Complexity of Uniform Operational Consistent Query Answering

Markus Schneider

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2018

Abstract

Obtaining meaningful answers from inconsistent databases is vital to a host of applications. This poses a challenge in the era of Big Data due to its inherent properties. Although the field of consistent query answering has attracted considerable attention, no practical solution exists to date. Most research has concentrated on finding tractable scenarios, which, unfortunately, have limited real world applicability. A recently proposed framework introduces a new approach which allows for efficient approximation algorithms. In our research, we tackle an open problem concerning a proposed variation of the presented framework. We formally introduce this variation and establish an intractability result for uniform operational consistent query answering, even in some easy settings of the problem.

Acknowledgements

I would like to thank my supervisor, Dr. Andreas Pieris, for his reliable support and expertise throughout this project. Also I would like to express my gratitude to Marco Calautti, who provided me with crucial insights vital for the completion of the project.

Further, I would like to thank my family and friends for their unconditional support throughout my studies. I am truly grateful for their steady encouragement and the honest feedback for this thesis.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Markus Schneider*)

Table of Contents

Chapter 1

Introduction

Big Data is omnipresent, produced and consumed by an increasing amount of applications, devices and users. In order to make effective use of this data, it has to be stored and well-maintained in a database, which becomes infeasible at scale. Accordingly, there is a discrepancy between the potential and the current capabilities of Big Data.

The most fundamental task for databases is query answering, which becomes harder with an increase in volume and veracity of the data. A common reason being that databases may not comply with their specifications, i.e. are inconsistent. It is widely accepted that removing all inconsistencies of a database is practically impossible. Therefore, one crucial problem for databases is to retrieve meaningful information from large, and simultaneously inconsistent data. This is the goal of consistent query answering.

1.1 Established Approach

Consistent query answering (CQA) was introduced as a concept in 1999 by Arenas, Bertossi and Chomicki [2]. The central elements of this approach are the ones of a *repair* and *certain answers* over these repairs. A repair to an inconsistent database is a consistent database, whose difference to the original is minimal, according to some measure. Further, we have a certain answer to a query posed on an inconsistent database if this answer is entailed in all repairs. Even if an answer is missing in just one repair, it is considered uncertain.

Example 1.1. Consider a database containing the two facts $R(a,b)$ and $R(a,c)$, while there is a key constraint imposed on the first attribute of relation R. This database would be inconsistent with respect to the key constraint. Now, we could delete either

one of the two facts to obtain a repair to the database. If we had a query that asks, whether there is a fact in the database whose first attribute is a, we could consistently say *yes*, as the query is true in both repairs. However, if instead we wanted to know whether there is a fact whose second attribute is b, we could not return *yes* as an answer, but would have to say *no* instead.

Generally, in this approach one has to argue over exponentially many repairs, which is why the problem of finding certain answers is coNP-hard. In light of this, much of the treatment CQA was given so far concentrated on finding conditions for which it is tractable. A extensive survey of previous results is presented in [4].

Nonetheless, the found scenarios have limited applicability. Moreover, for the intractable cases it is highly unlikely that efficient approximations exist, as this would imply the polynomial hierarchy collapses. Yet, "the ultimate goal of a practically applicable CQA approach should be efficient approximate query answering with explicitly stated guarantees" [5], as postulated by Calautti, Libkin and Pieris. This is why a new framework was proposed, which enables us to find approximation schemes to CQA.

Simultaneously, two major flaws of the declarative definition of the established CQA framework are corrected. First, it is counter intuitive to say an answer is uncertain if it is missing in just one of potentially exponentially many repairs. Therefore, instead of insisting on certain answers, we would rather like to know how likely an answer is. Second, in the traditional CQA we already assume a set of repairs on which we investigate certain answers, but it is not clear at all where these repairs come from. The current approach does not provide a procedure on how to arrive at repairs. We can only check, given a database, whether it is a repair or not. Further, the current situation does not allow to argue about the likelihood of repairs, or more crucially, why one repair is more likely than another.

1.2 Revised Approach

These issues are rectified by the recently proposed framework [5] by revising the relevant definitions of repairs and query answers on inconsistent databases. With the operational framework in place, we are able to explain how to construct repairs from an inconsistent database and assign probabilities to these. This allows us to argue about the likelihood of a fact occurring in a query answer. We achieve this the following way.

First, the notions of an *operation* and a *violation* are introduced. Basically, a violation explains why a database is inconsistent with respect to a set of constraints. An

operation is an update to the database, which ideally eliminates one or more violations. The updates considered in [5] are insertions and deletions of tuples, which we will focus on in this research, although the authors point out other forms of updates for future work. We now apply a *repairing sequence* of operations to the database, which is a sequence of database updates fulfilling certain requirements. For example, every operation must eliminate a violation, and the same fact cannot be inserted and deleted in the same sequence. If no more operations can be applied without violating one of the requirements, we say the sequence is complete. Note that this does not guarantee the consistency of the resulting database. If, however, the resulting database complies with the constraints, it is an *(operational) repair*.

Now, we can assign probabilities to operations and, based on these, compute the likelihood of a repairing sequence, and ultimately of the resulting repair. Note that multiple repairing sequences may lead to the same repair, such that the likelihood of a specific repair is the sum of the probabilities of all sequences leading to it. Further, we can calculate the probability of a fact being included in the answer. By only selecting answers that appear with probability 1, we can now simulate the old framework. Thus, it is not surprising that exact answers to this form of CQA are equally hard. However, and this is the significant advantage of the new approach, this operational framework admits efficient approximation algorithms, which was previously beyond reach.

In this work, we are looking at a slight variation of the original framework that considers equally likely repairing sequences. As we cannot always assign meaningful likelihoods to operations, it makes sense to look at a model in which every repairing sequence has the same probability, no matter which or how many operations it contains.

1.3 Outline

The outlined variation was proposed as future work in [5] and, thus, there was no progress made towards this open problem. In our work we first formalise the new setting to study this uniform approach. The main contribution of this research, besides formalising the uniform framework, is the establishment of #P-hardness of our problem in data-complexity for

1. conjunctive queries and inclusion dependencies,

2. conjunctive queries with negations and key constraints, even if only one atom occurs negated in the query,

3. conjunctive queries with negations and denial constraints with constants, even if only one atom occurs negated in the query, and

4. unions of conjunctive queries with negations and constant-free denial constraints, even if each conjunctive query only has one negation.

We will proceed to define the essential basic concepts followed by a detailed introduction into the new operational framework. After defining the precise problem we are studying in this research, we will continue to establish the results mentioned above. Finally, we will discuss the implications of our results.

Chapter 2

Background and Notation

In this chapter, we will introduce the fundamental concepts used throughout this research as well as the relevant notation. We start by defining the used database model and various query languages, followed by an introduction to database constraints. We conclude by outlining the established approach to consistent query answering.

2.1 Relational Databases

In this research, we work with the relational database model as introduced in [7]. That means, a relation R is a finite set of facts of equal arity n, denoted by R/n. A fact of arity n is an ordered n-tuple of constants c, drawn from a countably infinite set C. We write $R(c_1,\ldots,c_n)$ to say the fact $(c_1,\ldots,c_n)$ is contained in relation R/n, for some $n > 0$ and $c_i \in C$ for all $i \in [n] = \{1,\ldots,n\}$. A finite collection of relation symbols with annotated arity R/n is called a schema $\mathcal{S}$. A specific instance of a schema, i.e. a set of instances of all relations $R/n \in \mathcal{S}$, is a database D. We denote the active domain of a database D, i.e. all occurring constants in D, with $adom(D)$.

2.2 Queries

To retrieve information from such databases, we pose queries. These queries come in different shapes with different levels of expressiveness. We follow the established definitions of query languages as in [1]. A query Q operates over the same schema $\mathcal{S}$ as the underlying database D. The building block of queries are atoms, also called predicates, of the form $R(x_1,\ldots,x_n)$, for $x_i \in C \cup V$, $i \in [n]$. Here, V is a distinct set of variables, i.e. $V \cap C = \varnothing$. Let $Var(x_1,\ldots,x_n)$ be the variables occurring in the atom.

A *conjunctive query* (*CQ*) is a conjunction of atoms with existentially quantified variables. Throughout this work, we will use commas instead of $\wedge$ for conjunctions of atoms, as is common in the literature. For a query Q over relations $R_1/n_1, \ldots, R_m/n_m$ we write:

$$Q(x_1, \ldots, x_{n_Q}) = \exists \bar{x}\, R_1(\vec{x_1}), \ldots, R_m(\vec{x_m})$$

In the above query, each $\vec{x_i}$ is a vector over constants and variables of arity n_i, for all $i \in [m]$. The query is existentially quantified over all variables not appearing in the head of the query, i.e. $\bar{x} = \left(\bigcup_{i\in[m]} Var(\vec{x_i})\right) \setminus \{x_1, \ldots, x_{n_Q}\}$.

By permitting negated atoms in the query we obtain the class of *conjunctive queries with negations* ($CQ^\neg$). In other words, we allow negations directly in front of single atoms. However, variables in a negated atom must also occur in at least one other, positively appearing atom to allow for safe query evaluation, i.e. preventing infinitely large query answers.

Queries which are *unions of conjunctive queries with negations* ($UCQ^\neg$) are of the form $Q = \bigvee_{i\in[n]} Q_i$, where each $Q_i \in CQ^\neg$, for all $i \in [n]$. Note that all queries $Q, Q_1, \ldots, Q_n$ must share their free variables.

Last, we introduce *first order queries* (*FO*), which are allowed to use the full range of first order logic and are of the form $\{\vec{x} \mid \varphi(\vec{x})\}$, where φ is a first-order formula with free variables $\vec{x}$. The formula can be built from predicates, conjunctions, disjunctions, negations, existential and universal quantifiers, and implications. *FO* is the most expressive query language of those presented here and includes all previously defined query languages. Thus it is sufficient to define query answers for *FO*. The answer to a *FO* query $\{\vec{x} \mid \varphi\}$ on a database D is the set $\{\bar{c} \in adom(D)^{|x|} \mid D \models \varphi(\bar{c})\}$ [5].

If a query contains no free variables, we also call it a Boolean query or a sentence. The answer to such a query is either the empty tuple $\langle\rangle$, if the database entails the sentence, i.e. $D \models Q$, or the empty set otherwise. Instead of $Q(\langle\rangle)$ we simply write Q when defining the query.

2.3 Constraints

In this dissertation, we will deal with three standard database constraints, following the work of [5]. Namely, these are *equality-generating dependencies* (EGD), *tuple-generating dependencies* (TGD) and *denial constraints* (DC). Dependencies are FO-sentences defined over the same schema as the underlying database.

An EGD is of the form

$$\forall \bar{x} \left(\mathcal{C}(\bar{x}) \rightarrow x_i = x_j \right)$$

in which $\mathcal{C}(\bar{x})$ is a conjunction of atoms with variables $\bar{x}$ and $x_i, x_j \in \bar{x}$. EGDs can for example encode key constraints, as they can express the idea of a key in a relation. For instance, if we want to impose a key constraint on the first attribute of a relation $R/2$, we can establish the EGD $\forall x, y, z \left(R(x,y), R(x,z) \rightarrow y = z \right)$.

A TGD, which can encode an inclusion dependency, is of the following form:

$$\forall \bar{x} \, \forall \bar{y} \left(\mathcal{C}(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} \, \mathcal{C}'(\bar{x}, \bar{z}) \right)$$

Again, $\mathcal{C}(\bar{x}, \bar{y})$ and $\mathcal{C}'(\bar{x}, \bar{z})$ are conjunctions of atoms. This form of constraint can express, e.g., an inclusion of relations. We might want to express that all facts of relation $R/1$ are also included in the relation $S/1$. Therefore, the inclusion dependency $\forall x \left(R(x) \rightarrow S(x) \right)$ would have to be enforced to ensure $R \subseteq S$.

DCs can indicate disjointness of attributes or relations and are of the form

$$\forall \bar{x} \, \neg \mathcal{C}(\bar{x}) \quad \textit{or} \quad \forall \bar{x} \left(\mathcal{C}(\bar{x}) \rightarrow \bot \right)$$

for some conjunction of atoms $\mathcal{C}(\bar{x})$. Note that both of the above forms are equivalent. For example, if we want to impose that a fact can only be part of one of the relations $R/2$ and $S/2$, we create the denial constraint $\forall x, y \left(R(x,y), S(x,y) \rightarrow \bot \right)$.

In their more general form, only variables can be used in these constraints. If we allow for constants, however, we extend their expressiveness as we can, for example, work with specific values of the database. Later, we will work with such denial constraints and use the constants to distinguish between two sets of facts in the same relation. Note that, for brevity, we usually omit the universal quantifier in the constraints, as all variables that are not existentially quantified will be universally quantified.

Homomorphisms will enable us to argue about the satisfaction or violation of constraints. A homomorphism is a mapping from a set of atoms $\mathcal{C}$ to another set of atoms $\mathcal{C}'$. Note that we can view both databases as well as conjunctions of atoms as sets of atoms. Let $dom(\mathcal{C})$ denote the set of constants and variables appearing in $\mathcal{C}$. A homomorphism ϕ is a mapping from $dom(\mathcal{C})$ to $dom(\mathcal{C}')$, which is the identity on constants, such that for every atom $R(\vec{x}) \in \mathcal{C}$ we have an atom $R(\phi(\vec{x})) \in \mathcal{C}'$. Let $\phi(\mathcal{C}) = \{R(\phi(\vec{x})) \,|\, R(\vec{x}) \in \mathcal{C}\}$ be the set of atoms $\mathcal{C}$, where every variable is replaced by its assigned constant. We can now formalise the satisfaction of constraints by databases as follows [5].

An EGD as defined above is satisfied by a database D if for every homomorphism ϕ from $dom(\mathcal{C}(\bar{x}))$ to $adom(D)$, we have $\phi(x_i) = \phi(x_j)$. A database D satisfies a TGD as defined above if for every homomorphism ϕ from $dom(\mathcal{C}(\bar{x},\bar{y}))$ to $adom(D)$, there exists another homomorphism ψ from $dom(\mathcal{C}'(\bar{x},\bar{z}))$ to $adom(D)$ such that $\phi(x) = \psi(x)$ for all $x \in \bar{x}$. Lastly, a DC is satisfied by a database D if there is no homomorphism from $dom(\mathcal{C}(\bar{x}))$ to $adom(D)$.

If, for the purpose of an argument, we do not need to distinguish between an EGD, TGD and DC, we simply refer to it as constraint. We write Σ for the set of constraints imposed on a database. If a database D satisfies every constraint in Σ, we say D is *consistent* with respect to Σ, denoted $D \models \Sigma$. If it does not satisfy some constraint, we say the database is *inconsistent*.

2.4 Consistent Query Answering

When dealing with inconsistent databases, we still want to find answers to queries in these inconsistent databases, which is where *consistent query answering* (CQA) comes into play. We will now introduce the established approach to CQA as proposed in the late 1990s by Arenas, Bertossi and Chomicki [2].

The first key notion we will present is the one of a *repair*. According to [2], a repair of an inconsistent database is one that satisfies all constraints and which differs only minimally from the original database. Most commonly, the symmetric set difference (Δ) is used to measure the distance of databases. That means the difference between two databases D and D' is defined as $\Delta(D,D') = (D \setminus D') \cup (D' \setminus D)$. Further, a database D' is a repair to D with respect to Σ, if D' is consistent and there is no other consistent database D'' such that $\Delta(D,D'') \subseteq \Delta(D,D')$. The set of repairs to an inconsistent database D with respect to a set of constraints Σ is denoted by $Rep(D,\Sigma)$.

We can now define the second key notion of CQA, *certain answers* to a query. A certain answer is one that is entailed in all repairs $D' \in Rep(D,\Sigma)$. This causes a few issues, reflected in the limited applicability of this CQA approach as discussed in the introduction. Most importantly, the decision problem of CQA, i.e. deciding whether an answer is entailed in all repairs, is coNP-hard [6]. This rules out the existence of approximation algorithms with bounded error guarantees, as this would imply the polynomial hierachy collapses, which is generally believed not to be the case. With the new approach, we hope to improve this situation by opening up the possibility of approximating CQA.

Chapter 3

Operational Framework

We will now present the new operational approach to CQA as proposed in [5] for relational databases and the constraints introduced in Section 2.3. All definitions in this chapter are based on [5] unless specified otherwise.

3.1 Operations and Violations

We begin by defining the basic building block of this framework, *operations*. The operations *op* considered here are updates to the database that are either deletions or insertions of facts. For a set of facts F, we will refer to these operations as $-F$ and $+F$, respectively. Further, if we only delete a single fact $R(\vec{c})$ from the database, we write $-R(\vec{c})$ instead of $-\{R(\vec{c})\}$ for brevity. The same applies to insertions. If we apply an operation *op* to a database D, we write $op(D)$ to denote this. Note that also other forms of updates can be regarded, such as attribute-based operations. However, in our research we only consider deletions and additions of whole facts with constants from the database and constraints.

Now, the ultimate goal is to apply such operations to an inconsistent database until it is consistent. As an arbitrary succession of operations will obviously not achieve this, we have to keep track of all reasons why a database is inconsistent and then impose requirements on the operations that are applied to the database.

Formally, we capture these reasons under the notion of a *constraint violation*. Recall that a constraint is of the form $\varphi \to \vartheta$, where φ is a conjunction of atoms and ϑ is either an equality of variables (EGD), an existentially quantified conjunction of atoms (TGD) or the bottom symbol $\bot$ (DC). A database D violates a constraint $\sigma = \varphi \to \vartheta$ if there is a homomorphism ϕ from φ to D, such that $\phi(\varphi) \to \phi(\vartheta)$ is not satisfied. We

write (σ, ϕ) to denote the violation of σ by the database as evident by ϕ. We unite all violations of D for all constraints $\sigma \in \Sigma$ in the set $V(D, \Sigma)$.

We call an operation op (D, Σ)-fixing, if $V(D, \Sigma) - V(op(D), \Sigma) \neq \varnothing$, i.e. if op removes at least one violation from $V(D, \Sigma)$. We will later require every operation to remove at least one violation from the current set of violations.

Additionally, an operation should not only be fixing, but also justified. An operation would be considered unjustified, if it adds or removes facts that do not take part in the fixed violation. For example, consider the database $D = \{R(c_x, c_y), R(c_x, c_z), T(c_a, c_b)\}$. The key constraint $\sigma = R(x, y), R(x, z) \rightarrow y = z$ is violated by D, as both $R(c_x, c_y)$ and $R(c_x, c_z)$ are present in the database. Now, the operation $-\{R(c_x, c_y), T(c_a, c_b)\}$ would indeed be (D, σ)-fixing, but is unjustified as it also removes $T(c_a, c_b)$, which does not take part in the violation. For the above violation there are three justified operations, namely $-R(c_x, c_y)$, $-R(c_x, c_z)$ and $-\{R(c_x, c_y), R(c_x, c_z)\}$. We will now formally define justified operations.

Definition 3.1 (Justified Operations [5])**.** Let D be a database and Σ a set of constraints. For a database D', an operation $op \in \{+F, -F\}$ is called (D', Σ)-*justified* if there exists $(\sigma, \phi) \in V(D', \Sigma) - V(op(D'), \Sigma)$ such that for every non-empty set $G \subsetneq F$

(1) if $op = +F$ then $(\sigma, \phi) \in V(+G(D'), \Sigma)$

(2) if $op = -F$ then $(\sigma, \phi) \notin V(-G(D'), \Sigma)$.

Intuitively, that means that an operation $+F$ only adds to the database what is necessary to comply with the constraint. An operation of the form $-F$ is less cautious and can delete an arbitrary subset of atoms $\phi(\varphi)$ from the database D, for some constraint $\sigma = \varphi \rightarrow \vartheta$ and where $-F$ is fixing the violation (σ, ϕ). In particular, atoms that do not appear in $\phi(\varphi)$ are not removed from the database.

3.2 Repairing Sequences

Still, it is not sufficient to apply a sequence of only justified operations, as some anomalies may still occur. For example, we want to rule out the possibility that the same violation (σ, ϕ) is reintroduced after it has been fixed earlier in the sequence. Similarly, we would like to impose on a sequence of operations that the same fact is not removed and added by two operations of the same sequence. The last requirement enforced by

[5] is that the reason for the addition of a fact must remain valid throughout the sequence of operations. Specifically, if a fact $R'(\vec{c})$ was added because of the existence of another fact $R(\vec{c})$, i.e. to fix the violation of the inclusion dependency $R(\vec{x}) \rightarrow R'(\vec{x})$, that second fact $R(\vec{c})$ must not be deleted by a later operation, as this would eliminate the reason for the addition in the first place. All these requirements culminate in the definition of repairing sequences.

First, we introduce a notion for a sequence of operations $s = (op_i)_{i \geq 1}$ to describe the database obtained after a certain number of operations. Let D_0^s be the original (inconsistent) database D. Then, $D_i^s = op_i(D_{i-1}^s)$ for all $i \geq 1$. In other words D_i^s is the database we obtain after applying the first i operations of the sequence s.

Definition 3.2 (Repairing Sequence [5])**.** Consider a database D and a set Σ of constraints. A sequence of operations $s = (op_i)_{i \geq 1}$ is called (D, Σ)-*repairing* if it satisfies the following requirements for every $i \geq 1$:

(1) op_i is (D_{i-1}^s, Σ)-justified

(2) $(V(D_{i-1}^s, \Sigma) - V(D_i^s, \Sigma)) \cap V(D_j^s, \Sigma) = \varnothing$ for $j > i$

(3) $op_i = +F$ and $op_j = -G$ implies $F \cap G = \varnothing$ for every $i \neq j$

(4) For every $j > i$, $op_i = +F$ implies op_i is $(D_{i-1}^s - H, \Sigma)$-justified, where $H = \bigcup_{op_k = -G} G$ for all $i < k \leq j$

Let $\mathrm{RS}(D, \Sigma)$ be the set of all (D, Σ)-repairing sequences.

Note that the four requirements in the definition resemble the scenarios described above. Specifically, we impose that (1) every operation removes at least one violation, (2) no violation is reintroduced, (3) no fact is removed and added, and (4) every added fact stays justified. We observe further, that every repairing sequence itself and the set of repairing sequences is finite [5].

However, for some repairing sequence $s = (op_i)_{1 \leq i \leq n} \in RS(D, \Sigma)$, it is not guaranteed that the resulting database $D_n^s = s(D) = op_n(D_{n-1}^s)$ is consistent with respect to the constraints Σ. This comes from the fact that we may not be able to extend a repairing sequence without breaking one of the requirements (1) - (4), but the resulting database is still inconsistent.

3.3 Operational Repairs

An operational repair of a database D with respect to a set of constraints Σ is a database $D' = s(D)$ for some $s \in \mathrm{RS}(D,\Sigma)$ such that D' is consistent with respect to Σ. In the proposed framework we assign a probability to each operation depending on the current database and then derive the likelihood of a whole repairing sequence by multiplying the probabilities of the operations used in the sequence. Note that multiple repairing sequences can lead to the same operational repair. Thus, the likelihood of a particular repair is the sum of all sequences leading to this repair. In [5], this notion is formalised via a tree-shaped Markov Chain and its hitting distribution. However, as we are dealing with equally likely repairing sequences in this research, the concept of Markov Chains is not required, which is why we will omit the formal introduction into this topic and refer the interested reader to [5].

For a database D and a set of constraints Σ let (D', p) be the database-probability pair for each $D' = s(D)$, where $s \in \mathrm{RS}(D,\Sigma)$ and p is the sum of the likelihoods of all repairing sequences leading to this database. Let $\mathrm{OR}(D,\Sigma)$ be the set of all such (D', p), where D' is an operational repair with respect to Σ. We can now define the probability of a tuple $\bar{t}$ being in the answer to a query Q on an inconsistent database D with respect to constraints Σ as the following conditional probability [5]:

$$\mathrm{CP}_{D,\Sigma,Q}(\bar{t}) = \frac{\sum_{(D',p)\in \mathrm{OR}(D,\Sigma) \text{ and } \bar{t}\in Q(D')} p}{\sum_{(D',p)\in \mathrm{OR}(D,\Sigma)} p}$$

The above expression calculates the probability of the tuple being in the answer of the query on the resulting database of a repairing sequence, conditioned on the fact that this database is a repair. In [5] it is shown that obtaining the exact probability of a tuple being in the answer is intractable and therefore consider approximations to the problem. The goal of this research is to establish a similar result for the uniform setting, which is presented as an open problem in [5].

3.4 Uniform Operational Consistent Query Answering

As indicated before, in this research we are considering every repairing sequence to be equally likely. In order to be able to study this variation of the framework, we need to introduce new notations not provided with the original framework. The most important difference between the uniform and the probabilistic framework is that in our case we only have to count repairing sequences instead of adding their probabilities.

The *frequency* F of a database D' with respect to a database D and set of constraints Σ is defined as the count of repairing sequences leading to D'. Formally,

$$\mathrm{F}_{D,\Sigma}(D') = |\{s \in \mathrm{RS}(D,\Sigma) \,|\, D' = s(D)\}|.$$

As before, we call a database that was obtained by applying a repairing sequence s such that the resulting database conforms to its constraints a uniform operational repair. Now, we define the set of repair-frequency pairs analogously to the set of repair-probability pairs above.

$$\mathrm{UOR}(D,\Sigma) = \left\{(D', \mathrm{F}_{D,\Sigma}(D')) \,|\, D' \text{ is a uniform operational repair of } D \text{ w.r.t. } \Sigma\right\}$$

As every repairing sequence is assumed to be equally likely, we need to do the following in order to obtain the probability of a tuple being entailed in a query.Basically, we want to compute the ratio of the number of repairing sequences which lead to a consistent database and in which the tuple is in the query answer, and the number of repairing sequences which lead to a consistent database. Therefore, we need to compute each of the two numbers. For a database D, set of constraints Σ, query Q and tuple $\bar{t}$ let

$$\#q_{\Sigma,Q}(D,\bar{t}) = \sum_{(D',f)\in\mathrm{UOR}(D,\Sigma),\, \bar{t}\in Q(D')} f = \left|\left\{s \in \mathrm{RS}(D,\Sigma) \,|\, s(D) \models \Sigma \text{ and } \bar{t} \in Q(s(D))\right\}\right|$$

be the number of repairing sequences that lead to a consistent database such that $\bar{t}$ is in the query answer on that repair. Similarly, we denote the number of repairing sequences that lead to a repair as follows.

$$\#r_{\Sigma}(D) = \sum_{(D',f)\in\mathrm{UOR}(D,\Sigma)} f = \left|\left\{s \in \mathrm{RS}(D,\Sigma) \,|\, s(D) \models \Sigma\right\}\right|$$

Finally, we can define the relative frequency of $\bar{t}$ being in the query answer with respect to a database D, set of constraints Σ and query Q as the following ratio:

$$R_{D,\Sigma,Q}(\bar{t}) = \frac{\#q_{\Sigma,Q}(D,\bar{t})}{\#r_{\Sigma}(D)}$$

If D,Σ and Q are clear from context, we refer to this ratio simply as $R(\bar{t})$. Further, if Q is a Boolean query, we write $R_{D,\Sigma}(Q)$ instead of $R_{D,\Sigma,Q}(\langle\rangle)$, which we simplify to $R(Q)$, if D and Σ are clear from context.

In our uniform operational framework the answers to a query are tuple-frequency pairs $(\bar{t}, R(\bar{t}))$. The problem we are interested in is uniform operational consistent query answering (UOCQA), defined as follows.

Given a database D, a set of constraints Σ, a query Q, and a tuple $\bar{t}$ over constants from the database and constraints, we want to calculate $R_{D,\Sigma,Q}(\bar{t})$. In other words, we would like to calculate the ratio of repairing sequences in which the tuple is entailed. Note that we are generally interested in data-complexity [1]. This means the query and constraints are fixed and only the database is an input to the problem. We will now continue to study the complexity of UOCQA.

Chapter 4

Complexity Analysis

4.1 Complexity Classes NP and #P

Before we present the complexity analysis of UOCQA, we first give a brief introduction to the classes NP and #P. We say a decision problem L is in NP if there is a polynomial-time non-deterministic Turing Machine (NDTM) M which solves the problem [3]. That is, $x \in L \Leftrightarrow M(x) = 1$, or in other words, x is an instance of L if and only if an accepting path from the input x exists. Such an accepting path is a certificate for the membership of x in the problem.

Now, #P is the class of function problems which ask for the number of certificates to an NP problem. Specifically, we do not only want to know whether a certificate exists, but are rather interested in the number of certificates that satisfy the problem. More specifically, we define the class #P to contain all those functions f for which $f(x)$ is equal to the number of accepting paths in a polynomial-time NDTM with x as its input [3]. A canonical example of a problem in #P is #SAT, the problem of counting the satisfying assignments for a given Boolean formula.

In fact, #SAT is even #P-hard. That means every problem in #P is reducible to #SAT. To define reductions in #P, we need to introduce the class FP and the concept of an oracle. FP is the class of function problems computable by a deterministic Turing Machine (TM) in polynomial-time. We say a TM has oracle access to a function problem f if it can query this problem with a specific instance x in one computational step. The oracle then returns the answer f(x). We write FP^f for the "set of functions that are computable by polynomial-time TMs that have access to an oracle for f" [3]. In our research we are considering Turing-reductions. That means we reduce one problem f to another problem g by constructing a TM $M \in \text{FP}^g$ that computes f while

having oracle access to the problem g. Naturally, a function f is #P-complete if it is in #P and it is #P-hard, i.e. every $g \in$#P is in FP^f [3].

One might be tempted to claim that the counting version of a given problem is #P-hard, only if the problem is NP-hard. Surprisingly, this is not the case as there are also problems with an easy, sometimes even trivial, decision version, but the counting version is still #P-complete. Take for instance #IS, the problem that asks for the number of independent sets in a given graph. It is easy to see that an efficient, i.e. polynomial-time, algorithm for a #P-complete problem would imply that P=NP, since if we count the number of certificates of a problem, we can easily decide whether a certificate exists. More interestingly, according to Toda's Theorem, any problem in the polynomial hierarchy can be efficiently computed given access to an oracle for a #P-complete problem [9].

Note that a problem is #P-hard if and only if (iff) it is $\text{FP}^{\#\text{P}}$-hard. Analogously to other complexity classes, a problem is $\text{FP}^{\#\text{P}}$-hard, if every problem in $\text{FP}^{\#\text{P}}$ is reducible to that problem. The problems we are studying will all be #P-hard, but not in #P, as their output is a rational number rather than an integer. Thus, the problems will be $\text{FP}^{\#\text{P}}$-complete which still suggests that they are intractable, as just discussed.

4.2 Results

If we want to prove that a problem is $\text{FP}^{\#\text{P}}$-complete, we have to show that it is in $\text{FP}^{\#\text{P}}$ and that it is $\text{FP}^{\#\text{P}}$-hard. We will begin to show the first condition.

Theorem 4.1. *UOCQA is in $FP^{\#P}$ in data complexity.*

Proof. We first show that for every database D, set of constraints Σ, *FO* query $Q(\bar{x})$ and tuple $\bar{t} \in adom(D)^{|\bar{x}|}$, checking whether there exists a repairing sequence $s \in \text{RS}(D,\Sigma)$ such that $s(D) \models \Sigma$ and eventually $s(D) \models Q(\bar{t})$ is in NP. The claimed membership result follows from the fact that every sequence in $\text{RS}(D,\Sigma)$ is of length polynomial with respect to D and checking whether a given sequence s belongs to $\text{RS}(D,\Sigma)$ and satisfies $s(D) \models \Sigma$ and $s(D) \models Q(\bar{t})$ is feasible in polynomial time, in data complexity [5]. Since checking whether there exist such sequences is in NP, per definition of #P, counting such sequences is in #P. Thus, computing the numerator and denominator of $R(\bar{t})$ is in #P. It is now easy to devise a deterministic polynomial-time TM that makes two calls to a #P oracle in order to compute the numerator n and denominator d of $R(\bar{t})$. Finally, the machine outputs $\frac{n}{d}$. □

It is also easy to devise proofs for the #P-hardness of UOCQA when dealing with *FO* queries, which we will not lay out in detail as we continue to show stronger results. This means that we have $FP^{\#P}$-completeness for UOCQA in data-complexity for first-order queries, even if only one key or denial constraint or one inclusion dependency is imposed. However, we want to show that the problem remains hard, even for simpler classes of queries. Thus, the aim of this research was to find whether or not the intractability also holds for less expressive query languages. We first observe that the upper bound, i.e. the membership in $FP^{\#P}$, transfers to less expressive query languages that are contained in first-order queries. This is because a more general algorithm can of course also compute the answer to a special case of the possible input.

What remains to be done is proving the $FP^{\#P}$-hardness, or equivalently #P-hardness, for less powerful settings. If we could show, for example, that UOCQA is also $FP^{\#P}$-complete in data-complexity for conjunctive queries, we knew that the hardness is inherit in the problem and does not arise due to too complex input queries. In fact, we will obtain this result for inclusion dependencies in the next subsection. Following, we will first look at key constraints and lastly at denial constraints. Although we cannot show intractability for conjunctive queries in all cases, we are still able to significantly lower the expressiveness of the query language from *FO*.

4.2.1 Inclusion Dependencies

We will proceed to show our first result.

Theorem 4.2. *UOCQA is #P-hard in data complexity for conjunctive queries and inclusion dependencies.*

This will be proven by devising a Turing-reduction from the problem #MON-2DNF, the language of functions computing the number of satisfying assignments to a MON-2DNF formula φ. We will define a database and an inclusion dependency such that the resulting repairs encode assignments to the variables of φ. The query will then be entailed by the repair if and only if the associated assignment satisfies φ.

Definition 4.3 (#MON-2DNF)**.** A formula φ in MON-2DNF has the form $\varphi = \bigvee_{i\in[n]} C_i$, where each clause C_i is of the form $(u_i \wedge v_i)$ for all $i \in [n]$ and some positive variables u_i and v_i. The answer to the #P-complete problem #MON-2DNF [8] is the number of assignments to $Var(\varphi)$, the variables occurring in φ that satisfy the formula, denoted by $\#\varphi$.

Proof. To show #P-hardness, we will construct a Turing-reduction from #MON-2DNF. We will define a polynomial-time Turing Machine (TM) with access to an UOCQA-oracle for a fixed conjunctive query and a fixed inclusion dependency. The TM will take a MON-2DNF formula $\varphi = \bigvee_{i\in[n]} C_i$ as an input and compute $\#\varphi$. First, the TM constructs a database as follows. We associate a constant c_u with every variable u.

$$D = \{R(c_u) \mid u \in Var(\varphi)\} \cup \{J(c_u, c_v) \mid (u \wedge v) \in \{C_1, \ldots, C_n\}\}$$

The oracle for UOCQA calculates the ratio $R_{D,\Sigma}(Q)$ with respect to the following inclusion dependency

$$\Sigma = \{\sigma : R(x) \to S(x)\}$$

and the following conjunctive query

$$Q = \exists x, y\, J(x,y), S(x), S(y).$$

We will now take a closer look at the computed ratio $R_{D,\Sigma}(Q)$ with the previously defined input D, Σ and Q. First, we notice that there is precisely one violation for each constant c_u, namely $(\sigma, \{x \mapsto c_u\})$. Further, none of the violations interact with each other, which means we can look at each violation in isolation. There are exactly two different operations that repair one such violation, namely:

$$-R(c_u)$$
$$+S(c_u)$$

While the first one will be associated with setting the respective variable u to false, the latter is associated with setting it to true. This way, every repairing sequence $s \in \mathrm{RS}(D,\Sigma)$ naturally encodes an assignment to the Boolean variables of the formula. It is now easy to see that $s(D) \models Q$ iff the associated truth value assignment satisfies the given formula φ. In order for the formula to be satisfied, there has to exist a clause, encoded by the fact $J(c_u, c_v)$, such that both variables u and v are set to true. In our repair this would be encoded by the presence of the facts $S(c_u)$ and $S(c_v)$. Hence $\#\varphi$ equals the number of different repairs that entail the query.

Let $m = |Var(\varphi)|$. Note that there are $m!$ different repairing sequences leading to the same repair, and therefore representing the same assignment, due to the possible permutation of operations in the sequence. This means, we have $\#q_{\Sigma,Q}(D, \langle\rangle) = \#\varphi \cdot m!$ many repairing sequences entailing the query. As there are 2^m different assignments,

and therefore as many different repairs, the total number of repairing sequences is $\#r_\Sigma(D) = 2^m \cdot m!$. Hence, the oracle outputs

$$R(Q) = \frac{\#q_{\Sigma,Q}(D, \langle\rangle)}{\#r_\Sigma(D)} = \frac{\#\varphi \cdot m!}{2^m \cdot m!} = \frac{\#\varphi}{2^m}.$$

Our machine just has to multiply the oracle's answer, $R(Q)$, by 2^m to obtain $\#\varphi$, the desired number of satisfying assignments. As the constructed database is clearly of polynomial size with respect to the size of the given formula, the Turing Machine runs in polynomial time. □

4.2.2 Key Constraints

Next, we will establish the following theorem regarding key constraints.

Theorem 4.4. *UOCQA is #P-hard in data complexity for conjunctive queries with negations and key constraints.*

This will be proven by devising a Turing-reduction from the problem #IS, the number of independent sets in a given graph G.

Definition 4.5 (#Independence Set (#IS))**.** For a given graph $G = (V, E)$, $E \subseteq V \times V$, we want to count the number of independent sets. An independent set is a subset $S \subseteq V$ such that no two vertices in S are connected by an edge $e = (u, v) \in E$. Formally, we define our counting problem #IS as the problem of computing $\#\text{IS}(G)$, where G is the input graph and $\#\text{IS}(G)$ denotes the number of independent sets in G.

$$\#\text{IS}(G) = |\{S \subseteq V \mid \forall u, v \in S : (u, v) \notin E\}|$$

This problem is known to be #P-complete [8].

Proof. We will reduce this problem from #IS by constructing a Turing Machine which calculates the number of independent sets in a graph in polynomial time using an oracle for UOCQA with respect to a fixed conjunctive query with negations and a fixed key constraint. For every edge $(u, v) \in E$ either u or v or neither of both are included in any independent set. The basic idea of this construction is to use the constraint to have the edges "choose" which of the vertices are kept in the independent set. The query will then check whether for every vertex all edges agree on the membership of this vertex in the set, and thus filter out the invalid encodings of independent sets.

Given a graph $G = (V,E)$, where $|E| = m$, we enumerate the edges to be able to uniquely identify them and create an augmented set of edges. We assume therefore an arbitrary but fixed ordering of the edges of G.

$$E^{Enum} = \{(i,u,v) \mid (u,v) \text{ is the } i\text{-}th \text{ edge in } E\}$$

Now, we construct the following database, where we have a constant c_i for each edge i and a constant c_u for every vertex $u \in V$.

$$D = \{V(c_i,c_u), V(c_i,c_v) \mid (i,u,v) \in E^{Enum}\} \cup \{W(c_i,c_u), W(c_i,c_v) \mid (i,u,v) \in E^{Enum}\}$$

For each edge, we store its identifier with each of the vertices in the relation $V/2$ and a duplicate witness in the relation $W/2$. Next we define the fixed key constraint for the UOCQA-oracle.

$$\Sigma = \{\sigma : V(k,x), V(k,y) \rightarrow x = y\}$$

Now, every edge $(i,u,v) \in E^{Enum}$ participates in two violations $(\sigma, \{k \mapsto c_i, x \mapsto c_u, y \mapsto c_v\})$ and symmetrically $(\sigma, \{k \mapsto c_i, x \mapsto c_v, y \mapsto c_u\})$. We can still treat each edge in isolation, as the fixing operations are the same for both violations and no further violations occur. The three possible operations eliminating one such violation are the following:

$$\begin{aligned} &- V(c_i,c_u) \\ &- V(c_i,c_v) \\ &- \{V(c_i,c_u), V(c_i,c_v)\} \end{aligned}$$

We associate removing the vertex v from the independent set for that edge i with the deletion of the fact $V(c_i,c_v)$. As mentioned above, none of the vertices in an edge are required to be in the independent set and, therefore, we are allowed to remove both. Because one vertex can be connected by more than one edge, multiple facts in V can refer to the same vertex, being part of different edges. We want to filter out those repairing sequences, in which a vertex was removed for one edge, but not for another. This is what the following query accomplishes.

$$Q = \exists i,j,x\, W(j,x), V(i,x), \neg V(j,x)$$

We claim that there exists a bijection from the set of repairs not entailing the above query Q, denoted $\mathcal{R}(D,\Sigma,Q) = \{D' \mid \exists s \in \mathrm{RS}(D,\Sigma) : s(D) = D' \wedge D' \not\models Q\}$, to the set of independent sets of G, denoted $\mathrm{IS}(G)$. To show this, we introduce the notion of a

proper encoding of a set of vertices. Let $W(v) = \{(c_i, c_v) \,|\, W(c_i, c_v) \in D\}$ be the subset of witnesses involving the vertex v. A proper encoding of a subset $S \subseteq V$ is a database $D' = s(D), s \in \mathrm{RS}(D, \Sigma)$, for which the following holds:

$$\forall v \in V : \big(v \in S \Leftrightarrow \forall (c_i, c_v) \in W(v) : V(c_i, c_v) \in D'\big) \wedge \\ \big(v \notin S \Leftrightarrow \forall (c_i, c_v) \in W(v) : V(c_i, c_v) \notin D'\big)$$

Intuitively, for a vertex v this means that either all facts of the form $V(c_i, c_v)$, for some i, are simultaneously kept in the database or removed from it. Let $S_{D'}$ be the set of vertices encoded by D'. We note that every repair in $\mathcal{R}(D, \Sigma, Q)$ properly encodes a set of vertices, as shown by the following observation.

If a D' does not properly encode a subset $S_{D'} \subseteq V$, then it will entail the query Q and, thus, not be in the set $\mathcal{R}(D, \Sigma, Q)$. In an improper encoding, we have for at least one vertex v, connected by two or more edges i and j, that one of the facts $V(c_i, c_v)$ and $V(c_j, c_v)$ is removed in the repairing sequence while the other fact is kept. Let w.l.o.g. $V(c_j, c_v)$ be the removed fact. Because there is a witness for every fact, we know the fact $W(c_j, c_v)$ is present in the database D'. Now clearly $D' \models Q$, certified by the homomorphism ϕ defined by: $\phi(i) = c_i, \phi(j) = c_j, \phi(v) = c_v$.

We can now define a function from $\mathcal{R}(D, \Sigma, Q)$ to $\mathrm{IS}(G)$ as the set of vertices properly encoded by the repair, i.e. for a $D' \in \mathcal{R}(D, \Sigma, Q)$ we have $D' \mapsto S_{D'}$. What remains to be shown is that the encoded set of vertices is indeed an independent set and that the defined function is bijective, i.e. injective and surjective.

To prove that the encoded set is an independent set, it suffices to show that for all $(i, u, v) \in E^{Enum}$ no two facts $V(c_j, c_u), V(c_k, c_v)$ simultaneously exist in D', for some j and k. If these existed, both u and v would be in $S_{D'}$ as it is a proper encoded set of vertices. Therefore, $S_{D'}$ would not be an independent set. Let us fix an arbitrary edge $(i, u, v) \in E^{Enum}$ and let $(c_j, c_u) \in V$ for some j. As D' is a proper encoding, we know that $(c_i, c_u) \in V$, too. This in turn means that $V(c_i, c_v)$ was removed by an operation in the repairing sequence as both facts together violate the key constraint. Hence, for all k, $V(c_k, c_v)$ must also have been removed by one of the operations in the repairing sequence, because we would not have a proper encoding otherwise. Similarly, we can show if $(c_k, c_v) \in V$ then $(c_j, c_u) \notin V$. Thus, $S_{D'}$ is an independent set.

Next, we want to show that there are no two different repairs $D', D'' \in \mathcal{R}(D, \Sigma, Q)$, with $D' \neq D''$, which encode the same independent sets $S_{D'} = S_{D''}$, i.e. our function is injective. W.l.o.g. assume there is a fact contained in D' which is not contained in D'' and, thus, differentiates both repairs. As the relation W is untouched by the repairing

sequence, it is the same for both databases. Thus, there has to be a fact $V(c_i, c_u) \in D'$ while $V(c_i, c_u) \notin D''$. As both databases are proper encodings of sets of vertices, we know that $u \in S_{D'}$ but $u \notin S_{D''}$. Hence, $S_{D'} \neq S_{D''}$.

Lastly, we show that for every independent set in $\mathrm{IS}(G)$ there is a repair in $\mathcal{R}(D, \Sigma, Q)$ that encodes this independent set. In other words, given an independent set $S \subseteq V$, there has to exist a repair $D' = s(D), s \in \mathrm{RS}(D, \Sigma)$ which properly encodes S. We will define a repairing sequence leading to this repair. For every edge $(i, u, v) \in E^{Enum}$ apply the following operation in the repairing sequence:

$$\begin{aligned} -V(c_i, c_u), \text{ if } u \notin S, v \in S \\ -V(c_i, c_v), \text{ if } u \in S, v \notin S \\ -\{V(c_i, c_u), V(c_i, c_v)\}, \text{ if } u, v \notin S \end{aligned}$$

Note that it is not possible for both u and v to be in S, as S is an independent set. This basically means that for a vertex v, we keep all facts of the form $V(c_i, c_v)$ for all i, if v is part of the independent set and delete all of them otherwise. This is the very definition of a proper encoding.

Thus, we have a bijection from repairs not entailing the query to independent sets, which shows that both sets have the same cardinality. Hence, there are $\#\mathrm{IS}(G)$ many repairs that do not entail the query.

As before, there are $m!$ different repairing sequences leading to the same repair. Thus, the total number of repairing sequences is $\#r_\Sigma(D) = 3^m \cdot m!$. All of these, except those encoding an independent set, will produce a repair entailing the query Q. Hence, $\#q_{\Sigma,Q}(D, \langle\rangle) = (3^m - \#\mathrm{IS}(G)) \cdot m!$.

$$\begin{aligned} R(Q) &= \frac{\#q_{\Sigma,Q}(D, \langle\rangle)}{\#r_\Sigma(D)} = \frac{3^m \cdot m! - \#\mathrm{IS}(G) \cdot m!}{3^m \cdot m!} \\ &= 1 - \frac{\#\mathrm{IS}(G)}{3^m} \\ \#\mathrm{IS}(G) &= (1 - R(Q)) \cdot 3^m \end{aligned}$$

The Turing Machine simply performs the above calculation, subtracting the oracle's answer from 1 and multiplying the result by 3^m to obtain $\#\mathrm{IS}(G)$, the number of independent sets in the input graph. Clearly, our machine performs in polynomial time with respect to the given graph as the constructed database and enumerated set of edges are of polynomial size. □

4.2.3 Denial Constraints

The last category of constraints we explore are denial constraints, where we show two related results. In the first one, we show #P-hardness for denial constraints and unions of conjunctive queries with negations whereas in the second we only use queries from $CQ^{\neg}$, but allow constants in the constraint. We will first present the latter variation.

The proofs of this section are strongly related to the one for key constraints. In fact, the constructions are based on the ones above, only amended to accommodate the new form of constraint.

Theorem 4.6. *UOCQA is #P-hard in data complexity for conjunctive queries with negations and denial constraints with constants.*

Proof. In this case, we will adapt our previous reduction from #IS. Recall the definition of a graph $G = (V,E)$, $E \subseteq V \times V$ and the enumeration of edges E^{Enum}. We construct our database as follows:

$$D = \{V(c_i,c_u,c_0), V(c_i,c_v,c_1) \mid (i,u,v) \in E^{Enum}\} \cup$$
$$\{W(c_i,c_u,c_0), W(c_i,c_v,c_1) \mid (i,u,v) \in E^{Enum}\}$$

Again, we have a constant c_i for each edge i and a constant c_u for every vertex $u \in V$. Additionally, there are two constants c_0 and c_1 that will help us to distinguish both vertices of an edge in the constraint. The constraint and query with which the oracle will calculate the ratio $R_{D,\Sigma}(Q)$ are adapted as follows:

$$\Sigma = \{\sigma : V(k,x,c_0), V(k,y,c_1) \rightarrow \bot\}$$

$$Q = \exists i,j,x,a,b\, W(j,x,a), V(i,x,b), \neg V(j,x,a)$$

We can apply the same reasoning as before to establish the bijection from independent sets to repairs not entailing the query. The query also works the same way as before, as it checks whether a vertex x was deleted for some edge j but not for another edge i. As the edge and vertex together determine the third attribute of V, i.e. whether the vertex is on the c_0- or c_1-side of the edge, we know that for every homomorphism ϕ, the mappings $\phi(x), \phi(i)$ and $\phi(j)$ together imply $\phi(a)$ and $\phi(b)$. Thus, we can naturally incorporate the third attribute of V and W in our arguments.

Hence, $R(Q) = \frac{(3^m - \#\mathrm{IS}(G))}{3^m}$ and we can calculate the number of independent sets after the oracle call as follows:

$$\#\mathrm{IS}(G) = (1 - R(Q)) \cdot 3^m$$

□

Theorem 4.7. *UOCQA is #P-hard in data complexity for unions of conjunctive queries with negations and constant-free denial constraints.*

Proof. This variation of Theorem 4.6 trades a more expressive query language for a less powerful set of denial constraints. Again, the machine will construct a database D from a graph G and call an oracle for UOCQA with respect to some fixed query Q and a denial constraint σ. D, Σ and Q are defined as follows:

$$
\begin{aligned}
D = {} & \{V_0(c_i, c_u), V_1(c_i, c_v) \mid (i,u,v) \in E^{Enum}\} \cup \\
& \{W_0(c_i, c_u), W_1(c_i, c_v) \mid (i,u,v) \in E^{Enum}\} \\
\Sigma = {} & \{\sigma : V_0(k,x), V_1(k,y) \to \bot\} \\
Q = {} & \exists i,j,x \left(W_0(j,x), V_0(i,x), \neg V_0(j,x)\right) \vee \\
& \exists i,j,x \left(W_0(j,x), V_1(i,x), \neg V_0(j,x)\right) \vee \\
& \exists i,j,x \left(W_1(j,x), V_0(i,x), \neg V_1(j,x)\right) \vee \\
& \exists i,j,x \left(W_1(j,x), V_1(i,x), \neg V_1(j,x)\right)
\end{aligned}
$$

Analogue to the other constructions, there is a constant c_i for every edge i and a constant c_u for every vertex $u \in V$.

While in the previous proof we used a constant to distinguish the vertices of an edge, we now define different relations to store the vertices. Thus, we do not have to use constants in the denial constraint. As we cannot quantify over the relations in the query, we have to explicitly list all four options of how a repair might be an invalid encoding. Again, the arguments for this proof are analogue to those above and the machine outputs $\#\mathrm{IS}(G) = (1 - R(Q)) \cdot 3^m$. □

Chapter 5

Conclusion

In summary, we provided a formal framework for the study of uniform operational consistent query answering and have proven UOCQA to be $\text{FP}^{\#\text{P}}$-complete in data-complexity for a range of different configurations. The result holds for queries from CQ with inclusion dependencies, queries from $CQ^{\neg}$ with key constraints, queries from $CQ^{\neg}$ with denial constraints with constants, and queries from $UCQ^{\neg}$ with constant-free denial constraints.

Thus, for the above combinations of queries and constraints UOCQA cannot be efficiently computed. The main contribution of this research was to provide lower bounds for which the problem is intractable.

Future Work

However, further research is required to establish whether $\text{FP}^{\#\text{P}}$-hardness still holds for conjunctive queries and key or denial constraints, which was the aim of this research. Yet, we have made a significant step towards this goal. An approach which might be worthy to pursuit is to look at more restricted versions of #IS, like #BIS, that is concerned with the number of independent sets in a bipartite graph. This might relax the condition which we have to check with the query, such that it may become expressible in CQ. In general, #P-complete problems with an easy decision version seem to be suitable for this kind of reductions where we want to prove hardness with tools that are as restricted as possible.

Nonetheless, we also have to consider the possibility that there exists, in fact, a polynomial-time algorithm for conjunctive queries and key or denial constraints, as we could not find a proof for #P-hardness in our investigation.

When we encounter problems for which efficient exact solutions are unattainable, we look at possible approximations to the problem. It is planned to further investigate whether polynomial-time approximations either with multiplicative, i.e. relative, or additive, i.e. absolute, error guarantees exist. It was shown for the original, i.e. not uniform, framework for operational consistent query answering that the former type of approximation cannot exist, while an algorithm of the latter type was presented in [5]. Therefore we expect to find similar results in the uniform setting of OCQA.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 68–79. ACM, 1999.

[3] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[4] Leopoldo Bertossi. Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5), 2011.

[5] Marco Calautti, Leonid Libkin, and Andreas Pieris. An operational approach to consistent query answering. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 239–251. ACM, 2018.

[6] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1-2):90–121, 2005.

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

[8] J Scott Provan and Michael O Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal on Computing*, 12(4):777–788, 1983.

[9] Seinosuke Toda. Pp is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing*, 20(5):865–877, 1991.