i and the ws-descriptors $(B \wedge x = c')$ differ only in the valuation of x and together they enumerate all domain values of x, therefore

$$\omega(d_i) \cup \bigcup_{c' \in \textit{dom}(x), c' \neq c} \omega(B \wedge x = c') = \omega(B)$$

It follows that $\omega(d) \setminus \omega(d_i) = \omega(d) \setminus \omega(B)$ which is equivalent to equation 6.1. If d and d_i have more than one equality in common we can apply the above argument sequentially and remove one equality after another. □

Using Lemma 6.4.1 and 6.4.3 we can redefine the difference between two ws-sets as follows:

$$\textit{diff}(d, W) := \{\textit{concat}(d, d') \mid d' \in \textit{negate}(W'), \psi(d, d')\} \text{ where}$$
$$W' = \{\textit{remove}(d', d) \mid d' \in W, \psi(d, d')\}.$$

With the consistency check and with removing equalities we possibly reduce the size of the ws-set W' of which we compute the inverse. The cardinality of W' and the length of its ws-descriptors influence (exponentially) the complexity of computing the inverse of W', hence this is a clear optimization. The consistency check and removing equalities is also possible when using iws-descriptors or intws-descriptors.

Skipping

We are going to describe an extension of INV-NEGATE that generates equivalent but smaller results, by incorporating subset elimination into INV-NEGATE. For the extension we can show a better exponential bound for the worst case complexity and we can show polynomial time complexity for specific instances.

Example 4.5.3 already showed that INV-NEGATE (see Algorithm 3) produces ws-sets that possibly contain ws-descriptors that describe (strict) subsets of worlds of other ws-descriptors in the result; i.e. given a ws-set S, $S^i = \text{INV-NEGATE}(S)$, there can be ws-descriptors $d_1, d_2 \in S^i, d_1 \neq d_2$ such that $\omega(d_1) \supseteq \omega(d_2)$. In Section 6.2 we described how S^i can be cleaned of unneeded ws-descriptors as d_2 in polynomial time. It would be even better to avoid generating these unneeded ws-descriptors to save time. We will present a heuristics to achieve that. As stated by Lemma 6.2.1 it holds that $\omega(d_1) \supseteq \omega(d_2) \Leftrightarrow d_1 \subseteq d_2$. By definition of INV-NEGATE the ws-descriptors in S^i have all the same length, so there are only two possibilities that $d_1 \subseteq d_2$: d_1 and d_2 are equivalent or there is an equality that occurs more than once in d_1. We focus on the latter. Recall that on the i-th level INV-NEGATE iterates the inverse equalities of the i-th ws-descriptor in the input ws-set. We say that INV-NEGATE "branches". The following example shall explain the idea of skipping.

Example 6.4.4. We consider the ws-set

$$S = \{(y = 3), (x = 2 \land y = 2), (z = 1)\} \text{ and let}$$
$$dom(x) = dom(y) = dom(z) = \{1, 2, 3\}.$$

Let us see what happens when we apply INV-NEGATE to S. At the first level of the recursion the inverse of the first ws-descriptor $(y = 3)$ is built. There are two cases: $y = 1$ and $y = 2$. We consider the first and get a temporary ws-descriptor $tmp^1 = (y = 1)$. At the second level of the recursion the inverse of the second ws-descriptor $(x = 2 \land y = 2)$ is built. There are four cases: $x = 1$, $x = 3$, $y = 1$ and $y = 3$. Except for the last all are consistent with tmp^1 from the first level. Therefore at the second level we get three temporary ws-descriptors $tmp^2_1 = (y = 1 \land x = 1)$, $tmp^2_2 = (y = 1 \land x = 3)$ and $tmp^2_3 = (y = 1 \land y = 1)$ which we merge to $tmp^2_3 = (y = 1)$. tmp^2_3 describes a superset of worlds compared to the worlds described by tmp^2_1 and tmp^2_2, i.e.

$$\omega(tmp^2_3) = \omega(tmp^1) \supset (\omega(tmp^1) \cap \omega(x = 1)) = \omega(tmp^2_1)$$
$$\omega(tmp^2_3) = \omega(tmp^1) \supset (\omega(tmp^1) \cap \omega(x = 3)) = \omega(tmp^2_2).$$

This is exactly because the equality $y = 1$ is a repetition and so $tmp^2_3 = tmp^1$.

At the third level the inverse of the third ws-descriptor $(z = 1)$ is built. There are two cases: $z = 2$ and $z = 3$. We consider the first. The temporary ws-descriptors tmp^2_1, tmp^2_2 and tmp^2_3 are extended with $z = 2$ to ws-descriptors

$$tmp^3_1 = (y = 1 \land x = 1 \land z = 2),$$
$$tmp^3_2 = (y = 1 \land x = 3 \land z = 2) \text{ and}$$
$$tmp^3_3 = (y = 1 \land z = 2).$$

Now tmp^3_3 describes a superset of worlds compared to the worlds described by tmp^3_1 and tmp^3_2, i.e. $\omega(tmp^3_3) \supset \omega(tmp^3_1)$ and $\omega(tmp^3_3) \supset \omega(tmp^3_2)$. This is because tmp^2_3 described

already a superset of worlds compared to tmp_1^2 and tmp_2^2, and the three ws-descriptors were extended by the same equality $z = 2$. It would have been the same in case of extending with $z = 3$. If we extend tmp_1^2, tmp_2^2 and tmp_3^2 with the same equalities then the extended tmp_3^2 will always describe a superset of worlds compared to the extended tmp_1^2 and tmp_2^2, as stated by the following lemma.

Lemma 6.4.5. *Given two ws-descriptors d_1 and d_2 and a set of equalities E, it holds that $\omega(d_1) \subseteq \omega(d_2) \Rightarrow \omega(d_1 \wedge E) \subseteq \omega(d_2 \wedge E)$.*

Proof. The correctness follows from $\omega(d_1 \wedge E) = \omega(d_1) \cap \omega(E)$ and $\omega(d_2 \wedge E) = \omega(d_2) \cap \omega(E)$. □

No matter how we extend tmp_1^2, tmp_2^2 and tmp_3^2, the first two will never describe more worlds than tmp_3^2. This means that already at the second level we can forget about tmp_1^2 and tmp_2^2 and do not need to continue the recursion for them. No matter what is appended to tmp_1^2 and tmp_2^2 in the deeper levels of the recursion, the extended tmp_1^2 and tmp_2^2 describe only a subset of worlds compared to the extended tmp_3^2. We say that we can *skip* the second level because we continue only with $tmp_3^2 = tmp^1$ and go on to the third level. No branching is needed on the second level.

When can we skip a level in general? The main point in the above example is that $\omega(tmp_3^2) = \omega(tmp^1)$, i.e. there is a branch such that the temporary ws-descriptor from the previous level does not get more restricted. This is the case when an equality of the temporary ws-descriptor is repeated (if we use iws- or intws-descriptors this is different as we will describe later). And when is an equality repeated? Assume the temporary ws-descriptor contains an equality $x = c$. This equality is repeated in a branch if the current ws-descriptor $d = S[level]$ contains an equality $x = c'$ such that $c' \neq c$. Recall Example 6.4.4 where the temporary ws-descriptor tmp^1 contains the equality $y = 1$ and the ws-descriptor of level 2 contains the equality $y = 2$.

CAN-SKIP, which is defined in Algorithm 6, checks whether such a repetition would be possible. We define an extended version of the negation algorithm in Algorithm 7. INV-NEGATE-SKIP calls CAN-SKIP to check whether a certain level of the recursion can be skipped, given the temporary ws-descriptor built so far. Skipping does not completely avoid the generation of ws-descriptors that describe subsets of worlds compared to other ws-descriptors in the result of INV-NEGATE-SKIP. Therefore we do subset elimination every time another ws-descriptor is added to the result (line 2 in Algorithm 7). The method *add-using-subset-elimination* adds ws-descriptor d to the result set *result* only if it does not describe a subset of worlds described by any of the ws-descriptors already in *result*. If d is added, then the method removes all ws-descriptors from *result* that describe subsets of d.

The effect of skipping depends on the order of the ws-descriptors in the ws-set. Consider the ws-set S defined in Example 6.4.4, but now with a different order as a

Algorithm 6: CAN-SKIP(S, *level*, *tmp*)

Require: ws-set S, integer *level*, ws-descriptor *tmp*

```
1: d = S[level]
2: for i = 0 to |d| − 1 do
3:    for j = 0 to |tmp| − 1 do
4:       if tmp[j].var == d[i].var ∧ tmp[j].val ≠ d[i].val then
5:          return true
6:       end if
7:    end for
8: end for
9: return false
```

Algorithm 7: INV-NEGATE-SKIP(S, *level*, *tmp*, *result*)

Recursive function: first call is INV-NEGATE-SKIP($S, 0, (), \{\}$)
Require: ws-set S, integer *level*, ws-descriptor *tmp*, ws-set *result*

```
 1: if level ≥ |S| then
 2:    return result.add-using-subset-elimination(tmp)
 3: else if CAN-SKIP(S, level, tmp) then
 4:    INV-NEGATE-SKIP(S, level + 1, tmp, result)
 5: else
 6:    d = S[level]
 7:    for e = 0 to |d| − 1 do
 8:       v = d[e].var
 9:       for i = 1 to max(v), i ≠ d[e].val do
10:          if consistent((v = i), tmp) then
11:             result = INV-NEGATE-SKIP(S, level + 1, concat(tmp, (v = i)), result)
12:          end if
13:       end for
14:    end for
15:    return result
16: end if
```

ws-set $S' = \{(x = 2 \wedge y = 2), (y = 3), (z = 1)\}$. When applying INV-NEGATE-SKIP to S' CAN-SKIP returns *true* on the second level, but only in case the temporary ws-descriptor built by the first level is $(y = 1)$. Altogether INV-NEGATE-SKIP builds less ws-descriptors when applied to S than when applied to S'. We propose to sort the ws-descriptors in the ws-set by length in ascending order. The idea of this heuristic is that the later a ws-descriptor occurs in the ws-set, the higher the chance is that it can be

skipped. By skipping long ws-descriptors more time is saved than by skipping short ws-descriptors (without considering the domain sizes of the occurring variables).

We analyze the complexity of INV-NEGATE-SKIP and consider an arbitrary ws-set S consisting of n ws-descriptors. Let m be the maximum length of the ws-descriptors and d the maximum domain size of the occurring variables. The complexity of CAN-SKIP is in $O(nm)$ and it is called on each of the n levels of the recursion, giving n^2m for all n levels. The complexity of the consistency check on line 10 is linear in the length of the temporary ws-descriptor which we can neglect compared to CAN-SKIP. As without skipping, n is a limit for the number of levels where we branch and also a limit for the length of the temporary ws-descriptor.

Lemma 6.4.6. *Consider a ws-set S. When applying* INV-NEGATE-SKIP *to S, the number of levels where skipping is not possible is at most $|vars(S)|$.*

Proof. Assume that we have had already $|vars(S)|$ levels where skipping was not possible. It follows that the temporary ws-descriptor *tmp* contains $|vars(S)|$ equalities, one for each variable in S. This means that we can either skip the current level or that all possible extensions $v = i$ are inconsistent with *tmp*. So either skipping is possible or the recursion stops because there is no consistent branch. □

By Lemma 6.4.6 the number of levels where we branch is also bounded by the number of variables occurring in S, i.e. $|vars(S)|$. Each time we branch we have at most md possibilities. Putting it together, the complexity of INV-NEGATE-SKIP without subset elimination is in $O((md)^k n^2 m)$, where $k = min(|vars(S)|, n)$. The complexity of the subset elimination depends quadratically on the number of ws-descriptors in the resulting ws-set. Remember that the complexity of INV-NEGATE is exponential in n. This means that if $|vars(S)|$ is much smaller than n, then INV-NEGATE-SKIP is much faster than INV-NEGATE.

So we have an exponential upper bound for the complexity of INV-NEGATE-SKIP. In case no variable occurs twice in the input ws-set we obviously get an exponential blowup, because skipping is only possible if a variable occurs in more than one ws-descriptor. Now we show that due to skipping the complexity can drop to polynomial time on specific instances.

Theorem 6.4.7. *There are ws-sets on which* INV-NEGATE *needs exponential time, while* INV-NEGATE-SKIP *finishes in polynomial time.*

Proof. Consider the following ws-set S:

$$S = \big\{(x_1 = 2, x_2 = 1, x_3 = 1, \ldots, x_{n-1} = 1, x_n = 1),\\ (x_1 = 1, x_2 = 2, x_3 = 1, \ldots, x_{n-1} = 1, x_n = 1),\\ \vdots\\ (x_1 = 1, x_2 = 1, x_3 = 1, \ldots, x_{n-1} = 2, x_n = 1),\\ (x_1 = 1, x_2 = 1, x_3 = 1, \ldots, x_{n-1} = 1, x_n = 2)\big\}$$

The n ws-descriptors use the same n variables in their equalities and each ws-descriptor assigns the constant 2 to a different variable. Let the domain of each variable be $\{1, \ldots, d\}$. The cardinality of S is n and each ws-descriptor has length n. Obviously applying INV-NEGATE to S needs exponential time in n. We show that applying INV-NEGATE-SKIP to S needs only polynomial time.

Assume that we apply INV-NEGATE-SKIP to S and that we are at an arbitrary level i ($1 \leq i \leq n$) of the recursion. This means we consider the i-th ws-descriptor in S. We choose an arbitrary equality with index j ($1 \leq j \leq n$), hence we decide a value for the variable x_j. We distinguish three cases.

- $x_j = 1$. This is only possible if $j = i$.
- $x_j = 2$ and $j < i$. Then all the following $n - i$ levels of the recursion can be skipped because they can also set $x_j = 2$.
- $x_j = 2$ and $j > i$. Then all the following $n - i$ levels of the recursion can be skipped, except for the j-th level. At the j-th level there remain $O(dn)$ possibilities. This is depicted by Figure 6.3.
- $x_j = c, c \in \{3, \ldots d\}$. This is possible for every i and j. By setting $x_j = c$ all the following $n - i$ levels of the recursion can be skipped because they can also set $x_j = c$.

Note that it is not possible to set $x_j = 2$ if $j = i$.

In the second, third and fourth case it is easy to see that at most polynomial time is needed for the rest of the recursion. Either all the remaining levels or all but one can be skipped. Skipping a level means that there is no branching on this level, hence we do not get exponential in the number of levels as long as only a constant number of levels is not skipped.

In the first case none of the following levels can be skipped. But the first case affects only one branch because it is only possible if $j = i$. □

Skipping is also possible when using iws- or intws-descriptors. The general structure of INV-NEGATE-SKIP remains the same. Of course, the consistency check on line

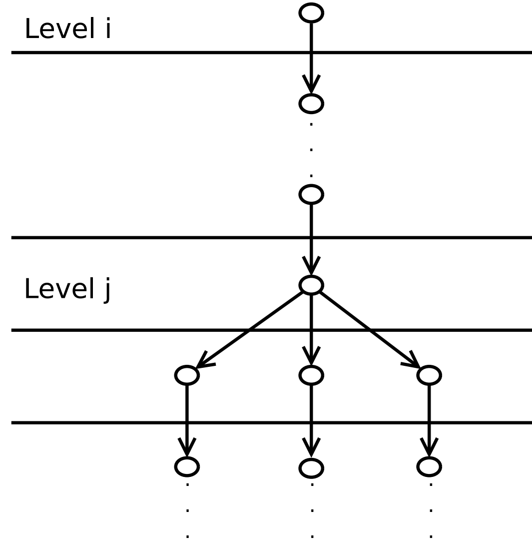

Figure 6.3: Except for level j, all levels below the i-th are skipped.

10 is a different one for iws- and intws-descriptors. The only thing that changes is the *if* condition in line 4 of CAN-SKIP. We have to check if there is a branch that does not further restrict the temporary descriptor *tmp*.

When using iws-descriptors this is the case if an (in-)equality is repeated, which happens if *tmp* contains an equality $x = c$ and the current descriptor $d = S[level]$ contains the inequality $x \neq c$ or vice versa. To check this we use the following condition on line 4 of CAN-SKIP:

$$tmp[j].var == d[i].var \land tmp[j].val = \text{-}d[i].val \qquad (6.2)$$

When using intws-descriptors a branch does not further restrict the temporary descriptor *tmp* if *tmp* contains an interval $(lo \leq x \leq hi)$ and the branch would extend *tmp* by an interval $(lo' \leq x \leq hi')$ such that $lo' \leq lo$ and $hi' \geq hi$. This happens if the current descriptor $d = S[level]$ contains an interval $c_1 \leq x \leq c_2$ and the intersection of $[c_1, c_2]$ with $[lo, hi]$ is empty. To check this we use the following condition on line 4 of CAN-SKIP:

$$tmp[j].var == d[i].var \land (tmp[j].lo > d[i].hi \lor tmp[j].hi < d[i].lo) \qquad (6.3)$$

The complexity bound we have shown for INV-NEGATE-SKIP with ws-descriptors does not hold in case of iws- and intws-descriptors. This follows from the conditions 6.2 and 6.3 which are “stronger” than the condition for ws-descriptors on line 4 of CAN-SKIP. Therefore fewer levels can be skipped. While the condition for iws-descriptors requires a value to be repeated (negated), the condition for intws-descriptors

only requires two intervals to have an empty intersection which is more likely in average. Is this relevant in practice? Our experiments will shed light on this question.

However, analogously to Theorem 6.4.7 (using the same exemplary ws-set) it can be shown that there are iws-set and intws-sets on which INV-NEGATE-SKIP (adapted for iws-/intws-descriptors) finishes in polynomial time, whereas INV-NEGATE needs exponential time.

In summary, skipping lowers the complexity of negation. With skipping we can generate equivalent but smaller inverses in less time.

6.5 Summary

After updating an uncertain database the representation is not necessarily optimal. We have shown that the problem of minimizing U-relations, U^i-relations and U^{int}-relations is Σ_2^P-complete, i.e. presumably harder than any problem in NP. In consequence we have presented two tractable optimization methods: *subset elimination* and *merging*. While the complexity of subset elimination is quadratic, the complexity of merging is cubic. With *skipping* we have integrated subset elimination into the algorithm INV-NEGATE. As a result of this integration, we have shown a better worst-case complexity for the extended algorithm INV-NEGATE-SKIP. Furthermore we have shown that the gap optimization problem – a special optimization problem on U^{int}-relations – is NP-complete.

So far, we have defined updates and set difference for U-relations, proposed new representations that speed up set difference and thus also updates, and we have discussed various optimization measures. After all the theoretic work we want to point out the practical value of our contributions. Therefore we have implemented a prototype. In the following two chapters we describe the prototype and discuss the results of the experimental evaluation.

7. Implementation

Based on the theoretical work described in this thesis we created a prototype of an uncertain database management system. The prototype extends MayBMS [28] which is the practical realization of U-relations. It is a probabilistic database management system based on PostgreSQL [39], a relational database management system. MayBMS is open source and the source code is available under the BSD license [33]. We use the latest version 2.1-beta, which is built on PostgreSQL version 8.3. MayBMS and PostgreSQL are both written in C and we also use C for our extensions. We modified MayBMS and created three variants: one using U-relations, one using U^i-relations and a third using U^{int}-relations.

MayBMS works by rewriting positive relational queries extended by the repair-key and the possible operator into pure SQL queries. It is integrated into the query engine of PostgreSQL. The main difference to the theoretical work of Antova et al. [6] is that in MayBMS relations are not transparently vertically partitioned. Nevertheless, the user can utilize the advantages of vertical partitions by adapting the table schemas and the queries manually. We kept it this way.

The main achievement of our extension of MayBMS is the implementation of the SQL *except* operator for uncertain tables. With it MayBMS now supports set difference and arbitrary updates. We implemented the except operator for U-relations, U^i-relations and U^{int}-relations, using the concepts described in the previous chapters: negation, subset elimination and skipping.

In the original MayBMS ws-descriptors are represented by several columns. For each equality of the ws-descriptor one column is needed for the variable name and one column for the value. We introduced a new data type *Wsdescriptor* that encapsulates an arbitrarily long ws/iws/intws-descriptor. Using the new data type one column is enough to represent any ws/iws/intws-descriptor. This is necessary because the except operator produces descriptors whose length is not based on the lengths of the descriptors in the input relations. Another advantage of this new data type is that the rewriting becomes less complex. Besides these changes we fixed a set of bugs in MayBMS.

The following Section 7.1 provides information about the new data type. Section 7.2 presents the general architecture of PostgreSQL and we describe where we have extended it. In Section 7.3 we describe why we had to reimplement the repair-key operator from scratch and how we did it. In Section 7.4 we discuss the implementation of the except operator and in Section 7.5 we give more detailed information on the code structure. Section 7.6 is about the correctness of our code.

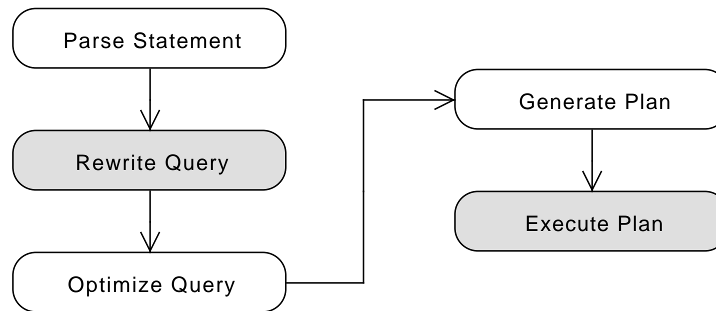

Figure 7.1: Query processing in PostgreSQL.

7.1 Data Type Wsdescriptor

We use the term *descriptor* to refer to the general data type that can represent either a ws-descriptor, an iws-descriptor or an intws-descriptor. As described in Section 3.4 we order the (in-)equalities/intervals of a descriptor by the variable names.

A ws-descriptor can be seen as a list of equalities between variables and respective domain values. We use positive integers for both variable names and domain values. Iws-descriptors consist of equalities and inequalities. We encode inequalities by negating the domain value, i.e. the inequality $x \neq 5$ is encoded by the pair $(x, -5)$. Intws-descriptors are lists of variable names with lower and upper bounds.

When concatenating two descriptors we optimize the result. Double occurrences of identical equalities/inequalities/intervals are skipped. In case of iws-descriptors an inequality $x \neq c$ is skipped in presence of an equality $x = c'$ ($c \neq c'$) on the same variable. In case of intws-descriptors intervals on the same variable are intersected, for example $((1 \leq x \leq 5) \land (3 \leq x \leq 7))$ is turned into $(3 \leq x \leq 5)$.

7.2 Architecture

We describe at which points we intercept PostgreSQL/MayBMS. Figure 7.1 depicts how the database server processes queries received from clients. The gray background marks the steps where our main changes occur. The queries are first parsed and analyzed. The rewriting step is traditionally used to rewrite queries on views to queries on the tables behind the views. MayBMS rewrites queries on uncertain tables and queries containing one of the additional operators (repair-key, possible) according to the translation function $[\![\cdot]\!]$ presented in Chapter 3. We changed the rewriting such that it considers the new data type instead of multiple columns to represent descriptors for world sets.

In the next step queries are optimized, i.e. the best table join order is computed.

Person	Id	Name
	1	Martin
	1	Thomas
	2	Ali
	2	Ruslan
	3	Jussi

Person'	D	Id	Name
	(x=1)	1	Martin
	(x=2)	1	Thomas
	(y=1)	2	Ali
	(y=2)	2	Ruslan
	()	3	Jussi

Figure 7.2: The relation Person before and after being repaired.

Then a query plan is generated which means that the optimized query is transformed into a tree of execution nodes. An execution node is for example a node that reads tuples from disk, a node that performs a nested loop join or a node that sorts a relation. In this step we create execution nodes for repair-key and uncertain except, if needed.

Finally the query plan is executed. Here the results of the query are generated. The next two sections describe our implementation of the repair-key node and the uncertain except node.

7.3 Repair-Key

The repair-key operator is essential to introduce uncertain data in an uncertain database. In Chapter 3 we have described that it is part of world-set algebra [29], a powerful query algebra for uncertain databases. It can be applied to certain tables to resolve violations of functional dependencies. Let us have a look at a small example. Consider a table $Person[Id, Name]$ where the functional dependency $Id \to Name$ is violated, as shown in Figure 7.2. The person with id 1 has two names (Martin and Thomas) and so does the person with id 2. To repair the table we issue the query

```
repair key id in Person;
```

The result of it is the table $Person'$. The two violations are repaired by introducing two new variables x and y that distinguish the possible cases. Now in each of the four worlds the functional dependency $Id \to Name$ is satisfied.

In the MayBMS-Project repair-key is implemented via query rewriting. On the one hand this is a nice approach because rewriting can be done outside the core of the database system (the execution step in Figure 7.1). On the other hand there are severe disadvantages. Side-effects like changing the world table are not possible, and due to the limits of rewriting the MayBMS implementation has to sort the input relation repeatedly.

To be able to compute the inverse of a ws/iws/intws-set the domains of the occurring variables need to be known. Hence a materialized world table is needed and it has to

Algorithm 8: REPAIR-KEY(R, K)

```
Require: certain table R, set of key attributes K ⊆ sch(R)
 1: result = empty table with schema sch(R) ∪ {_wsdesc}
 2: R.sort_by(K)
 3: while R.hasNext() do
 4:    tuple = R.next()
 5:    count = 1
 6:    while R.hasNext() ∧ R.next().K == tuple.K do
 7:       count++
 8:    end while
 9:    R.goBackBy(count+1)
10:    var = new and unused variable name
11:    for i = 1 → count do
12:       tuple = R.next()
13:       wsdescriptor = (var, i)
14:       result.insert(tuple, wsdescriptor)
15:       worldTable.insert(var, i)
16:    end for
17: end while
18: return result
```

be filled by repair-key. That is why we had to implement repair-key as a new operator in the core of the database system. REPAIR-KEY is depicted in Algorithm 8. We describe it shortly. The result relation contains the same tuples as the input relation, but augmented by ws-descriptors such that the functional dependency $K \rightarrow sch(R)$ is maximally repaired. First (line 2) the input relation is ordered by the key attributes so that tuples with the same keys can be accessed sequentially. The outer while loop is used to iterate all tuples in the ordered input relation. The first part within the loop (line 4-9) counts the number of tuples with equivalent key attributes. Let us call them the current group. In the second part (line 10-16) the tuples of the current group get annotated with ws-descriptors. For each group a new variable is introduced (line 10) and for each tuple within the group another domain value is introduced (the variable i in line 11). We use an SQL sequence to keep track of used variable names and to get new variable names. The tuple, now augmented with a ws-descriptor, is added to the result relation and the world table is updated. Note that we actually use empty ws-descriptors if the size of the current group is only 1. In this way we avoid needless variables with a domain size of only 1.

Algorithm 9: EXCEPT(R, S)

Require: tables R and S, both of schema [_*wsdesc*, $\bar{A}$]
1: result = empty table of schema [_*wsdesc*, $\bar{A}$]
2: T = append(R, S)
3: T.sort_by($\bar{A}$)
4: **while** T.hasNext() **do**
5: data = T.peek().$\bar{A}$
6: W_R, W_S = empty set
7: **while** T.hasNext() $\wedge$ T.peek().$\bar{A}$ == data **do**
8: t = T.next()
9: **if** t comes from R **then**
10: W_R.add(t._*wsdesc*)
11: **else if** t comes from S **then**
12: W_S.add(t._*wsdesc*)
13: **else**
14: break
15: **end if**
16: **end while**
17: W = wsset_diff(W_R, W_S)
18: **for** d in W **do**
19: result.insert(d, data)
20: **end for**
21: **end while**
22: **return** result

7.4 Except

The except operator computes the difference of two tables. If both tables are certain then we use the original except operator of PostgreSQL. In case any of the two input tables is uncertain we use our implementation of except for uncertain tables If one table is certain and the other one uncertain then we make the first one "uncertain" by adding empty ws-descriptors to the tuples.

We implemented except for uncertain tables as a new operator in the core of the database system. EXCEPT is depicted in Algorithm 9. It expects two tables of the same schema as arguments and the result relation has this schema too. First we create a table T which contains all the tuples of R and S. Note that each tuple in T gets extended by a flag that describes whether the tuple originates in R or S. To append R to S we use the implementation of the union operator of PostgreSQL. The tuples in T get sorted by the data values $\bar{A}$ so that we can access all tuples that have the same data values sequentially.

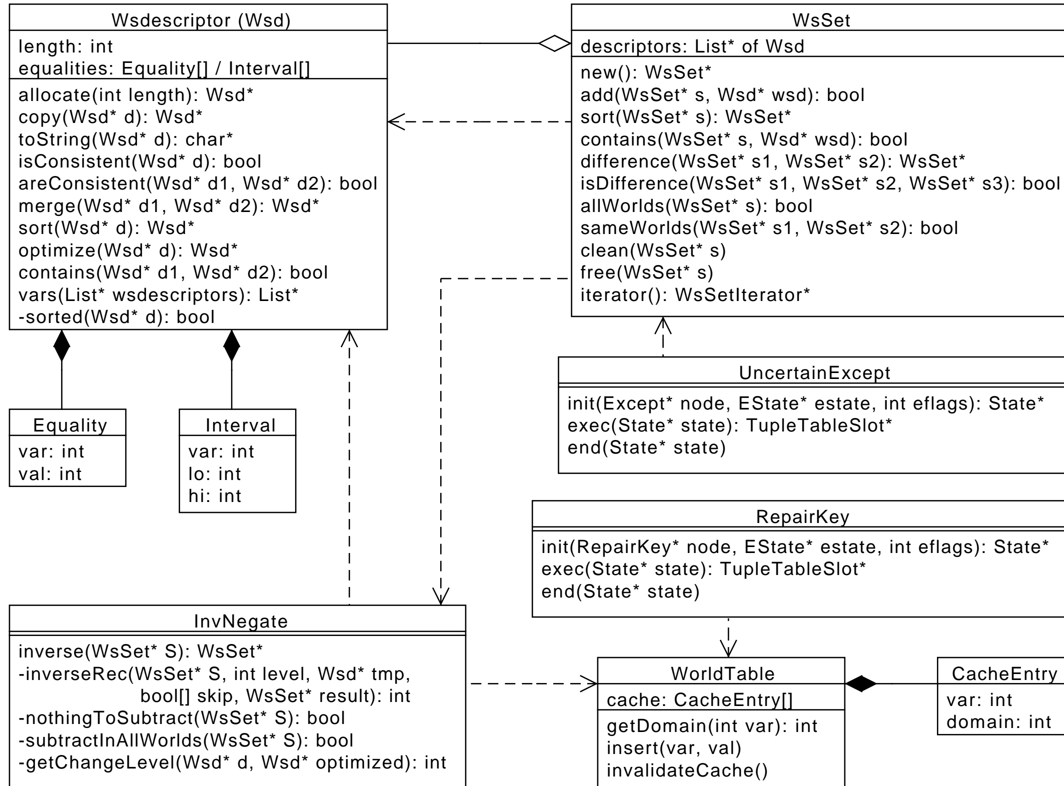

Figure 7.3: Code Structure.

To sort T we use the implementation of the order by operator of PostgreSQL. The outer while loop is used to iterate all tuples in T. The inner while loop collects the descriptors of tuples that have the same data values. The descriptors of tuples in R are added to W_R, whereas the descriptors of tuples in S are added to W_S. For this differentiation we use the before mentioned flag. Then we compute the set W which is the difference of W_R and W_S (line 17). For each descriptor in W we add one tuple to the result relation.

7.5 Details

Our implementation uses C which is purely iterative. Nevertheless we applied object-oriented thinking to achieve a better modularity and abstraction. We use C structures to model objects and pass them as first argument to “member” methods.

A configuration option (the C macro `WSD_TYPE`) defines whether the database uses ws-descriptors, iws-descriptors or intws-descriptors. Most of the methods we describe exist in three variants, one for each descriptors.

The diagram in Figure 7.3 depicts the parts of our code which are interesting for the discussion. Note that it is not complete. The diagram follows the style for UML class diagrams. Filled respectively empty diamonds represent composition respectively aggregation. The dashed arrows denote usage, for example the methods of *WsSet* call the methods of *Wsdescriptor*. The main structures are *Wsdescriptor*, *WsSet* and *WorldTable*.

A *Wsdescriptor* is composed either of equalities (which can also represent inequalities through negative values) or intervals, depending on the configuration option mentioned above.

A *WsSet* is a list of elements of type *Wsdescriptor*. Its *add* function adds a descriptor to the set only if needed (by checking with *contains*). The *contains* function checks for a descriptor d whether the set contains a descriptors d' such that $\omega(d) \subseteq \omega(d')$. The functions *allWorlds* and *sameWorlds* are intended for debugging. They decide whether a set describes all worlds respectively whether two sets describe the same worlds. To do this they iterate all total descriptors relative to the occurring variables, like the algorithm INV-DECOMPRESS. To compute the *difference* of two sets we use the *InvNegate* component. *isDifference* serves to check the correctness of the result of *InvNegate*, as we will describe in the next section. *inverseRec* in *InvNegate* is the implementation of the algorithm INV-NEGATE. To speed up skipping we use a precomputed array *skip*.

The component *UncertainExcept* encapsulates the algorithm EXCEPT and the component *RepairKey* the algorithm REPAIR-KEY. Both are execution nodes in the query plan, like for instance nodes that perform a nested loop join or that sort a relation. The actual work is done in the *exec* methods which get tuples from child nodes and return the processed tuples to parent nodes. In both cases the child node is a sort node because both EXCEPT and REPAIR-KEY need a sorted input.

WorldTable is the internal interface to the special relation `_world` which holds the world table. It uses SQL queries to find out the domain of a variable and to add new variable/value combinations to the world table. The queries are executed via the server programming interface (SPI) of PostgreSQL. The domains of variables are needed to compute the inverse of a ws/iws/intws-set and in the feasibility check for iws-descriptors. To speed up queries for the domains of variables we implemented a cache for them. Every time the world table is changed (by an insert, update or delete statement) the cache is invalidated.

7.6 Correctness

It is important to ensure the correctness of the code to get valid experimental results. We cannot prove total correctness of the code but we can take measures to greatly reduce the probability of errors.

- Regression tests. We manually defined test cases and expected results to check the correctness of the implemented code on a high level (SQL). For example we create two uncertain relations R and S and verify the result of `select * from R except select * from S`. Besides simple test cases we identified many corner cases.
- Assertions. We augmented the code with assertions to verify properties of data structures. For example every time we access a ws/iws/intws-descriptor we check its consistency and whether its equalities are sorted by the variable names.
- Dual computation. The most complex and critical code is involved in the computation of the inverse of a ws/iws/intws-set. Therefore it makes sense to double-check it. Besides INV-NEGATE we have described INV-DECOMPRESS as a naive method to compute the inverse. While INV-DECOMPRESS is not useful in practice it can be used to verify the results of INV-NEGATE (as long as the number of variables is limited). We use large numbers of randomly generated ws/iws/intws-sets and check if the worlds described by the result of INV-DECOMPRESS and INV-NEGATE are equivalent. This complements the regression tests which work only with manually defined test cases.

In the productive code used for the experiments dual computation and assertions are deactivated because they would influence the measured time values.

7.7 Summary

To point out the practical value of our contributions, we have implemented a prototype based on MayBMS. In total about 10000 lines of code have been added or modified. In this chapter we have described the prototype in which we had to omit many details. For instance we have not discussed the new rewriting system because it requires in-depth knowledge of the PostgreSQL query processing.

We have presented in more detail the implementation of the repair-key operator and the implementation of the except operator for uncertain relations. And we have also discussed the measures we have taken to ensure the correctness of the code. Besides regression tests and assertions, the dual computation of the critical functions helps to greatly reduce the probability of errors.

With the prototype we have performed a set of experiments to evaluate the practicability of our approach. We describe the results in the next chapter.

8. Experimental Results

In this chapter we show the practicability of our approach for updates and set difference on uncertain databases, and we experimentally compare U-relations to U^i-relations and U^{int}-relations. We update tables and measure the time needed for the updates and the size of the updated table. Then we measure the time needed for select-join queries on the updated tables to determine whether using iws/intws-descriptors is beneficial in later queries. The expectation is that with iws/intws-descriptors we can compute set difference and updates faster than with ws-descriptors, because the updated tables are smaller, as shown theoretically in Chapter 5. When using iws-descriptors feasibility has to be checked at every join, which means that it is necessary to look up the domains of the involved variables in the world table. We examine how this affects the time needed to answer queries.

Unfortunately there does not exist any benchmark for uncertain databases yet. Many interesting applications for uncertain and probabilistic databases have been described, and the approaches to solve them are manifold. As a result the field of research has not agreed yet upon one or a few typical use cases which could be treated in a benchmark. For certain databases the Transaction Processing Performance Council [35] (TPC) is accepted widely as an organization defining benchmarks. One of its benchmarks is the decision support benchmark TPC-H [48]. It defines a database schema that mimics typical business databases about products, suppliers and more. We use the database schema defined in the TPC-H benchmark and a modified version of the TPC-H data generator [6] that generates uncertain databases.

Section 8.1 gives more details on the test system and the test data. In Section 8.2 we present the results for update queries, whereas in Section 8.3 we present the result for queries with set difference. In Section 8.5 we discuss the experimental results. As we have also implemented the repair-key operator, we compare our implementation with the original implementation of the MayBMS project in Section 8.4.

8.1 Test System and Test Data

All the queries are run in the psql shell of PostgreSQL and the answers of the queries are materialized into a new table (by using `create table as select ...`). We run the queries ten times and report the mean evaluation time, using the PostgreSQL statement `\timing`. After loading the test data into the database and before running the update queries we call the PostgreSQL statement `vacuum analyze` to have more

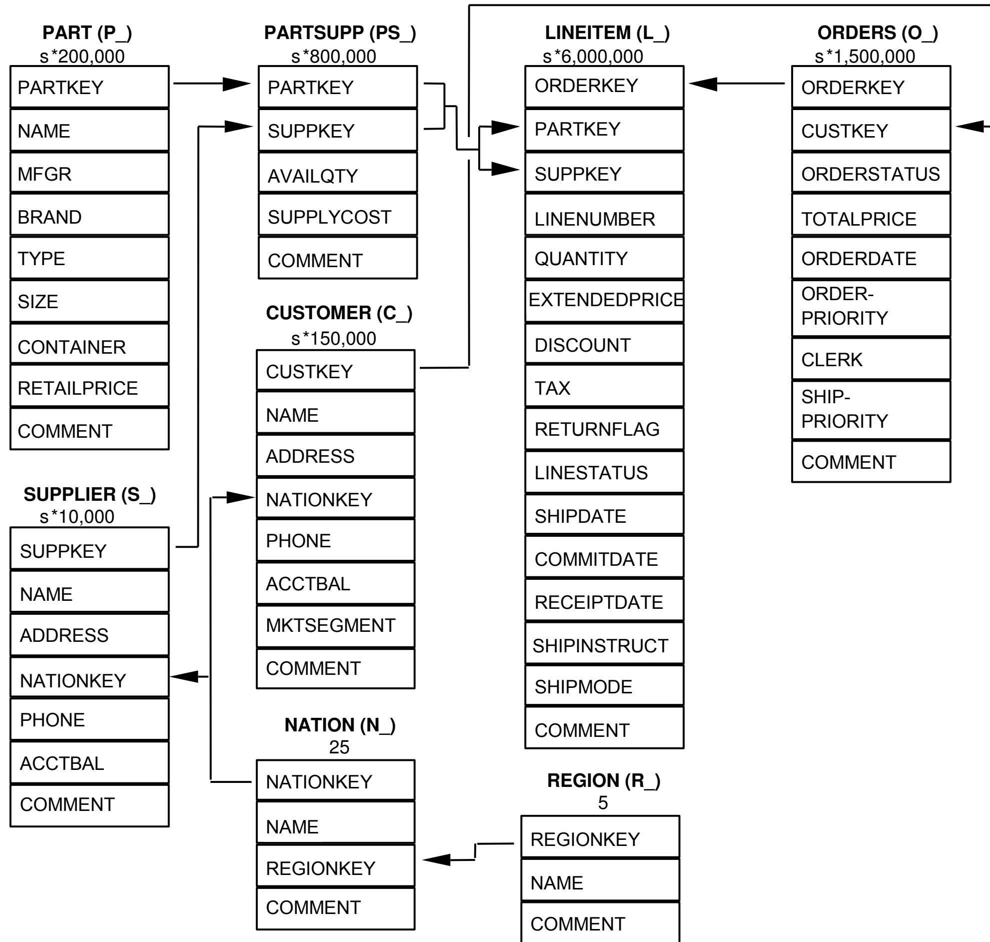

Figure 8.1: TPC-H Schema [48].

similar initial conditions (e.g.: the tables are pruned of “dead” tuples). The code is compiled and all the experiments are run on the following test system:

- Linux version 2.6.37 SMP PREEMPT
- gcc version 4.5.1
- Pentium(R) Dual-Core CPU E5200 @2.50GHz
- 2 GB RAM

Figure 8.1 shows the schema of the TPC-H benchmark which we use for our queries. The arrows denote foreign keys and the numbers denote the numbers of tuples in the relations, mostly multiplied with the scale factor s. The parentheses after the table names denote the prefixes used for the column names. For example the table *Part* has the attribute *p_partkey*.

For each attribute of the relations in the schema we use a separate vertical partition. For instance the table *partsupp* is represented by five vertical partitions in our test cases. To speed up the join of the vertical partitions we use indexes on the tuple id attributes.

To create the test databases we use the data generator described in [6]. We vary three parameters:

- Scale factor s. It is identical to the TPC-H scale factor and defines the size of a single possible world. A scale factor of 1 corresponds to a database of approximately 1GB. The number of tuples in the individual tables can be read off Figure 8.1. We vary the scale factor from 0.01 to 1.
- Uncertainty ratio x. It denotes the ratio of data fields (in the vertical partitions) that are uncertain, i.e. that do not have the same values in all worlds. We vary the uncertainty ratio from 0.001 to 0.2.
- Maximum alternatives per attribute m. This is the maximum number of different possible values an attribute can have. We vary m from 2 to 10.

Note that m influences the domain sizes of the variables indirectly. The reason is that m is only a lower bound for the domain sizes of the variables, but not an upper bound. For example consider a variable v with $dom(v) = \{1, \ldots, 14\}$. Assume in case $v \leq 7$ a tuple t_1 exists with various different values, and in case $v \geq 8$ a tuple t_2 exists with various different values. They are mutually exclusive, there is no world in which both tuples exist. The number of possible values for the attributes of both tuples is only seven, but the domain size of v is 14.

The data generator expects another parameter, the correlation ratio. It denotes how strongly the uncertain attributes are correlated, i.e. for how many attributes a variable is used. We fixed it to 0.25.

8.2 Updates

The first set of test cases simulates a sequence of updates of the attribute *ps_supplycost* in the relation *partsupp*, followed by a query on the updated table. Figure 8.2 shows the performed queries.

Q_1 increases the supply cost of all parts that are manufactured by "Manufacturer#1". The actually executed update involves four U-relations, namely the two vertical partitions of *partsupp* that hold the attributes *ps_supplycost* and *ps_partkey*, and the two vertical partitions of *part* that hold the attributes *p_partkey* and *p_mfgr*. Note that the

```
Q1 : update partsupp set ps_supplycost = ps_supplycost * 1.1
     from part
     where ps_partkey = p_partkey and p_mfgr='Manufacturer#1';
Q2 : update partsupp set ps_supplycost = ps_supplycost * 1.1
     where ps_availqty < 1000;
Q3 : update partsupp set ps_supplycost = ps_supplycost * 0.9
     where ps_availqty > 9000;
Q4 : update partsupp set ps_supplycost = ps_supplycost * 0.9
     where ps_availqty > 8000;
Q5 : select * from partsupp;
```

Figure 8.2: Test case 1: Update queries.

vertical partitions are joined implicitly along the tuple ids. Q_2, Q_3 and Q_4 update the supply cost based on the available quantity, using different conditions on the quantity. They implicitly involve a join of two tables, namely the join over the vertical partitions of *partsupp* for the attributes *ps_supplycost* and *ps_availqty*. Finally Q_5 selects all attributes from the relation *partsupp* which implicitly means a join of all the five vertical partitions of *partsupp*.

We analyze the influence of the scale factor s, the uncertainty ratio x and the maximum number of alternatives per field m. The base values we use are $s = 0.1$, $x = 0.05$ and $m = 7$. The different lines in each of the diagrams correspond to updates with ws-descriptors, iws-descriptors and intws-descriptors. We report the total time needed for the update queries $Q_1, \ldots Q_4$, as well as the number of tuples in the updated table after applying the updates. For comparison we also give the number of tuples in the "base table", i.e. the number of tuples in the table before the updates. This is equal for ws-/iws- and intws-descriptors. Further we report the time needed for a select-join query (Q_5) on the updated table.

Influence of the Scale Factor

We let $x = 0.05$, $m = 7$ and vary the scale factor. Figure 8.3 illustrates the influence of the scale factor. The first diagram shows the time needed to perform the updates. In the second diagram we can see that the number of tuples in the updated table rises linearly with the scale factor. When using ws-descriptors the linear factor is higher than when using iws- or intws-descriptors. In the third diagram we can see the time needed for the select-join query on the updated table. We conclude that, given a moderate amount of uncertainty, our approach for updating uncertain databases is feasible in practice even for larger databases.

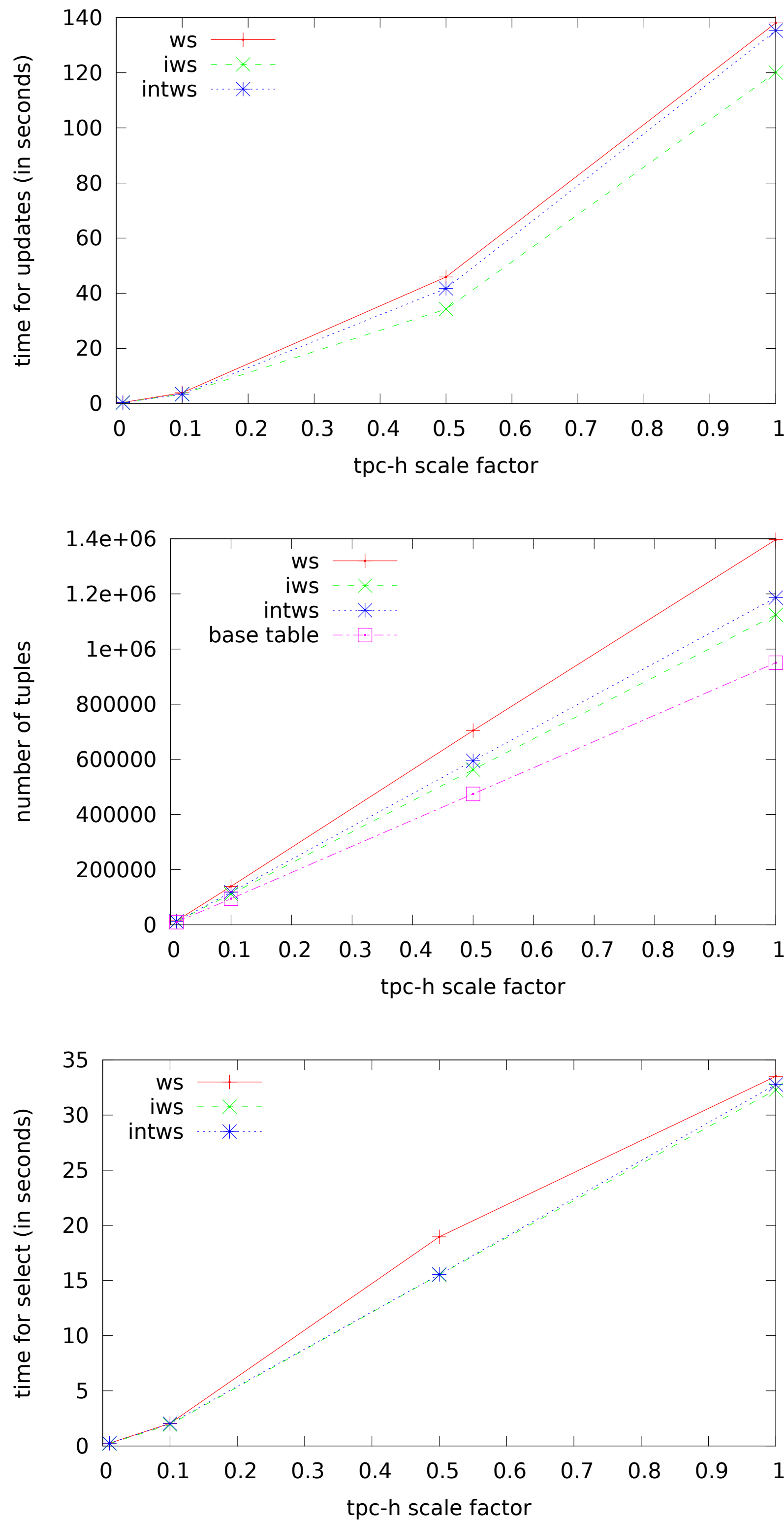

Figure 8.3: Influence of the scale factor on updates.

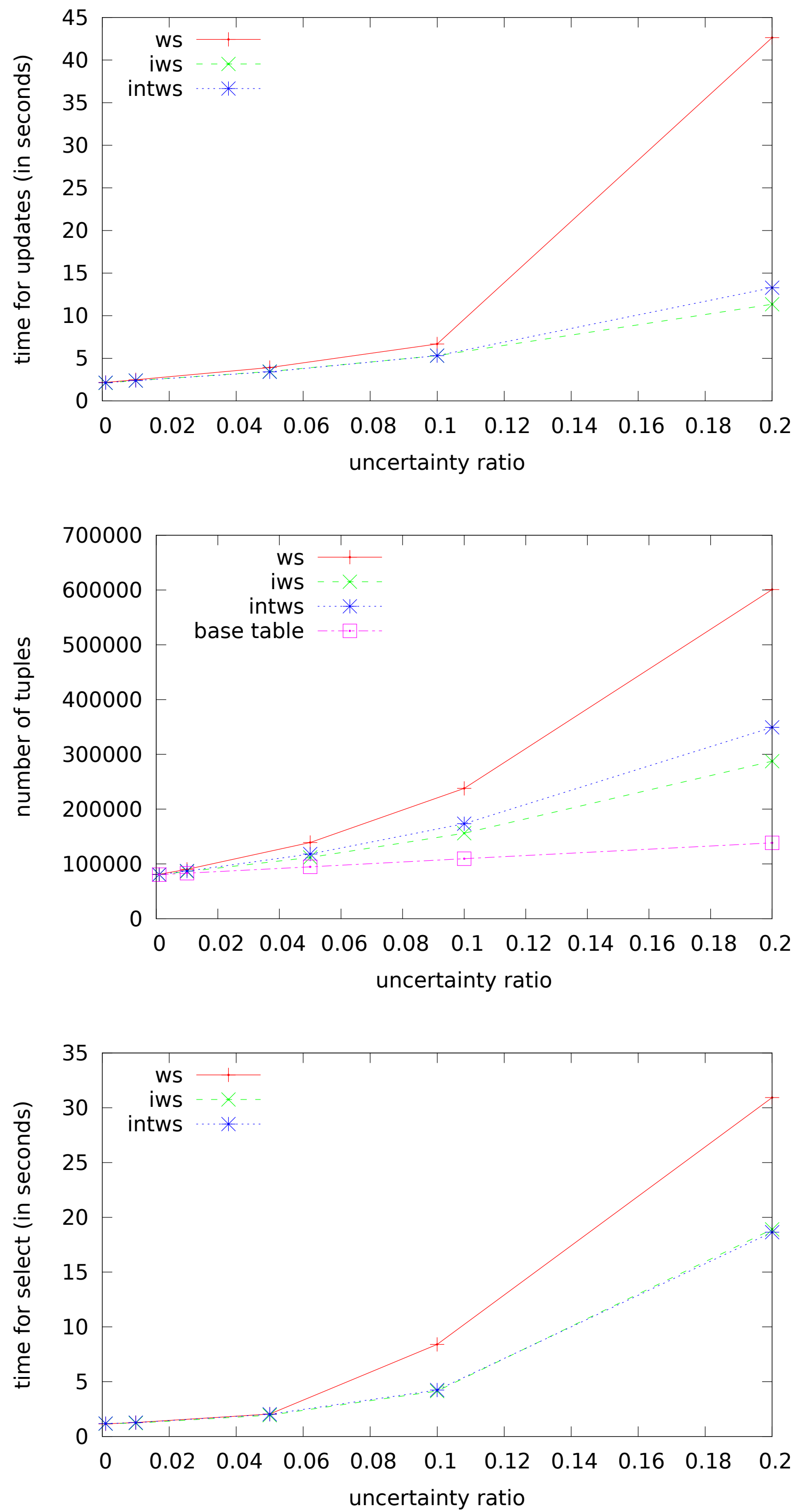

Figure 8.4: Influence of the uncertainty ratio on updates.

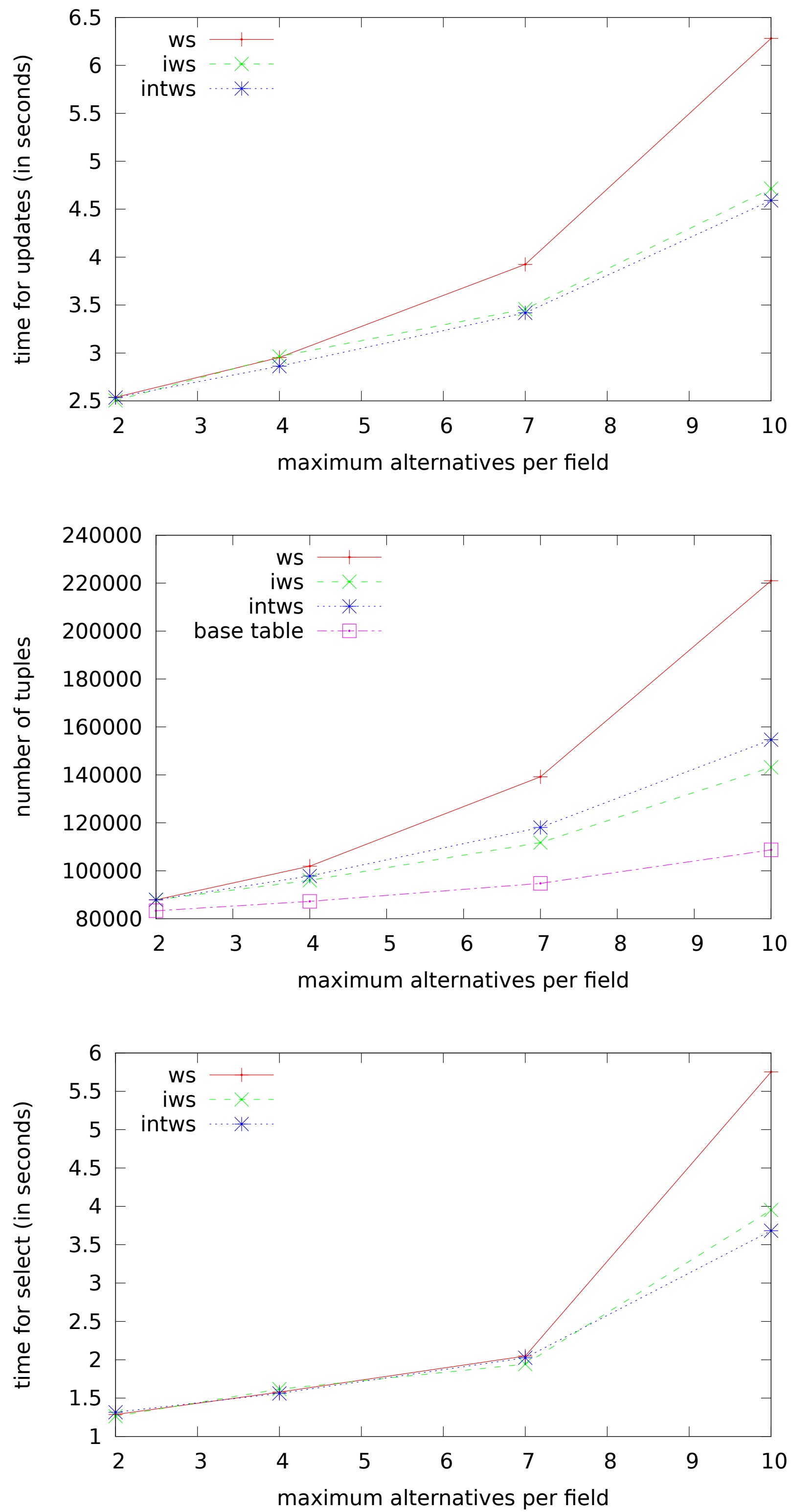

Figure 8.5: Influence of the domain size of the variables on updates.

Influence of the Uncertainty Ratio

We let $s = 0.1$, $m = 7$ and vary the uncertainty ratio. Figure 8.4 illustrates the influence of the uncertainty ratio. One can see that, using ws-descriptors, more than three times as much time is needed to perform the updates for a uncertainty ratio of 0.2. With a rising uncertainty ratio, the number of tuples in the updated table rises more than linearly, especially in the case of ws-descriptors. When using ws-descriptors the updated table is about twice as large as with iws- or intws-descriptors. As the third diagram shows, this influences the time needed for the subsequent select-join query.

Influence of the Domain Size

We let $s = 0.1$, $x = 0.05$ and vary m, the maximum number of alternatives per field. Figure 8.5 illustrates the influence of m. With a rising value for m the difference between ws-descriptors on the one hand and iws-/intws-descriptors on the other hand becomes greater. Iws- and intws-descriptors do not give very different results.

8.3 Set Difference

The second set of test cases consists of queries that use the set difference operator for queries not expressible in positive relational algebra. Figure 8.6 shows the queries.

The query Q_6 selects parts that were only sold at quantities of 5 or more. Q_7 selects parts that were only sold at quantities of 5 or more and whose availability is smaller than 8000 at all suppliers. Q_8 selects parts that are available at most at one supplier. Q_9 selects suppliers that offer all parts of a specific brand and size. It uses a subquery which selects the suppliers that do not offer all parts of that brand and size.

We analyze the influence of the parameter m, the maximum number of alternatives per field. We set the other parameters to $s = 0.1$ and $x = 0.05$. The Figures 8.7, 8.8, 8.9 and 8.10, respectively, illustrate the experimental results for query Q_6, Q_7, Q_8 and Q_9, respectively. Note that Figure 8.7 and 8.8 use a logarithmic scale for the y-axis. The different lines in each of the diagrams correspond to using ws-descriptors, iws-descriptors and intws-descriptors. For each query we report the time needed to answer it and the number of tuples in the result of the query (except for Q_9 where the answers are empty relations).

In case $m = 10$, the results for Q_6 and Q_7 exhibit a difference of several orders of magnitude for both the time needed to answer the queries and the size of the query results. With ws-descriptors the most time and space is needed. They are clearly outperformed by iws-descriptors and intws-descriptors. It is interesting that with intws-descriptors clearly less time is needed than with iws-descriptors, while with iws-descriptors the query results are smaller. We will refer to that in the summary.

Q_6 :
```
select p_partkey from part
except
select l_partkey from lineitem_l where l_quantity < 5;
```
Q_7 :
```
select p_partkey from part
except
select l_partkey from lineitem, partsupp
where l_partkey = ps_partkey and l_quantity < 5 and
   ps_availqty >= 8000;
```
Q_8 :
```
select p_partkey from part
except
select ps1.ps_partkey from partsupp ps1, partsupp ps2
where ps1.ps_partkey = ps2.ps_partkey and ps1.ps_suppkey
   <> ps2.ps_suppkey;
```
Q_9 :
```
select s_suppkey from supplier
except
select s_suppkey from
( select s_suppkey, p_partkey
  from supplier, part p
  where part.p_brand ='Brand#15' and part.p_size = 13
  except
  select ps_suppkey, ps_partkey from
  partsupp
) notallparts;
```

Figure 8.6: Test case 2: Queries with set difference.

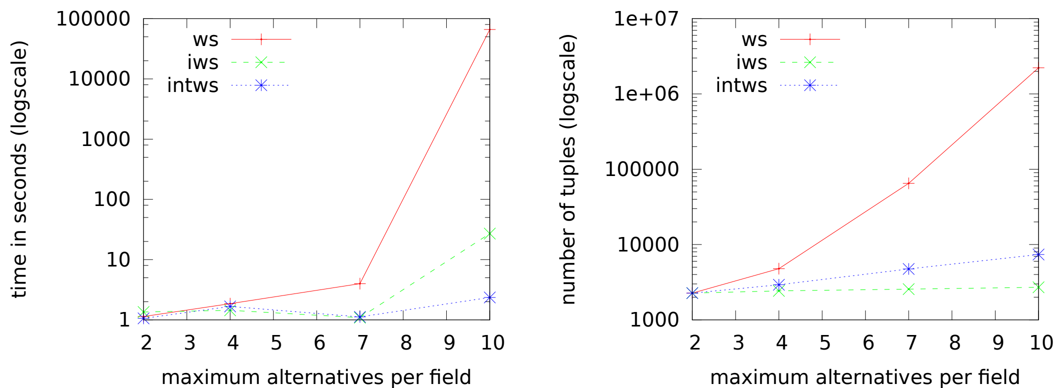

Figure 8.7: Comparison for query Q_6.

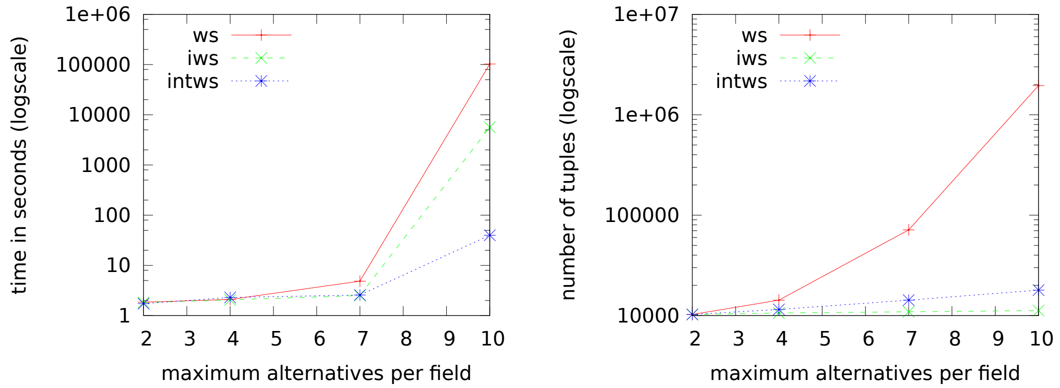

Figure 8.8: Comparison for query Q_7.

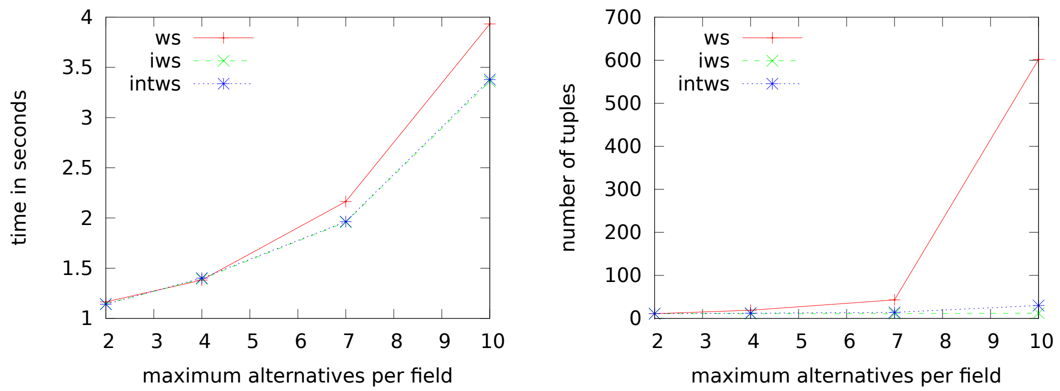

Figure 8.9: Comparison for query Q_8.

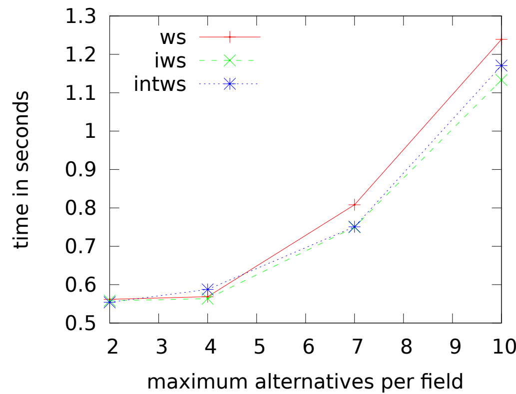

Figure 8.10: Comparison for query Q_9.

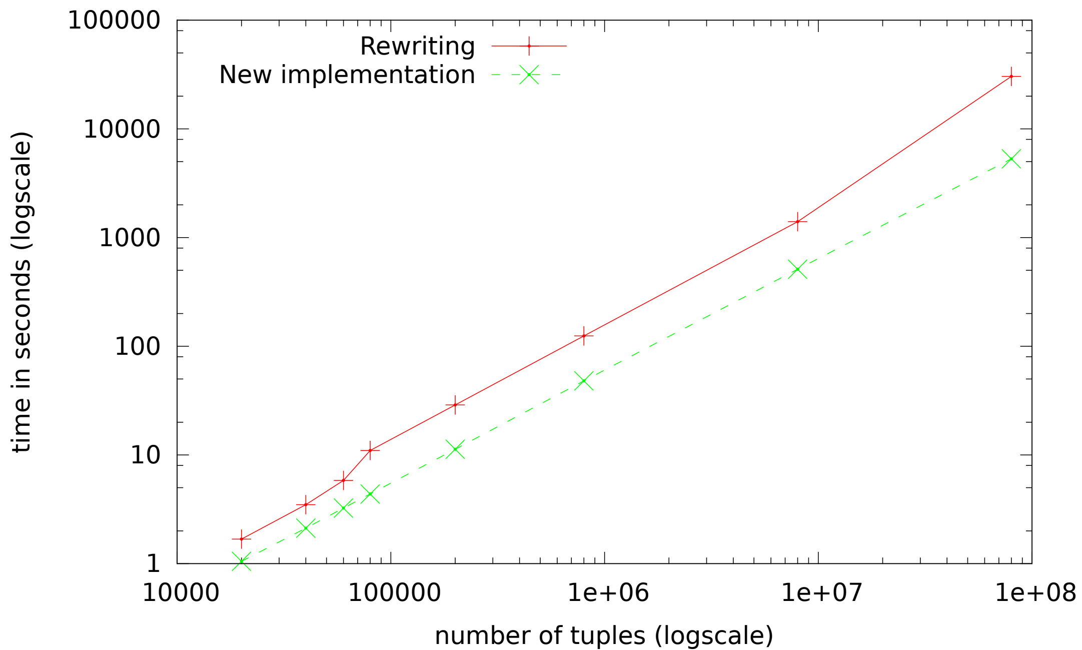

Figure 8.11: Runtime comparison for repair-key.

The queries Q_8 and Q_9 are processed by all three variants in a short time, even for high values of m.

8.4 Repair-Key

To compare our new implementation of the repair-key operator with the implementation of the MayBMS project we use a table *Measurements*$[k,$ *Weather*, *Ground*$]$. k is the key attribute (imagine a time stamp), *Weather* a weather condition (for example rain) and *Ground* describes whether the ground is dry or wet. The goal is to repair the key constraint k, i.e. k shall uniquely identify the attributes $weather$ and $ground$ in every possible world.

To do that we introduce uncertainty by means of the repair-key operator, using the following query:

```
repair key k in Measurements;
```

Figure 8.11 compares the runtime of the implementation of the repair-key operator in MayBMS (based on rewriting) and of our new implementation. In the figure the average value of 10 runs is shown for different numbers of tuples in the table *Measurements*. Our new implementation is in all cases significantly faster, although it additionally fills the world table which requires a lot of slow disk writings.

The reason for this difference is that in our implementation the table on which repair-key is carried out is sorted only once, while the original implementation of MayBMS sorts the table several times.

8.5 Summary

We have described the test environment and the experiments we have carried out. They show first of all the practicability of our theoretical work. We summarize the observations.

- *Database size*. The variation of the scale factor has shown that an increase of the size of the database results only in a linear or slightly greater increase of time needed to answer a query. This means that our solutions scale well with increasing database sizes.

- *Uncertainty ratio and maximum alternatives per field*. The incrementation of these two parameters has led to substantially longer query processing times and also larger results. This is explained by the unavoidable exponential worst-case complexity.

- *New representations*. As the theoretical work has already suggested, ws-descriptors are outperformed by iws- and intws-descriptors in every place. This means that both U^i-relations and U^{int}-relations are a clear improvement over U-relations.

- *Feasibility check*. Recall that when using iws-descriptors, the feasibility of the iws-descriptors has to be checked after every join operation. Thus one might expect that U^i-relations are slower in a join. In our experiments we have not observed any slowdown of the select-join query following the updates, which could have been caused by the feasibility check. The reason is the clever implementation of the feasibility check, which avoids querying the world table in most cases.

- *Iws- vs. intws-descriptors*. There are cases where with intws-descriptors considerably less time is needed to compute set difference, than with iws-descriptors. At the same time, with iws-descriptors the query results are smaller.

The last point needs an explanation. We believe it is due to skipping (see Section 6.4). The skipping condition for intws-descriptors allows one to skip more levels than the skipping condition for iws-descriptors, which in turn means that (possibly) less time is needed with intws-descriptors.

Decompression vs. Negation

We have presented the algorithms INV-DECOMPRESS and INV-NEGATE-SKIP to compute the inverse of a ws-set. The complexity of the first is always exponential whereas the complexity of the second is only exponential in the worst case. We performed our tests with INV-NEGATE-SKIP and argue now that those tests could not have been finished with INV-DECOMPRESS in reasonable time.

In all the tests that we performed we monitored the ws-sets of which we computed the inverse. Up to 20 different variables occurred in the ws-sets, and the product of the domain sizes of the variables was up to 10^{15}. This means that INV-DECOMPRESS would have had to iterate sets of 10^{15} ws-descriptors to compute the inverse of one ws-set. This is impossible in reasonable time.

9. Conclusion

We have studied the problem of updating uncertain databases in the form of U-relations. Updates of uncertain databases and the need for decompression were shortly mentioned by Antova et al. [7], but not analyzed in depth. We have shown how to use set difference to model arbitrary updates. Set difference is also important as an operator on its own because it allows more powerful queries. But due to its hardness it has received only little attention so far. We have proposed algorithms, studied their complexity, and we have introduced new representation systems for uncertain databases. For the research questions stated in the beginning of this thesis, we can give now the following answers:

- *Updates*. We have considered a comprehensive update language that allows one to join relations and that includes update queries with subqueries. We have shown how to model such update and delete statements using set difference. In case some constraints are fulfilled, the semi-join update approach can be applied. It relies on positive relational algebra only and thus has polynomial time data complexity.

- *Set Difference*. We have reduced the problem of computing set difference to computing the inverse of a ws-set. For this purpose we have proposed two methods: *decompression* and *negation*. Whereas decompression is a brute-force approach, negation considers the structure of the ws-set. Additionally we have introduced two new representation systems: U^i-relations and U^{int}-relations. They exponentially reduce the worst-case complexity of negation and thus also of set difference. We have shown that positive relational algebra queries extended by the possible operator have the same complexity on U^i-relations and U^{int}-relations as on U-relations. In addition, with a technically involved proof we have shown that by using U^i-relations one can never get larger query results than by using U-relations.

- *Optimizing U-relations*. We have proven that the problem of minimizing ws-sets (as well as iws/intws-sets) is Σ_2^P-complete. Hence there is no efficient algorithm that shrinks a U-relation to the minimally necessary size. However, we have presented two optimization heuristics: *subset elimination* and *merging*. We have integrated the first into the computation of inverses of ws-sets and shown that this integration results in a significantly better worst-case complexity. For U^{int}-relations we have pointed out a special optimization problem and shown its intractability.

- *Experimental evaluation*. Computing set difference on U-relations has an unavoidable exponential worst-case complexity. However, the experiments with our

prototype have shown the practicability of the described concepts for moderate amounts of uncertainty, and that our solutions scale well with increasing database sizes. Furthermore, the experiments have confirmed the predominance of U^i-relations and U^{int}-relations over U-relations in practice.

Future work

We have demonstrated how to perform updates and how to compute set difference on U-relations. Although we have answered many important questions, new ones have appeared and should be investigated.

- *Probability Computation*. For probabilistic U-relations there exist exact and approximate algorithms that compute the probability that a given tuple actually is part of a relation. Can they be adapted to compute the probabilities of tuples in probabilistic U^i-relations and U^{int}-relations?
- *Further Optimization*. We have described several ways to optimize the computation of set difference, one of them being skipping. Are there other optimizations possible that further reduce the complexity of set difference?

While these two questions point out research topics that directly build on the work of this thesis, we want to suggest another open issue: benchmarks. At the moment there does not exist any benchmark for uncertain databases. But to be able to compare experimental results a common benchmark, based on practically relevant use cases, is needed. What use cases are relevant in practice? To answer this question, a thorough analysis of the applications of uncertain databases is required.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. (Cited on page 4.)

[2] Serge Abiteboul, Paris Kanellakis, and Gösta Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78:159–187, 1991. (Cited on page 1.)

[3] Charu C. Aggarwal. *Managing and Mining Uncertain Data*. Springer, 2009. (Cited on page 1.)

[4] Charu C. Aggarwal and Philip S. Yu. A survey of uncertain data algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 21:609–623, 2009. (Cited on page 1.)

[5] Parag Agrawal and Jennifer Widom. Confidence-aware join algorithms. In *Proc. of the 25th International Conference on Data Engineering (ICDE)*, pages 628–639. IEEE, 2009. (Cited on page 1.)

[6] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and simple relational processing of uncertain data. In *24th International Conference on Data Engineering (ICDE)*, pages 983–992. IEEE, 2008. (Cited on page vii, 1, 6, 9, 14, 15, 16, 74, 82, 84.)

[7] Lyublena Antova and Christoph Koch. On APIs for probabilistic databases. In *Proc. of the International Workshop on Quality in Databases and Management of Uncertain Data (QDB/MUD)*, pages 41–56, 2008. (Cited on page 1, 2, 19, 95.)

[8] Lyublena Antova, Christoph Koch, and Dan Olteanu. From complete to incomplete information and back. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 713–724. ACM, 2007. (Cited on page 2, 6.)

[9] Lyublena Antova, Christoph Koch, and Dan Olteanu. $10^{(10^6)}$ worlds and beyond: efficient representation and processing of incomplete information. *VLDB Journal*, 18(5):1021–1040, 2009. (Cited on page 1.)

[10] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. (Cited on page 5.)

[11] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. ULDBs: Databases with uncertainty and lineage. In *Proc. of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 953–964. ACM, 2006. (Cited on page 1.)

[12] Jihad Boulos, Nilesh N. Dalvi, Bhushan Mandhani, Shobhit Mathur, Christopher Ré, and Dan Suciu. MYSTIQ: A system for finding more answers by using probabilities. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 891–893. ACM, 2005. (Cited on page 1.)

[13] Reynold Cheng and Sunil Prabhakar. Managing uncertainty in sensor database. *SIGMOD Record*, 32:41–46, 2003. (Cited on page 1.)

[14] Edgar F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems (TODS)*, 4(4):397–434, 1979. (Cited on page 6.)

[15] Nilesh Dalvi, Karl Schnaitter, and Dan Suciu. Computing query probability with incidence algebras. In *Proc. of the 29th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 203–214. ACM, 2010. (Cited on page 2.)

[16] Nilesh Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, 16:523–544, 2007. (Cited on page 2, 13.)

[17] Nilesh Nilesh Dalvi, Christopher Ré, and Dan Suciu. Probabilistic databases: diamonds in the dirt. *Communications of the ACM*, 52(7):86–94, 2009. (Cited on page 1.)

[18] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Prentice Hall International, sixth edition, 2010. (Cited on page 4.)

[19] Robert Fink and Dan Olteanu. On the optimal approximation of queries using tractable propositional languages. In *Proc. of the 14th International Conference on Database Theory (ICDT)*, pages 174–185. ACM, 2011. (Cited on page 2.)

[20] Robert Fink, Dan Olteanu, and Swaroop Rath. Providing support for full relational algebra in probabilistic databases. In *Proc. of the 27th International Conference on Data Engineering (ICDE)*, pages 315–326. IEEE, 2011. (Cited on page 2, 24.)

[21] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book*. Pearson Education, second edition, 2009. (Cited on page 4.)

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. (Cited on page 5.)

[23] Salim Haddadi and Zoubir Layouni. Consecutive block minimization is 1.5-approximable. *Information Processing Letters*, 108(3):132 – 135, 2008. (Cited on page 64.)

[24] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing, third edition, 2006. (Cited on page 5.)

[25] Wen-Lian Hsu. A simple test for the consecutive ones property. *Journal of Algorithms*, 43(1):1 – 16, 2002. (Cited on page 64.)

[26] Tomasz Imieliński and Witold Lipski Jr. Incomplete information in relational databases. *Journal of the ACM*, 31:761–791, 1984. (Cited on page 1, 6.)

[27] Abhay Jha and Dan Suciu. Knowledge compilation meets database theory: compiling queries to decision diagrams. In *Proc. of the 14th International Conference on Database Theory (ICDT)*, pages 162–173. ACM, 2011. (Cited on page 1.)

[28] Christoph Koch. MayBMS: A system for managing large uncertain and probabilistic databases. In *Managing and Mining Uncertain Data*. Springer, 2008. (Cited on page 1, 74.)

[29] Christoph Koch. A compositional query algebra for second-order logic and uncertain databases. In *Proc. of the 12th International Conference of Database Theory (ICDT)*, pages 127–140. ACM, 2009. (Cited on page 6, 76.)

[30] Christoph Koch and Dan Olteanu. Conditioning probabilistic databases. *Proc. of the VLDB Endowment*, 1:313–325, 2008. (Cited on page 6, 13, 18.)

[31] Lawrence T. Kou. Polynomial complete consecutive information retrieval problems. *SIAM Journal on Computing*, 6(1):67–75, 1977. (Cited on page 63, 64.)

[32] Jian Li and Amol Deshpande. Ranking continuous probabilistic datasets. *Proc. of the VLDB Endowment*, 3(1):638–649, 2010. (Cited on page 1.)

[33] MayBMS. `http://maybms.sourceforge.net`, last access: August 2011. (Cited on page 74.)

[34] João Meidanis, Oscar Porto, and Guilherme P. Telles. On the consecutive ones property. *Discrete Applied Mathematics*, 88(1-3):325 – 354, 1998. (Cited on page 64.)

[35] Raghunath Nambiar, Nicholas Wakou, Forrest Carman, and Michael Majdalany. Transaction processing performance council (TPC): State of the council 2010. In *Performance Evaluation, Measurement and Characterization of Complex Systems*, pages 1–9. Springer, 2011. (Cited on page 82.)

[36] Dan Olteanu, Jiewen Huang, and Christoph Koch. Approximate confidence computation in probabilistic databases. In *Proc. of the 26th International Conference on Data Engineering (ICDE)*, pages 145–156. IEEE, 2010. (Cited on page 13.)

[37] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. (Cited on page 5.)

[38] Reinhard Pichler, Vadim Savenkov, Sebastian Skritek, and Hong-Linh Truong. Uncertain databases in collaborative data management. In *Proc. of the Fourth International VLDB workshop on Management of Uncertain Data (MUD)*, pages 129–143. CTIT, University of Twente, 2010. (Cited on page 18.)

[39] PostgreSQL. `http://www.postgresql.org`, last access: August 2011. (Cited on page 74.)

[40] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw-Hill, third edition, 2003. (Cited on page 4.)

[41] Christopher Ré and Dan Suciu. The trichotomy of having queries on a probabilistic database. *The VLDB Journal*, 18:1091–1116, 2009. (Cited on page 2.)

[42] Raymond Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977. (Cited on page 6.)

[43] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *Proc. of the 24th International Conference on Data Engineering (ICDE)*, pages 1023–1032. IEEE, 2008. (Cited on page 1.)

[44] Saket Sathe, Hoyoung Jeung, and Karl Aberer. Creating probabilistic databases from imprecise time-series data. In *Proc. of the 27th International Conference on Data Engineering (ICDE)*, pages 327–338. IEEE, 2011. (Cited on page 1.)

[45] Sarvjeet Singh, Chris Mayfield, Sagar Mittal, Sunil Prabhakar, Susanne Hambrusch, and Rahul Shah. Orion 2.0: native support for uncertain data. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 1239–1242. ACM, 2008. (Cited on page 1.)

[46] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976. (Cited on page 54.)

[47] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Morgan & Claypool, 2011. (Cited on page 1.)

[48] Transaction Processing Performance Council. TPC benchmark H (decision support). `http://www.tpc.org/tpch/spec/tpch2.14.2.pdf`, last access: August 2011. (Cited on page vii, 82, 83.)

[49] Christopher Umans. The minimum equivalent DNF problem and shortest implicants. In *Proc. of the 39th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 556–563. IEEE, 1998. (Cited on page 54.)

[50] Ting-You Wang, Christopher Ré, and Dan Suciu. Implementing not exists predicates over a probabilistic database. In *Proc. of the International Workshop on Quality in Databases and Management of Uncertain Data (QDB/MUD)*, pages 73–86, 2008. (Cited on page 2, 24.)

[51] Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *Managing and Mining Uncertain Data*. Springer, 2008. (Cited on page 1.)

[52] Fei Xu, Kevin Beyer, Vuk Ercegovac, Peter J. Haas, and Eugene J. Shekita. $E = mc^3$: managing uncertain enterprise data in a cluster-computing environment. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 441–454. ACM, 2009. (Cited on page 1.)