UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ANSWERING QUERIES OVER INCONSISTENT DATABASES USING SAT SOLVERS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Akhil A. Dixit

September 2021

The Dissertation of Akhil A. Dixit
is approved:

Professor Phokion G. Kolaitis, Chair

Professor Seshadhri Comandur

Professor Benny Kimelfeld

Professor Carlos Maltzahn

Peter F. Biehl
Vice Provost and Dean of Graduate Studies

Contents

List of Figures

List of Tables

Abstract

Answering Queries Over Inconsistent Databases Using SAT Solvers

by

Akhil A. Dixit

An inconsistent database is a database that violates one or more integrity constraints, such as key constraints and functional dependencies. Consistent Query Answering (CQA) is a rigorous and principled approach to the semantics of queries posed against inconsistent databases. The consistent answers to a query on an inconsistent database are the intersection of the answers to the query on every repair, i.e., on every consistent database that differs from the given inconsistent one in a minimal way. Computing the consistent answers of a fixed conjunctive query on a given inconsistent database can be a coNP-hard problem, even though every fixed conjunctive query is efficiently computable on a given consistent database.

The notion of consistent answers is extended to the notion of range consistent answers for queries with aggregation operators with or without the grouping constructs. The range consistent answers to an aggregation query is the interval [glb, lub] of the greatest lower bound and the least upper bound such that the answer to the query on every repair falls within the range. Computing the range consistent answers to a fixed aggregation query can be an NP-hard problem. In fact, we show that computing the range consistent answers to an aggregation query can be NP-hard even if the consistent answers to its underlying conjunctive query (i.e., the query without aggregation

operators) are first-order rewritable, thus, computable in polynomial-time.

Despite several attempts towards building practical systems for consistent query answering, no comprehensive and scalable system for consistent query answering exists at present; this state of affairs has impeded the broader adoption of the framework of repairs and consistent answers as a principled alternative to data cleaning. We designed, implemented, and evaluated CAvSAT, the first SAT-based system for consistent query answering. CAvSAT leverages a set of natural reductions from the problem of computing the consistent answers to variants of the Boolean Satisfiability problem. The system is capable of handling unions of conjunctive queries and arbitrary denial constraints, which include functional dependencies as a special case. Moreover, it is also the first system capable of computing the range consistent answers of general aggregation queries with the COUNT(A), COUNT($*$), and SUM(A) operators, and with or without grouping constructs. We report results from experiments evaluating CAvSAT on both synthetic and real-world databases. We carry out an extensive set of experiments on both synthetic and real-world databases that demonstrate the usefulness and the scalability of CAvSAT.

Acknowledgments

It is overwhelming, profoundly joyous, and a bittersweet moment, and I am experiencing a whole bunch of mixed emotions that are difficult to put in words. This incredible journey wouldn't have been the same without the strong support system.

I want to express my deepest gratitude to my advisor Prof. Phokion G. Kolaitis, a mentor and a well-wisher who has always brought the best out of me. In 2017, he not only let me work on my Master's project under his supervision, but also kindly agreed to serve as my PhD advisor when I showed interest in expanding on my work. His immense knowledge, invaluable advice, and plentiful experience have encouraged me in all the time of my academic research and daily life. Over the years, he has provided me with several opportunities to attend and present my work at some of the top conferences, seminars, and workshops all across the globe.

I'm utmost grateful to Prof. Carlos Maltzahn and the Center for Research in Open Source Software (CROSS) at UC Santa Cruz for generously supporting me for more than three years, because of which I was able to work without distractions and deliver the best. Due to CROSS, I was constantly able to receive a valuable feedback on my work from the industry experts which, in my opinion, is rare in an academic setting. I am thankful to the Baskin School of Engineering at UC Santa Cruz for awarding me the Dissertation-Year Fellowship in the year 2020-2021.

I also want to thank the Simons Institute for the Theory of Computing at UC Berkeley, where I virtually spent the Spring 2021 semester as a visiting graduate student

and participated in the “Satisfiability: Theory, Practice, and Beyond” program; some interesting conversations with the fellow participants of the program led me to explore new directions of research in my area of study.

Last, but definitely not the least, I am thankful to the members of my dissertation reading committee – Prof. Seshadhri Comandur and Prof. Carlos Maltzahn from UC Santa Cruz, and Prof. Benny Kimelfeld from Technion - Israel Institute of Technology, for taking out their valuable time to read and thoroughly review my work.

Chapter 1

Introduction

Managing inconsistencies in databases is old, but a recurring, problem. An inconsistent database is a database that violates one or more integrity constraints, such as key constraints or functional dependencies. In the real world, inconsistent databases arise due to various reasons and in several different contexts, including data warehousing and information integration, where dealing with inconsistency is regarded as a key challenge [21]. For example, the gathering of data from heterogeneous sources where the data sources obey their own integrity constraints can result in violations, causing inconsistency at the time of data integration. This is not always preventable, because the data sources may be completely independent, and unaware of each other's integrity constraints. The incapability of database engines to handle complex integrity constraints may also be a reason behind real-world databases becoming inconsistent. Clearly, the inconsistency in the data is unwanted and it cannot simply be ignored because it may affect the quality of the answers to the queries posed against the databases. In the real

world, inconsistent databases are more of a norm than an exception, and they play a major role in a more general and broadly used term, *poor quality data*, which, according to a report from IBM, cost $3.1 trillion to the US economy in 2016 alone [2].

A widely used engineering method to deal with inconsistent databases is called *Data Cleaning*. The main idea of data cleaning is to perform simple database operations such as tuple-deletions, tuple-insertions, and tuple-updates to resolve the violations of the integrity constraints and come up with a single consistent version of the database, and use that version against the queries. This is the most common approach adopted by the industry today, with commercial data cleaning solutions like IBM Infosphere Quality Stage and Microsoft SQL Server Data Quality Services (DQS) available in the market. The problem with such an approach is that we often have to make arbitrary choices about what data to keep and what to remove, while transforming the inconsistent database into the consistent one [54]. For example, if a person has two social security numbers in a database, there may not be any apriori reason to decide which one to keep and which one to remove. Researchers have used heuristics, probabilistic inferences, learning algorithms, data deduplication, and many other techniques [49, 51, 74, 78] to make such choices seem less arbitrary, but inherently, data cleaning does not provide a strong guarantee about what data will be kept or removed in the cleaning process. The downside of this is that the same database cleaned up in different ways provides different answers to the same query, and this is an undesirable scenario.

The framework of *database repairs* and *consistent query answering*, introduced by Arenas, Bertossi, and Chomicki [15], is an alternative, principled, and more scientific

approach to managing inconsistent databases. In this framework, inconsistencies are handled at the query evaluation time by considering all possible *repairs* of the inconsistent database, where a repair of an inconsistent database $\mathcal{I}$ w.r.t. a set Σ of integrity constraints is a consistent database $\mathcal{J}$ that differs from $\mathcal{I}$ in a "minimal" way. The notion of minimality can be interpreted in different ways, thus giving rise to various types of repairs (see Section 2.1 and the monograph [22]). The *consistent answers* to a query q on a given database $\mathcal{I}$ w.r.t. a set Σ of integrity constraints is the intersection of the results of q applied on each repair of $\mathcal{I}$. Thus, a consistent answer provides the guarantee that it will be found no matter on what repair the query has been evaluated. The main algorithmic problem in this framework is to compute the *consistent answers* to a query q on a given database $\mathcal{I}$ (denoted by $\textsc{Cons}(q, \mathcal{I}, \Sigma)$), that is, the tuples that lie in the intersection of the results of q applied on each repair of $\mathcal{I}$, i.e.,

$$\textsc{Cons}(q, \mathcal{I}, \Sigma) = \bigcap \Big\{ q(\mathcal{J}) \mid \mathcal{J} \text{ is a repair of } \mathcal{I} \Big\}.$$

Computing the consistent answers can be an intractable problem, because an inconsistent database may have exponentially many repairs. In particular, computing the consistent answers to a fixed conjunctive query can be a coNP-complete problem, even though the query evaluation problem for a fixed conjunctive query is in P. By now, there is an extensive body of work on the complexity of consistent answers for conjunctive queries (see Section 2.2).

Although many practical queries can be expressed as conjunctive queries, the most frequently asked queries on real-world databases often involve aggregation and grouping; in SQL, these are the queries of the form

$$Q := \texttt{SELECT } Z, f(A) \texttt{ FROM } T(U, Z, A) \texttt{ GROUP BY } Z,$$

where $f(A)$ is one the standard aggregation operators `COUNT(`A`)`, `COUNT(*)`, `SUM(`A`)`, `AVG(`A`)`. `MIN(`A`)`, `MAX(`A`)`, and $T(U, Z, A)$ is the relation returned by a non-aggregation query q expressed in SQL. What is the semantics of an aggregation query over an inconsistent database? Since an aggregation query may return different answers on different repairs of an inconsistent database, there is typically *no* consistent answer as per the earlier definition of consistent answers. To obtain meaningful semantics to aggregation queries, the notion of consistent answers has been extended to the notion of the *range consistent answers*, first introduced by Arenas et al. [16] for aggregation queries without grouping, and then extended by Fuxman et al. [40] for aggregation queries with grouping. Here, we briefly describe the notion of range consistent answers; see Sections 2.1 and 2.2 for formal definitions. For an aggregation query Q without grouping, the set of *possible answers* to Q on an inconsistent instance $\mathcal{I}$ w.r.t. a set Σ of integrity constraints is the set of the answers to Q over all repairs of $\mathcal{I}$ w.r.t. Σ, i.e.,

$$\text{Poss}(Q, \mathcal{I}, \Sigma) = \bigcup \Big\{ Q(\mathcal{J}) \mid \mathcal{J} \text{ is a repair of } \mathcal{I} \text{ w.r.t. } \Sigma \Big\}.$$

By definition, the *range consistent answers* to Q on $\mathcal{I}$ is the interval $[glb(Q, \mathcal{I}), lub(Q, \mathcal{I})]$, where the endpoints of this interval are, respectively, the greatest lower bound (glb) and the least upper bound (lub) of the set $\text{Poss}(Q, \mathcal{I}, \Sigma)$ of possible answers to Q on $\mathcal{I}$. If Q is an aggregation query, $\textsc{Cons}(Q, \mathcal{I}, \Sigma)$ denotes the problem: given a database instance $\mathcal{I}$, compute the range consistent answers to Q on $\mathcal{I}$.

Range consistent answers have become the standard semantics of aggregation

queries in the framework of database repairs (see [22, Section 5.6]). Furthermore, range semantics have been adapted to give semantics to aggregation queries in several other contexts, including data exchange [13] and ontologies [56]. Finally, range semantics have been suggested as an alternative way to overcome some of the issues arising from SQL's handling of null values [48].

There have been multiple efforts to bridge the gap between theory and practice by building systems to compute the consistent answers or the range consistent answers to queries answering that leverage (some of) the known complexity results, but the theoretical research has not yet penetrated the mainstream industry. Several academic prototype systems for consistent query answering have been developed [17, 20, 26, 40, 41, 46, 55, 68, 71]. These systems use different approaches, including logic programming [20, 46], compact representations of repairs [25], or reductions to solvers [68, 55], and handle different classes of queries and integrity constraints. Among all these systems, however, only the ConQuer system by Fuxman et al. [40, 41] is capable of handling aggregation queries. Actually, ConQuer can only handle a restricted class of aggregation query, namely, those aggregation queries w.r.t. key constraints for which the underlying conjunctive query, i.e, the query without aggregation operators, belongs to the class called C_{forest}. For such a query Q, the range consistent answers of Q are SQL-rewritable, which means that there is a SQL query Q' such that the range semantics answers of Q on an instance $\mathcal{I}$ can be obtained by directly evaluating Q' on $\mathcal{I}$. This leaves out, however, many aggregation queries, including all aggregation queries whose range consistent answers are not SQL-rewritable or are NP-hard to compute. Up to now, no

system supports such queries. Therefore it is fair to say, that no comprehensive and scalable system for consistent query answering exists at present, and this state of affairs has impeded the broader adoption of the framework of repairs and consistent answers as a principled alternative to data cleaning. The main reason behind this appears to be that the high computational complexity of $\textsc{Cons}(q, \mathcal{I}, \Sigma)$ even for restricted classes of queries and integrity constraints makes it difficult to build a fast, comprehensive, and scalable system capable of handling real-world databases.

In this thesis, we use SAT solving, for the first time, to build a system for consistent query answering that can handle broad classes of practical queries and integrity constraints. Boolean Satisfiability (SAT) is a prototypical NP-complete problem [52] and there has been extensive amount of work on different aspects of satisfiability and SAT solving (see Section 2.1.8 and the handbook [23]). Modern SAT solvers are capable of solving quickly SAT-instances with millions of clauses and variables. Over the years, SAT solvers have been widely used in both academia and industry as general-purpose problem-solving tools. Indeed, many real-world problems from a variety of domains, including scheduling, protocol design, software verification, and model checking, can be naturally encoded as SAT instances, and solved quickly using solvers, such as Glucose [19] and CaDiCaL [1]. Furthermore, SAT solvers have been used in solving open problems in mathematics [50, 73]. In our work, we leverage the progress made by the SAT solving community to build a fast, comprehensive, and scalable real-world system for answering queries over inconsistent databases. In what follows, we summarize the contributions of this thesis.

Summary of Contributions

- On the foundational side, we develop several polynomial-time reductions from computing the consistent answers to different classes of database queries, including aggregation queries, to SAT and its optimization variants. The most basic reduction is for the conjunctive queries over databases having primary key constraints and it is heavily based on the reduction to Binary Integer Programming from the EQUIP system [55]. For broader classes of queries and integrity constraints such as unions of conjunctive queries with aggregation and grouping over databases with arbitrary denial constraints, we develop much more sophisticated reductions.
- We give a new hardness result about computing the range consistent answers to an aggregation query whose underlying conjunctive query has SQL-rewritable consistent answers. Specifically, we show that, unlike the conjunctive queries in the class C_{forest}, the SQL-rewritability of the consistent answers is not preserved after adding an aggregation operator on top of an arbitrary self-join-free conjunctive query having SQL-rewritable consistent answers.
- We present a comprehensive and scalable system, CAvSAT (Consistent Answers via SAT Solving), for answering queries over inconsistent databases. The distinctive feature of CAvSAT is that it uses several natural reductions from computing the consistent answers of queries to SAT and its optimization variants, and then deploys powerful SAT solvers. CAvSAT can handle a wide range of practical queries and integrity constraints, namely, unions of conjunctive queries with or without aggregation

and grouping, over database schemata with arbitrary denial constraints.

- We report an extensive experimental evaluation of CAvSAT. We carried out a suite of experiments on both synthetically generated databases and real-world databases, and for a variety of queries with and without aggregation and grouping. We also demonstrate the features and usefulness of the system with a Graphical User Interface.

The rest of the thesis is organized as follows. In Chapter 2, we familiarize the reader with the necessary background material and basic notions and give an account of the past research on consistent query answering. In Chapter 3, we present the natural polynomial-time reductions from computing the consistent answers to queries without aggregation to SAT and its optimization variants. We published this work as an extended abstract in the Proceedings of the 2019 International Conference on Management of Data (ACM SIGMOD 2019) [32] and as a conference paper in the Proceeding of the International Conference on Theory and Applications of Satisfiability Testing (SAT 2019) [33, 34]. In Chapter 4, we present reductions from computing the range consistent answers to aggregation queries to SAT and its variants; this work is currently under peer review (see the arXiv version [36]). We then present the architecture and the implementation details of the CAvSAT system in Chapter 5. We demonstrated CAvSAT at the 2021 International Conference on Management of Data (ACM SIGMOD 2021) [35]. In Chapter 6, we report the experimental analysis of CAvSAT, and we conclude the thesis in Chapter 7 with a discussion on the open problems in consistent query answering and the directions for future research.

Chapter 2

Preliminaries and Related Work

2.1 Background and Basic Notions

2.1.1 Relational Database Schemata and Database Instances

A *relational database schema* or, *database schema* $\mathcal{R}$ is a finite collection of relation symbols, each with a fixed positive integer as its arity. The attributes of a relation symbol are names for its columns; attributes can also be identified by their positions, thus $Attr(R) = \{1, ..., n\}$ denotes the set of attributes of R. An $\mathcal{R}$-*database instance* or, simply, an $\mathcal{R}$-*instance* is a collection $\mathcal{I}$ of finite relations $R^{\mathcal{I}}$, one for each relation symbol R in $\mathcal{R}$. An expression of the form $R^{\mathcal{I}}(a_1, ..., a_n)$ is a *fact* of the instance $\mathcal{I}$ if $(a_1, ..., a_n) \in R^{\mathcal{I}}$. Every $\mathcal{R}$-instance can be identified with the (finite) set of its facts. When it is clear from the context, we often drop the superscript in $R^{\mathcal{I}}$ and simply use the symbol R to refer to the relation $R^{\mathcal{I}}$. The *active domain* of $\mathcal{I}$ is the set of all values occurring in facts of $\mathcal{I}$. For example, a university database schema in Example 2.1.1

has six relation symbols, namely, *Students*, *Courses*, *Enrollments*, *Faculty*, *Teaches*, and *Departments* with arities three, four, two, four, two, and three respectively. We will use this database schema and its instance shown in Example 2.1.1 as a running example throughout this section.

Example 2.1.1. *A database schema $\mathcal{R}$ of university records and an $\mathcal{R}$-instance $\mathcal{I}$.*

$\mathcal{R}$ = *Students(SID, NAME, DEPT), Courses(CID, NAME, DEPT, CREDIT),*
Enrollments(SID, CID), Faculty(FID, NAME, DEPT, RANK),
Teaches(FID, CID), Departments(DID, CHAIR, LOCATION).

An $\mathcal{R}$-instance $\mathcal{I}$:

Students

SID	NAME	DEPT
S1	Sai	CSE
S2	John	AMS
S3	Mary	CSE
S4	Seb	ECE

Courses

CID	NAME	DEPT	CREDIT
C1	Algorithms	CSE	5
C2	Topology	AMS	2
C3	Databases	CSE	2
C4	Mechatronics	ECE	5

Enrollments

SID	CID
S1	C1
S1	C3
S2	C2
S3	C4

Faculty

FID	NAME	DEPT	RANK
F1	Ben	CSE	Professor
F2	Jen	AMS	Lecturer
F3	Mick	ECE	Professor

Teaches

FID	CID
F1	C1
F1	C3
F2	C2
F3	C4

Departments

DID	CHAIR	LOCATION
CSE	F1	Engineering 2
AMS	F4	Baskin Engineering
ECE	F3	Baskin Engineering

2.1.2 Integrity Constraints

Relational database schemata are often accompanied by a set of integrity constraints that impose semantic restrictions on the allowable instances. We give definitions and examples of some of the commonly used integrity constraints here, specifically, the ones that are heavily used in the subsequent chapters.

Definition 2.1.1. Functional Dependency. *A functional dependency (FD) $\vec{x} \rightarrow \vec{y}$ on a relation symbol R is an integrity constraint asserting that if two facts agree on the attributes in $\vec{x}$, then they must also agree on the attributes in $\vec{y}$.*

For example, a constraint that one faculty member can serve as a chair of at most one department can be expressed using functional dependency CHAIR $\rightarrow$ DID on the *Departments* relation. Another constraint that no course is taught by more than one faculty can be written as a functional dependency CID $\rightarrow$ FID on the *Teaches* relation.

Definition 2.1.2. Primary Key. *A primary key (or, simply, key) is a minimal subset $\vec{x}$ of $Attr(R)$ such that the functional dependency $\vec{x} \rightarrow Attr(R)$ holds.*

Thus, every key constraint is a functional dependency, but not vice versa. In a key constraint $\vec{x} \rightarrow Attr(R)$, the attributes in $\vec{x}$ are called the *key attributes* of R and they are denoted by underlining their corresponding positions. As shown in Example 2.1.1, *Courses*(<u>CID</u>, NAME, DEPT, CREDIT) denotes that the attribute CID is a key of the *Courses* relation, while *Enrollments* and *Teaches* relations have no key constraints.

Every functional dependency is expressible in first-order logic. For example, the functional dependency CHAIR $\rightarrow$ DID in *Departments*(<u>DID</u>, CHAIR, LOCATION),

is expressed by the first-order formula

$$\forall x, x', y, z, z' (\mathit{Departments}(x, y, z) \wedge \mathit{Departments}(x', y, z') \rightarrow x = x').$$

A more generalized class of dependencies is the *equality generating dependencies (EGDs)*, which are the first-order sentences of the form

$$\forall x_1, ..., x_n (\varphi(x_1, ..., x_n) \rightarrow x_i = x_j),$$

where $\varphi(x_1, ..., x_n)$ is a conjunction of atomic formulas. An equality generating dependency enforces an equality between two attributes, if certain tuples appear together in a database. For example, we may want to enforce a natural condition that a chair of a department must be a faculty of the same department; this can be expressed using an equality generating dependency

$$\forall w, x, y, y', z, u \ (\mathit{Faculty}(w, x, y, z) \wedge \mathit{Departments}(y', w, u) \rightarrow (y = y')).$$

Even though the equality generating dependencies are greatly helpful in designing the database schema with the desired constraints, there is still a big space of integrity constraints that cannot be captured by EGDs. The primary keys, the FDs, and the EGDs are important special cases of *denial constraints (DCs)*, which are expressible by first-order formulas of the form

$$\forall x_1, ..., x_n \neg (\varphi(x_1, ..., x_n) \wedge \psi(x_1, ..., x_n)),$$

or, equivalently,

$$\forall x_1, ..., x_n (\varphi(x_1, ..., x_n) \rightarrow \neg \psi(x_1, ..., x_n)),$$

where $\varphi(x_1, ..., x_n)$ is a conjunction of atomic formulas and $\psi(x_1, ..., x_n)$ is a conjunction of expressions of the form $(x_i \text{ op } x_j)$ with each op a built-in predicate, such as $=, \neq, <, >, \leq, \geq$. Since all variables in a denial constraint are universally quantified, we often skip them when it is clear from the context, and simply write the constraint as

$$\neg(\varphi(x_1, ..., x_n) \wedge \psi(x_1, ..., x_n)).$$

In words, a denial constraint prohibits a set of tuples that satisfy certain conditions from appearing together in a database instance. For example, a restriction that a single course cannot offer more than five credits can be expressed using a denial constraint

$$\forall w, x, y, z \neg(\mathit{Courses}(w, x, y, z) \wedge (z > 5))$$

or, equivalently,

$$\forall w, x, y, z \ (\mathit{Courses}(w, x, y, z) \rightarrow (z \leq 5)),$$

or, simply,

$$\neg(\mathit{Courses}(w, x, y, z) \wedge (z > 5)).$$

There are broader classes of integrity constraints, e.g., *universal constraints (UCs)*, but they are out of scope of this thesis. We have the following relationship between the aforementioned classes of integrity constraints, in terms of their expressive power [22].

$$\text{Keys} \subset \text{FDs} \subset \text{EGDs} \subset \text{DCs} \subset \text{UCs}.$$

2.1.3 Inconsistent Database Instances and Database Repairs

Definition 2.1.3. Inconsistent Database Instance. *Let $\mathcal{R}$ be a database schema, Σ a set of integrity constraints on $\mathcal{R}$, and $\mathcal{I}$ an $\mathcal{R}$-instance. The $\mathcal{R}$-instance $\mathcal{I}$ is called an inconsistent database instance if $\mathcal{I}$ violates Σ (denoted by $\mathcal{I} \not\models \Sigma$).*

For example, suppose we have a constraint that one faculty can teach at most one course, i.e., a functional dependency FID $\rightarrow$ CID on the *Teaches* relation, then the database instance $\mathcal{I}$ from Example 2.1.1 would be inconsistent because the facts *Teaches*(F1, C1) and *Teaches*(F1, C3) form a violation, as they imply that the faculty F1 teaches two courses, namely, C1 and C3.

Clearly, the inconsistency in database instances is unwanted. To restore consistency in an inconsistent database instance, Arenas et al. [15] proposed a notion of *database repairs*. Informally, the database repairs are the consistent instances that are "close" to the original inconsistent database instance. It is quite natural to have the condition that a database repair should be "close" or "minimally" different from the original inconsistent database. To make this precise, Arenas et al. [15] proposed a partial order to compare the database instances in terms of their distances to a given database instance $\mathcal{I}$. We first define this partial order and then the database repairs.

Definition 2.1.4. Partial Order of Database Instances. *Let $\mathcal{R}$ be a database schema and $\mathcal{I}$ be an $\mathcal{R}$-instance. For two database instances $\mathcal{J}$ and $\mathcal{J}'$, we say that $\mathcal{J}$ is at least as close to $\mathcal{I}$ as $\mathcal{J}'$ (denoted as $\mathcal{J} \leq_{\mathcal{I}} \mathcal{J}'$) if and only if $\Delta(\mathcal{J},\mathcal{I}) \subseteq \Delta(\mathcal{J}',\mathcal{I})$, where $\Delta(\mathcal{J},\mathcal{I})$ is the symmetric difference between the instances $\mathcal{J}$ and $\mathcal{I}$ when they are*

viewed as the sets of facts, i.e., $\Delta(\mathcal{J},\mathcal{I}) = (\mathcal{J}\backslash\mathcal{I}) \cup (\mathcal{I}\backslash\mathcal{J})$. *Naturally,* $\mathcal{J} <_{\mathcal{I}} \mathcal{J}'$ *holds if and only if* $\mathcal{J} \leq_{\mathcal{I}} \mathcal{J}'$, *but not* $\mathcal{J}' \leq_{\mathcal{I}} \mathcal{J}$.

Definition 2.1.5. Database Repairs. *Let* $\mathcal{R}$ *be a database schema,* Σ *a set of integrity constraints on* $\mathcal{R}$, *and* $\mathcal{I}$ *an* $\mathcal{R}$*-instance. A database instance* $\mathcal{J}$ *is a database repair of* $\mathcal{I}$ *w.r.t.* Σ *if* $\mathcal{J}$ *satisfies* Σ, *i.e.,* $\mathcal{J} \models \Sigma$, *and* $\mathcal{J}$ *is* $\leq_{\mathcal{I}}$*-minimal in the* $\mathcal{R}$*-instances that satisfy* Σ.

It is crucial to decide the types of database operations such as tuple-deletions, tuple-insertions, and tuple-updates that are allowed to be performed on an inconsistent database instance to obtain consistent database instances. Based on the allowable database operations, several repair semantics such as tuple- and set-inclusion-based repairs, tuple-deletion and set-inclusion-based repairs, tuple- and cardinality-based-repairs have been investigated (see [22] for a comprehensive survey on different repair semantics). Here, we focus on *subset repairs*, the most widely studied type of database repairs. The subset repairs are the database repairs obtained by performing only tuple-deletions on the inconsistent database instance. We formally define *subset repairs* next.

Definition 2.1.6. Subset Repairs. *Let* $\mathcal{R}$ *be a database schema,* Σ *a set of integrity constraints on* $\mathcal{R}$, *and* $\mathcal{I}$ *an* $\mathcal{R}$*-instance. A database instance* $\mathcal{J}$ *is a subset repair (or simply, a repair) of* $\mathcal{I}$ *w.r.t.* Σ *if* $\mathcal{J} \subset \mathcal{I}$, $\mathcal{J}$ *satisfies* Σ, *i.e.,* $\mathcal{J} \models \Sigma$, *and there is no other database instance* $\mathcal{J}'$ *such that* $\mathcal{J} \subset \mathcal{J}' \subset \mathcal{I}$ *and* $\mathcal{J}' \models \Sigma$.

In other words, subset repairs are the maximal consistent subsets of the original inconsistent database instance. For example, consider again the database instance $\mathcal{I}$

in Example 2.1.1 and a functional dependency CID $\rightarrow$ FID on the *Teaches* relation. Clearly, there are two possible ways to repair $\mathcal{I}$, i.e., by deleting one of the two facts *Teaches*(F1, C1) and *Teaches*(F1, C3). The consistent database instance obtained by deleting both facts is not $\leq_{\mathcal{I}}$-minimal in all consistent subsets of $\mathcal{I}$, hence we do not obtain a subset repair. From now on, we refer to subset repairs as, simply, repairs.

It is easy to see that there are multiple ways to repair a single inconsistent database instance, and it is important to realize that, in general, the number of repairs can be exponential in the size of the original database even in case of just primary key constraints. We illustrate this using Example 2.1.2.

Example 2.1.2. *Consider a schema $\mathcal{R} = R(\underline{A}, B)$ and a set Σ of integrity constraints on $\mathcal{R}$ that contains just one key constraint $A \rightarrow B$ on the relation R. Let an $\mathcal{R}$-instance $\mathcal{I}$ be a set of facts $R(a_1, b_1)$, $R(a_1, b_2)$, $R(a_2, b_1)$, $R(a_2, b_2)$, $\cdots$, $R(a_n, b_1)$, $R(a_n, b_2)$. Clearly, there are $2n$ facts in $\mathcal{I}$, but the number of repairs of $\mathcal{I}$ w.r.t. Σ is 2^n.*

2.1.4 Database Queries: Conjunctive Queries

Let k be a positive integer. A *k-ary query* on a relational database schema $\mathcal{R}$ is a function q that takes an $\mathcal{R}$-instance $\mathcal{I}$ as argument and returns a k-relation $q(\mathcal{I})$ on the active domain of $\mathcal{I}$ as value. A *boolean query* on $\mathcal{R}$ is a function that takes an $\mathcal{R}$-instance $\mathcal{I}$ as argument and returns true or false as value. As is well known, first-order logic has been successfully used as a query language. In fact, it forms the core of SQL, the main commercial database query language.

A *conjunctive query* is a first-order formula built using relational symbols,

conjunctions, and existential quantifiers. Thus, each conjunctive query is expressible by a first-order formula of the form

$$q(\vec{z}) := \exists \vec{w}\ (R_1(\vec{x_1}) \wedge ... \wedge R_m(\vec{x_m})),$$

where each $\vec{x_i}$ is a tuple consisting of variables and constants, $\vec{z}$ and $\vec{w}$ are tuples of variables, and the variables in $\vec{x_1}, ..., \vec{x_m}$ appear in exactly one of $\vec{z}$ and $\vec{w}$. The variables in $\vec{z}$ are called the free variables while the ones in $\vec{w}$ are existentially quantified. The existentially quantified variables are sometimes omitted while writing a conjunctive query because it is obvious that the variables that are not free are quantified existentially. Clearly, a conjunctive query with k free variables in $\vec{z}$ is a k-ary query, while a conjunctive query with no free variables (i.e., all variables are existentially quantified) is a boolean query. A conjunctive query is *self-join-free* if no relation symbol appears more than once in it. Conjunctive queries are also known as the *select-project-join* (SPJ) queries and are among the most frequently asked queries in databases. For example, a database query that returns the set of all pairs (x, y) of a student name x and a faculty name y such that the student x is enrolled in a course taught by faculty y can be expressed as a binary conjunctive query $q(x, y)$ as follows.

$$q(x, y) := \exists s, t, u, v, w, z\ (\mathit{Enrollments}(u, z) \wedge \mathit{Teaches}(v, z) \wedge \mathit{Faculty}(v, y, s, t) \wedge \mathit{Students}(u, x, w))$$

Equivalently, this query can be written in SQL as follows.

```
SELECT STUDENTS.NAME, FACULTY.NAME
FROM ENROLLMENTS, TEACHES, FACULTY, STUDENTS
```

```
WHERE ENROLLMENTS.CID = TEACHES.CID
      AND TEACHES.FID = FACULTY.FID
      AND ENROLLMENTS.SID = STUDENTS.SID.
```

Similarly, a query to check whether the CSE department offers a five-credit course can be written as a boolean conjunctive query q' as follows.

$$q'() := \exists x, y\ (Courses(x, y, \text{'CSE'}, 5))$$

In SQL, it can be written as

```
SELECT 1 FROM COURSES
WHERE COURSES.DEPT = 'CSE'
      AND COURSES.CREDITS = 5.
```

In the subsequent chapters, we will also be dealing with a class of queries, namely, *unions of conjunctive queries*. A k-ary union q of n conjunctive queries is a disjunction

$$q(\vec{z}) := q_1(\vec{z_1}) \cup q_2(\vec{z_2}) \cup \cdots \cup q_n(\vec{z_n})$$

where each q_i is a conjunctive query of arity k. Importantly, unions of conjunctive queries are strictly more expressive than conjunctive queries.

2.1.5 Consistent Answers to Conjunctive Queries

Arenas, Bertossi, and Chomicki [15] used the notion of database repairs to give rigorous semantics to query answering on inconsistent databases. They precisely defined the consistent information in a database as the information that is invariant w.r.t. all

possible repairs. In particular, when this characterization is applied to query answers, we obtain what is known as *consistent answers* (or, *certain answers*) to the query on an inconsistent database instance. The consistent answers offer a strong semantic guarantee that no matter how is the inconsistent database instance repaired, the consistent answers to the given query remain the same.

Definition 2.1.7. Consistent Answers. *Assume that $\mathcal{R}$ is a database schema, Σ is a set of integrity constraints on $\mathcal{R}$, $\mathcal{I}$ is an $\mathcal{R}$-instance, q is a query, and $\vec{t}$ is a tuple of values. We say that $\vec{t}$ is a consistent answer to q on $\mathcal{I}$ w.r.t. Σ if $\vec{t} \in q(\mathcal{J})$, for every repair $\mathcal{J}$ of $\mathcal{I}$. We write* $\textsc{Cons}(q, \mathcal{I}, \Sigma)$ *to denote the set of all consistent answers to q on $\mathcal{I}$ w.r.t. Σ, i.e.,* $\textsc{Cons}(q, \mathcal{I}, \Sigma) = \bigcap \{q(\mathcal{J}) : \mathcal{J}$ *is a repair of $\mathcal{I}$ w.r.t. $\Sigma\}$.*

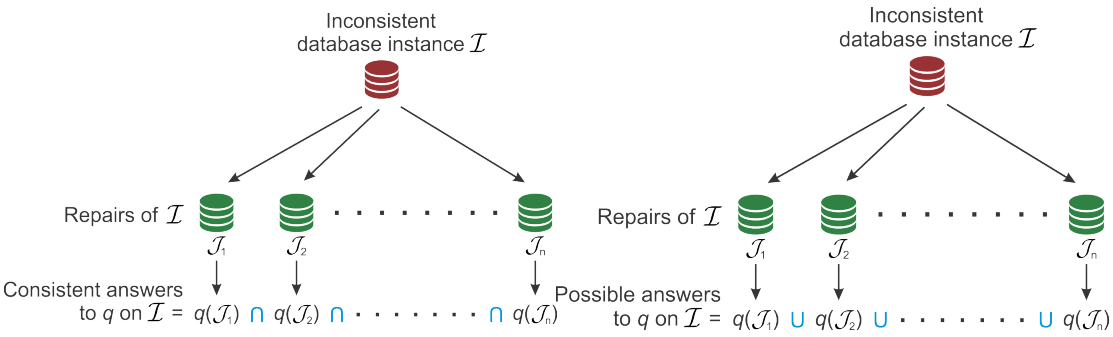


Figure 2.1: Consistent answers $\textsc{Cons}(q, \mathcal{I})$ and possible answers $\textsc{Poss}(q, \mathcal{I})$

If Σ is a fixed set of integrity constraints and q is a fixed query, then the main computational problem associated with the consistent answers asks: given an $\mathcal{R}$-instance $\mathcal{I}$, compute $\textsc{Cons}(q, \mathcal{I}, \Sigma)$. If q is a boolean query, then computing the consistent answers becomes the decision problem $\textsc{Certainty}(q, \Sigma)$: given an $\mathcal{R}$-instance

$\mathcal{I}$, is q true on every repair $\mathcal{J}$ of $\mathcal{I}$ w.r.t. Σ? When the set Σ of integrity constraints is understood from the context, we will write $\textsc{Cons}(q, \mathcal{I})$ and $\textsc{Certainty}(q)$, instead of $\textsc{Cons}(q, \mathcal{I}, \Sigma)$ and $\textsc{Certainty}(q, \Sigma)$. Computing the consistent answers to a query q on a database instance $\mathcal{I}$ can be computationally harder than evaluating q on $\mathcal{I}$, because the instance $\mathcal{I}$ may have exponentially many repairs. In the subsequent chapters, we also use the notions of *potential answers* and *possible answers*, that we introduce next.

Definition 2.1.8. Potential Answers. *For a database schema $\mathcal{R}$, an $\mathcal{R}$-instance $\mathcal{I}$, and a query q, the potential answers to q on $\mathcal{I}$ is the set $q(\mathcal{I})$, i.e., the set of answers obtained by evaluating q on $\mathcal{I}$.*

Definition 2.1.9. Possible Answers. *Assume that $\mathcal{R}$ is a database schema, Σ is a set of integrity constraints on $\mathcal{R}$, $\mathcal{I}$ is an $\mathcal{R}$-instance, q is a query, and $\vec{t}$ is a tuple of values. We say that $\vec{t}$ is a possible answer to q on $\mathcal{I}$ w.r.t. Σ if $\vec{t} \in q(\mathcal{J})$, for some repair $\mathcal{J}$ of $\mathcal{I}$. We write $\textsc{Poss}(q, \mathcal{I}, \Sigma)$ to denote the set of all possible answers to q on $\mathcal{I}$ w.r.t. Σ, i.e., $\textsc{Poss}(q, \mathcal{I}, \Sigma) = \bigcup \{q(\mathcal{J}) : \mathcal{J}$ is a repair of $\mathcal{I}$ w.r.t. $\Sigma\}$.*

See Figure 2.1 for a pictorial representation of the consistent answers and the possible answers to a query. The unions of conjunctive queries are monotonous in nature, i.e., for a union q of conjunctive queries over two database instances $\mathcal{I}$ and $\mathcal{I}'$, we have that $\mathcal{I} \subseteq \mathcal{I}' \implies q(\mathcal{I}) \subseteq q(\mathcal{I}')$. Therefore, as far as the subset repairs are considered, we have the following relationship between the consistent answers, the possible answers, and the potential answers to a union q of conjunctive queries over an $\mathcal{R}$-instance $\mathcal{I}$.

$$\textsc{Cons}(q, \mathcal{I}) \subseteq \textsc{Poss}(q, \mathcal{I}) \subseteq \textsc{Pot}(q, \mathcal{I}).$$

Example 2.1.3. *Consider the database instance from Example 2.1.1 with a functional dependency constraint FID → CID on the Teaches relation, and a query q that asks for the name of the faculty who teaches the Algorithms course, i.e,*

$$q(x) := \mathit{Faculty}(w, x, y, z) \wedge \mathit{Teaches}(w, v) \wedge \mathit{Courses}(v, \text{`Algorithms'}, t, u).$$

It is easy to see that 'Ben' is the only potential answer and the possible answer to q on $\mathcal{I}$. However, 'Ben' is not a consistent answer because the database instance leaves open the possibility that Ben may be teaching a course on Databases and not Algorithms; thus, we have that $\textsc{Cons}(q, \mathcal{I}) = \emptyset$.

2.1.6 Database Queries: Aggregation Queries

In Section 2.1.4, we introduced unions of conjunctive queries. Although they cover a broad class of practical queries, the most frequently asked database queries in practice often involve standard aggregation operators `COUNT(`A`)`, `COUNT(*)`, `SUM(`A`)`, `AVG(`A`)`, `MIN(`A`)`, `MAX(`A`)`, and, possibly, a `GROUP BY` clause for grouping of results. In what follows, we will use the term *aggregation queries* to refer to queries with aggregate operators and with or without a `GROUP BY` clause. Thus, in full generality, an aggregation query can be expressed as

$$Q := \texttt{SELECT}\ Z, f(A)\ \texttt{FROM}\ T(U, Z, A)\ \texttt{GROUP BY}\ Z,$$

where $f(A)$ is one the aforementioned aggregation operators and $T(U, Z, A)$ is the relation returned by a query q, which typically is a conjunctive query or a union of conjunctive queries expressed in SQL. This query q is called the *underlying query* of Q,

the attribute represented by the variable A is called the *aggregation attribute*, and the attributes represented by Z are called the *grouping attributes*. An aggregation query is called *scalar* if it does not contain the `GROUP BY` clause, i.e., an aggregation query of the form $Q :=$ `SELECT` $f(A)$ `FROM` $T(U, A)$.

For example, the total number of students belonging to the CSE department can be found by using a scalar aggregation query

```
SELECT COUNT(*) FROM Students
WHERE Students.DEPT = 'CSE',
```

while one can find the total number of credits taught by each faculty by evaluating a non-scalar aggregation query

```
SELECT Teaches.FID, SUM(Courses.CREDIT)
FROM Courses, Teaches
WHERE Teaches.FID = Courses.FID
GROUP BY Teaches.FID.
```

2.1.7 Range Consistent Answers to Aggregation Queries

What is the semantics of an aggregation query over an inconsistent database? It is often the case that an aggregation query returns different answers on different repairs of an inconsistent database; thus, even for a scalar aggregation query, there is typically *no* consistent answer as per Definition 2.1.7 of consistent answers given earlier. In fact, to produce an empty set of consistent answers, it suffices to have just two repairs on which a scalar aggregation query returns different answers. Aiming to

obtain more meaningful answers to aggregation queries, Arenas et al. [16] proposed the *range consistent answers*, as an alternative notion of consistent answers.

The notion of range consistent answers is based on the set of possible answers to a query. For a scalar aggregation query Q, the set of *possible answers* to Q on an inconsistent instance $\mathcal{I}$ consists of the answers to Q over all repairs of $\mathcal{I}$, i.e., $\textsc{Poss}(Q, \mathcal{I}, \Sigma) = \bigcup \{Q(\mathcal{J}) : \mathcal{J} \text{ is a repair of } \mathcal{I} \text{ w.r.t. } \Sigma\}$. With this, we now define the range consistent answers.

Definition 2.1.10. Range Consistent Answers. *(Scalar Aggregation Queries). Assume $\mathcal{R}$ is a database schema, Σ is a set of integrity constraints on $\mathcal{R}$, $\mathcal{I}$ is an $\mathcal{R}$-instance, and Q is a scalar aggregation query. The range consistent answers to Q on $\mathcal{I}$ is the interval $[glb(Q, \mathcal{I}), lub(Q, \mathcal{I})]$, where the endpoints of this interval are, respectively, the greatest lower bound (glb) and the least upper bound (lub) of the set $Poss(Q, \Sigma)$ of possible answers to Q on $\mathcal{I}$.*

Arenas et al. [17] focused on scalar aggregation queries only. Fuxman, Fazli, and Miller [40] extended the notion of range consistent answers to aggregation queries with grouping, i.e., to aggregation queries of the form

$$Q := \texttt{SELECT } Z, f(A) \texttt{ FROM } T(U, Z, A) \texttt{ GROUP BY } Z.$$

Definition 2.1.11. Range Consistent Answers. *(Non-Scalar Aggregation Queries). Assume $\mathcal{R}$ is a database schema, Σ is a set of integrity constraints on $\mathcal{R}$, $\mathcal{I}$ is an $\mathcal{R}$-instance, and Q is a non-scalar aggregation query. A tuple $(T, [glb, lub])$ is a range consistent answer to Q on $\mathcal{I}$ if the following conditions hold.*

- *For every repair $\mathcal{J}$ of $\mathcal{I}$, there exists d s.t. $(T, d) \in Q(\mathcal{J})$ and $glb \leq d \leq lub$.*
- *For some repair $\mathcal{J}$ of $\mathcal{I}$, we have that $(T, glb) \in Q(\mathcal{J})$.*
- *For some repair $\mathcal{J}$ of $\mathcal{I}$, we have that $(T, lub) \in Q(\mathcal{J})$.*

For a database schema $\mathcal{R}$ with a set Σ of integrity constraints, if Q is an aggregation query on $\mathcal{R}$, then $\textsc{Cons}(Q, \mathcal{I}, \Sigma)$ denotes the problem: given an $\mathcal{R}$-instance $\mathcal{I}$, compute the range consistent answers to Q on $\mathcal{I}$ w.r.t. Σ. When the set Σ is understood from the context, we write $\textsc{Cons}(Q, \mathcal{I})$ instead of $\textsc{Cons}(Q, \mathcal{I}, \Sigma)$.

Example 2.1.4. *Consider again the database instance $\mathcal{I}$ from Table 2.1.1 with key constraints shown with the underlined attributes and one functional dependency FID $\rightarrow$ CID on the Teaches relation. Let Q be the scalar aggregation query that finds the total number of students belonging to the CSE department, i.e.,*

```
SELECT COUNT(*) FROM Students
WHERE Students.DEPT = 'CSE'.
```

Clearly, the Students relation does not violate any integrity constraint, and we have the range consistent answers $\textsc{Cons}(Q, \mathcal{I}) = [2, 2]$, i.e., no matter how $\mathcal{I}$ is repaired, it is guaranteed that there will be two students in the CSE department. Now, let Q' be the aggregation query to compute the total number of credits taught by each faculty:

```
SELECT Teaches.FID, SUM(Courses.CREDIT)
FROM Courses, Teaches
WHERE Teaches.FID = Courses.FID
GROUP BY Teaches.FID.
```

There exists a repair in which Faculty F1 teaches a two-credit course, while there is another repair in which they teach a five-credit course. In all repairs, the faculty F2 teaches a two-credit course and the faculty F3 teaches a five-credit course. As a result, the range consistent answers $\textsc{Cons}(Q', \mathcal{I})$ *are* $\{(F1, [2, 5]), (F2, [2, 2]), (F3, [5, 5])\}$.

2.1.8 Boolean Satisfiability (SAT) and SAT Solvers

Boolean Satisfiability (SAT) is the prototypical and arguably the most widely studied NP-complete problem. SAT is the following decision problem: *given a boolean formula* φ*, is* φ *satisfiable?* There has been an extensive body of research on different aspects of boolean satisfiability (see the handbook [23]). Significant progress has been made on developing SAT-solvers, so much so that the advances in this area of research are often referred to as the "SAT Revolution" [81]). Typically, a SAT-solver takes a boolean formula φ in *conjunctive normal form* (CNF) as an input and outputs a satisfying assignment for φ (if one exists) or tells that the formula φ is unsatisfiable. Recall that a formula φ is in CNF if it is a conjunction of clauses, where each clause is a disjunction of literals. For example, the formula $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4)$ has a satisfying assignment $\{x_1, x_2, x_3, x_4\} = \{1, 0, 0, 1\}$.

Researchers have found several optimization variants of SAT to be useful in encoding naturally many real-world problem instances. Some of the widely studied optimization variants of SAT are also of interest to us and we list them here.

Weighted MaxSAT is the maximization variant of SAT in which each clause is assigned a positive weight and the goal is to find an assignment that maximizes the

sum of the weights of the satisfied clauses. We write $(l_1 \vee \cdots \vee l_k, w)$ to denote a clause $(l_1 \vee \cdots \vee l_k)$ with weight w.

Partial MaxSAT is the maximization variant of SAT in which some clauses of the formula are assigned an infinite weight (*hard clauses*), while each of the rest is assigned weight one (*soft clauses*). The goal is to find an assignment that satisfies all hard clauses and the maximum number of soft clauses. If the hard clauses of a Partial MaxSAT instance are not simultaneously satisfiable, then we say that the instance is *unsatisfiable*. For simplicity, a hard clause $(l_1 \vee \cdots \vee l_k, \infty)$ is denoted as $(l_1 \vee \cdots \vee l_k)$.

Weighted Partial MaxSAT is the maximization variant of SAT where some of the clauses of the formula are assigned an infinite weight (*hard clauses*), while each of the rest is assigned a positive weight (*soft clauses*). The goal is to find an assignment that satisfies all hard clauses and maximizes the sum of weights of the satisfied soft clauses. Clearly, Weighted Partial MaxSAT is a generalization of both Weighted MaxSAT (no hard clauses) and Partial MaxSAT (each soft clause has weight one).

Modern solvers, such as MaxHS [31], can efficiently solve large instances of these maximization variants of SAT. Note that these maximization problems have dual minimization problems, called Weighted MinSAT, Partial MinSAT, and Weighted Partial MinSAT, respectively. For example, in Weighted Partial MinSAT, the goal is to find an assignment that satisfies all hard clauses and minimizes the sum of weights of the satisfied soft clauses. These minimization problems are of interest to us, because some of the computations of the range consistent answers have natural reductions to such minimization problems. At present, the only existing Weighted Partial

MinSAT solver is MinSatz [67]. Since this solver has certain size limitations, we will deploy Kügel's technique [64] to first reduce Weighted Partial MinSAT to Weighted Partial MaxSAT, and then use the Weighted Partial MaxSAT solvers. This technique uses the concept of *CNF-negation* [64, 88] that we introduce next. By definition, if $C = (l_1 \vee \cdots \vee l_k)$ is a clause, then the CNF-negation $\mathrm{CNF}(\overline{C})$ of C is the CNF-formula $\neg l_1 \wedge (l_1 \vee \neg l_2) \wedge \cdots \wedge (l_1 \vee l_2 \vee \cdots \vee l_{k-1} \vee \neg l_k)$. It is easy to verify that the following properties hold: (i) if an assignment s does not satisfy C, then s satisfies every clause of $\mathrm{CNF}(\overline{C})$; (ii) if an assignment s satisfies C, then s satisfies all but one of the clauses of $\mathrm{CNF}(\overline{C})$, namely, the clause $(\neg l_1 \vee \cdots \vee \neg l_{j-1} \vee l_j)$, where j is the smallest index such that $s(l_j) = 1$. It follows that C is satisfiable if and only if $\mathrm{CNF}(\overline{C})$ is unsatisfiable. For a weighted clause $C = (l_1 \vee \cdots \vee l_k, w)$, the CNF-negation $\mathrm{CNF}(\overline{C})$ of C is the formula $(\neg l_1, w) \wedge (l_1 \vee \neg l_2, w) \wedge \cdots \wedge (l_1 \vee l_2 \vee \cdots \vee l_{k-1} \vee \neg l_k, w)$.

2.2 Related Work

Consistent Query Answering (CQA) was first introduced by Arenas et al. [15] as a principled approach to deal with inconsistent databases. There has been extensive amount of research on this topic over the years, and a comprehensive survey can be found in a book by L. Bertossi [22]. In the quest of finding a way to deal with inconsistent databases that gives a strong semantic guarantee, Arenas et al. defined the notions of database repairs, consistent answers, and associated problems. The main problem associated with consistent query answering, $\textsc{Cons}(q, \mathcal{I})$ (or $\textsc{Certainty}(q)$ for

boolean queries), is the problem of computing the consistent answers to a query q on an inconsistent database instance $\mathcal{I}$. It is often studied from the perspectives of two complexity measures, namely, *data complexity*, i.e., the complexity of the problem in the size of the database instance $\mathcal{I}$ where the query q is fixed, and *query complexity*, i.e., the database instance $\mathcal{I}$ is fixed but the complexity is studied as the function of the size of the query q [80]. When both the database instance and the query are parts of the input, the complexity is referred as the *combined complexity*. For the rest of this document, whenever we talk about the complexity of $\textsc{Cons}(q, \mathcal{I})$ or $\textsc{Certainty}(q)$, we mean the data complexity. In what follows, we summarize the complexity results associated with consistent query answering.

2.2.1 The Complexity of Consistent Answers

If $\mathcal{R}$ is a database schema, Σ is a fixed finite set of denial constraints on $\mathcal{R}$, and q is a k-ary conjunctive query, where $k \geq 1$, then the following problem is in coNP: given an $\mathcal{R}$-instance $\mathcal{I}$ and a tuple $\vec{t}$, is $\vec{t}$ a certain answer to q on $\mathcal{I}$ w.r.t. Σ? This is so because to check that $\vec{t}$ is not a certain answer to q on $\mathcal{I}$ w.r.t. Σ, we guess a repair $\mathcal{J}$ of $\mathcal{I}$ and verify that $\vec{t} \notin q(\mathcal{J})$. Note that $\mathcal{J}$ is a subset of $\mathcal{I}$, evaluating a fixed conjunctive query on a given database is a polynomial-time task, and testing if $\mathcal{J}$ is a repair of $\mathcal{I}$ w.r.t. denial constraints (i.e., *repair checking* w.r.t. denial constraints) is a polynomial-time task as well. Similarly, if q is a boolean conjunctive query, then the decision problem $\textsc{Certainty}(q, \Sigma)$ is in coNP.

Even for primary key constraints and boolean conjunctive queries, $\textsc{Certainty}(q, \Sigma)$

exhibits a variety of behaviors within coNP. Indeed, consider the queries

1. $\textsc{Path}() := \exists x, y, z\ R(\underline{x}, y) \wedge S(\underline{y}, z)$;
2. $\textsc{Cycle}() := \exists x, y\ R(\underline{x}, y) \wedge S(\underline{y}, x)$;
3. $\textsc{Sink}() := \exists x, y, z\ R(\underline{x}, z) \wedge S(\underline{y}, z)$.

Fuxman and Miller [42] showed that $\textsc{Certainty}(\textsc{Path})$ is first-order-rewritable (or, FO-rewritable), i.e., there is a first-order definable boolean query q' such that $\textsc{Cons}(\textsc{Path}, \mathcal{I}) = q'(\mathcal{I})$. As a matter of fact, the query q' is expressible by the first-order sentence $q'() := \exists x, y, z\ R(\underline{x}, y) \wedge S(\underline{y}, z) \wedge \forall y'(R(\underline{x}, y') \rightarrow \exists z' S(\underline{y'}, z'))$.

Wijsen [84], showed that $\textsc{Certainty}(\textsc{Cycle})$ is in P, but it is not FO-rewritable, while Fuxman and Miller [42] showed that $\textsc{Certainty}(\textsc{Sink})$ is coNP-complete via a reduction from the complement of Monotone 3-SAT.

The preceding state of affairs sparked a series of investigations aiming to obtain classification results concerning the computational complexity of computing the consistent answers (e.g., see [47, 54, 65, 82, 84]). Researchers initially looked at the sub-classes of conjunctive queries for which computing consistent answers is a tractable problem [42, 82, 83, 54]. Fuxman and Miller [42] identified a class of self-join-free conjunctive queries under the set of integrity constraints restricted to primary keys, for which the consistent answers to a query q on an inconsistent database instance $\mathcal{I}$ are first-order rewritable (or, FO-rewritable), i.e., there exists a first-order query q' such that for every database instance $\mathcal{I}$ over the same schema as q, the consistent answers to q on $\mathcal{I}$ are precisely the result of evaluating q' on $\mathcal{I}$, i.e., $\textsc{Cons}(q, \mathcal{I}) = q'(\mathcal{I})$. Such a

first-order query q' is called a *certain first-order rewriting* of q. They called this class of queries C_{tree}, and it is defined using a notion of a *join graph* of a query. In a join graph of a self-join-free conjunctive query, each atom of the query is treated as a node in the graph, and there is a directed edge from a node R_i to R_j $(i \neq j)$ if a some non-key variable in R_i appears in R_j, and there is a self-loop on R_i if some non-key variable in R_i appears more than once in R_i. A self-join-free conjunctive query q belongs to C_{tree} if its join graph is a forest (i.e. each connected component is a tree), and for every pair $R_i(\underline{\vec{x_i}}, \vec{y_i})$ and $R_j(\underline{\vec{x_j}}, \vec{y_j})$ of connected nodes in the join graph, all variables in $\vec{x_j}$ appear in $\vec{y_i}$. They also provided an efficient algorithm that, given a query $q \in C_{tree}$, returns a first-order rewriting q' of q. The idea behind this algorithm is to use recursion on the tree structure of each of the components of the join graph of q. The results from [42] have been extended to exclusion dependencies [47] and to the unions of conjunctive queries [65]. The necessary and sufficient conditions for a query to have FO-rewritable consistent answers were later given by Koutris and Wijsen [57].

It was a ten-year-old open problem, that for self-join-free conjunctive queries under one key constraint per relation, is it the case that $\textsc{Certainty}(q, \mathcal{I}, \Sigma)$ is in either P or coNP-complete. A special case of this problem was first shown to be true, where the authors proved that for queries with exactly two atoms, such a dichotomy holds and it is effective [54]. They showed that for relation symbols R_1 and R_2 appearing in the query q, the decision problem $\textsc{Certainty}(q)$ is in P if and only if $keys(R_1) \cup keys(R_2) \subseteq L$, where $keys(R_i)$ is the set of variables corresponding to the primary key attributes of R_i, and L is the set of variables shared between the atoms of R_1 and R_2.

Later, this problem was settled with a *trichotomy* theorem, established by Koutris and Wijsen [57, 58, 59], for self-join-free boolean conjunctive queries. This trichotomy theorem asserts that if q is a self-join free boolean conjunctive query under a fixed set of integrity constraints restricted to primary keys, then CERTAINTY(q) is FO-rewritable, or in P but not FO-rewritable, or coNP-complete. Moreover, there is a quadratic algorithm to decide, given such a query, which of the three cases of the trichotomy holds. This algorithm is based on the notion of *attack graphs* of queries, first introduced by Wijsen [83] and then refined by Koutris and Wijsen [59]. The cyclicity properties of the attack graph determine the complexity of its certain answer; specifically, for a self-join-free boolean conjunctive query q, if the attack graph is acyclic, then CERTAINTY(q) is in FO, if the attack graph contains a cycle but not a *strong* cycle, then CERTAINTY(q) is in P but not in FO, otherwise, CERTAINTY(q) is coNP-complete.

In 2019, Koutris and Wijsen further sharpened their trichotomy result by proving that for every self-join-free boolean conjunctive query q, the problem CERTAINTY(q) is either in FO, or it is L-complete, or it is coNP-complete [61]. This is the most definitive classification result regarding the computational complexity of computing the consistent answers to date. Such complexity results are important not only from a theoretical standpoint but also from a practical perspective. They can be used in pre-processing a query, before actually solving it for CERTAINTY(q). For example, if CERTAINTY(q) for a query belongs to a lower complexity class, then there is no need to use heavy machinery such as SAT or BIP solvers or linear programming to solve it, we could just send the certain first-order rewriting to the SQL engine or to a Datalog engine, and it

will compute consistent answers efficiently.

In the recent past, researchers have also investigated the complexity of consistent answers w.r.t. broader classes of queries and integrity constraints [60, 62, 63]. For example, self-join-free conjunctive queries with negated atoms were studied in [60], the first-order rewritability of consistent answers to self-join-free conjunctive queries on database schemata having multiple key constraints was investigated in [62], while consistent answers to path queries with self-joins, i.e., the queries of the form $\exists x_1, \cdots, \exists x_{k+1}$ $R_1(\underline{x_1}, x_2) \wedge R_2(\underline{x_2}, x_3) \wedge \cdots \wedge R_k(\underline{x_k}, x_{k+1})$ were studied in [63]. It remains an open problem whether or not a classification result such as the Koutris-Wijsen trichotomy extends to arbitrary boolean conjunctive queries and to arbitrary functional dependencies or denial constraints.

2.2.2 The Complexity of Range Consistent Answers

The complexity of the consistent answers has been extensively studied for self-join-free conjunctive queries, but very little work has been done on aggregation queries and range consistent answers. To obtain more meaningful information out of the answers to scalar aggregation queries on inconsistent databases, the notion of range consistent answers was first proposed as an alternative to consistent answers by Arenas et al. in [17]. In the same paper, the authors studied the complexity of the range consistent answers to the scalar aggregation queries of the form

$$\texttt{SELECT } f(A) \texttt{ FROM } R(U, A),$$

where R is a relation symbol in the database schema under consideration. Their results

can be summarized as follows.

- If the relation $R(U, A)$ has at most one functional dependency and $f(A)$ is one of the aggregation operators `MIN(`A`)`, `MAX(`A`)`, `SUM(`A`)`, `COUNT(*)`, `AVG(`A`)`, then the range consistent answers to the query `SELECT` f(A) `FROM` $R(U, A)$ are computable in polynomial time in the size of the database instance.
- There exists a database schema $\mathcal{R}$ consisting of a relation symbol R having one key dependency such that computing the range consistent answers of the query `SELECT COUNT(`A`)` `FROM` $R(U, A)$ on a given $\mathcal{R}$-instance is an NP-complete problem.
- There exists a database schema $\mathcal{R}$ consisting of a relation symbol R with two functional dependencies, such that computing the range consistent answers of the query `SELECT` $f(A)$ `FROM` $R(U, A)$ on a given $\mathcal{R}$-instance is a NP-complete problem, where $f(A)$ is one of the standard aggregation operators.

It remains an open problem to pinpoint the complexity of the range consistent answers for richer aggregation queries of the form

$$Q := \texttt{SELECT } Z, f(A) \texttt{ FROM } T(U, Z, A) \texttt{ GROUP BY } Z,$$

where $T(U, Z, A)$ is the relation returned by a conjunctive query q or by a union $q := q_1 \cup \cdots \cup q_k$ of conjunctive queries. It can be shown, however, that if computing the consistent answers $\textsc{Cons}(q)$ of the underlying query q is a hard problem, then computing the range consistent answers $\textsc{Cons}(Q)$ of the aggregation query Q is a hard problem as well (see Section 4.1). This gives rise to the following question: how does the complexity of the consistent answers of the underlying query q impact the complexity of the range

consistent answers of the aggregation query Q?

Fuxman and Miller [42] identified a class, called C_{forest}, of self-join-free conjunctive queries whose consistent answers are FO-rewritable. In his PhD thesis, Fuxman [39] introduced the class $C_{aggforest}$ consisting of all aggregation queries such that the aggregation operator is one of `MIN(`A`)`, `MAX(`A`)`, `SUM(`A`)`, `COUNT(*)`, the underlying query q is a conjunctive query in C_{forest}, and there is one key constraint for each relation in the underlying query q. Fuxman [39] showed that the range consistent answers of every query in $C_{aggforest}$ are SQL-rewritable (earlier, similar results for a proper subclass of $C_{aggforest}$ were obtained by Fuxman, Fazli, and Miller [40]).

It is known that there are self-join-free conjunctive queries outside the class C_{forest} whose consistent answers are FO-rewritable. In fact, Koutris and Wijsen [59] have characterized the self-join-free conjunctive queries whose consistent answers are FO-rewritable. However, the SQL-rewritability of aggregation queries beyond those in $C_{aggforest}$ has not been investigated (except our hardness result from Section 4.1).

2.2.3 Existing Systems for Consistent Query Answering

Several systems for consistent query answering have been proposed in the past [17, 20, 26, 40, 41, 46, 55, 68, 71], but for multiple reasons, they mostly remained as academic prototypes. In particular, the ConQuer (Consistent Querying) system [40, 41] is tailored to queries in the class C_{forest} and the class $C_{aggforest}$. Other systems use logic programming [20, 46], compact representations of repairs [25], or reductions to solvers. Specifically, the system in [68] uses reductions to answer set programming,

while the EQUIP system [55] uses reductions to binary integer programming and the subsequent deployment of CPLEX. The ConsEx (Consistency Extractor) system [70] was built with an idea of disjunctive logic programming, that computes the consistent answers to queries by evaluating logic programs with stable model semantics. Hippo [26] is an early system that leverages a polynomial-time algorithm for computing consistent answers on database schemata having denial constraints for a certain class of boolean queries. Table 2.1 lists some of the consistent query answering systems built in the past and outlines the supported classes of integrity constraints and queries and the methods used to compute the consistent answers or the range consistent answers. In the end, we also list for comparison CAvSAT, a SAT-based system for answering queries over inconsistent databases, that we will present in Chapter 5 of this thesis.

Table 2.1: Summary of the systems built for Consistent Query Answering

System	**Constraints**	**Queries**	**Method**
Hippo	Universal constraints	Projection-free queries with $\cup$ and $\setminus$	Direct algorithm
ConQuer	Primary key constraints	Subclass of conjunctive queries with aggregation and grouping	SQL-rewriting
ConsEx	Universal constraints, acyclic referential integrity constraints, and not-null constraints	Datalog with $\neg$	Answer set programming
EQUIP	Primary key constraints	Arbitrary conjunctive queries	Reductions to Binary Integer Programming
CAvSAT	Arbitrary denial constraints	Arbitrary unions of conjunctive queries with aggregation and grouping	Reductions to SAT and its variants

Chapter 3

Consistent Answers via SAT Solving

In this chapter, we give polynomial-time reductions from computing the consistent answers to different classes of non-aggregation queries to variants of SAT. In Section 3.1, we first give a reduction from computing the certain answer to a union of boolean conjunctive queries to UnSAT where the database schema has one key constraint per relation; we then extend this reduction to support unions of non-boolean conjunctive queries, and in Section 3.2, we show how it can be extended further to handle schemata with arbitrary denial constraints. We use the inconsistent database instance about flight information records shown in Table 3.1 as a running example throughout this section. This database instance has three relations, namely, *Airlines*, *Tickets*, and *Flights*, each with one key constraint (the key attributes are underlined in Table 3.1). For convenience, we associate an attribute *FactID* to each relation to uniquely refer to each fact of the database instance.

Table 3.1: An inconsistent database instance of flight information records

Airlines		
FactID	AIRLINE	COUNTRY
f_1	Southwest	United States
f_2	Jazz Air	Canada
f_3	Southwest	Canada

Tickets				
FactID	PNR	CODE	CLASS	FARE
f_4	MJ9C8R	SWA 1568	Economy	430
f_5	KLF88V	MI 471	First	914
f_6	NJ5RT3	SWA 1568	First	112

Flights							
FactID	CODE	DATE	AIRLINE	FROM	TO	DEPARTURE	ARRIVAL
f_7	JZA 8329	01/29/19	Jazz Air	GEG	OAK	16:12 PST	18:00 PST
f_8	SWA 1568	01/29/19	Silkair	YYZ	YAM	18:55 EST	18:44 EST
f_9	SWA 1568	01/29/19	Southwest	LAX	OAK	16:18 PST	17:25 PST

3.1 Consistent Query Answering for Key Constraints

In this section, we assume that $\mathcal{R}$ is a database schema and Σ is a finite set of primary key constraints on $\mathcal{R}$, i.e., there is one key constraint per each relation of $\mathcal{R}$. We first consider unions of boolean conjunctive queries. For a fixed union $q := q_1 \cup \ldots \cup q_k$ of boolean conjunctive queries $q_1, \ldots, q_k$, we give a natural polynomial-time reduction from CERTAINTY(q) to UNSAT. We then extend this reduction to unions of non-boolean conjunctive queries, so that for every fixed union $q := q_1 \cup \ldots \cup q_k$ of non-boolean conjunctive queries $q_1, \ldots, q_k$, the consistent answers to q can be computed by iteratively solving Weighted MaxSAT instances. Note that for a union $q := q_1 \cup \ldots \cup q_k$ of conjunctive queries, CONS(q) is not, in general, equal to the CONS(q_1) $\cup \ldots \cup$ CONS(q_k). In what follows, we heavily use the notions of *key-equal groups* of facts and *minimal witnesses* to a union of conjunctive queries, that we introduce next.

Definition 3.1.1. ***Key-Equal Group***. *Let $\mathcal{I}$ be an $\mathcal{R}$-instance. We say that two facts of a relation R of $\mathcal{I}$ are key-equal if they agree on the key attributes of R. A set S of facts of $\mathcal{I}$ is called a key-equal group of facts if every two facts in S are key-equal, and no fact in S is key-equal to some fact in $\mathcal{I} \backslash S$.*

For example, the key-equal groups of facts of the database instance from Table 3.1 are $\{f_1, f_3\}$, $\{f_2\}$, $\{f_4\}$, $\{f_5\}$, $\{f_6\}$, $\{f_7\}$, and $\{f_8, f_9\}$.

Definition 3.1.2. ***Minimal Witness***. *Let $\mathcal{I}$ be an $\mathcal{R}$-instance and let $\mathcal{S}$ be a sub-instance of $\mathcal{I}$. We say that $\mathcal{S}$ is a minimal witness to a union q of conjunctive queries on $\mathcal{I}$, if $S \models q$, and for every proper subset $\mathcal{S}'$ of $\mathcal{S}$, we have that $\mathcal{S}' \not\models q$.*

3.1.1 Computing Key-Equal Groups and Minimal Witnesses

For the reduction from CERTAINTY(q) to UNSAT to be a polynomial-time reduction, it is crucial that we can compute the key-equal groups of facts of the inconsistent database instance and the minimal witnesses to the union of boolean conjunctive queries in consideration in polynomial time in the size of the database instance. It is easy to see that for each relation R of $\mathcal{I}$, the key-equal groups of R can be computed efficiently by first computing all distinct values of the key attributes of R and then finding for each key-value all facts of R having that key value. The set of minimal witnesses to a union q of boolean conjunctive queries on a database instance $\mathcal{I}$ can be computed by simply evaluating q and $\mathcal{I}$, and it is well-known that the query evaluation problem for a fixed union of conjunctive queries is solvable in polynomial-time. If a boolean conjunctive query in q contains a self-join, then the tuples in the query result $q(\mathcal{I})$ may

contain duplicate facts. In that case, the duplicates can be removed in linear time to obtain the set of minimal witnesses. Now, we are ready to present Reduction 3.1.1, a natural polynomial-time reduction from CERTAINTY(q) to UNSAT.

Reduction 3.1.1. *Let q be a fixed union of boolean conjunctive queries over $\mathcal{R}$. Given an $\mathcal{R}$-instance $\mathcal{I}$, we construct a CNF-formula ϕ as follows. For each fact f_i of $\mathcal{I}$, introduce a boolean variable x_i, $1 \leq i \leq n$. Let $\mathcal{G}$ be the set of key-equal groups of facts of $\mathcal{I}$, and let $\mathcal{W}$ be the set of minimal witnesses to q on $\mathcal{I}$.*

- *For each key-equal group $G_j \in \mathcal{G}$, construct the clause $\alpha_j = \bigvee_{f_i \in G_j} x_i$.*
- *For each minimal witness $W_j \in \mathcal{W}$, construct the clause $\beta_j = \bigvee_{f_i \in W_j} \neg x_i$.*
- *Construct the CNF-formula $\phi = \left(\bigwedge_{i=1}^{|\mathcal{G}|} \alpha_i \right) \wedge \left(\bigwedge_{j=1}^{|\mathcal{W}|} \beta_j \right)$.*

Proposition 3.1.1. *For a CNF-formula ϕ constructed using Reduction 3.1.1, the following two statements hold.*

1. *The size of ϕ is polynomial in the size of $\mathcal{I}$.*
2. *The formula ϕ is satisfiable if and only if* CERTAINTY*(q, Σ) is false on $\mathcal{I}$.*

Proof. Let n be the number of facts in $\mathcal{I}$. There are exactly n boolean variables used in ϕ. Clearly, $|\mathcal{G}| \leq n$, therefore the number of α-clauses is bounded above by n. Similarly, for each $G_j \in \mathcal{G}$, we have that $|G_j| \leq n$. Hence, the length of each α-clause is also at most n. If d is the number of atoms in q, then we have that $|\mathcal{W}| \leq n^d$; moreover, for every $W_j \in \mathcal{W}$, we have that $|W_j| \leq d$. Hence, the number of β-clauses in ϕ is at most n^d, and the length of each β-clause is bounded above by d. Since the union q of boolean conjunctive queries is fixed, we have that d is a constant. Therefore, the first statement

of Proposition 3.1.1 holds.

To prove the second part of Proposition 3.1.1, assume first that CERTAINTY(q) is false on $\mathcal{I}$. Hence, there exists a repair $\mathcal{J}$ of $\mathcal{I}$ that falsifies q. Construct an assignment s to the variables in ϕ by setting $s(x_i) = 1$ if and only if $f_i \in \mathcal{J}$. Since exactly one fact from each key-equal group of $\mathcal{I}$ is present in $\mathcal{J}$, exactly one variable from each α-clause is set to 1 in s. Also, since $\mathcal{J} \not\models q$, no minimal witness to q is in $\mathcal{J}$. Therefore, at least one variable from each β-clause is set to 0 in s. Hence, s is a satisfying assignment to ϕ. For the other direction, let s be a satisfying assignment to ϕ. Since no two α-clauses share a variable, we can construct a set X of variables by arbitrarily choosing exactly one x_i from each α-clause, such that $s(x_i) = 1$. Construct a set $\mathcal{J}$ of facts of $\mathcal{I}$, such that $f_i \in \mathcal{J}$ if and only if $x_i \in X$. It is easy to see that $\mathcal{J}$ contains exactly one fact from each key-equal group of $\mathcal{I}$, and no minimal witness to q on $\mathcal{I}$ is present in $\mathcal{J}$. Hence, $\mathcal{J}$ is a repair of $\mathcal{I}$ that falsifies q. □

Example 3.1.1. *Consider the database instance from Table 3.1 and a union q of boolean conjunctive queries that ask whether at least one of Jazz Air and Southwest airlines belongs to Canada, i.e., $q() := q_1 \cup q_2$, where $q_1() :=$ Airlines('Jazz Air', 'Canada') and $q_2() :=$ Airlines('Southwest', 'Canada').*

We introduce variables x_1, x_2, and x_3 corresponding to the facts f_1, f_2, and f_3 respectively. We do not need variables corresponding to the facts of the relations whose symbols do not appear in the queries in consideration. We construct two α-clauses $(x_1 \vee x_3)$ and (x_2) since the facts f_1 and f_3 form a key-equal group of size

two, and $\{f_2\}$ *is a key-equal group of size one. The sets* $\{f_2\}$ *and* $\{f_3\}$ *are minimal witnesses to* q *so we construct two* β*-clauses* $(\neg x_2)$ *and* $(\neg x_3)$*, resulting in a CNF-formula* $\phi = (x_1 \vee x_3) \wedge (x_2) \wedge (\neg x_2) \wedge (\neg x_3)$*. Clearly, the formula* ϕ *is unsatisfiable due to the presence of the clauses* (x_2) *and* $(\neg x_2)$*, implying that* CERTAINTY(q) *is true by Proposition 3.1.1. Observe that the two conflicting clauses* (x_2) *and* $(\neg x_2)$ *arise because the fact Airlines('Jazz Air', 'Canada') is present in every repair of the database instance, which is precisely the reason why* CERTAINTY(q) *is true.*

Next, we show how Reduction 3.1.1 can be modified to support unions of non-boolean conjunctive queries. Let q be a fixed union of non-boolean conjunctive queries on $\mathcal{R}$, i.e., q has one or more free variables. We extend Reduction 3.1.1 to Reduction 3.1.2, so that one can reason about the consistent answers to q on an $\mathcal{R}$-instance $\mathcal{I}$ using the satisfying assignments of the CNF-formula ϕ constructed via Reduction 3.1.2. We use the term *potential answers* to refer to the answers to q on $\mathcal{I}$. If $\vec{a}_l$ is such a potential answer, we write $q[\vec{a}_l]$ to denote the boolean conjunctive query obtained from q by replacing the free variables in the body of q by corresponding constants from $\vec{a}_l$. To reason about the potential answers via the satisfying assignments of the CNF formula, we associate each potential answer $\vec{a}_l$ with a boolean variable p_l, and it is set to true in only those assignments from which a repair $\mathcal{J}$ can be constructed such that $\vec{a}_l \notin q(\mathcal{J})$.

Reduction 3.1.2. *Given an* $\mathcal{R}$*-instance* $\mathcal{I}$*, we construct a CNF-formula* ϕ *as follows. For each fact* f_i *of* $\mathcal{I}$*, introduce a boolean variable* x_i*,* $1 \leq i \leq n$*. Let* $\mathcal{G}$ *be the set of key-equal groups of facts of* $\mathcal{I}$ *and let* $\mathcal{A}$ *be the set of potential answers to* q *on* $\mathcal{I}$*. For*

each $\vec{a}_l \in \mathcal{A}$, let $\mathcal{W}^l$ denote the set of minimal witnesses to the union $q[\vec{a}_l]$ of boolean conjunctive queries on $\mathcal{I}$. For each $\vec{a}_l \in \mathcal{A}$, introduce a boolean variable p_l, $1 \leq l \leq |\mathcal{A}|$.

- *For each $G_j \in \mathcal{G}$, construct the clause $\alpha_j = \bigvee_{f_i \in G_j} x_i$.*
- *For each $\vec{a}_l \in \mathcal{A}$ and for each $W_j^l \in \mathcal{W}^l$, construct the clause $\beta_j^l = \left(\bigvee_{f_i \in W_j^l} \neg x_i \right) \vee \neg p_l$.*
- *Construct the boolean formula $\phi = \left(\bigwedge_{i=1}^{|\mathcal{G}|} \alpha_i \right) \wedge \left(\bigwedge_{l=1}^{|\mathcal{A}|} \left(\bigwedge_{j=1}^{|\mathcal{W}^l|} \beta_j^l \right) \right)$.*

Proposition 3.1.2. *For a CNF-formula ϕ constructed using Reduction 3.1.2, the following two statements hold.*

1. *The size of ϕ is polynomial in the size $\mathcal{I}$.*
2. *There exists a satisfying assignment to ϕ in which a variable p_l is set to true if and only if $\vec{a}_l \notin \textsc{Cons}(q, \mathcal{I})$.*

Proof. Let n be the number of facts in $\mathcal{I}$. Let m be the arity of the query q and let d and the number of atoms of q. Since an answer to q is a set of at most m facts, we have that $|\mathcal{A}| \leq n^m$. For each l, the number of witnesses in $\mathcal{W}^l$ is bounded by n^d. Therefore, there are at most n^{m+d} β-clauses in ϕ, each of length at most d. Since the query q is not part of the input to $\textsc{Cons}(q)$, the quantities m and d are constants. It follows directly from Proposition 3.1.1 that both the number of α-clauses and the length of each α-clause in ϕ are bounded above by n.

To prove the second part of the proposition, assume first that $\vec{a}_l \notin \textsc{Cons}(q)$. Hence, there exists a repair $\mathcal{J}$ of $\mathcal{I}$, such that no minimal witness to $q[\vec{a}_l]$ is in $\mathcal{J}$. Construct an assignment s to the variables in ϕ as follows. Set $s(x_i) = 1$ if and only

if $f_i \in \mathcal{J}$. Set $s(p_l) = 1$, and set $s(p_j) = 0$ for all $j \neq l$. Since exactly one fact from each key-equal group of $\mathcal{I}$ is in $\mathcal{J}$, the assignment sets to 1 exactly one variable from each α-clause. Since no minimal witness to $q[\vec{a}_l]$ is in $\mathcal{J}$, at least one variable from each β^l-clause is set to 0 in s, thus satisfying all β^l-clauses, even when p_l is set to 1. All other β-clauses are satisfied trivially because of the assignment $s(p_j) = 0$, for all $j \neq l$. In the other direction, let s be the satisfying assignment to ϕ, such that $s(p_l) = 1$. Since no two α-clauses share a variable, we can construct a set X of variables by arbitrarily choosing exactly one x_i from each α-clause, such that $s(x_i) = 1$. Construct a set $\mathcal{J}$ of facts of $\mathcal{I}$, such that $f_i \in \mathcal{J}$ if and only if $x_i \in X$. It is easy to see that exactly one fact from each key-equal group of $\mathcal{I}$ is present in $\mathcal{J}$. Since $s(p_l) = 1$ and since all β^l-clauses are satisfied by s, at least one fact from each minimal witness to $q[\vec{a}_l]$ is missing in $\mathcal{J}$. Hence, $\mathcal{J}$ must be a repair of $\mathcal{I}$ such that $\mathcal{J} \not\models q[\vec{a}_l]$. □

Example 3.1.2. *Suppose we want to find out the codes of the flights that belong to an airline from Canada and fly to the airport OAK. This can be expressed by the unary conjunctive query* $q(x) := \mathit{Flights}(x, y, z, p, \text{'OAK'}, q, r) \wedge \mathit{Airlines}(z, \text{'Canada'})$.

There are two potential answers to q, namely, 'JZA 8329' and 'SWA 1568', so we introduce their corresponding variables p_1 *and* p_2. *Since the facts* f_1 *and* f_3 *form a key-equal group, we construct an* α*-clause* $(x_1 \vee x_3)$. *Similarly, since the set* $\{f_2, f_7\}$ *of facts is a minimal witness to* $q[\text{'JZA 8329'}]$, *we construct the* β*-clause* $(\neg x_2 \vee \neg x_7 \vee \neg p_1)$. *By continuing this way, we obtain the following CNF-formula* ϕ*:*

$(x_1 \vee x_3) \wedge x_2 \wedge x_4 \wedge x_5 \wedge x_6 \wedge x_7 \wedge (x_8 \vee x_9) \wedge (\neg x_2 \vee \neg x_7 \vee \neg p_1) \wedge (\neg x_3 \vee \neg x_9 \vee \neg p_2)$.

The clauses (x_2), (x_7), and $(\neg x_2 \vee \neg x_7 \vee \neg p_1)$ force p_1 to take value 0 in each satisfying assignment of ϕ, because the facts f_2 and f_7 appear in every repair of the database, thus making 'JZA 8329' a consistent answer. In contrast, there is a satisfying assignment of ϕ in which p_2 is set to 1, implying that 'SWA 1568' is not a consistent answer.

3.2 Consistent Query Answering for Denial Constraints

Primary keys are an important but limited class of integrity constraints. In what follows, we consider the broader class of denial constraints, which includes primary keys and functional dependencies are special cases. We give a polynomial-time reduction from $\textsc{Cons}(q, \mathcal{I}, \Sigma)$ to UnSAT, where Σ is a fixed finite set of arbitrary denial constraints and q is a fixed union of non-boolean conjunctive queries. The potential answers to q are treated in the same way as the potential answers to the conjunctive query q in Reduction 3.1.2; to this effect, we introduce a boolean variable for each potential answer. As a result, it should not be difficult to imagine the union of boolean conjunctive queries as a special case of this reduction. Reduction 3.1.2 relies on the notions of *minimal violations* and *near-violations* to the set of denial constraints that we introduce next.

Definition 3.2.1. ***Minimal violation.*** *Assume that Σ is a set of denial constraints, $\mathcal{I}$ is an $\mathcal{R}$-instance, and $\mathcal{S}$ is a sub-instance of $\mathcal{I}$. We say that $\mathcal{S}$ is a minimal violation to Σ, if $\mathcal{S} \not\models \Sigma$ and for every set $\mathcal{S}' \subset \mathcal{S}$, we have that $\mathcal{S}' \models \Sigma$.*

Definition 3.2.2. ***Near-violation.*** *Assume that Σ is a set of denial constraints, $\mathcal{I}$ is an $\mathcal{R}$-instance, $\mathcal{S}$ is a sub-instance of $\mathcal{I}$, and f is a fact of $\mathcal{I}$. We say that $\mathcal{S}$ is a*

near-violation w.r.t. Σ *and* f*, if* $\mathcal{S} \models \Sigma$ *and* $\mathcal{S} \cup \{f\}$ *is a minimal violation to* Σ*. As a special case, if* $\{f\}$ *itself is a minimal violation to* Σ*, then we say that there is exactly one near-violation w.r.t.* f*, and it is the singleton* $\{f_t\}$*, where* f_t *is an auxiliary fact.*

3.2.1 Computing Minimal Violations and Near-Violations

For Reduction 3.2.1 to be a polynomial-time reduction, it is important that the minimal violations and the near-violations are computed in the time that is polynomial in the size of the database instance. Here, we show for a fixed finite set Σ of arbitrary denial constraints on a database schema $\mathcal{R}$, the minimal violations of an $\mathcal{R}$-instance $\mathcal{I}$ can be computed efficiently. The body of a denial constraint $d \in \Sigma$ is treated as a boolean conjunctive query q_d, possibly containing atomic formulas from d that use built-in predicates such as $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$, in addition to the relation symbols. The set of minimal witnesses to q_d on $\mathcal{I}$ is computed as described in Section 3.1.1, which is also, precisely, the set of minimal violations to d. The union of the sets of minimal violations over all denial constraints in Σ gives us the set of minimal violations to Σ. For each fact $f \in \mathcal{I}$, the set of near-violations to Σ w.r.t. f can be obtained by removing f from every minimal violation to Σ that contains f unless $\{f\}$ itself is a minimal violation to Σ. That special case can be handled separately. With this, we now give Reduction 3.2.1. Let $\mathcal{R}$ be a database schema, Σ be a fixed finite set of arbitrary denial constraints on $\mathcal{R}$, and let $q := q_1 \cup \cdots \cup q_k$ be a union of non-boolean conjunctive queries $q_1, \cdots, q_k$.

Reduction 3.2.1. *Given an* $\mathcal{R}$*-instance* $\mathcal{I}$*, we construct a boolean formula* ϕ' *as follows. Compute the following sets:*

- $\mathcal{V}$: *the set of minimal violations to* Σ *on* $\mathcal{I}$.
- $\mathcal{N}^i$: *the set of near-violations to* Σ, *on* $\mathcal{I}$, *w.r.t. each fact* $f_i \in \mathcal{I}$.
- $\mathcal{A}$: *the set of potential answers to* q *on* $\mathcal{I}$.
- $\mathcal{W}^l$: *the set of all minimal witnesses to* $q[\vec{a}_l]$ *on* $\mathcal{I}$, *for each* $\vec{a}_l \in \mathcal{A}$.

For each fact f_i *of* $\mathcal{I}$, *introduce a boolean variable* x_i, $1 \leq i \leq n$. *For the auxiliary fact* f_t, *introduce a boolean constant* $x_{true} = true$. *For each* $N_j^i \in \mathcal{N}^i$, *introduce a boolean variable* y_j^i, *and for each* $\vec{a}_l \in \mathcal{A}$, *introduce a boolean variable* p_l.

1. *For each* $V_j \in \mathcal{V}$, *construct a clause* $\alpha_j = \bigvee\limits_{f_i \in V_j} \neg x_i$.
2. *For each* $\vec{a}_l \in \mathcal{A}$ *and for each* $W_j^l \in \mathcal{W}^l$, *construct a clause* $\beta_j^l = \Big(\bigvee\limits_{f_i \in W_j^l} \neg x_i \Big) \vee \neg p_l$.
3. *For each* $f_i \in \mathcal{I}$, *construct a clause* $\gamma_i = x_i \vee \Big(\bigvee\limits_{N_j^i \in \mathcal{N}^i} y_j^i \Big)$.
4. *For each variable* y_j^i, *construct an expression* $\theta_j^i = y_j^i \leftrightarrow \Big(\bigwedge\limits_{f_d \in N_j^i} x_d \Big)$.
5. *Construct the following boolean formula* ϕ':

$$\phi' = \Big(\bigwedge_{i=1}^{|\mathcal{V}|} \alpha_i \Big) \wedge \Big(\bigwedge_{l=1}^{|\mathcal{A}|} \Big(\bigwedge_{j=1}^{|\mathcal{W}^l|} \beta_j^l \Big) \Big) \wedge \Big(\bigwedge_{i=1}^{|\mathcal{I}|} \Big(\Big(\bigwedge_{j=1}^{|\mathcal{N}^i|} \theta_j^i \Big) \wedge \gamma_i \Big) \Big)$$

Proposition 3.2.1. *For a boolean formula* ϕ' *constructed using Reduction 3.2.1, the following two statements hold.*

1. *The formula* ϕ' *can be transformed to an equivalent CNF-formula* ϕ *whose size is polynomial in the size of* $\mathcal{I}$.
2. *There exists a satisfying assignment to* ϕ' *in which a variable* p_l *is set to 1 if and only if* $\vec{a}_l \notin \textsc{Cons}(q, \mathcal{I}, \Sigma)$.

Proof. Let n be the number of facts of $\mathcal{I}$. Let d_1 be the smallest number such that there exists no denial constraint in Σ whose number of database atoms is bigger than d_1. Also,

let d_2 be the smallest number such that there exists no conjunctive query in q whose number of database atoms is bigger than d_2. Since Σ and q are not part of the input to $\textsc{Cons}(q)$, the quantities d_1 and d_2 are fixed constants. We also have that $|\mathcal{V}| \leq n^{d_1}$, $|\mathcal{N}^i| \leq n^{d_1}$ for $1 \leq i \leq n$, $|\mathcal{A}| \leq n^{d_2}$, and $|\mathcal{W}^l| \leq n^{d_2}$ for $1 \leq l \leq |\mathcal{A}|$. The number of x-, y-, and p-variables in ϕ' is therefore bounded by n, n^{d_1+1}, and n^{d_2}, respectively. The formula ϕ' contains as many α-clauses as $|\mathcal{V}|$, and none of the α-clause's length exceeds n. Similarly, there are at most n^{d_2} β-clauses, and none of their lengths exceeds $d_2 + 1$. The number of γ-clauses is precisely n, and each γ-clause is at most $n^{d_1+1} + 1$ literals long. There are as many θ-expressions as there are y-variables. Every θ-expression is of the form $y \leftrightarrow (x_1 \wedge ... \wedge x_d)$, where d is a constant obtained from the number of facts in the corresponding near-violation. Each θ-expression can be equivalently written in a constant number of CNF-clauses as $((\neg y \vee x_1) \wedge ... \wedge (\neg y \vee x_d)) \wedge (\neg x_1 \vee ... \vee \neg x_d \vee y)$, in which the length each clause is constant. This makes it possible to transform ϕ' into an equivalent CNF-formula ϕ, whose size is polynomial in the size of $\mathcal{I}$.

To prove the second part of the proposition, assume first that s is a satisfying assignment to the variables in ϕ' such that $s(p_l) = 1$. Construct a database instance $\mathcal{J}$ such that $f_i \in \mathcal{J}$ if and only if $s(x_i) = 1$. The α-clauses make sure that no minimal violation to Σ is present in $\mathcal{J}$, meaning that $\mathcal{J}$ is a consistent subset of $\mathcal{I}$. The γ-clauses and the θ-expressions encode the condition that, for every fact $f \in \mathcal{I}$, either $f \in \mathcal{J}$ or at least one near-violation w.r.t. Σ and f is in $\mathcal{J}$. This condition makes sure that $\mathcal{J}$ is indeed a repair of $\mathcal{I}$. Since $s(p_l) = 1$, the β^l-clauses ensure that at least one fact from each minimal witness to $q[\vec{a}_l]$ is missing from $\mathcal{J}$, meaning that $\vec{a}_l \notin q(\mathcal{J})$.

In the other direction, given a repair $\mathcal{J}$ that falsifies $q[\vec{a}_l]$, build an assignment s as follows. Set $s(x_i) = 1$ if and only if $f_i \in \mathcal{J}$. Set $s(p_l) = 1$, and set $s(p_{l'}) = 0$ for all $l' \neq l$. Since $\mathcal{J} \models \Sigma$, no minimal violation to Σ is a subset of $\mathcal{J}$, meaning that s satisfies all α-clauses in ϕ'. Also, for every fact $f \in \mathcal{I}$, it must be the case that either $f \in \mathcal{J}$ or at least one near-violation w.r.t. Σ and f is in $\mathcal{J}$ (otherwise $\mathcal{J}$ would not have been a repair of $\mathcal{I}$). Therefore, all γ-clauses and θ-expressions are also satisfied by the assignment s. Since $\mathcal{J} \not\models q[\vec{a}_l]$, at least one fact from each minimal witness to $q[\vec{a}_l]$ must be missing from $\mathcal{J}$, meaning that there is at least one variable x_i in each β^l-clause such that $s(x_i) = 0$. Hence, all β^l-clauses are satisfied by s, even when $s(p_l) = 1$. All other β-clauses are satisfied trivially, since $s(p_{l'}) = 0$, for all $l' \neq l$. □

Example 3.2.1. *Consider again the database instance from Table 3.1 but now with two additional integrity constraints as follows:*

(a) if a flight departs from YYZ, then its airline must be Jazz Air; and

(b) for Southwest airlines, if two tickets have the same code, then the ticket with an economy class must have a lower fare than the one with the first class.

These can be expressed as the following denial constraints:

(a) $\forall x, y, z, w, p, q \; \neg(\mathit{Flights}(x, y, z, \text{`YYZ'}, w, p, q) \wedge z \neq \text{`Jazz Air'})$

(b) $\forall x, y, z, w, p, q \; \neg(\mathit{Flights}(x, y, \text{`Southwest'}, z, w, p, q) \wedge \mathit{Tickets}(r, x, \text{`First'}, t)$

$$\wedge \mathit{Tickets}(r', x, \text{`Economy'}, t') \wedge t \leq t')$$

Suppose we want to find the PNR numbers of the tickets booked with first-class, or with Silkair airlines. This can be expressed as the union $q(x) := q_1 \cup q_2$ *of two unary*

conjunctive queries, where $q_1(x) := \exists y, z\ \mathit{Tickets}(x, y, \text{'First'}, z)$ *and*

$q_2(x) := \exists y, z, w, p, q, r, s, t\ \mathit{Tickets}(x, y, z, w) \wedge \mathit{Flights}(y, \text{'Silkair'}, p, q, r, s, t)$.

The minimal violations to Σ *are* $\{f_1, f_3\}$, $\{f_8\}$, *and* $\{f_4, f_6, f_9\}$, *while the minimal witnesses to* q *are* $\{f_5\}$, $\{f_6\}$, *and* $\{f_4, f_8\}$. *The near violations to* Σ *w.r.t. the facts* $f_1, \ldots, f_9$ *are* $\{f_3\}$, *none,* $\{f_1\}$, $\{f_6, f_9\}$, *none,* $\{f_4, f_9\}$, *none,* $\{f_t\}$, *and* $\{f_4, f_6\}$, *respectively. If* $|N^i_j| = 1$, *there is no need to introduce a variable* y^i_j, *since it is going to be equivalent to the* x*-variable corresponding to the only fact in* N^i_j. *With this, we construct the* α*-,* β*-,* γ*-clauses, and the* θ*-expressions of* ϕ' *as follows:*

α*-clauses:* $(\neg x_1 \vee \neg x_3), (\neg x_8), (\neg x_4 \vee \neg x_6 \vee \neg x_9)$

β*-clauses:* $(\neg x_5 \vee \neg p_1), (\neg x_6 \vee \neg p_2), (\neg x_4 \vee \neg x_8 \vee \neg p_3)$

γ*-clauses:* $(x_1 \vee x_3), (x_2), (x_3 \vee x_1), (x_4 \vee y^4_1), (x_5), (x_6 \vee y^6_1), (x_7), (x_8 \vee x_{true}), (x_9 \vee y^9_1)$

θ*-expressions:* $(y^4_1 \leftrightarrow (x_6 \wedge x_9)), (y^6_1 \leftrightarrow (x_4 \wedge x_9)), (y^9_1 \leftrightarrow (x_4 \wedge x_6))$

From this, the following CNF-formula ϕ *is constructed.*

$$\begin{aligned}\phi = {} & (\neg x_1 \vee \neg x_3) \wedge (\neg x_8) \wedge (\neg x_4 \vee \neg x_6 \vee \neg x_9) \wedge (\neg x_5 \vee \neg p_1) \wedge (\neg x_6 \vee \neg p_2) \\ & \wedge (\neg x_4 \vee \neg x_8 \vee \neg p_3) \wedge (x_1 \vee x_3) \wedge (x_2) \wedge (x_4 \vee y^4_1) \wedge (x_5) \wedge (x_6 \vee y^6_1) \wedge (x_7) \\ & \wedge (x_9 \vee y^9_1) \wedge (\neg y^4_1 \vee x_6) \wedge (\neg y^4_1 \vee x_9) \wedge (y^4_1 \vee x_6 \vee x_9) \wedge (\neg y^6_1 \vee x_4) \\ & \wedge (\neg y^6_1 \vee x_9) \wedge (y^6_1 \vee x_4 \vee x_9) \wedge (\neg y^9_1 \vee x_4) \wedge (\neg y^9_1 \vee x_6) \wedge (y^9_1 \vee x_4 \vee x_6)\end{aligned}$$

Observe that, in each satisfying assignment to ϕ, *the variable* p_1 *must take the value 0 due to the unit clause* (x_5). *In contrast, this is not the case for* p_2 *and* p_3. *By Proposition 3.2.1, 'KLF88V' is a consistent answer, but 'MJ9C8R' and 'NJ5RT3' are not.*

3.3 Computing Consistent Answers via Partial MaxSAT

By Proposition 3.1.1, the certain answer to a union of boolean conjunctive queries over a schema $\mathcal{R}$ with primary key constraints can be computed by solving the CNF-formula constructed in Reduction 3.1.1 using a SAT solver and checking for its unsatisfiability. For non-boolean queries, however, in a CNF-formula ϕ constructed using Reduction 3.1.2 or Reduction 3.2.1, one needs to identify each variable p_l such that there exists at least one satisfying assignment to ϕ in which p_l gets set to 1. By Proposition 3.2.1, the corresponding potential answers can then be discarded for being inconsistent. One way to do this is as follows. Add a clause $(p_1 \vee ... \vee p_{|A|})$ to ϕ, and solve ϕ using a SAT solver. For each p_l that gets set to 1 in the solution of ϕ, remove the literal p_l from ϕ and then solve ϕ again. Repeat this process until ϕ is no longer satisfiable. At the end of this iterative process, the potential answers corresponding to the p-variables that still occur positively in ϕ are precisely the consistent answers. This approach, however, requires many SAT instances to be solved when the number of potential answers is large. In the worst case, it needs to solve as many SAT instances as there are potential answers, which is polynomial in the size of the database instance. For this reason, we developed Algorithm 1, a different method that relies on iteratively solving Partial MaxSAT instances. Construction 3.3.1 describes the construction of the initial Partial MaxSAT instance that Algorithm 1 uses in its first iteration. In every iteration, Algorithm 1 computes an optimal solution to the Partial MaxSAT instance using a state-of-the-art solver and then modifies the instance based on the solution. This

iterative process computes the consistent answers to the union of conjunctive queries in consideration by means of eliminating all inconsistent potential answers. In our experiments, we found that Algorithm 1 takes less than four iterations to terminate even on large databases with queries of high selectivity.

Construction 3.3.1. *Let ϕ be the CNF-formula constructed using Reduction 3.1.2 or Reduction 3.2.1. Construct a Partial MaxSAT instance ψ as follows.*

1. *For each $\vec{a}_l \in \mathcal{A}$, construct a unit clause $\epsilon_l = (p_l)$.*
2. *Construct a CNF-formula $\psi = \phi \wedge \left(\bigwedge_{l=1}^{|\mathcal{A}|} \epsilon_l \right)$.*
3. *Treat all clauses in ψ that come from ϕ as hard and all ϵ_l-clauses as soft.*

Algorithm 1 Eliminating Inconsistent Potential Answers

```
1:  procedure EliminateWithPMaxSAT(ψ, A)
2:      let Ans = bool array[|A|]
3:      for l = 1 to |A| do
4:          Ans[l] ← true
5:      let bool moreAnswers ← true
6:      while moreAnswers do
7:          moreAnswers ← false
8:          let opt ← MaxSAT(ψ)                    ▷ Use Partial MaxSAT solver
9:          for l = 1 to |A| do
10:             if opt[p_l] = 1 then
11:                 moreAnswers ← true
12:                 Ans[l] ← false
13:                 Remove the unit clause (p_l) from ψ
14:                 Remove all clauses containing the literal ¬p_l from ψ
15:                 Add a new unit hard clause (¬p_l) to ψ
16:     return Ans
```

The idea of Algorithm 1 is to eliminate, in each iteration, as many inconsistent answers from $\mathcal{A}$ as possible by solving ψ, and after each call to a solver, modifying ψ in such a way that additional inconsistent answers, if any are still left, can be eliminated in subsequent iterations. Proposition 4.3.3 proves the correctness of Algorithm 1, but first, we state and prove Lemma 3.3.1, that reasons about the satisfying assignments to the Partial MaxSAT instance ψ, constructed using Construction 3.3.1. This lemma is

used in proving Proposition 4.3.3.

Lemma 3.3.1. *Let ϕ' be the CNF-formula constructed in Step 1 of Reduction 3.3.1, and ψ_i be the Weighted MaxSAT instance at the beginning of i^{th} iteration of Algorithm 1. For all i, every optimal solution of ψ_i satisfies all clauses in ϕ'.*

Proof. We prove Lemma 3.3.1 by induction on i. The CNF-formula ϕ' constructed in Step 1 of Reduction 3.3.1 can always be satisfied by setting all x-variables to 1, and all p-variables to 0. The clauses in ϕ being hard, ϵ being soft ensures that every optimal solution of ψ_0 satisfies all clauses in ϕ. Assume that for some $i \geq 0$, every optimal solution to ψ_i satisfies all clauses in ϕ. At the end of iteration $i+1$, if *moreAnswers* is true, the formula ψ_{i+1} is constructed from ψ_i by adding to ψ_i the unit hard clauses $(\neg p_l)$. This forces every optimal solution of ψ_{i+1} to satisfy all of these added clauses. Since no p_l variable occurs positively in ϕ, we have that, for all i, every optimal solution to ψ_{i+1} still satisfies ϕ. □

Proposition 3.3.1. *Algorithm 1 returns an array* Ans *such that $\vec{a}_l \in$ Cons$(q, \mathcal{I}, \Sigma)$ if and only if the entry* Ans$[l]$ *is true.*

Proof. In one direction, for every l, if $\vec{a}_l \in$ Cons$(q, \mathcal{I}, \Sigma)$, then, by the second part of Proposition 3.2.1, the variable p_l takes value 0 in every assignment that satisfies ϕ. By Lemma 3.3.1, for every i, the optimal solution of ψ_i also assigns value 0 to the variable p_l. As a result, Line 12 never gets executed, and the entry Ans$[l]$ remains true.

For the other direction, we first prove that Algorithm 1 always terminates. At the end of the i^{th} iteration, for every l, a unit clause (p_l) is present in ψ_i if and only

if ANS[l] is true. Hence, at the end of i^{th} iteration, if *moreAnswers* is true, then the optimal solution to ψ_i must have assigned value 1 to at least one variable p_l such that ANS[l] was previously true. Therefore, at the end of i^{th} iteration, at least i entries in ANS are false. It follows that Algorithm 1 terminates after at most $|\mathcal{A}|$ iterations.

Now, since no clause in ϕ contains a positive literal p_l, the addition of a unit hard clause $(\neg p_l)$ to ψ_i does not suppress any satisfying assignments to ϕ while finding the optimal solution to ψ_{i+1}. Therefore, in every iteration, the optimal solution of ψ guarantees to satisfy the maximum number of p_l variables for which the unit clause (p_l) is still in ψ. As a result, Algorithm 1 does not terminate until it marks the entries ANS[l] false, for all l, for which there exists a satisfying assignment to ϕ in which p_l gets assigned to 1. In other words, by the second part of Proposition 3.2.1, for every inconsistent answer $\vec{a}_l$, the entry ANS gets marked as false. □

Chapter 4

Range Consistent Answers via SAT Solving

In this chapter, we give polynomial-time reductions from computing the range consistent answers of aggregation queries to variants of SAT. The reductions Section 4.2 and Section 4.3 assume that the aggregation query does not have the grouping construct, and the database schema has one key constraint per relation; in Section 4.4, we show how these reductions can be extended to schemata with arbitrary denial constraints, and in Section 4.5 we describe an iterative algorithm to obtain the range consistent answers to aggregation queries with grouping. The range consistent answers to the aggregation queries with `SUM(`A`)`, `COUNT(`A`)`, and `COUNT(*)` operators are obtained in a similar fashion; Reduction 4.2.1 handles these operators. The reductions to handle queries with `MIN(`A`)` and `MAX(`A`)` operators are different and we discuss them in Section 4.3. We do not yet have a natural reduction from the range consistent answers to

queries with the aggregation operator AVG(A) to any variant of SAT, and the problem is left open for future research (see Chapter 7 for more details). Before we provide the reductions to compute the range consistent answers using SAT solvers, in Section 4.1, we first corroborate the need for a system that goes well beyond ConQuer by showing two hardness results about the range consistent answers. Specifically, in Theorem 4.1.1, we prove that there exists an aggregation query Q involving SUM(A) such that the consistent answers of the underlying conjunctive query q w.r.t. key constraints are FO-rewritable, but the range consistent answers of Q are NP-hard.

4.1 The Complexity of Range Consistent Answers

Recall that Arenas et al. [16] investigated the computational complexity of the range consistent answers for scalar aggregation queries of the form

$$\texttt{SELECT } f(A) \texttt{ FROM } R(U, A) \quad ,$$

where $f(A)$ is one of the standard aggregation operators and $R(U, A)$ is a relational schema with functional dependency constraints. They proved a dichotomy in the complexity of consistent answers to such queries, i.e., for queries without grouping and with exactly one relation symbol, where the integrity constraints in consideration were functional dependencies (see Section 2.2 for details). However, it remains an open problem to pinpoint the complexity of the range consistent answers to a richer class of aggregation queries, namely, the queries of the form

$$Q := \texttt{SELECT } Z, f(A) \texttt{ FROM } T(U, Z, A) \texttt{ GROUP BY } Z,$$

where $T(U, Z, A)$ is the relation returned by a conjunctive query q or by a union $q := q_1 \cup \cdots \cup q_k$ of conjunctive queries.

First, we show in Proposition 4.1.1 that if computing the consistent answers $\textsc{Cons}(q)$ of the underlying query q is a hard problem, then computing the range consistent answers $\textsc{Cons}(Q)$ of the aggregation query Q is a hard problem as well.

Proposition 4.1.1. *Let* $Q :=$ `SELECT` $Z, f(A)$ `FROM` $T(U, Z, A)$ `GROUP BY` Z *be an aggregation query where* $T(U, Z, A)$ *is the relation returned by a self-join-free conjunctive query* $q(U, Z, A)$ *such that* $\textsc{Cons}(q) \notin P$. *Then,* $\textsc{Cons}(Q) \notin P$.

Proof. We use the notion of attack graphs [57, 58, 59] for this proof. Let G be the attack graph of the boolean self-join-free conjunctive query q_b where q_b is obtained from $q(U, Z, A)$ by replacing the variable corresponding to the aggregation attribute A and the variables corresponding to the attributes in U and Z by some constants. Since $\textsc{Cons}(q) \notin P$, we must have that $\textsc{Certainty}(q_b) \notin P$ and that G has a strong cycle (because $\textsc{Certainty}(q_b) \in P$ would imply $\textsc{Cons}(q) \in P$). Now, consider a query q' such that the atoms in q' are the same as that in q, but the variables corresponding to the grouping attribute Z are the only free variables in q'. Hence, q' can be written as $q'(Z) := \exists U \exists A\ T(U, Z, A)$. Therefore, for every atom $F \in q$ (or $F \in q'$), we have that the set $F^{+,q}$ of variables, the transitive closure of the key variables of F w.r.t. the dependencies arising from the key constraints on the relation symbols in $q \backslash F$, is a subset of $F^{+,q'}$. Hence, the graph G is a spanning subgraph of the attack graph of q', implying that the attack graph of q' also contains a strong cycle and $\textsc{Cons}(q') \notin P$.

Now, it is easy to see that if $\textsc{Cons}(Q) \in P$, the attributes in Z can be projected out from a materialized view of $\textsc{Cons}(Q)$ to obtain $\textsc{Cons}(q')$ in polynomial time, leading to a contradiction. □

This gives rise to the following question: what can we say about the complexity of the range consistent answers $\textsc{Cons}(Q)$ if computing the consistent answers $\textsc{Cons}(q)$ of the underlying query is an easy problem?

Recall that Fuxman [39] introduced a class called $C_{aggforest}$ consisting of all aggregation queries such that the aggregation operator is one of `MIN(`A`)`, `MAX(`A`)`, `SUM(`A`)`, `COUNT(*)`, the underlying query q is a conjunctive query in C_{forest}, and there is one key constraint for each relation in the underlying query q. Fuxman [39] showed that the range consistent answers of every query in $C_{aggforest}$ are SQL-rewritable but the SQL-rewritability of aggregation queries beyond those in $C_{aggforest}$ has not been investigated. Here, we show that there exists a self-join-free conjunctive query whose consistent answers are FO-rewritable, but the SQL-rewritability property does not hold when an aggregation operator is added on top of it. Specifically, we reduce the Maximum Cut problem to the problem of computing the range consistent answers to an aggregation query without grouping that involves the `SUM` operator and whose underlying conjunctive query has FO-rewritable consistent answers. We begin by recalling the definition of the Maximum Cut, a fundamental NP-complete problem [52].

Definition 4.1.1. Maximum Cut. *For an undirected graph $G = (V, E)$, a* cut *of G is a partition $(S, \overline{S})$ of V, where $S \subseteq V$ and $\overline{S} = V \backslash S$. The set of edges with one vertex*

in S and one vertex in $\overline{S}$ is denoted by $E(S, \overline{S})$, and the the size of the cut $(S, \overline{S})$ is $|E(S, \overline{S})|$. The Maximum Cut *problem asks: Given an undirected graph G and an integer k, is there a cut of G that has size at least k?*

Theorem 4.1.1. *Let $\mathcal{R}$ be a schema with three relations $R_1(\underline{A_1}, B_1)$, $R_2(\underline{A_2}, B_2)$, and $R_3(\underline{A_1, B_1, A_2, B_2}, C)$. Let Q be the following aggregation query:*

$$Q := \texttt{SELECT SUM(A) FROM } \; q(A),$$

where $q(A)$ is the following self-join-free conjunctive query:

$$q(A) := \exists x \exists y \; R_1(\underline{x}, \text{'red'}) \wedge R_2(\underline{y}, \text{'blue'}) \wedge R_3(\underline{x, \text{'red'}, y, \text{'blue'}, A}).$$

Then the following two statements hold.

1. Cons(q) *is FO-rewritable.*
2. Cons(Q) *is NP-hard.*

Proof. To show that Cons(q) is FO-rewritable, consider the first-order query q':

$$q'(A) := \exists x \exists y (R_1(\underline{x}, \text{'red'}) \wedge R_2(\underline{y}, \text{'blue'}) \wedge R_3(\underline{x, \text{'red'}, y, \text{'blue'}, A})$$
$$\wedge \forall z (R_1(x, z) \rightarrow z = \text{'red'}) \wedge \forall w (R_2(y, w) \rightarrow w = \text{'blue'})).$$

We will show that for every instance $\mathcal{I}$ and every value a, we have that $a \in q'(\mathcal{I}) \Leftrightarrow a \in \text{Cons}(q, \mathcal{I})$. Since q' filters out the tuples from R_1 and R_2 that participate in the violations of the key constraints, we have that if $a \in q'(\mathcal{I})$, then $a \in q(\mathcal{J})$, for every repair $\mathcal{J}$ of $\mathcal{I}$, which means that $a \in \text{Cons}(q, \mathcal{I})$. In the other direction, we claim that if $a \in \text{Cons}(q, \mathcal{I})$, then $a \in q'(\mathcal{I})$. Indeed, if $a \notin q'(\mathcal{I})$, then for all x and y such that $R_1(x, \text{'red'}) \wedge R_2(y, \text{'blue'}) \wedge R_3(x, \text{'red'}, y, \text{'blue'}, a)$, we would have that there is some z

such that $R_1(x, z)$ and $z \neq$ 'red' or there is some w such that $R_2(y, w)$ and $w \neq$ 'blue'. Construct a repair $\mathcal{J}$ of $\mathcal{I}$ as follows. First, for every x, if 'red' is the only value z such that $R_1(x, z)$ is a fact of $\mathcal{I}$, then put $R_1(x, \text{'red'})$ in $\mathcal{J}$; otherwise, pick an element $z^* \neq$ 'red' such that $R_1(x, z^*)$ is a fact of $\mathcal{I}$ and put $R_1(x, z^*)$ in $\mathcal{J}$. Second, for every y, if 'blue' is the only value w such that $R_2(y, w)$ is a fact of $\mathcal{I}$, then put $R_2(y, \text{'blue'})$ in $\mathcal{J}$; otherwise, pick an element $w^* \neq$ 'blue' such that $R_1(y, w^*)$ is a fact of $\mathcal{I}$ and put $R_2(x, w^*)$ in $\mathcal{J}$. Third, put every tuple of the relation R_3 of $\mathcal{I}$ into $\mathcal{J}$. Clearly, $\mathcal{J}$ is a repair of $\mathcal{I}$. Moreover, $a \notin q(\mathcal{J})$. Indeed, if $a \in q(\mathcal{J})$, then there are elements x and y such that $\mathcal{J} \models R_1(x, \text{'red'}) \wedge R_2(y, \text{'blue'}) \wedge R_3(x, \text{'red'}, y, \text{'blue'}, a)$. Since $a \notin q'(\mathcal{I})$, we have that there is some z' such that $R_1(x, z')$ and $z' \neq$ 'red' or there is some w' such that $R_2(y, w')$ and $w' \neq$ 'blue'. In the first case, the construction of $\mathcal{J}$ implies that $R_1(x, \text{'red'})$ is not a fact of $\mathcal{J}$, while in the second case, the construction of $\mathcal{J}$ implies that $R_2(y, \text{'blue'})$ is not a fact of $\mathcal{J}$; in either case, we have arrived at a contradiction.

In the other direction, if $a \in \textsc{Cons}(q, \mathcal{I})$, there must exist x and y such that the dependencies $(R_1(x, z) \rightarrow z =$ 'red') and $(R_2(y, w) \rightarrow w =$ 'blue') hold in $\mathcal{I}$, since, otherwise, we could construct a repair $\mathcal{J}$ of $\mathcal{I}$ such that for every x, y that satisfies $R_1(x, \text{'red'}) \wedge R_2(y, \text{'blue'}) \wedge R_3(x, \text{'red'}, y, \text{'blue'}, u)$, we pick a z other than 'red' and put $R_1(x, z)$ in $\mathcal{J}$, and pick a w other than 'blue' and put $R_2(y, w)$ in $\mathcal{J}$. This would imply $a \notin q(\mathcal{J})$, which is a contradiction.

To show that $\textsc{Cons}(Q)$ is NP-hard, consider the following reduction from undirected graphs to $\mathcal{R}$-instances, where $\mathcal{R}$ is the schema with relations $R_1(\underline{A_1}, B_1)$, $R_2(\underline{A_2}, B_2)$, and $R_3(\underline{A_1, B_1, A_2, B_2, C})$.

Reduction 4.1.1. *Given an undirected graph $G = (V, E)$, let $m = -|E| - 1$, and construct an $\mathcal{R}$-instance $\mathcal{I}$ as follows.*

- *For each $v \in V$, add tuples $R_1(v,$ 'red'$)$, $R_1(v,$ 'blue'$)$, $R_2(v,$ 'red'$)$, and $R_2(v,$ 'blue'$)$ to $\mathcal{I}$.*
- *For each $v \in V$, add a tuple $R_3(v,$ 'red'$, v,$ 'blue'$, m)$ to $\mathcal{I}$.*
- *For each edge $(u, v) \in E$, add tuples $R_3(u,$ 'red'$, v,$ 'blue'$, 1)$ and $R_3(v,$ 'red'$, u,$ 'blue'$, 1)$ to $\mathcal{I}$.*

We will show that Reduction 4.1.1 reduces Maximum Cut to computing the range semantics of the aggregation query Q. Let G be an undirected graph and $\mathcal{I}$ be the database instance constructed from G using Reduction 4.1.1. We say that a repair $\mathcal{J}'$ of $\mathcal{I}$ produces a *red-blue coloring* of G if for every vertex $v \in V$, we have that the tuples $R_1(v,$'red'$)$ and $R_2(v,$'red'$)$ are either both present in $\mathcal{J}'$ or both absent in $\mathcal{J}'$. We now prove a useful lemma.

Lemma 4.1.1. *For every repair $\mathcal{J}$ of $\mathcal{I}$, there exists a repair $\mathcal{J}'$ of $\mathcal{I}$ (not necessarily different from $\mathcal{J}$) such that $\mathcal{J}'$ produces a red-blue coloring of G and $Q(\mathcal{J}') \geq Q(\mathcal{J})$.*

Proof. Let $\mathcal{J}$ be a repair of $\mathcal{I}$. Construct an $\mathcal{R}$-instance $\mathcal{J}'$ from $\mathcal{J}$ as follows. For every vertex $v \in V$, if both tuples $R_1(v, x)$ and $R_2(v, x)$ are present in $\mathcal{J}$ for $x \in \{$'red', 'blue'$\}$, then add them to $\mathcal{J}'$. Otherwise, add the tuples $R_1(v,$'red'$)$ and $R_2(v,$'red'$)$ to $\mathcal{J}'$. Also, copy all tuples from relation R_3 of $\mathcal{J}$ to relation R_3 of $\mathcal{J}'$. Clearly, $\mathcal{J}'$ is a repair of $\mathcal{I}$ and $\mathcal{J}'$ produces a red-blue coloring of G. Observe that $Q(\mathcal{J}')$ can be different than $Q(\mathcal{J})$ only if there exists at least one vertex $v \in V$ such that either

$R_1(v, \text{'red'}), R_2(v, \text{'blue'}) \in \mathcal{J}$ or $R_1(v, \text{'blue'}), R_2(v, \text{'red'}) \in \mathcal{J}$. We will show that $Q(\mathcal{J}') \geq Q(\mathcal{J})$ holds in both cases.

Case 1: Let v be a node such that $R_1(v, \text{'red'}), R_2(v, \text{'blue'}) \in \mathcal{J}$. In this case, while populating the database instance $\mathcal{J}'$, vertex v changes its color in relation R_2, i.e., we have that $R_2(v, \text{'red'}) \in \mathcal{J}'$ and $R_2(v, \text{'blue'}) \notin \mathcal{J}'$. Therefore, the summands arising from the tuples of the form $R_1(u, \text{'red'})$, $R_2(v, \text{'blue'})$, and $R_3(u, \text{'red'}, v, \text{'blue'}, 1)$ of $\mathcal{J}$ (for some vertex $u \neq v \in V$) do not appear in $Q(\mathcal{J}')$. Notice that each of these summands contributes value 1 to $Q(\mathcal{J})$ and the number of these summands is at most $|E|$. At the same time, the summand that contributes value m to $Q(\mathcal{J})$ arising from the tuples $R_1(v, \text{'red'})$, $R_2(v, \text{'blue'})$, and $R_3(v, \text{'red'}, v, \text{'blue'})$ of $\mathcal{J}$ also does not appear in $Q(\mathcal{J}')$. Since $m = -|E| - 1$, it follows that $Q(\mathcal{J}')$ cannot be made smaller than $Q(\mathcal{J})$ on account of such a node v.

Case 2: Let v be a node such that $R_1(v, \text{'blue'}), R_2(v, \text{'red'}) \in \mathcal{J}$. In this case, while populating $\mathcal{J}'$, vertex v changes its color in relation R_1, i.e., we have that $R_1(v, \text{'red'}) \in \mathcal{J}'$ and $R_1(v, \text{'blue'}) \notin \mathcal{J}'$. Compared to $Q(\mathcal{J})$, this can only increase the number of summands that contribute 1 to $Q(\mathcal{J}')$, by possibly having new summands arising from the tuples of type $R_1(v, \text{'red'})$, $R_2(w, \text{'blue'})$, and $R_3(v, \text{'red'}, w, \text{'blue'}, 1)$ of $\mathcal{J}'$ (for some vertex $w \neq v \in V$). Moreover, for every vertex $u \in V$, it is true that, if $R_1(u, \text{'red'}) \in \mathcal{J}$ then $R_1(u, \text{'red'}) \in \mathcal{J}'$; similarly, if $R_2(u, \text{'blue'}) \in \mathcal{J}$ then $R_2(u, \text{'blue'}) \in \mathcal{J}'$. Therefore, every summand that contributes 1 to $Q(\mathcal{J})$ also contributes 1 to $Q(\mathcal{J}')$. Hence, $Q(\mathcal{J}')$ cannot be made smaller than $Q(\mathcal{J})$ on account of such a node v. The preceding analysis implies that $Q(\mathcal{J}') \geq Q(\mathcal{J})$. □

By Lemma 4.1.1, there exists a repair $\mathcal{J}$ of $\mathcal{I}$ such that $\mathcal{J}$ produces a red-blue coloring of G and $Q(\mathcal{J})$ is the *lub*-answer in $\textsc{Cons}(Q, \mathcal{I})$. We will show that, for every non-negative integer k, there is a cut $(S, \overline{S})$ of G such that $|E(S, \overline{S})| \geq k$ if and only if there exists a repair $\mathcal{J}$ of $\mathcal{I}$ such that $\mathcal{J}$ produces a red-blue coloring of G and $Q(\mathcal{J}) \geq k$. This will be sufficient to prove the NP-hardness of $\textsc{Cons}(Q, \mathcal{I})$ since it will follow that it is NP-hard to even compute the *lub*-answer in $\textsc{Cons}(Q, \mathcal{I})$.

Let $(S, \overline{S})$ be a cut of G such that $|E(S, \overline{S})| \geq k$. Construct an $\mathcal{R}$-instance $\mathcal{J}$ as follows. For each vertex $v \in S$, add tuples $R_1(v, \text{'red'})$ and $R_2(v, \text{'red'})$ to $\mathcal{J}$. For each vertex $v \in \overline{S}$, add tuples $R_1(v, \text{'blue'})$ and $R_2(v, \text{'blue'})$ to $\mathcal{J}$. Add all tuples from relation R_3 of $\mathcal{I}$ to $\mathcal{J}$. Observe that $\mathcal{J}$ is a repair of $\mathcal{I}$ and that $\mathcal{J}$ produces a red-blue coloring of G. Also, every edge $(u, v) \in E$ such that $u \in S$ and $v \in \overline{S}$ is part of a witness to a summand that contributes 1 to $Q(\mathcal{J})$. Moreover, no summand in $Q(\mathcal{J})$ arises from a tuple of the form $R_3(v, \text{'red'}, v, \text{'blue'}, m)$ for some $v \in V$. Since we have that $|E(S, \overline{S})| \geq k$, it must be the case that $Q(\mathcal{J}) \geq k$. In the other direction, let $\mathcal{J}$ be a repair of $\mathcal{I}$ such that $\mathcal{J}$ produces a red-blue coloring of G and $Q(\mathcal{J}) \geq k$. Construct two sets S and $\overline{S}$ of vertices of G as follows. Let $v \in S$ if $R_1(v, \text{'red'}) \in \mathcal{J}$, and let $v \in \overline{S}$ if $R_1(v, \text{'blue'}) \in \mathcal{J}$. Clearly, $(S, \overline{S})$ is a cut of G. Every edge $(u, v) \in E$ such that $u \in S$ and $v \in \overline{S}$ is part of a witness to a summand that contributes 1 to $Q(\mathcal{J})$ since the tuples $R_1(u, \text{'red'})$, $R_2(v, \text{'blue'})$, and $R_3(u, \text{'red'}, v, \text{'blue'}, 1)$ of $\mathcal{J}$ satisfy the underlying conjunctive query of Q. In fact, since $\mathcal{J}$ produces a red-blue coloring of G, every summand that contributes to $Q(\mathcal{J})$ must arise from such tuples. Since $Q(\mathcal{J}) \geq k$, it must be the case that $|E(S, \overline{S})| \geq k$. □

In Step 2 of Reduction 4.1.1, we add a tuple of relation R_3 to $\mathcal{I}$ in which we pass the value $-|E|-1$ for the aggregation attribute A because this value being strictly smaller than $-|E|$ is crucial for Lemma 4.1.1 to hold. As a consequence, Reduction 4.1.1 cannot be extended to show the NP-hardness of the `COUNT(*)` or the `COUNT(`A`)` operator in a straightforward way as the value of a count can never be negative. We note that the NP-hardness of the range consistent answers to an aggregation query with the `COUNT(`A`)` operator where the underlying conjunctive query has FO-rewritable consistent answers is already known [17], but such a result remains open for `COUNT(*)`, `MIN(`A`)`, `MAX(`A`)`, and `AVG(`A`)` operators.

4.2 Answering Queries with SUM/COUNT without Grouping

Let $\mathcal{R}$ be a database schema with one key constraint per relation, and Q be the aggregation query

$$Q := \texttt{SELECT } f \texttt{ FROM } T(U, A),$$

where f is one of the operators `COUNT(*)`, `COUNT(`A`)`, `SUM(`A`)`, and $T(U, A)$ is a relation expressed as a union of conjunctive queries over $\mathcal{R}$. We now give reductions from computing the range consistent answers $\textsc{Cons}(Q)$ on an $\mathcal{R}$-instance $\mathcal{I}$ to Partial MaxSAT and to Weighted Partial MaxSAT.

4.2.1 Reductions to Partial MaxSAT and Weighted Partial MaxSAT

Reduction 4.2.1. *Let $Q :=$ `SELECT` f `FROM` $T(U, A)$ be an aggregation query, where f is one of the operators `COUNT(*)`, `COUNT(`A`)`, and `SUM(`A`)`. Let $\mathcal{I}$ be an $\mathcal{R}$-instance and*

$\mathcal{G}$ be the set of key-equal groups of facts of $\mathcal{I}$. For each fact f_i of $\mathcal{I}$, introduce a boolean variable x_i. Let $\mathcal{W}$ be the set of minimal witnesses to the query q^* on $\mathcal{I}$, where

$$q^* := \begin{cases} \exists U \exists A\ T(U, A) & \text{if } f \text{ is } \texttt{COUNT(*)} \\ \exists U\ T(U, A) & \text{if } f \text{ is } \texttt{COUNT(A)} \text{ or } \texttt{SUM(A)}. \end{cases}$$

Construct a Partial MaxSAT instance ϕ (if f is `COUNT(*)` *or* `COUNT(A)`*) or a Weighted Partial MaxSAT instance ϕ (if f is* `SUM(A)`*) as follows:*

(1) For each $G_j \in \mathcal{G}$,

- *construct a hard clause $\alpha_j = \bigvee_{f_i \in G_j} x_i$.*
- *for each pair (f_m, f_n) of facts in G_j such that $m \neq n$, construct a hard clause $\alpha_j^{mn} = (\neg x_m \vee \neg x_n)$.*

(2a) If f is `COUNT(*)` *or* `COUNT(A)`*, then for each $W_j \in \mathcal{W}$, construct a soft clause β_j (i.e., a clause of weight 1), where*

$$\beta_j = \Big(\bigvee_{f_i \in W_j} \neg x_i, 1 \Big).$$

Construct a Partial MaxSAT instance

$$\phi = \Big(\bigwedge_{j=1}^{|\mathcal{G}|} \alpha_j \Big) \wedge \Big(\bigwedge_{j=1}^{|\mathcal{G}|} \Big(\bigwedge_{\substack{f_m \in \mathcal{G}_j \\ f_n \in \mathcal{G}_j}} \alpha_j^{mn} \Big) \Big) \wedge \Big(\bigwedge_{j=1}^{|\mathcal{W}|} \beta_j \Big).$$

(2b) If f is `SUM(A)`*, let $\mathcal{W}_P$ and $\mathcal{W}_N$ be the subsets of $\mathcal{W}$ such that for each $W_j \in \mathcal{W}$, we have $W_j \in \mathcal{W}_P$ iff $q^*(W_j) > 0$, and $W_j \in \mathcal{W}_N$ iff $q^*(W_j) < 0$. Let also $w_j = ||q^*(W_j)||$, where $||q^*(W_j)||$ is the absolute value of $q^*(W_j)$. Construct a weighted soft clause β_j and a conjunction γ_j of hard clauses as follows. If $W_j \in \mathcal{W}_{\mathcal{N}}$,*

introduce a new variable y_j *and let*

$$\beta_j = (y_j, w_j) \text{ and}$$

$$\gamma_j = \Big(\Big(\bigvee_{f_i \in W_j} \neg x_i\Big) \vee y_j\Big) \wedge \Big(\bigwedge_{f_i \in W_j} (\neg y_j \vee x_i)\Big);$$

otherwise, let $\beta_j = \Big(\bigvee_{f_i \in W_j} \neg x_i, w_j\Big)$ *and do not construct* γ_j.

Construct a Weighted Partial MaxSAT instance

$$\phi = \Big(\bigwedge_{j=1}^{|\mathcal{G}|} \alpha_j\Big) \wedge \Big(\bigwedge_{j=1}^{|\mathcal{G}|} \Big(\bigwedge_{\substack{f_m \in \mathcal{G}_j \\ f_n \in \mathcal{G}_j}} \alpha_j^{mn}\Big)\Big) \wedge \Big(\bigwedge_{j=1}^{|\mathcal{W}|} \beta_j\Big) \wedge \Big(\bigwedge_{W_j \in \mathcal{W}_\mathcal{N}} \gamma_j\Big).$$

Proposition 4.2.1. *Let* $Q :=$ `SELECT` f `FROM` $T(U, A)$ *be an aggregation query, where* f *is one of the operators* `COUNT(*)`, `COUNT(A)`, *and* `SUM(A)`. *In a maximum satisfying assignment of the Partial MaxSAT instance or the Weighted Partial MaxSAT instance* ϕ *constructed using Reduction 3.1.1, the sum of weights of the falsified clauses is the glb-answer in the range consistent answers* $\textsc{Cons}(Q, \mathcal{I})$.

Proof. Let $Q :=$ `SELECT COUNT(*) FROM` $T(U, A)$, and let s be an assignment of the formula ϕ constructed using Reduction 3.1.1. Let $g(\phi, s)$ denote the sum of weights of the soft clauses of ϕ satisfied by s. Construct a database sub-instance $\mathcal{J}$ from s such that $f_i \in \mathcal{J}$ if and only if $s(x_i) = 1$. The hard clauses of ϕ constructed in Step (1) of Reduction 3.1.1 encode the condition that exactly one fact from each key-equal group of facts of $\mathcal{I}$ is in $\mathcal{J}$, ensuring that $\mathcal{J}$ is a repair of $\mathcal{I}$. Moreover, the soft clauses of ϕ falsified by s have a one-to-one correspondence with the minimal witnesses to q^* in $\mathcal{J}$. Therefore, we have that $g(\phi, s) = |\mathcal{W}| - Q(\mathcal{J})$. Since $|\mathcal{W}|$ does not depend on $\mathcal{J}$, the answer $Q(\mathcal{J})$ is minimized (i.e., $Q(\mathcal{J})$ is a *glb*-answer in $\textsc{Cons}(Q)$ on $\mathcal{I}$) when

s is a maximum satisfying assignment. Essentially the same argument works for the case where $Q :=$ `SELECT COUNT(`A`) FROM` $T(U, A)$. A dual argument to this proves that a repair of $\mathcal{I}$ constructed from a minimum satisfying assignment of ϕ realizes the *lub*-answer in $\textsc{Cons}(Q)$ on $\mathcal{I}$.

Now, let $Q :=$ `SELECT SUM(`A`) FROM` $T(U, A)$. Construct a repair $\mathcal{J}$ of $\mathcal{I}$ from s by choosing $f_i \in \mathcal{J}$ if and only if $s(x_i) = 1$. Also, construct a database instance $\mathcal{I}_p$ as follows. For every fact $f \in \mathcal{I}$, let $f \in \mathcal{I}_p$ if and only if $f \in W_j$ for some $W_j \in \mathcal{W}_p$. Thus, $Q(\mathcal{I}_p)$ is the sum of values of the aggregation attribute evaluated on the witnesses in $\mathcal{W}_p$. Observe that, for every $W_j \in \mathcal{W}_P$, the clause β_j is falsified by s if and only if $W_j \in \mathcal{J}$. Similarly, for every $W_j \in \mathcal{W}_N$, the clause β_j is satisfied by s if and only if $W_j \in \mathcal{J}$. Therefore, we have that,

$$\begin{aligned} Q(\mathcal{J}) &= Q(\mathcal{I}_p) - \Sigma_{W_j \in \mathcal{W}_P \wedge s(\beta_j)=1}(w_j) - \Sigma_{W_j \in \mathcal{W}_N \wedge s(\beta_j)=1}(w_j) \\ &= Q(\mathcal{I}_p) - g(\phi, s) \end{aligned}$$

Since $Q(\mathcal{I}_p)$ does not depend on $\mathcal{J}$, the answer $Q(\mathcal{J})$ is minimized (i.e., $Q(\mathcal{J})$ is a *glb*-answer in $\textsc{Cons}(Q)$ on $\mathcal{I}$) when s is a maximum satisfying assignment. □

Corollary 4.2.1. *Let $Q :=$ `SELECT` f `FROM` $T(U, A)$ be an aggregation query, where f is one of the operators `COUNT(*)`, `COUNT(`A`)`, and `SUM(`A`)`. In a minimum satisfying assignment of the Partial MinSAT instance or the Weighted Partial MinSAT instance ϕ constructed using Reduction 3.1.1, the sum of weights of the falsified clauses is the lub-answer in the range consistent answers $\textsc{Cons}(Q, \mathcal{I})$.*

Proof. A dual argument to that of Proposition 4.2.1 proves that a repair of $\mathcal{I}$ constructed

from a minimum satisfying assignment to ϕ realizes the *lub*-answer in $\textsc{Cons}(Q, \mathcal{I})$. $\square$

We will use the inconsistent database instance of bank account records shown in Table 4.1 as a running example throughout this section. This database instance has three relations, namely, *Customer*, *Accounts*, and *CustAcc*, with relations *Customer* and *Accounts* having one key constraint each (the key attributes are underlined in Table 4.1). Similar to the database instance in Table 3.1, we associate the attribute *Fact* to each relation for convenience to uniquely refer to each fact of the database instance.

Table 4.1: An inconsistent database instance of bank account records

Customer			
Fact	CID	NAME	CITY
f_1	C1	John	LA
f_2	C2	Mary	LA
f_3	C2	Mary	SF
f_4	C3	Don	SF
f_5	C4	Jen	LA

Accounts				
Fact	ACCID	TYPE	CITY	BAL
f_6	A1	Check.	LA	900
f_7	A2	Check.	LA	1000
f_8	A3	Saving	SJ	1200
f_9	A3	Saving	SF	-100
f_{10}	A4	Saving	SJ	300

CustAcc		
Fact	CID	ACCID
f_{11}	C1	A1
f_{12}	C2	A2
f_{13}	C2	A3
f_{14}	C3	A4

Example 4.2.1. *Let Q be the following aggregation query that counts the number of customers having an account in their own city:*

```
SELECT COUNT(*) FROM CUSTOMER, ACCOUNTS, CUSTACC
WHERE CUSTOMER.CID = CUSTACC.CID
  AND ACCOUNTS.ACCID = CUSTACC.ACCID
  AND CUSTOMER.CITY = ACCOUNTS.CITY
```

From Reduction 3.1.1, we construct the following clauses:

α-clauses: $x_1, (x_2 \vee x_3), x_4, x_5, x_6, x_7, (x_8 \vee x_9), x_{10}$*;*

α^{mn}-clauses: $(\neg x_2 \vee \neg x_3), (\neg x_8 \vee \neg x_9)$*;*

β-clauses: $(\neg x_1 \vee \neg x_6, 1), (\neg x_2 \vee \neg x_7, 1), (\neg x_3 \vee \neg x_9, 1)$.

Observe that it is okay to omit the variables corresponding to the facts in `CUSTACC` *since* `CUSTACC` *does not violate a key constraint. A maximum satisfying assignment to the Partial MaxSAT instance ϕ constructed from above clauses is $x_i = 0$ for $i \in \{2, 9\}$, and $x_i = 1$ otherwise. It falsifies one clause, namely, $(\neg x_1 \vee \neg x_6, 1)$. Similarly, an assignment $x_i = 0$ for $i \in \{2, 8\}$, and $x_i = 1$ otherwise is a minimum satisfying assignment to the Partial MinSAT instance ϕ, and it falsifies two clauses, namely, $(\neg x_1 \vee \neg x_6, 1)$ and $(\neg x_3 \vee \neg x_9, 1)$. Thus, $\textsc{Cons}(Q, \mathcal{I})$ w.r.t. range semantics is $[1, 2]$ by Proposition 3.1.1.*

Example 4.2.2. *Now, consider the following aggregation query Q:*

```
SELECT SUM(ACCOUNTS.BAL) FROM CUSTOMER, ACCOUNTS, CUSTACC
WHERE CUSTOMER.CID = CUSTACC.CID
  AND ACCOUNTS.ACCID = CUSTACC.ACCID
  AND CUSTOMER.CNAME = 'Mary'
```

The hard clauses constructed using Reduction 3.1.1 are same as the ones from Example 4.2.1. The rest of the clauses are as follows:

β-clauses: $(\neg x_2 \vee \neg x_7, 1000)$, $(\neg x_3 \vee \neg x_7, 1000)$, $(\neg x_2 \vee \neg x_8, 1200)$, $(\neg x_3 \vee \neg x_8, 1200)$, $(y_1, 100)$, $(y_2, 100)$.

γ-clauses: $(\neg x_2 \vee \neg x_9 \vee y_1)$, $(\neg y_1 \vee x_2)$, $(\neg y_1 \vee x_9)$, $(\neg x_3 \vee \neg x_9 \vee y_2)$, $(\neg y_2 \vee x_3)$, $(\neg y_2 \vee x_9)$.

The witnesses $\{f_2, f_9, f_{13}\}$ *and* $\{f_3, f_9, f_{13}\}$ *belong to* $\mathcal{W}_N$ *because the account balance is -100 in both cases, so we introduce new variables* y_1 *and* y_2 *respectively, and construct hard* γ*-clauses as described above. The* β*-clauses corresponding to these witnesses are* $(y_1, 100)$ *and* $(y_2, 100)$*. We omit* x_{13} *in all of these clauses since* `CUSTACC` *does not violate* Σ*. Note that* $Q(\mathcal{I}_p) = 4400$*. An assignment in which* $x_8 = 0$ *and* $x_9 = 1$ *is a maximum satisfying assignment to the Partial MaxSAT instance* ϕ *constructed. The sum of satisfied soft clauses by this assignment is 3500 since it satisfies two clauses with weights 1200 each, one with weight 1000 and one with weight 100. Thus, by Proposition 3.1.1, we have that* $Q(\mathcal{J}) = 4400 - 3500 = 900$ *where* $\mathcal{J}$ *is a repair corresponding to the assignment in consideration. Similarly, setting* $x_8 = 1$ *and* $x_9 = 0$ *yields a minimum satisfying assignment in which the sum of satisfied soft clauses is 2200 since it satisfies one clause with weight 1200 and one with weight 1000. Thus, we have that the range consistent answers* $\textsc{Cons}(Q, \mathcal{I}) = [900, 2200]$.

4.2.2 Handling the `DISTINCT` Keyword

Let $Q :=$ `SELECT` f `FROM` $T(U, A)$ be an aggregation query, where f is either `COUNT(DISTINCT` A`)` or `SUM(DISTINCT` A`)`. Solving a Partial MaxSAT or a Weighted Partial MaxSAT instance constructed using Reduction 3.1.1 may yield incorrect *glb* and *lub* answers to Q, if the database contains multiple witnesses with the same value for attribute A. For example, consider a query

$Q :=$ `SELECT COUNT(DISTINCT ACCOUNTS.TYPE) FROM ACCOUNTS`.

The correct *glb* and *lub*-answers in $\textsc{Cons}(Q, \mathcal{I})$ are both 2, but the solutions to the

Partial MaxSAT and Partial MinSAT instances constructed using Reduction 3.1.1 yield both answers as 4. The reason behind this is that the soft clauses $\neg x_6$ and $\neg x_7$ both correspond to the account type Checking, and similarly $\neg x_8$, $\neg x_9$, and $\neg x_{10}$ all correspond to the account type Saving. The hard clauses in the formula ensure that x_6, x_7, x_{10}, and one of x_8 and x_9 are true, thus counting both Checking and Saving account types exactly twice in every satisfying assignment to the formula. This can be handled by modifying the β-clauses in Reduction 3.1.1 as follows.

Let $\mathcal{A}$ denote a set of distinct answers to the query $q^*(A) := \exists U\ T(U, A)$. For each answer $b \in \mathcal{A}$, let $\mathcal{W}^b$ denote a subset of $\mathcal{W}$ such that for every minimal witness $W \in \mathcal{W}^b$, we have that $q^*(W) = b$. The idea is to use auxiliary variables to construct one soft clause for every distinct answer $b \in \mathcal{A}$, such that it is true if and only if no witness in $\mathcal{W}^b$ is present in a repair corresponding to the satisfying assignment. First, for every witness $W_j^b \in \mathcal{W}^b$, we introduce an auxiliary variable z_j^b that is true if and only if W_j^b is not present in the repair. Then, we introduce an auxiliary variable v^b which is true if and only if all z^b-variables are true. These constraints are encoded in the set H^b returned by Algorithm 2, and are forced by making clauses in H^b hard. For every answer $b \in \mathcal{A}$, Algorithm 2 also returns one β^b-clause, which serves the same purpose as the β-clauses in Reduction 3.1.1. Now, a Partial MaxSAT or a Weighted Partial MaxSAT instance can be constructed by taking in conjunction all α-clauses from the key-equal groups, the hard γ-clauses if any, the hard clauses from all H^b-sets, and all soft β^b-clauses. With this, it is easy to see that a maximum (*or minimum*) satisfying assignment to Partial MaxSAT or Weighted Partial MaxSAT instance give us

the *glb*-answer (*or lub-answer*) in $\textsc{Cons}(Q)$. This is illustrated in Example 4.2.3.

Algorithm 2 Handling `DISTINCT`

procedure HANDLEDISTINCT($\mathcal{W}^b$)
  let $H^b = \emptyset$ //Empty set of clauses
  for $W_j^b \in \mathcal{W}^b$ **do**
    $H^b = H^b \bigcup \left\{ \left(\neg z_j^b \vee \left(\bigvee_{f_i \in W_j^b} \neg x_i \right) \right) \right\}$
    for $f_i \in W_j^b$ **do**
      $H^b = H^b \bigcup \{(z_j^b \vee x_i)\}$
  $H^b = H^b \bigcup \left\{ \left(\neg v^b \vee \left(\bigvee_{W_j^b \in \mathcal{W}^b} \neg z_j^b \right) \right) \right\}$
  for $W_j^b \in \mathcal{W}^b$ **do**
    $H^b = H^b \bigcup \{(\neg v^b \vee z_j^b)\}$
  let $\beta^b = (v^b, 1)$
  if (f is `SUM(DISTINCT` A`)`) **then**
    $\beta^b = (v^b, ||b||)$
    if $b < 0$ **then** $\beta^b = (\neg v^b, ||b||)$
  return H^b, β^b

Example 4.2.3. *Consider the following aggregation query Q that counts the number of distinct account types present in the* `ACCOUNTS` *relation:*

```
SELECT COUNT(DISTINCT ACCOUNTS.TYPE) FROM ACCOUNTS.
```

We have that $\mathcal{A} = \{$'Checking', 'Saving'$\}$. Let us denote these two answers by a_1 and a_2 respectively. Since every witness to the query consists of a single fact, every y^a-variable is equivalent to a single literal, for example, $y_1^{a_1} \leftrightarrow \neg x_6$ and $y_2^{a_1} \leftrightarrow \neg x_7$. As a result,

it is unnecessary to introduce any y^a-variables at all. Thus, we construct the following clauses from Reduction 3.1.1 and Algorithm 2:

α-clauses: $x_6, x_7, (x_8 \vee x_9), x_{10}$; *$\alpha^{mn}$-clauses:* $(\neg x_8 \vee \neg x_9)$;

$H^{a_1} : (x_6 \vee x_7 \vee v^{a_1}), (\neg v^{a_1} \vee \neg x_6), (\neg v^{a_1} \vee \neg x_7)$;

$H^{a_2} : (x_8 \vee x_9 \vee x_{10} \vee v^{a_2}), (\neg v^{a_2} \vee \neg x_8), (\neg v^{a_2} \vee \neg x_9), (\neg v^{a_2} \vee \neg x_{10})$;

β-clauses: $(v^{a_1}, 1), (v^{a_2}, 1)$

The maximum and minimum satisfying assignments to the Partial MaxSAT and Partial MinSAT instances constructed using these clauses falsify both β-clauses, since $\textsc{Cons}(Q, \mathcal{I})$ *w.r.t. range semantics is* $[2, 2]$.

4.3 Answering Queries with MIN/MAX without Grouping

Let $\mathcal{R}$ be a database schema with one key constraint per relation, and let

$$Q := \texttt{SELECT } f \texttt{ FROM } T(U, A),$$

where f is one of the operators `MIN(`A`)` and `MAX(`A`)`, and $T(U, A)$ is a relation expressed as a union of conjunctive queries over $\mathcal{R}$. The semantics of the range consistent answers to aggregation queries with `MIN` and `MAX` operators are similar to aggregation queries with `SUM` or `COUNT` operators, but here we need to address one additional special case. We illustrate this special case using Example 4.3.1.

Example 4.3.1. *Consider the database instance $\mathcal{I}$ from Table 4.1 and two aggregation queries Q_1 and Q_2 as follows.*

Q_1 :=*SELECT SUM(ACCOUNTS.BAL) FROM ACCOUNTS WHERE ACCOUNTS.CITY = 'SF'*

Q_2 :=*SELECT MIN(ACCOUNTS.BAL) FROM ACCOUNTS WHERE ACCOUNTS.CITY = 'SF'*

It is clear that the range consistent answers to $\textsc{Cons}(Q_1, \mathcal{I}) = [-100, 0]$. *The lub-answer of 0 in* $\textsc{Cons}(Q_1, \mathcal{I})$ *comes from a repair on which there is no account in the city of SF, and therefore the* `SUM` *function returns 0. For* Q_2*, however, the range consistent answers are unclear because the* `MIN` *function is not defined on an empty set.*

In such scenarios, various different semantics can be considered. One natural semantics is that if there exists a repair on which the underlying conjunctive query evaluates to an empty set, we could say that there is no consistent answer to the aggregation query. Another one could be to return the interval [*glb*, *lub*] of values that come from the repairs on which the underlying conjunctive query evaluates to a non-empty set of answers, and additionally return the information about the existence of the repair on which the underlying conjunctive query returns the empty set of answers. The reductions we give in this Section can be used regardless of which of the two above-mentioned semantics is chosen.

In what follows, we first show that the *glb*-answer to an aggregation query with the `MIN(`A`)` operator and the *lub*-answer to an aggregation query with the `MAX(`A`)` operator can be computed in polynomial time in the size of the original inconsistent database instance $\mathcal{I}$ (Proposition 4.3.1 and Corollary 4.3.1). Then, in Reduction 4.3.1, we describe a Weighted MaxSAT-based approach to computing the *lub*-answer for the scalar aggregation queries with the `MIN(`A`)` operator. We then point out a downside of Reduction 4.3.1 and discuss an alternative method based on iterative SAT solving. We do not explicitly give a reduction or discuss alternative approaches to obtain the *glb*-

answer to scalar aggregation queries with the `MAX(`A`)` operator since it is straightforward that the *lub*-answer for `MIN(`A`)` is a dual of the *glb*-answer for `MAX(`A`)` in the sense that computing the *lub*-answer for the `MIN(`A`)` operator yields the same result as negating all values of the aggregation attribute A in the database instance and then computing the *glb*-answer for the `MAX(`A`)` operator.

Proposition 4.3.1. *Let $\mathcal{R}$ be a database schema, $\mathcal{I}$ an $\mathcal{R}$-instance, and Q the aggregation query* `SELECT MIN(`A`) FROM` $T(U, A)$*. Let q_1 be the union $q_1(A) := \exists U\ T(U, A)$ of conjunctive queries and W_{glb} be the witness to q_1 on $\mathcal{I}$ such that no two facts in W_{glb} are key-equal, and there is no W' such that $q(W') < q(W_{glb})$ and no two facts in W' are key-equal. Then, $q_1(W_{glb})$ is the glb-answer in* $\textsc{Cons}(Q, \mathcal{I})$.

Proof. For every witness W' to q_1 on $\mathcal{I}$ such that $q_1(W') < q_1(W_{glb})$, we have that no repair of $\mathcal{I}$ contains W' because W' contains at least two key-equal facts. Moreover, since no two facts in W_{glb} are key-equal, there exists a repair $\mathcal{J}$ of $\mathcal{I}$ such that $W_{glb} \in \mathcal{J}$. Therefore, $q_1(W_{glb})$ must be the smallest possible answer to Q on $\mathcal{I}$, i.e., the *glb*-answer in $\textsc{Cons}(Q, \mathcal{I})$. Since the number of witnesses to q_1 is polynomial in the size of $\mathcal{I}$, a desired witness W_{glb} can be obtained efficiently from the result of evaluating q_1 on $\mathcal{I}$. □

Corollary 4.3.1. *Let $\mathcal{R}$ be a database schema, $\mathcal{I}$ an $\mathcal{R}$-instance, and Q the aggregation query* `SELECT MAX(`A`) FROM` $T(U, A)$*. Let q_1 be the union $q_1(A) := \exists U\ T(U, A)$ of conjunctive queries and W_{lub} be the witness to q_1 on $\mathcal{I}$ such that no two facts in W_{lub} are key-equal, and there is no W' such that $q(W') > q(W_{lub})$ and no two facts in W' are key-equal. Then, $q_1(W_{lub})$ is the lub-answer in* $\textsc{Cons}(Q, \mathcal{I})$.

Proof. It follows from a dual argument to that of Proposition 4.3.1. □

Now, we describe two SAT-based approaches to compute the least upper bound in the range consistent answers to a scalar aggregation query with `MIN(`A`)` operator.

Reduction 4.3.1. *Given an $\mathcal{R}$-instance $\mathcal{I}$, construct a Weighted Partial MaxSAT instance ϕ as follows. For each fact f_i of $\mathcal{I}$, introduce a boolean variable x_i. Let $\mathcal{G}$ be the set of key-equal groups of facts of $\mathcal{I}$, and $\mathcal{W} = \{W_1, \cdots, W_m\}$ denote the set of minimal witnesses to a conjunctive query q on $\mathcal{I}$, where $q(w) := \exists \vec{u}\ T(\vec{u}, w)$. Assume that the set $\mathcal{W}$ is sorted in descending order of the answers, i.e., for $1 \leq i < m$, we have that $q(W_i) \geq q(W_{i+1})$.*

- *For each $G_j \in \mathcal{G}$, construct a hard clause $\alpha_j = \bigvee_{f_i \in G_j} x_i$.*
- *For each $G_j \in \mathcal{G}$, for each pair (f_m, f_n) of facts such that $f_m \in G_j$, $f_n \in G_j$, and $m \neq n$, construct a hard clause $\alpha_j^{mn} = \neg x_m \vee \neg x_n$.*
- *Let $w_c = 1$, $sum = 0$. For $j = 1$ to m, do the following.*
  - *Construct a clause $\beta_j = \bigvee_{f_i \in W_j} \neg x_i$, and let its weight be w_c.*
  - *$sum = sum + w_c$.*
  - *If ($j > 1$ and $q(W_j) < q(W_{j-1})$) then $w_c = sum + 1$.*
- *Construct the Weighted Partial MaxSAT instance $\phi = \left(\bigwedge_{j=1}^{|\mathcal{G}|} \alpha_j\right) \wedge \left(\bigwedge_{j=1}^{|\mathcal{W}|} \beta_j\right)$.*

Proposition 4.3.2. *Let s be a maximum satisfying assignment to the Weighted Partial MaxSAT instance ϕ constructed using Reduction 4.3.1. Let β_j be a clause such that $s(\beta_j) = false$, and there is no $j' < j$ such that $s(\beta_{j'}) = false$. Then, $q(W_j)$ is the lub-answer in* $\textsc{Cons}(Q, \mathcal{I})$.

Proof. To see why Proposition 4.3.2 is true, we first make the following observation. If l is the *lub*-answer in $\textsc{Cons}(Q, \mathcal{I})$, then i) there exists a repair on which Q evaluates to l, i.e., a repair that does not contain any witness W such that $q(W) < l$ holds, and ii) there exists no repair on which Q evaluates to a value strictly greater than l.

Now, the weight distribution among the soft clauses of ϕ makes sure the following condition holds for any two potential answers l_1 and l_2. If $l_1 < l_2$, then the sum of all clauses corresponding to the witnesses to l_2 and to all potential answers greater than l_2 is smaller than the weight of each clause corresponding to a witness of l_1. Hence, every solution to ϕ will first try to satisfy the β-clauses corresponding to the witnesses to l_1, even if it is at an expense of falsifying all clauses corresponding to the witnesses to the potential answers strictly greater than l_1. Therefore, from the preceding observations, a repair extracted (*in a similar manner to that in Reduction 3.1.1*) from a maximum satisfying assignment to ϕ must contain a witness to the *lub*-answer in $\textsc{Cons}(Q)$. Clearly, this answer can be obtained from a falsified clause with the highest weight, which is precisely what Proposition 4.3.3 states. □

Note that Reduction 4.3.1 is a polynomial-size reduction because the number of clauses in ϕ is bounded by $|\mathcal{W}|$, the size of each clause is bounded by the number of atoms in the query, and the numeric value w_c of the weight of a clause can be represented in $log_2(w_c)$ bits. With this, it seems possible to use Weighted MaxSAT solvers to compute the range consistent answers to aggregation queries with `MIN(`A`)` and `MAX(`A`)` operators. However, Reduction 4.3.1 has one major downside and that is

the numeric values of the weights of the clauses, i.e, the values taken by the variable w_c, increase exponentially in the size of the database. For example, if the potential answers from the witnesses to the conjunctive query q are spanned across a hundred distinct values, then the largest weight that needs to be assigned to a clause in ϕ will be $O(2^{100})$, which is impractical from the perspective of the solvers because the solvers need the weights to be entered in the decimal system in the DIMACS file format.

It is not hard to see that this exponential increase in the weights of the clauses occurs because of expressing the problem at hand as a Weighted Partial MaxSAT instance. Observe that in Reduction 4.3.1, the numeric values of the weights do not matter, as long as a clause corresponding to the witness W_j is *strictly harder* than all clauses corresponding to the witnesses on which the query q evaluates to an answer that is strictly smaller than $q(W_j)$. Therefore, instead of Weighted Partial MaxSAT, one can view this problem as *hierarchical* Partial MaxSAT, where, instead of having just two levels of hardness (*hard* and *soft*) like in Partial MaxSAT, one can imagine a hierarchy of the levels of hardness of clauses. Here, the semantics is that each clause at the hardness level i is weighted more than all clauses at the hardness levels less than i combined. Thus, while representing the instance in a solver-readable file, one could simply write a clause along with its hardness level. To our knowledge, there has not been any work on solving hierarchical Partial MaxSAT instances, and via the range consistent answers to aggregation queries, we provide an interesting application to a hierarchical Partial MaxSAT solver if one is built in the future.

To compute the range consistent answers to aggregation queries with `MIN(`A`)`

and `MAX(`A`)` operators with available solvers, we opt for an iterative SAT solving approach. The idea is similar to solving a hierarchical Partial MaxSAT instance described earlier, but we solve for one hardness level at each iteration. In what follows, we formalize the construction of the SAT instance for the first iteration (Construction 4.3.1) and give Algorithm 3 that computes the *lub*-answer in $\textsc{Cons}(Q, \mathcal{I})$.

Construction 4.3.1. *Given an $\mathcal{R}$-instance $\mathcal{I}$, construct a CNF formula ϕ as follows. For each fact f_i of $\mathcal{I}$, introduce a boolean variable x_i. Let $\mathcal{G}$ be the set of key-equal groups of facts of $\mathcal{I}$, and $\mathcal{W} = \{W_1, \cdots, W_m\}$ denote the set of minimal witnesses to a conjunctive query q on $\mathcal{I}$, where $q(w) := \exists \vec{u}\, T(\vec{u}, w)$. Assume that the set $\mathcal{W}$ is sorted in descending order of the answers, i.e., for $1 \leq i < m$, we have that $q(W_i) \geq q(W_{i+1})$.*

- *For each $G_j \in \mathcal{G}$, construct a clause $\alpha_j = \bigvee\limits_{f_i \in G_j} x_i$.*
- *Construct a CNF formula $\phi = \bigwedge\limits_{j=1}^{|\mathcal{G}|} \alpha_j$.*

Proposition 4.3.3. *Let $Q :=$ `SELECT MIN(A) FROM` $T(U, A)$ be an aggregation query, and $\mathcal{I}$ be a database instance. Algorithm 3 returns the lub-answer in $\textsc{Cons}(Q, \mathcal{I})$.*

Proof. The α-clauses of ϕ make sure that a repair $\mathcal{J}$ of $\mathcal{I}$ can be constructed from every assignment s of ϕ that satisfies the α-clauses, by arbitrarily choosing exactly one fact f_i from each key-equal group of $\mathcal{I}$ such that $s(x_i) = 1$. Let $\mathcal{A} = \{A_1, \cdots, A_{lub}, \cdots, A_{|\mathcal{A}|}\}$ denote the set of distinct answers to a conjunctive query $q(w) := \exists \vec{u}\, T(\vec{u}, w)$ on $\mathcal{I}$, where A_{lub} is the *lub*-answer to Q on $\mathcal{I}$. For a witness W to q, a clause $(\bigvee\limits_{f_i \in W} \neg x_i)$ is satisfied by an assignment s if and only if W is not present in any repair constructed from s. At iteration j of the while-loop, if the formula ϕ is checked for satisfiability (line 8 of

Algorithm 3 Computing the *lub*-answer in $\textsc{Cons}(Q, \mathcal{I})$ for `MIN` via Iterative SAT

1: **procedure** LubAnswer-IterativeSAT($\phi, \mathcal{W}$)
2: **let** $v = q(W_1), j = 1$
3: **while** $j \leq |\mathcal{W}|$ **do**
4: **if** $v = q(W_j)$ **then**
5: **let** $\phi = \phi \wedge \left(\bigvee_{f_i \in W_j} \neg x_i \right)$
6: **let** $j = j + 1$
7: **else**
8: **if** UNSAT(ϕ) **then**
9: **return** $q(W_{j-1})$
10: **else**
11: **let** $v = q(W_j)$
12: **return** $q(W_{|\mathcal{W}|})$

Algorithm 3), the formula contains the α-clauses corresponding to the key-equal groups of $\mathcal{I}$ in conjunction to all clauses corresponding to the minimal witnesses to q on which the q evaluates to an answer strictly smaller than $q(W_j)$. At this point, if the formula ϕ is satisfiable, then there exists a repair $\mathcal{J}$ of $\mathcal{I}$ such that $q(W_{j-1}) \notin q(\mathcal{J})$, and also for all potential answers $A_i \leq q(W_{j-1})$, we have that $A_i \notin q(\mathcal{J})$. On the other hand, if the formula is unsatisfiable, then there exists no repair $\mathcal{J}$ of $\mathcal{I}$ such that $q(W_{j-1}) \notin q(\mathcal{J})$ and $A_i \notin q(\mathcal{J})$ for all $A_i \leq q(W_{j-1})$. Since the clauses are added in the ascending order of the answers, we have that ϕ satisfiable at iteration j if and only if $q(W_{j-1}) < A_{lub}$. Therefore, if ϕ becomes unsatisfiable for the first time at iteration j, it must be the the case that $q(W_{j-1})$ is the *lub*-answer in $\textsc{Cons}(Q, \mathcal{I})$. □

Why not a Binary Search?

In essence, Algorithm 3 works like a linear search on a sorted array. It may sound appealing to perform the binary search instead of the linear search for obvious reasons. Clearly, the SAT solver will only have to solve $O(log_2\ |\mathcal{A}|)$ instances of SAT instead of $O(|\mathcal{A}|)$ instances of SAT. The problem with this approach is the following. Observe that, on an average, half of the SAT instances that the solver needs to solve in the binary search approach will be unsatisfiable. In the linear search approach, however, all but the last instance given to the solver are satisfiable. Typically, the proofs of unsatisfiability produced by the SAT solvers are significantly large compared to the proofs of satisfiability as the unsatisfiability of an instance needs to be proven with a refutation tree that can be exponential in size of the formula, while just one satisfying assignment is enough to prove the satisfiability of an instance. As a result, SAT solvers typically take considerably long amounts of time to solve unsatisfiable instances but they are very quick on most real-world satisfiable instances. Therefore, at a practical level, the linear search often works better than the binary search.

4.4 Range Consistent Answers Beyond Key Constraints

If Σ is a fixed finite set of denial constraints and Q is an aggregation query without grouping, then the following problem is in coNP: given a database instance $\mathcal{I}$ and a number t, is t the *lub*-answer *(or the glb-answer)* in $\textsc{Cons}(Q, \mathcal{I})$ w.r.t. Σ? This is so because to check that t is not the *lub*-answer *(or the glb-answer)*, we guess a repairs $\mathcal{J}$

of $\mathcal{I}$ and verify that $t > Q(\mathcal{J})$ *(or* $t > Q(\mathcal{J})$*)*. In all preceding reductions, the α-clauses capture the inconsistency in the database arisen due to the key violations to enforce every satisfying assignment to uniquely correspond to a repair of the initial inconsistent database instance. Importantly, the α-clauses are independent of the input query. In what follows, we provide a way to construct clauses to capture the inconsistency arising due to the violations of denial constraints. Thus, replacing the α-clauses in the reductions from Sections 4.2 and 4.3 by the ones provided below allows us to compute consistent answers over databases with a fixed finite set of arbitrary denial constraints. The intuition behind Reduction 4.4.1 is similar to the one in Reduction 3.2.1, so recall the notions of *minimal violations* and *near-violations* to the set of denial constraints (Definitions 3.2.1 and 3.2.2). Let $\mathcal{R}$ be a database schema and Σ be a fixed finite set of arbitrary denial constraints on $\mathcal{R}$. Let Q be an aggregation query without grouping on $\mathcal{R}$ and let $\mathcal{I}$ be an $\mathcal{R}$-instance.

Reduction 4.4.1. *Given an $\mathcal{R}$-instance $\mathcal{I}$, compute the sets:*

1. *$\mathcal{V}$: the set of minimal violations to Σ on $\mathcal{I}$.*
2. *$\mathcal{N}^i$: the set of near-violations to Σ, on $\mathcal{I}$, w.r.t. each fact $f_i \in \mathcal{I}$.*

For each fact f_i of $\mathcal{I}$, introduce a boolean variable x_i, $1 \leq i \leq n$. For the auxiliary fact f_{true}, introduce a constant $x_{true} = true$, and for each $N_j^i \in \mathcal{N}^i$, introduce a boolean variable p_j^i.

1. *For each $V_j \in \mathcal{V}$, construct a clause $\alpha_j = \bigvee\limits_{f_i \in V_j} \neg x_i$.*
2. *For each $f_i \in \mathcal{I}$, construct a clause $\gamma_i = x_i \vee \Big(\bigvee\limits_{N_j^i \in \mathcal{N}^i} p_j^i \Big)$.*

3. For each variable p_j^i, construct an expression $\theta_j^i = p_j^i \leftrightarrow \left(\bigwedge_{f_d \in N_j^i} x_d \right)$.

4. Construct the following boolean formula ϕ:

$$\phi = \left(\bigwedge_{i=1}^{|\mathcal{V}|} \alpha_i \right) \wedge \left(\bigwedge_{i=1}^{|\mathcal{I}|} \left(\left(\bigwedge_{j=1}^{|\mathcal{N}^i|} \theta_j^i \right) \wedge \gamma_i \right) \right)$$

Proposition 4.4.1. *The boolean formula ϕ constructed using Reduction 3.2.1 can be transformed to an equivalent CNF-formula ϕ whose size is polynomial in the size of $\mathcal{I}$. The satisfying assignments to ϕ and the repairs of $\mathcal{I}$ w.r.t. Σ are in one-to-one correspondence.*

Proof. Let n be the number of facts of $\mathcal{I}$. Let d_1 be the smallest number such that there exists no denial constraint in Σ whose number of database atoms is bigger than d_1. Also, let d_2 be the smallest number such that there exists no conjunctive query in Q whose number of database atoms is bigger than d_2. Since Σ and Q are not part of the input to CONS(Q), the quantities d_1 and d_2 are fixed constants. We also have that $|\mathcal{V}| \leq n^{d_1}$, $|\mathcal{N}^i| \leq n^{d_1}$ for $1 \leq i \leq n$, $|\mathcal{A}| \leq n^{d_2}$, and $|\mathcal{W}^l| \leq n^{d_2}$ for $1 \leq l \leq |\mathcal{A}|$. The number of x-, y-, and p-variables in ϕ' is therefore bounded by n, n^{d_1+1}, and n^{d_2}, respectively. The formula ϕ' contains as many α-clauses as $|\mathcal{V}|$, and none of the α-clause's length exceeds n. Similarly, there are at most n^{d_2} β-clauses, and none of their lengths exceeds $d_2 + 1$. The number of γ-clauses is precisely n, and each γ-clause is at most $n^{d_1+1} + 1$ literals long. There are as many θ-expressions as there are y-variables. Every θ-expression is of the form $y \leftrightarrow (x_1 \wedge ... \wedge x_d)$, where d is a constant obtained from the number of facts in the corresponding near-violation. Each θ-expression can be equivalently written in a constant number of CNF-clauses as $((\neg y \vee x_1) \wedge ... \wedge (\neg y \vee x_d)) \wedge (\neg x_1 \vee ... \vee \neg x_d \vee y)$, in

which the length each clause is constant. Thus, one can transform ϕ' into an equivalent CNF-formula ϕ with size polynomial in the size of $\mathcal{I}$.

For the second part of Proposition 4.4.1, consider a satisfying assignment s to ϕ and construct a database instance $\mathcal{J}$ such that $f_i \in \mathcal{J}$ if and only if $s(x_i) = 1$. The α-clauses assert that no minimal violation to Σ is present in $\mathcal{J}$, i.e., $\mathcal{J}$ is a consistent subset of $\mathcal{I}$. The γ-clauses and the θ-expressions encode the condition that, for every fact $f \in \mathcal{I}$, either $f \in \mathcal{J}$ or at least one near-violation w.r.t. Σ and f is in $\mathcal{J}$, making sure that $\mathcal{J}$ is indeed a repair of $\mathcal{I}$. In the other direction, one can construct a satisfying assignment s to ϕ from a repair $\mathcal{J}$ of $\mathcal{I}$ by setting $s(x_i) = 1$ if and only if $f_i \in \mathcal{J}$. □

4.5 Answering Aggregation Queries with Grouping

Let Q be the aggregation query

$$Q := \texttt{SELECT } Z, f(A) \texttt{ FROM } T(U, Z, A) \texttt{ GROUP BY } Z,$$

where $f(A)$ is one of $\texttt{COUNT}(*)$, $\texttt{COUNT}(A)$, $\texttt{SUM}(A)$, $\texttt{MIN}(A)$, or $\texttt{MAX}(A)$, and $T(U, A)$ is a relation expressed by a union of conjunctive queries on $\mathcal{R}$. For aggregation queries with grouping, it does not seem feasible to reduce $\textsc{Cons}(Q, \mathcal{I})$ to a single Partial MaxSAT or a Weighted Partial MaxSAT instance because for each group of consistent answers, the *glb*-answer and the *lub*-answer may realize in different repairs of the inconsistent database instance $\mathcal{I}$. To illustrate this, consider the database from Table 4.1 and a query $Q :=$ `SELECT COUNT(*) FROM CUSTOMER GROUP BY CUSTOMER.CITY`. Notice that, the *glb*-answers (LA, 2) and (SF, 1) in $\textsc{Cons}(Q, \mathcal{I})$ come from two different repairs of relation

CUST, namely, $\{f_1, f_3, f_4, f_5\}$ and $\{f_1, f_2, f_4, f_5\}$ respectively. However, the reductions from the preceding sections of this chapter can be used to compute the bounds to each consistent group of answers independently. For a given aggregation query Q with grouping, we first compute the consistent answers to the conjunctive query $q(Z) := \exists U, A\ T(U, Z, A)$. Then, for each answer b in $\textsc{Cons}(q, \mathcal{I})$, we compute the *glb*-answer and the *lub*-answer to the scalar query $Q' :=$ `SELECT` $f(A)$ `FROM` $T(U, Z, A) \wedge (Z = b)$ via Partial MaxSAT or Weighted Partial MaxSAT solving using the reductions from earlier sections, as shown in Algorithm 4.

Algorithm 4 The Range Consistent Answers to Aggregation Queries With Grouping

Let $\mathcal{I}$ be an inconsistent database instance, and Q be an aggregation query of the form $Q :=$ `SELECT` Z, f `FROM` $T(U, Z, A)$ `GROUP BY` Z.

1: **procedure** ConsAggGrouping(Q)
2: **let** $Ans = \emptyset$
3: **let** $q(Z) := \exists U, A\ T(U, Z, A)$
4: **let** $\mathcal{A}_c = \textsc{Cons}(q, \mathcal{I})$
5: **for** $b \in \mathcal{A}_c$ **do**
6: **let** $Q' :=$ `SELECT` $f(A)$ `FROM` $T(U, Z, A) \wedge (Z = b)$
7: **let** $[glb_A, lub_A] = \textsc{Cons}(Q', \mathcal{I})$
8: $A_{ans} = A_{ans} \cup \{(b, [glb_A, lub_A])\}$
9: **return** A_{ans}

Chapter 5

CAvSAT: A SAT-based System for Consistent Query Answering

In this chapter, we present CAvSAT – "Consistent Answers via SAT Solving", a comprehensive and scalable SAT-based system capable of computing the consistent answers or the range consistent answers to broad classes of practical queries over database schemata that are inconsistent w.r.t. a set of arbitrary denial constraints. We demonstrated CAvSAT at the 2021 International Conference on Management of Data (ACM SIGMOD) [35]. In Section 5.1, we describe the modular architecture of the CAvSAT system and the detailed working of each of the modules, while in Section 5.2 we give the details of CAvSAT's implementation including its graphical user interface.

5.1 CAvSAT System Architecture

The architecture of CAvSAT shown in Figure 5.1 leverages the extensive research that has been done on the complexity of computing the consistent answers to a variety of classes of queries and integrity constraints. To the core of CAvSAT lie, the two SAT-solving modules that implement the reductions we described in Chapter 3 and Chapter 4. In what follows, we describe the working of each of the modules.

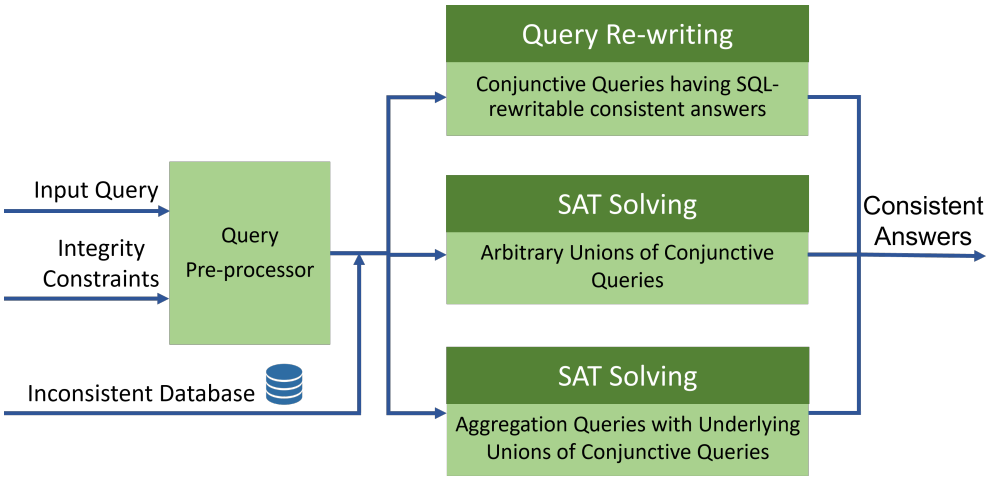


Figure 5.1: The modular architecture of CAvSAT

5.1.1 The Query Pre-processor Module

The Query Pre-processor module takes the query and the set of integrity constraints on the database schema as inputs and attempts to determine the complexity of computing the consistent answers to the query. If the database schema has only primary key constraints, then this module first checks whether the input query belongs to the class C_{forest} or the class $C_{aggforest}$ [40]. If it does, the query is forwarded to the

Query Re-writing module to derive the SQL query (i.e., the SQL-rewriting based on the algorithms from ConQuer [40]) that will compute the consistent answers or the range consistent answers to the original input query. If the input query is a conjunctive query without self-joins over a database schema with only primary key constraints, the Query Pre-processor module additionally constructs the *attack graph* [59] whose cyclicity properties determine whether or not the consistent answers to the query are SQL-rewritable. If the consistent answers are SQL-rewritable, the query is forwarded to the Query Re-writing module for deriving the SQL query to compute its consistent answers using the rewriting algorithms of Koutris and Wijsen [59]. If the consistent answers are not SQL-rewritable or their complexity is not known in the pre-processing stage due to the presence of self-joins, the presence of aggregation operators, or the presence of integrity constraints on the schema that are beyond primary keys, we still know that the consistent answers are in coNP, so the query is forwarded to one of the SAT-solving modules depending on whether or not the query has aggregate operators.

5.1.2 The Query Re-writing Module

This module implements the query-rewriting algorithms in [40, 41, 59]. For queries with SQL-rewritable consistent answers, this module produces the consistent rewriting and evaluates it on the inconsistent database instance directly to obtain the consistent answers to the original input query. The consistent rewritings based on the algorithms from ConQuer are directly constructed in SQL, while the rewritings based on the algorithms of Koutris and Wijsen are first constructed in tuple-relational calculus

and then translated in SQL to evaluate within a standard database engine.

5.1.3 The SAT Solving Modules

These two modules reduce a given CQA instance to an instance of Boolean satisfiability (SAT) or one of its variants via polynomial-time reductions described in Chapter 3 and Chapter 4. The SAT instances are stored in the DIMACS file format, which is one of the standard ways to represent CNF formulas. After constructing an instance of SAT or one of its variants, these modules invoke a suitable state-of-the-art SAT solver to compute an optimal solution. For example, for solving Boolean SAT instances, the SAT-solving module invokes the Glucose solver [19] or the CaDiCaL solver [1], while the MaxHS solver [31] is invoked if a Partial MaxSAT or a Weighted Partial MaxSAT instance needs to be solved. The solver writes an optimal assignment (if found) to a file that is read by CAvSAT to decode the information about the consistent answers to the range consistent answers to the query. In the case of iterative SAT approaches, CAvSAT reads a solution output by a solver, modifies the SAT instance in the DIMACS file as needed (typically by appending new clauses or modifying the weights of existing clauses), and calls the solver again for the next iteration. The SAT solving modules and the SAT solvers are loosely coupled in the sense that CAvSAT uses the solver as a black box. Since the DIMACS file format is readable by most of the modern solvers, CAvSAT can easily accommodate a new more efficient solver if one is developed in the future. Table 5.1 summarizes the different methods, algorithms, and the variants of SAT used by the SAT solving modules of CAvSAT for a variety of

classes of the input queries.

Table 5.1: SAT-variants and the solvers used by CAvSAT for different classes of queries

Query Type	**SAT-variant / Method**	**Solvers**
Unions of boolean conjunctive queries	SAT	Glucose [19], CaDiCaL [1]
Unions of non-boolean conjunctive queries	Iterative algorithm with Partial MaxSAT	MaxHS [31]
Unions of conjunctive queries with aggregation (SUM/COUNT)	Weighted Partial MaxSAT for *glb*, Weighted Partial MinSAT along with Kügel's encoding for *lub*	MaxHS [31]
Unions of conjunctive queries with aggregation (MIN/MAX)	Linear search with iterative SAT	Glucose [19], CaDiCaL [1]
Unions of conjunctive queries with aggregation and grouping	Weighted Partial MaxSAT for *glb*, Weighted Partial MinSAT along with Kügel's encoding for *lub*, with iterative algorithm for groups	MaxHS [31]

5.2 CAvSAT System Implementation

Figure 5.2 depicts the implementation of CAvSAT at a high level. CAvSAT uses a standard database management system such as the Microsoft SQL Server, MySQL, or PostgreSQL to store the inconsistent database instance and the set of integrity constraints. Since we need the database server to store inconsistent data and handle inconsistencies at query time, the integrity constraints are not applied on the database instance within the server but are stored separately as a relation. The reductions from computing the consistent answers or the range consistent answers to variants of SAT are implemented in Java 8 and are hosted as a Java SpringBoot application on a web server. This application also has access to running powerful SAT solvers listed in

Table 5.1. The web-based graphical user interface of CAvSAT is implemented using the ReactJS front-end development framework that connects to the CAvSAT back-end via REST API calls. The CAvSAT source code is available at the GitHub repository `https://github.com/uccross/cavsat` via a BSD-style open-source license.

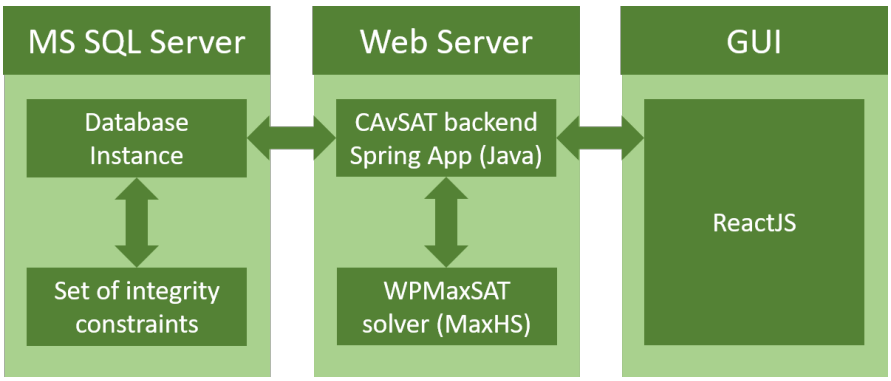


Figure 5.2: Implementation overview of CAvSAT

5.2.1 Performance Optimizations

In this section, we describe several optimizations we implemented in CAvSAT to speed up the computation of consistent answers and the range consistent answers to the queries. These implementation-level optimizations primarily apply to the SAT solving modules of CAvSAT and help either to speed up the construction of the SAT instances or to reduce the sizes of the SAT instances to be solved.

Data Pre-Processing

Data Pre-processing is the first stage in the overall workflow of CAvSAT, and it is performed offline because it is independent of the input query. In this stage,

CAvSAT first adds an attribute named *FactID* to every relation in the original database instance $\mathcal{I}$, and numbers the tuples of each relation in the database to uniquely identify them across all relations in $\mathcal{I}$ using their FactID. The FactID attributes are useful in computing and dealing with the minimal witnesses to the input query more efficiently.

Pre-computing Clauses Corresponding to Violations of Constraints

In the reductions from computing the consistent answers or the range consistent answers to variants of SAT, we construct clauses that encode the inconsistency present in the database. Clearly, the inconsistency in the data is independent of the query, so the clauses corresponding to it can be computed before the user inputs the query. Specifically, CAvSAT computes the key-equal groups of facts in case of primary key constraints and the minimal violations and the near-violations in case of broader classes of integrity constraints prior to the user entering the input query. This way, CAvSAT gets to pre-compute the α-clauses of Reduction 3.1.1, and the α-clauses, γ-clauses, and the θ-expressions of Reduction 3.2.1. Of course, if new tuple additions, tuple deletions, or any other modifications are performed on the database instance, CAvSAT may have to adjust the values of the FactID attributes and re-compute the clauses corresponding to the violations of the integrity constraints.

Computing Answers from the Consistent Part of the Database

For each relation in the original database instance, CAvSAT creates an auxiliary relation with the prefix *CAvSAT_Clean_* that stores only those tuples that do not

participate in the inconsistency (i.e., consistent part of the data). In most real-world databases, the percentage of inconsistency is low, typically less than 20%, and as a result, many answers to the query lie completely within the consistent part of the data. Clearly, these answers are consistent, because every fact of the consistent part of the database must also be present in every possible repair of the database. It is helpful to separately store the consistent part of the data because these answers can be computed directly by evaluating the input query on the consistent part of the data and the clauses corresponding to their minimal witnesses need not be encoded in the SAT instances. This reduces the size of the SAT instances, improving the overall performance.

Identifying Database Facts Relevant to Query Answers

We described in the reductions from Chapter 3 and Chapter 4 that we need one boolean variable for every fact of the inconsistent database instance, and we construct clauses corresponding to all key-equal groups of facts or all minimal violations of the integrity constraints on the database schema. In practice, this is often unnecessary as we can easily identify which facts are relevant and which are irrelevant w.r.t. the answers to the query and discard the irrelevant facts during the construction of the SAT instances. A trivial example of the facts of a relation R being irrelevant w.r.t. the answers to a query is when the relation symbol R does not appear in the input query. More precisely, CAvSAT does the following to decide which facts are relevant. Initially, all facts of the database instance $\mathcal{I}$ are marked irrelevant. If a fact f of $\mathcal{I}$ appears in at least one minimal witness to a query, then CAvSAT marks the fact f as relevant. For

primary key constraints, CAvSAT marks every fact g of $\mathcal{I}$ as relevant if g is key-equal to some other fact f that is already marked relevant. For broader classes of constraints, this notion extends naturally in the sense that CAvSAT marks every fact g as relevant if g appears in at least one minimal violation V to the integrity constraints on the database schema such that there is another fact f that participates in V and that f is already marked as relevant. With this, CAvSAT introduces variables corresponding to only relevant facts and constructs clauses using them, which helps in reducing the sizes of the SAT instances to be solved.

5.2.2 Workflow of CAvSAT

To better understand how are the aforementioned performance optimizations are applied in practice, we go over the workflow of CAvSAT with an example, starting from a user entering a query till CAvSAT computing its consistent answers. Recall the database schema $\mathcal{R}$ consisting of three relation symbols *Customer*, *Accounts*, and *CustAcc*, and the $\mathcal{R}$-instance $\mathcal{I}$ as shown in Table 5.2. Let Σ be the set of integrity constraints on $\mathcal{R}$, consisting of two primary keys, namely, CID $\rightarrow$ NAME, CITY on the *Customer* relation, and ACCID $\rightarrow$ TYPE, CITY, BAL on the *Accounts* relation. The key attributes of *Customer* and *Accounts* are underlined in Table 5.2.

We will consider the following union q of conjunctive queries that asks for the names of the customers who belong to Los Angeles or have bank accounts in the same city that they belong, i.e., $q := q_1 \cup q_2$, where

q_1 := `SELECT CUSTOMER.CNAME FROM CUSTOMER`

Table 5.2: An inconsistent database instance of bank account records

Customer			
Fact	CID	NAME	CITY
f_1	C1	John	LA
f_2	C2	Mary	LA
f_3	C2	Mary	SF
f_4	C3	Don	SF
f_5	C4	Jen	LA

Accounts				
Fact	ACCID	TYPE	CITY	BAL
f_6	A1	Check.	LA	900
f_7	A2	Check.	LA	1000
f_8	A3	Saving	SJ	1200
f_9	A3	Saving	SF	-100
f_{10}	A4	Saving	SJ	300

CustAcc		
Fact	CID	ACCID
f_{11}	C1	A1
f_{12}	C2	A2
f_{13}	C2	A3
f_{14}	C3	A4

```
    WHERE CUSTOMER.CITY = 'LA',
```

$q_2 :=$ `SELECT CUSTOMER.CNAME FROM CUSTOMER, ACCOUNTS, CUSTACC`

```
    WHERE CUSTOMER.CID = CUSTACCT.CID
        AND ACCOUNTS.ACCID = CUSTACCT.ACCID
        AND CUSTOMER.CITY = ACCOUNTS.CITY.
```

We will now describe the workflow of CAvSAT as how it computes the consistent answers to the query q on the database instance $\mathcal{I}$ from Table 5.2.

Prior to receiving the query q from the user, CAvSAT adds the attribute FactID to each relation of $\mathcal{I}$ and assigns unique identifiers $f_1, \cdots, f_{14}$ to the facts of $\mathcal{I}$ as part of the Data Pre-processing stage (see Table 5.2). Moreover, CAvSAT creates auxiliary relations to separately store the consistent part of the data (see Table 5.3) and also computes the key-equal groups of facts of $\mathcal{I}$. The key-equal groups of facts of $\mathcal{I}$ are used to construct the α-clauses (x_1), $(x_2 \vee x_3)$, (x_4), (x_5), (x_6), (x_7), $(x_8 \vee x_9)$, (x_{10}), (x_{11}), (x_{12}), (x_{13}), and (x_{14}).

Once the user inputs the query, the Query Pre-processor module attempts

Table 5.3: Auxiliary tables constructed by CAvSAT during Data Pre-processing

Clean_Customer			
Fact	CID	NAME	CITY
f_1	C1	John	LA
f_4	C3	Don	SF
f_5	C4	Jen	LA

Clean_Accounts				
Fact	ACCID	TYPE	CITY	BAL
f_6	A1	Check.	LA	900
f_7	A2	Check.	LA	1000
f_{10}	A4	Saving	SJ	300

Clean_CustAcc		
Fact	CID	ACCID
f_{11}	C1	A1
f_{12}	C2	A2
f_{13}	C2	A3
f_{14}	C3	A4

to determine the complexity of computing its consistent answers. Since there is no known classification of the unions of conjunctive queries based on the complexity of their consistent answers, we do not have a mechanism to determine the exact complexity of $\textsc{Cons}(q, \mathcal{I})$; we only know that it is in coNP. Therefore, the query q is forwarded to the SAT solving module tailored for queries without aggregation operators.

The next stage of CAvSAT's workflow is to compute the consistent answers from the consistent part of the database. CAvSAT stores these answers in a temporary relation named *CAvSAT_ans_from_cons*. Thus, the answers 'John' and 'Jen' in the table *CAvSAT_ans_from_cons*, since they come from the minimal witnesses $\{f_1\}$ and $\{f_5\}$ respectively, both of which are in the consistent part of the database. Notice that the answer 'John' also comes from a witness $\{f_1, f_7, f_{11}\}$ since it satisfies the conjunctive query q_2, but $\{f_1, f_7, f_{11}\}$ is not a minimal witness to q since $\{f_1\} \subset \{f_1, f_7, f_{11}\}$ and $\{f_1\}$ satisfies q.

The next step in the workflow of CAvSAT is to look for additional consistent answers in the inconsistent part of the database. For this, it first computes the minimal witnesses to q on $\mathcal{I}$ and identifies the facts that are relevant to the consistent

Table 5.4: Minimal witnesses to conjunctive queries q_1 and q_2 on $\mathcal{I}$

*CAvSAT_minimal_witnesses_*q_1		
CUSTOMER.FactID	CUSTOMER.CNAME	
f_1	John	×
f_2	Mary	
f_5	Jen	×

*CAvSAT_minimal_witnesses_*q_2				
CUSTOMER.FactID	ACCOUNTS.FactID	CUSTACC.FactID	CUSTOMER.CNAME	
f_1	f_6	f_{11}	John	×
f_2	f_7	f_{12}	Mary	×
f_3	f_9	f_{13}	Mary	

answers to q. For computing the minimal witnesses to a union of conjunctive queries, the FactID attribute of each relation is projected out in addition to the attributes corresponding to the free variables (i.e., the attributes already present in the `SELECT` clause) in each of the individual conjunctive queries participating in the union, they are evaluated on $\mathcal{I}$, and the results are stored in temporary relations with the prefix *CAvSAT_minimal_witnesses_*. Thus, CAvSAT computes the minimal witnesses to q_1 and q_2 as shown in Table 5.4. The minimal witnesses corresponding to the answers that are already known to be consistent are discarded (shown with a symbol '×' in Table 5.4). Also, the witness $\{f_2, f_7, f_{12}\}$ is discarded because even though it is a minimal witness to the conjunctive query q_2, it is not a minimal witness to the boolean union q['Mary'] of conjunctive queries, as $\{f_2\} \subset \{f_2, f_7, f_{12}\}$ and $\{f_2\}$ satisfies q['Mary'].

In the next step, CAvSAT identifies that f_2, f_3, f_8, f_9, and f_{13} are the only relevant facts of $\mathcal{I}$ w.r.t. the answers to q. Since q is a non-boolean union of conjunctive

queries, CAvSAT associates a p-variable p_1 to the only potential answer 'Mary' that is not already known to be consistent or inconsistent. Even though there are three minimal witnesses with the answer 'Mary', CAvSAT associates only one variable for it because for the unions of conjunctive queries, the current implementation of CAvSAT follows the set semantics on the answers to the query. It is easy to see that introducing a distinct p-variable for each occurrence of an answer will result in consistent answers w.r.t. the bag semantics (see Section 4.2.2 for a discussion on handling the `DISTINCT` keyword in aggregation queries for more details). Finally, a Partial MaxSAT instance ϕ is constructed as follows:

$$\phi = (x_2 \vee x_3) \wedge (x_8 \vee x_9) \wedge (x_{13}) \wedge (\neg x_2 \vee \neg p_1) \wedge (\neg x_3 \vee \neg x_9 \vee \neg x_{13} \vee \neg p_1)$$

Solving the instance ϕ using Algorithm 1 confirms that there exists a satisfying assignment $\hat{a}$ to ϕ such that $\hat{a}(p_1) = 1$, and hence by Proposition 3.1.2, 'Mary' is not a consistent answer to q. Finally, CAvSAT returns the only consistent answers 'John' and 'Jen' to the user.

5.2.3 Graphical User Interface

The CAvSAT system is implemented with both a command-line and a graphical user interface. The command-line user interface is a basic and functional way to interact with CAvSAT; we used it for running experiments discussed in Chapter 6. The graphical user interface is more feature-rich. In this section, we focus on the graphical user interface and describe its features and usefulness.

To start interacting with CAvSAT, the user connects to a remote database

server via SQL server authentication, as shown in Figure 5.3. After a connection is

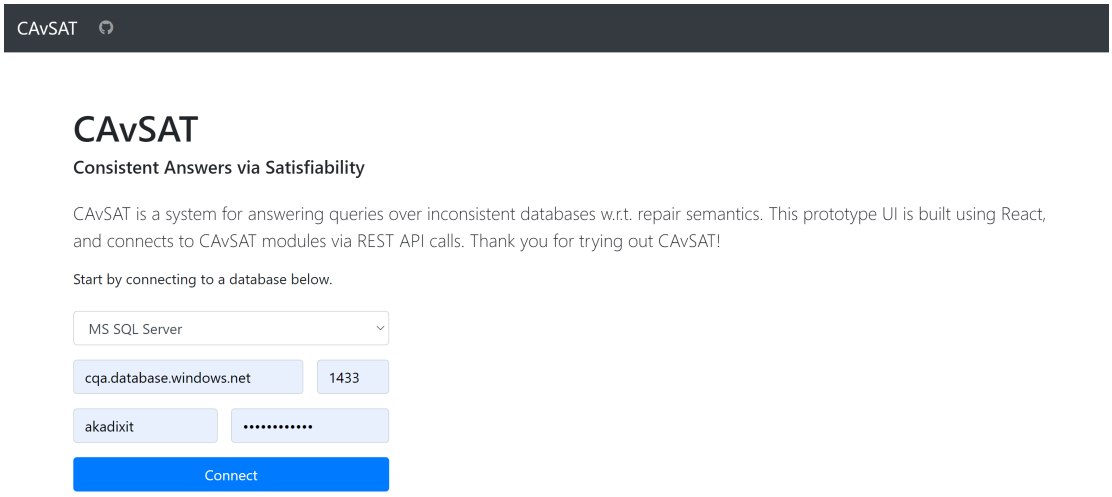


Figure 5.3: CAvSAT log-in screen to connect to a remote database instance

successfully established, the user is asked to choose the database instance on which they intend to compute the consistent answers of the queries. The top right corner of the user interface in Figure 5.4 shows that the user is connected to a Microsoft SQL Server at localhost and selected the database instance named `bank_accounts`.

At this point, if CAvSAT has not yet performed the data pre-processing on the database instance selected by the user (this happens only when the user connects to an instance for the first time), CAvSAT informs the user about it and starts the data pre-processing in the background. By default, the *Preview Schema and Raw Data* tab is displayed to the user where they can see a few sample rows from each relation of the database instance. Clicking on the *Constraints* button in the top bar shows the set of integrity constraints on the chosen schema (see Figure 5.5). In the main text box titled *Enter a Query*, CAvSAT allows the user to specify the input query either using

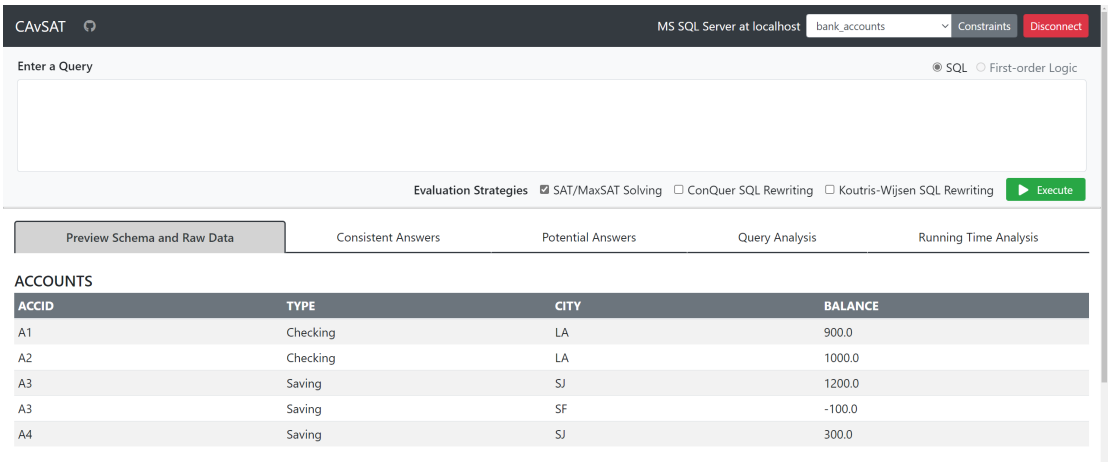


Figure 5.4: Preview of the data from the selected database instance

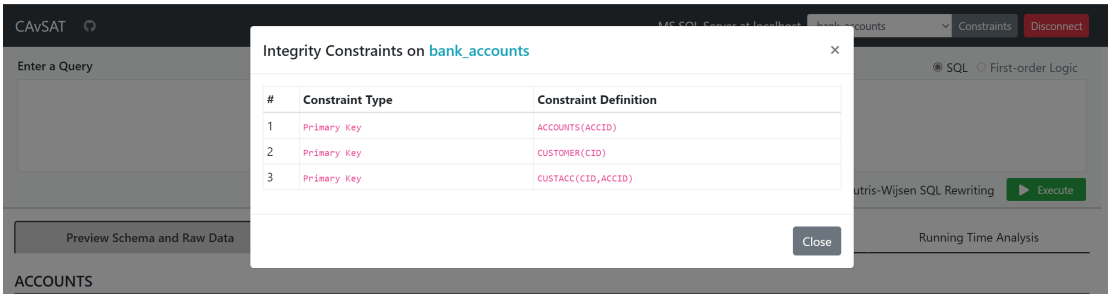


Figure 5.5: The set of integrity constraints on the selected database schema

SQL or with the first-order logic syntax. The user also has a choice between evaluation strategies for computing consistent answers: the default is to use SAT solving, while two SQL-rewriting algorithms from [40, 59] can be used for queries known to be having SQL-rewritable consistent answers or range consistent answers.

Clicking the *Execute* button starts the computation of the consistent answers or the range consistent answers to the query entered by the user. The answers are displayed to the user along with the evaluation strategy that CAvSAT opted for. When

multiple evaluation strategies are applicable, CAvSAT returns the answers to the user with the fastest of the strategies. Figure 5.6 shows that CAvSAT returned with 'John' as the only consistent answer to the conjunctive query entered by the user and Figure 5.7 shows the range consistent answers to the aggregation query that asks for the number of bank accounts grouped by city. In both cases, CAvSAT used Partial MaxSAT solving.

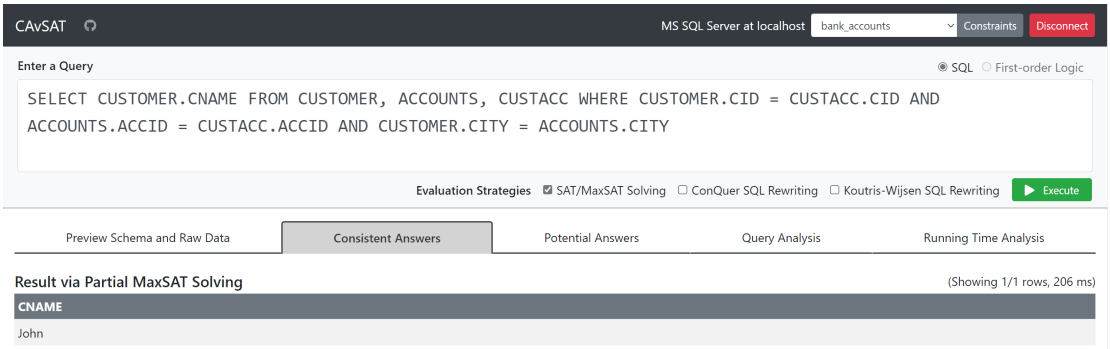


Figure 5.6: The consistent answers to a conjunctive query

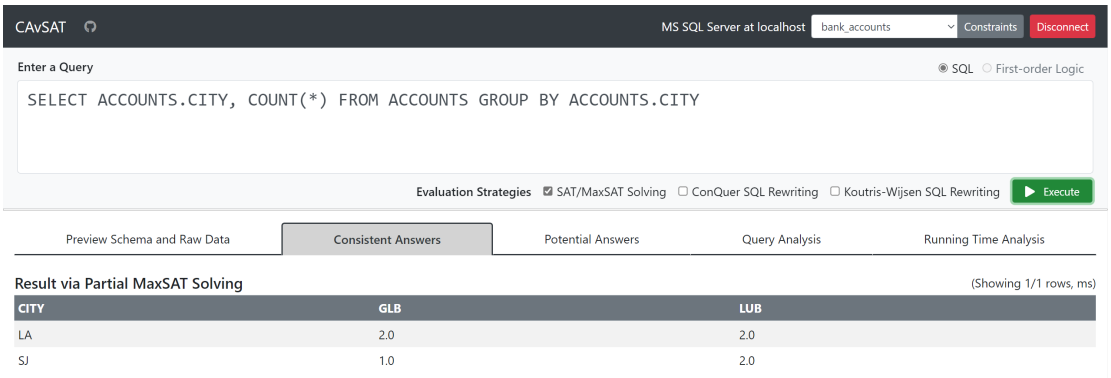


Figure 5.7: The range consistent answers to an aggregation query with grouping

The graphical user interface has several other features, including the following. The *Potential Answers* tab displays the answers to the input query evaluated directly

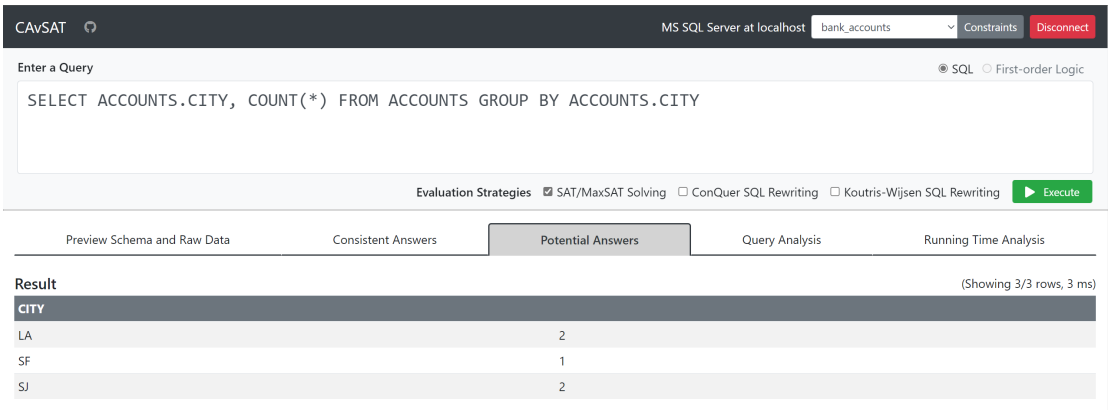


Figure 5.8: The potential answers to an aggregation query with grouping

on the inconsistent database instance (see Figure 5.8). This not only helps users to visualize the difference between potential and consistent answers but to also determine the overhead of computing consistent answers compared to evaluating the input query on the database directly. For example, the city 'SF' appears in the potential answers (Figure 5.8) with one bank account, but it is not a consistent answer because there is a repair with no bank accounts in 'SF' (Figure 5.7).

The *Query Analysis* tab shows the complexity of computing consistent answers to the query if it is determined by the Query Pre-processor module. The attack graph and the join graph of the input query are also visualized. As depicted in Figure 5.9, computing the consistent answers to the conjunctive query that asks for the names of customers having bank accounts in the same city as they belong is coNP-complete because there is a strong cycle of length two in its attack graph.

For queries having SQL-rewritable consistent answers, the *Data Complexity* pane of the *Query Analysis* tab lets the user view the rewriting computed using the

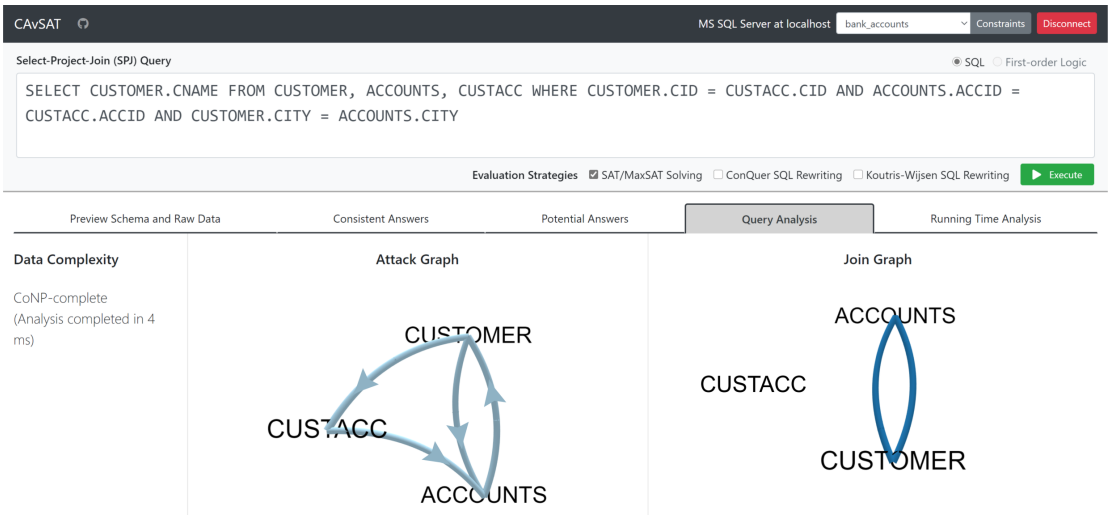


Figure 5.9: Query Analysis: a visual explanation on why computing the consistent answers to the input query is a coNP-complete task

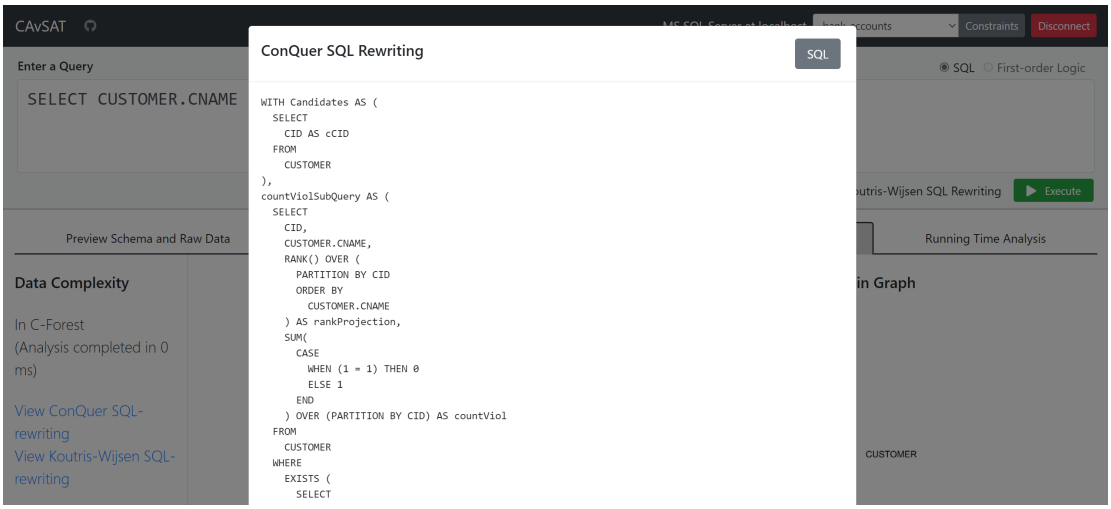


Figure 5.10: ConQuer-based SQL-rewriting of the consistent answers

algorithms from [40, 59] (see Figure 5.10). Finally, the *Running Time Analysis* tab shows the breakdown of the internal tasks performed by the SAT solving module of CAvSAT while computing the consistent answers or the range consistent answers, along

with the time taken to complete each of those tasks (Figure 5.11).

SAT / Partial MaxSAT Solving

Attribute	Value
Time to compute answers from the consistent part of the database (ms)	18
Time to compute minimal witnesses to the query (ms)	107
Time to compute relevant facts (ms)	27
Time to attach FactIDs to the relevant facts (ms)	26
Time to create positive clauses from key-equal groups (ms)	4
Time to create negative clauses from minimal witnesses (ms)	60
Time to write the clauses to a DIMAC file (ms)	3
Time to eliminate inconsistent potential answers (ms)	125
Total SAT-solving time (ms)	69
Time to write the final consistent answers to a table (ms)	75
Total Evaluation Time (ms)	498

Figure 5.11: The breakdown of the internal tasks and the time taken to compute the consistent answers using the SAT-solving module

Chapter 6

Experimental Analysis

We carried out a suite of extensive experiments with CAvSAT and evaluated its performance in several different scenarios. We divide our experiments in the following two main parts. In the first part, we report on the results obtained using CAvSAT on conjunctive queries and unions of conjunctive queries without aggregation and grouping (Section 6.1). Naturally, the SAT solving module of CAvSAT used in these experiments is based on the reductions from Chapter 4. In the second part, we give an account of our experiments with aggregation queries (with and without grouping), where the SAT solving module of CAvSAT uses the reductions from Chapter 4 (Section 6.2). In both of these parts, we conducted experiments using both synthetically generated databases and real-world databases, and with a variety of integrity constraints, such as primary keys, functional dependencies, and denial constraints.

6.1 Experiments on Queries Without Aggregation

6.1.1 Experimental Setup

The experiments on queries without aggregation were carried out on a machine running on Intel Core i7 2.7 GHz, 64 bit Ubuntu 16.04, with 8GB of RAM. We used PostgreSQL as an underlying DBMS, and MaxHS v3.0 solver [31] for solving the Weighted MaxSAT instances. Our system is implemented in Java 9.04. In the first set of experiments, synthetically generated database instances with primary key constraints were used to evaluate a set of 21 conjunctive queries. On the queries that have SQL-rewritable consistent answers, we compared the performance of CAvSAT and the existing SQL-rewriting approaches from Koutris and Wijsen [59] and ConQuer [40]. In the second set of experiments, we evaluated the performance of CAvSAT on a real-world database with key constraints and one functional dependency constraint.

6.1.2 Synthetic Data Generation

For this set of experiments, the synthetic data were generated in two phases: (a) generation of consistent data; (b) injection of inconsistency into consistent data. The parameters used to generate the data were the number of tuples per relation (*rSize*), degree of inconsistency (*inDeg*), i.e., the percentage of tuples participating in the violations of integrity constraints, and the size of each key-equal group (*kSize*).

Generating consistent data

For this set of experiments, we used a database schema with seven relations $R_1, \cdots, R_7$, of arity two or three each. This schema has been used before to evaluate EQUIP [55], a consistent query answering system based on Integer Linear Programming solving. Each relation in the consistent database was generated with the same number of tuples so that injecting inconsistency with specified *kSize* and *inDeg* will make the total number of tuples in the relation equal to *rSize*. For each query used in the experiment, the data was generated so the evaluation of the query on the consistent database results in a relation that has the size 15% to 20% of *rSize*. The values of the third attribute in all of the ternary relations were chosen from a uniform distribution in the range [1, *rSize*/10]. This was done to simulate a reasonably large number of potential answers. The remaining attributes take values from randomly generated strings of length 10.

Injecting inconsistency

In each relation, the inconsistency was injected by inserting new tuples to the consistent data, that share the values of the key attributes with some already existing tuples from the consistent data. We conducted experiments with varying values of *inDeg*, ranging from 5% to 15%. The values of *kSize* were uniformly distributed between two to five. The non-key attributes of the newly injected tuples were uniform random alphanumeric strings of length 10.

6.1.3 Conjunctive Queries on Synthetic Databases

For the set of experiments on the synthetically generated databases, we used a set of 21 self-join-free conjunctive queries $q_1, \cdots, q_{21}$, which was also used to evaluate EQUIP [55] in the past. These queries have been carefully designed so that the first seven queries $q_1, \cdots, q_7$ have SQL-rewritable consistent answers, the next seven queries, $q_8, \cdots, q_{14}$, have consistent answers that are polynomial-time computable but not SQL-rewritable, and the consistent answers to the last seven queries, $q_{15}, \cdots, q_{21}$, are coNP-complete. Here, we list down all conjunctive queries used in this set of experiments.

SQL-rewritable consistent answers

$$q_1(z) := \exists x, y, v, w\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, v, w))$$

$$q_2(z, w) := \exists x, y, v\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, v, w))$$

$$q_3(z) := \exists x, y, v, u, d\ (R_1(\underline{x}, y, z) \wedge R_3(\underline{y}, v) \wedge R_2(\underline{v}, u, d))$$

$$q_4(z, d) := \exists x, y, v, u\ (R_1(\underline{x}, y, z) \wedge R_3(\underline{y}, v) \wedge R_2(\underline{v}, u, d))$$

$$q_5(z) := \exists x, y, v, w\ (R_1(\underline{x}, y, z) \wedge R_4(\underline{y, v}, w))$$

$$q_6(z) := \exists x, y, x', w, d\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w) \wedge R_5(\underline{x}, y, d))$$

$$q_7(z) := \exists x, y, w, d\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w) \wedge R_5(\underline{x}, y, d))$$

In P but not SQL-rewritable consistent answers

$$q_8(z, w) := \exists x, y\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w))$$

$$q_9(z) := \exists x, y, w, u, d\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w) \wedge R_4(\underline{y}, u, d))$$

$$q_{10}(z, w, d) := \exists x, y, u\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w) \wedge R_4(\underline{y}, u, d))$$

$$q_{11}(z) := \exists x, y, w\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w))$$

$$q_{12}(v, d) := \exists x, y, z, u\ (R_3(\underline{x}, y) \wedge R_6(\underline{y}, z) \wedge R_1(\underline{z}, x, d) \wedge R_4(\underline{x, u}, v))$$

$$q_{13}(v) := \exists x, y, z, u\ (R_3(\underline{x}, y) \wedge R_6(\underline{y}, z) \wedge R_7(\underline{z}, x) \wedge R_4(\underline{x, u}, v))$$

$$q_{14}(d) := \exists x, y, z, u\ (R_3(\underline{x}, y) \wedge R_6(\underline{y}, z) \wedge R_1(\underline{z}, x, d) \wedge R_7(\underline{x}, u))$$

CoNP-complete consistent answers

$$q_{15}(z) := \exists x, y, x', w\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w))$$

$$q_{16}(z, w) := \exists x, y, x'\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w))$$

$$q_{17}(z) := \exists x, y, x', w, u, d\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w) \wedge R_4(\underline{y}, u, d))$$

$$q_{18}(z, w) := \exists x, y, x', u, d\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w) \wedge R_4(\underline{y}, u, d))$$

$$q_{19}(z, w, d) := \exists x, y, x', u\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w) \wedge R_4(\underline{y}, u, d))$$

$$q_{20}(z) := \exists x, y, x', w, u, d, v\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w) \wedge R_4(\underline{y}, u, d) \wedge R_3(\underline{u}, v))$$

$$q_{21}(z, w) := \exists x, y, x', u, d, v\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w) \wedge R_4(\underline{y}, u, d) \wedge R_3(\underline{u}, v))$$

6.1.4 Experimental Results on Synthetic Databases

CAvSAT on SQL-rewritable Conjunctive Queries

In this set of experiments, we compare the performance of CAvSAT against the SQL-rewritings of seven queries over the database with primary key constraints. For queries $q_1, \ldots, q_7$, we computed the SQL-rewritings using the algorithm of Koutris and Wijsen in [59]. We refer to these SQL-rewritings as the *KW-SQL-rewritings*. Since the queries $q_1, \ldots, q_4$ happen to be in the class C_{forest}, we computed additional SQL-rewritings for them using the algorithm from ConQuer [40]; we refer to these rewritings

as the *ConQuer-SQL-rewritings*.

We first computed the consistent answers to these queries using the SAT solving module of CAvSAT on database instances of size one million tuples per relation and with 10% inconsistency. Figure 6.1 shows computing the answers to the queries from the consistent part of the database first significantly improves the performance of CAvSAT. The letter 'E' indicates the time taken by the SAT solving module of CAvSAT to encode the problem into a SAT instance, and the letter 'S' denotes the time taken to solve the SAT instance. The legends 'Opt' and 'Unopt' denote whether the performance optimization of computing the answers from the consistent part of the data first was on or off. The gain in the performance is not surprising; since 90% of the data were consistent, it is expected that most of the consistent answers lie in the consistent part of the database. The SAT instances corresponding to the unoptimized way of computing the answers had an average of over 2.5 million variables and clauses, while the size was reduced to 45 thousand variables and 49 thousand clauses for the optimized approach. For the rest of the experiments, we use the optimized approach, i.e., we compute the answers to the queries from the consistent part of the data first.

Figure 6.2 shows that for the queries $q_1, \ldots, q_4$ in the class C_{forest}, the performance of CAvSAT is slightly worse, but comparable, to their ConQuer-SQL-rewritings. For all seven queries $q_1, \ldots, q_7$, however, CAvSAT significantly outperformed their KW-SQL-rewritings, as PostgreSQL hit the two hours timeout while evaluating each KW-SQL-rewriting. In fact, this timeout was hit by all seven queries even for databases of size as small as 100K tuples per relation. For $q_1, \ldots, q_7$, the average number of iterations

taken by Algorithm 1 to eliminate all inconsistent potential answers was 2.85.

Figure 6.1: Performance of CAvSAT on conjunctive queries with SQL-rewritable consistent answers (1M tuples/relation with 10% inconsistency).

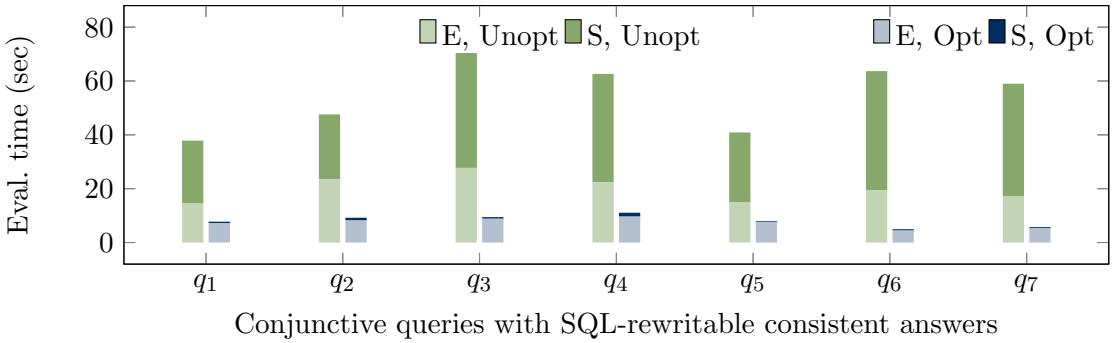
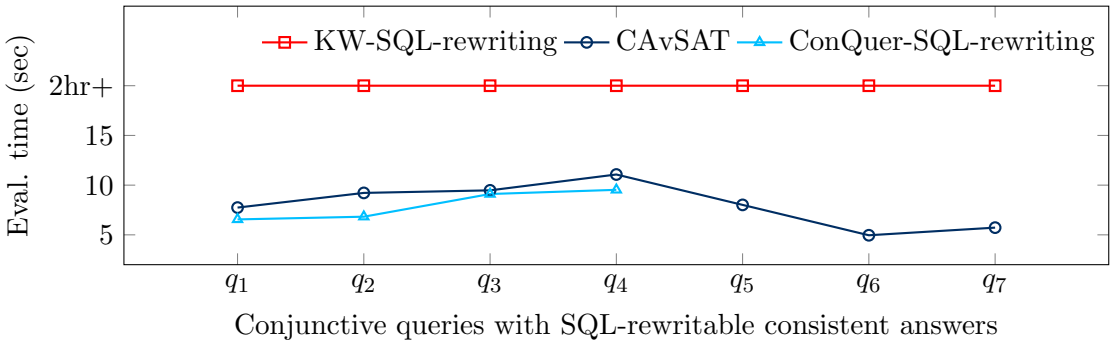


Figure 6.2: Performance of CAvSAT, ConQuer, and Koutris-Wijsen rewriting



CAvSAT on Harder Queries

In this set of experiments, we considered fourteen additional non-boolean conjunctive queries $q_8, \cdots, q_{21}$ whose consistent answers are coNP-complete or in P but not

SQL-rewritable. Figure 6.3 shows that the time required to encode the CQA instance into a Partial MaxSAT instance dominates over the time taken by the SAT solver and the Algorithm 1 to eliminate all inconsistent potential answers. The solver takes comparatively more time for the queries that have more free variables or more atoms. Table 6.1 shows the size of the Partial MaxSAT instances constructed by Reduction 3.1.2 in this experiment. The average number of iterations taken by Algorithm 1 to eliminate all inconsistent potential answers to a query was 3.2.

Figure 6.3: Performance of CAvSAT on conjunctive queries with consistent answers of varying data complexity (1M tuples/relation, varying inconsistency).

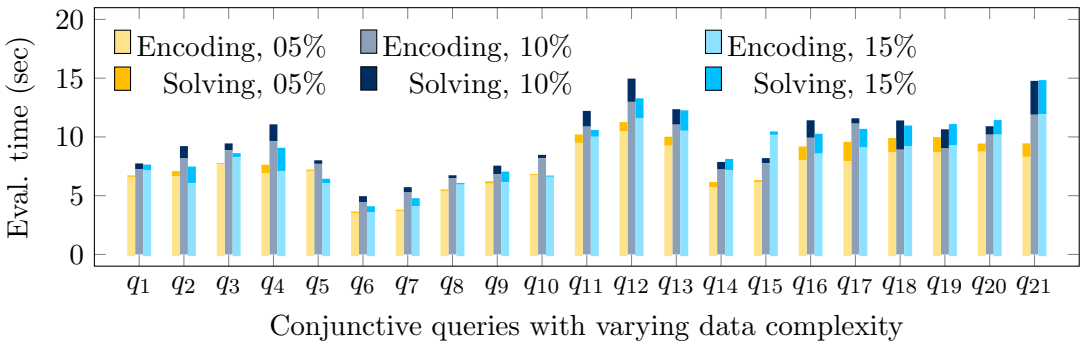


Table 6.1: The size of the CNF-formulas with optimization (1M tuples/relation)

Query	Variables	Clauses	Query	Variables	Clauses	Query	Variables	Clauses
q_1	16.5K	20.9K	q_8	16.6K	16.8K	q_{15}	14.9K	15K
q_2	68.6K	76.0K	q_9	58K	57.7K	q_{16}	58.8K	58.4K
q_3	31.9K	36.8K	q_{10}	31.3K	36.6K	q_{17}	40.1K	41.4K
q_4	117.2K	123.7K	q_{11}	105K	118.1K	q_{18}	107.5K	121.4K
q_5	16.3K	20.6K	q_{12}	116.8K	123.4K	q_{19}	114.4K	120.7K
q_6	32.8K	33.2K	q_{13}	63.2K	65.7K	q_{20}	53.4K	63.7K
q_7	32.5K	33.8K	q_{14}	53.9K	59.2K	q_{21}	170K	199K

6.1.5 Real-world Database and Queries

For the next set of experiments, we used real-world data having key constraints on each relation, along with one functional dependency. The data are about inspections of food establishments in Chicago and New York and are taken from [6, 7]. Parts of this data have been previously used for evaluating data cleaning systems, such as HoloClean [74]. Since the structure of the schema or the constraints on the database were not specified by the source, we decomposed the data into four relations, and assumed reasonable key constraints for all relations and also one additional functional dependency, as shown in Table 6.2.

Table 6.2: The schema and the constraints of the real-world database

Relation	# Tuples
NY_Insp (LicenseNo, Risk, InspDate, InspType, Result)	229K
NY_Rest (Name, LicenseNo, Cuisine, Address, Zip)	26.5K
CH_Insp (LicenseNo, Risk, InspDate, InspType, Result)	167K
CH_Rest (Name, LicenseNo, Facility, Address, Zip)	31.1K

Constraint	Type	Violations
NY_Insp (LicenseNo, InspDate, InspType $\rightarrow$ Risk, Result)	Key	25.6%
NY_Rest (LicenseNo $\rightarrow$ Name, Cuisine, Address, Zip)	Key	0%
CH_Insp (LicenseNo, InspDate, InspType $\rightarrow$ Risk, Result)	Key	0.07%
CH_Rest (LicenseNo $\rightarrow$ Name, Cuisine, Address, Zip)	Key	5.86%
CH_Rest (Name $\rightarrow$ Zip)	FD	9.73%

We evaluated the performance of CAvSAT on the six queries $q_1^r, \cdots, q_6^r$, their definitions are given next. For example, the query q_3^r returns the names of the restaurants, such that their outlets are present in both New York and Chicago, and they were

inspected on the same day.

$$q_1^r() := \exists x, y, z, w, v, y', z', w', v' \ (\text{NY\_Rest}(x, y, z, w, v) \wedge \text{CH\_Rest}(x, y', z', w', v')),$$

$$q_2^r(x) := \exists y, z, w, v, y', z', w', v' \ (\text{NY\_Rest}(x, y, z, w, v) \wedge \text{CH\_Rest}(x, y', z', w', v')),$$

$$q_3^r(x) := \exists y, z, w, v, y', z', w', v', q, r, s, t, q', s', t' \ (\text{NY\_Rest}(x, y, z, w, v)$$
$$\wedge \text{CH\_Rest}(x, y', z', w', v') \wedge \text{NY\_Insp}(y, q, r, s, t) \wedge \text{CH\_Insp}(y', q', r, s', t')),$$

$$q_4^r(x, y) := \exists z, w, v, q, r, s \ (\text{CH\_Rest}(x, y, z, w, v) \wedge \text{CH\_Insp}(y, q, r, s, \text{`Pass'})),$$

$$q_5^r(x) := \exists y, z, w, v, q, r, s \ (\text{CH\_Rest}(x, y, z, w, v) \wedge \text{CH\_Insp}(y, q, r, s, \text{`Fail'})) \cup$$
$$\exists y, z, w, v, q, r, s \ (\text{NY\_Rest}(x, y, z, w, v) \wedge \text{NY\_Insp}(y, q, r, s, \text{`Fail'})),$$

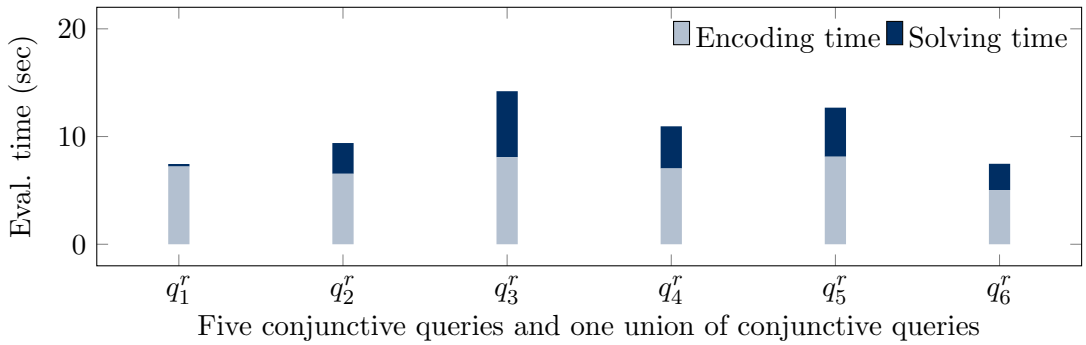
$$q_6^r(x, v) := \exists y, z, w, y', z', w', v', q, r, s \ (\text{CH\_Rest}(x, y, z, w, v) \wedge \text{NY\_Rest}(x, y', z', w', v')$$
$$\wedge \text{NY\_Insp}(y', \text{`Not Critical'}, q, r, s)).$$

6.1.6 Experimental Results on the Real-world Database

Here, we observed that the encoding time for all six queries is pretty much identical, but it is higher compared to the queries in the earlier set of experiments considering that the database instance used here is much smaller in size. The reason behind this is that the functional dependency constraint on the schema shown in Figure 6.4 forces CAvSAT to opt for Reduction 3.2.1 since Reduction 3.1.2 is applicable only to primary key constraints. The Partial MaxSAT instance generated from Reduction 3.2.1 are larger compared to the ones from Reduction 3.2.1 due to the way minimal violations and the near-violations are encoded. As a result, the solver also takes a considerably longer time to solve these instances compared to the queries on the synthetically generated database. Not surprisingly, the evaluation time increases as the number of atoms

or the number of free variables in the query grow.

Figure 6.4: Performance of CAvSAT on the real-world food inspection database



6.2 Experiments on Aggregation Queries

6.2.1 Experimental Setup

We evaluated the performance of CAvSAT over both synthetic and real-world databases. The first set of experiments includes a comparison of CAvSAT with an existing SQL-rewriting-based CQA system, namely, ConQuer, over synthetically generated TPC-H databases having one key constraint per relation. This set of experiments is divided into two parts, based on the method used to generate the inconsistent database instances. In the first part, we use the `DBGen` tool from TPC-H and artificially inject inconsistencies in the generated data; in the second part, we employ the PDBench inconsistent database generator from MayBMS [14] (see Section 6.2.2 for details). Next, we assess the scalability of CAvSAT by varying the size of the database and the amount of inconsistency present in it. Lastly, to evaluate the performance of the reductions

from Section 4.4, we use a real-world Medigap [3] database that has three functional dependencies and one denial constraint. All experiments were carried out on a machine running on Intel Core i7 2.7 GHz, 64 bit Ubuntu 16.04, with 8GB of RAM. We used Microsoft SQL Server 2017 as an underlying DBMS, and MaxHS v3.2 solver [31] for solving the Weighted MaxSAT instances.

6.2.2 Experiments with TPC-H Data and Queries

TPC-H Data Generation

For the first part of the experiments, the data is generated using the `DBGen` data generation tool from the TPC-H Consortium. The TPC-H schema comes with exactly one key constraint per relation, which was ideal for comparing CAvSAT against ConQuer [40, 42] (the only existing system for computing the range consistent answers to aggregation queries) because ConQuer does not support more than one key constraint per relation or classes of integrity constraints broader than keys. The `DBGen` tool generates consistent data, so we artificially injected inconsistency by updating the key attributes of randomly selected tuples from the data with the values taken from existing tuples of the same relation. The sizes of the key-equal groups that violate the key constraints were uniformly distributed between two and seven. The database instances were generated in such a way that every repair had the specified size. We experimented with varying degrees of inconsistency, ranging from 5% up to 35% of the tuples violating a key constraint, and with a variety of repair sizes, starting from 500 MB (4.3 million tuples) up to 5 GB (44 million tuples). For the second part, we employed the PDBench

database generator from MayBMS [14] to generate four inconsistent database instances with varying degrees of inconsistency (see Table 6.3). In all four instances, the data is generated in such a way that every repair is of size 1 GB.

Table 6.3: Details of the TPC-H instances generated using PDBench

	Percentage of Inconsistency			
Relation	**Instance 1**	**Instance 2**	**Instance 3**	**Instance 4**
CUSTOMER	4.42%	8.5%	16.14%	29.49%
LINEITEM	6.36%	12.09%	22.53%	39.82%
NATION	7.69%	0%	7.69%	7.69%
ORDERS	3.51%	6.77%	12.87%	23.9%
PART	4.93%	9.33%	17.66%	32.16%
PARTSUPP	1.53%	2.96%	5.77%	11.13%
REGION	0%	0%	0%	0%
SUPPLIER	3.69%	7.44%	14.11%	26.51%
Overall	5.36%	10.25%	19.29%	34.72%
	Database size and Repair size (in GB)			
	1.04 & 1.00	1.07 & 1.01	1.14 & 1.02	1.3 & 1.02
	Size of the Largest Key-equal Groups			
	8 tuples	16 tuples	16 tuples	32 tuples

TPC-H Queries

The standard TPC-H specification comes with 22 queries (constructed using the `QGen` tool). Here, we focus on queries 1, 3, 4, 5, 6, 10, 12, 14, and 19; the other 13 queries have features such as nested subqueries, left outer joins, and negation that are beyond the aggregation queries defined in Section 2.1. In Section 6.2.3, we describe our results for queries without grouping. Since six out of the nine queries under consideration contained the `GROUP BY` clause, we removed it and added appropriate conditions in

the `WHERE` clause based on the original grouping attributes to obtain queries without grouping. We refer to these queries as $Q'_1, Q'_3, \cdots, Q'_{19}$ and their definitions are given in Table A.1 of the Appendix.

6.2.3 Experimental Results on Queries without Grouping

In the first set of experiments, we computed the range consistent answers of the TPC-H-inspired aggregation queries without grouping via WPMaxSAT solving over a database instance with 10% inconsistency and having repairs of size 1 GB (8 million tuples). Figure 6.5 shows that much of the evaluation time used by CAvSAT is consumed in encoding the CQA instance into a WPMaxSAT instance, while the solver comparatively takes less time to compute the optimal solution. Note that, Q'_5 is not in the class $C_{aggforest}$, and thus ConQuer cannot compute its range consistent answers. CAvSAT performs better than ConQuer on seven out of the remaining eight queries.

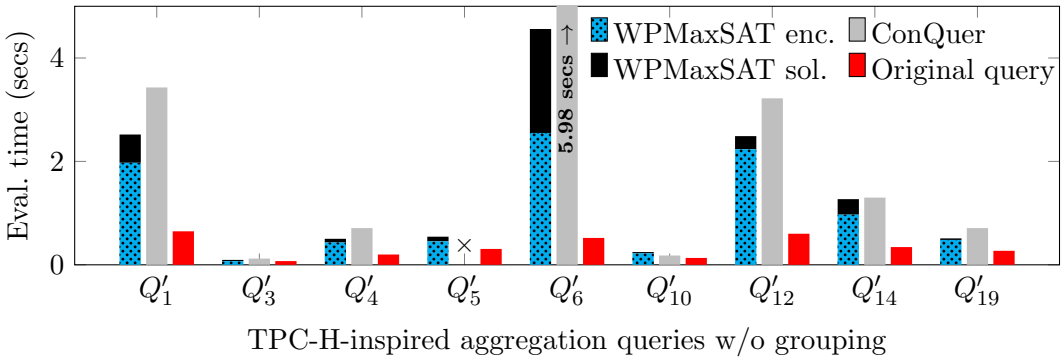


Figure 6.5: CAvSAT vs. ConQuer on TPC-H data generated using the `DBGen`-based tool (10% inconsistency, 1 GB repairs)

Next, we compared the performance of CAvSAT and ConQuer on database

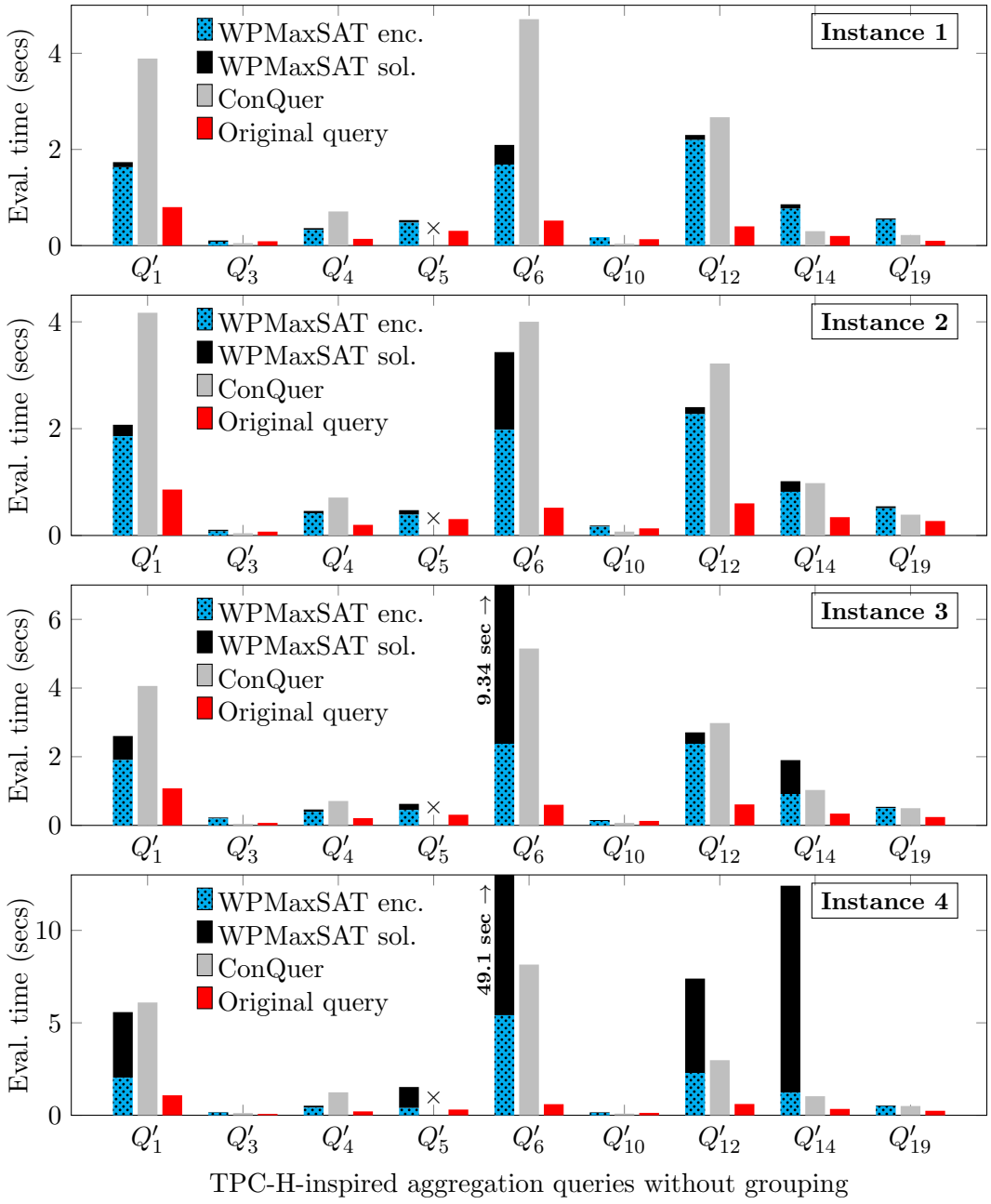


Figure 6.6: CAvSAT vs. ConQuer on PDBench instances

instances generated using PDBench. Figure 6.6 shows that CAvSAT performs better than ConQuer on PDBench instances with low inconsistency. As the inconsistency

increases, the Weighted Partial MaxSAT solver requires a considerably long time to compute the optimal solutions (especially for Q'_6, Q'_{12}, and Q'_{14}). One reason is that the sizes of key-equal groups in PDBench instances with higher inconsistency percentages are large, which translates into clauses of large sizes in the Weighted Partial MaxSAT instances, hence the solver works hard to solve them. Also, Kügel's reduction [64] from Weighted Partial MinSAT to Weighted Partial MaxSAT significantly increases the size of the CNF formula, resulting in higher time for the *lub*-answers to the queries.

Table 6.4: The average size of the CNF formulas for Q'_1, Q'_6, and Q'_{14}

(a) # of variables (in thousands)

	5%	15%	25%	35%
Q'_1	10.2	34.3	60.6	95.3
Q'_6	28.4	96.2	175.0	271.2
Q'_{14}	6.4	21.1	40.6	62.3

(b) # of clauses (in thousands)

5%	15%	25%	35%
27.6	92.2	163.6	258.1
76.8	259.9	472.9	734.0
15.6	51.9	101.0	156.6

(c) # of variables (in thousands)

	1 GB	3 GB	5 GB
Q'_1	21.3	44.1	105.6
Q'_6	60.9	127.13	304.4
Q'_{14}	13.9	32.9	67.7

(d) # of clauses (in thousands)

1 GB	3 GB	5 GB
57.7	104.1	258.8
165.3	300.7	823.1
34.0	73.7	166.6

Next, we varied the inconsistency in the database instances created using the `DBGen`-based data generator while keeping the size of the database repairs constant (1 GB). Figure 6.7 shows that the evaluation time of CAvSAT stays well under ten seconds (except for Q'_6), even if there is more inconsistency in the data. Tables 6.4a and 6.4b show the average size of the CNF formulas for the top three queries that exhibited the

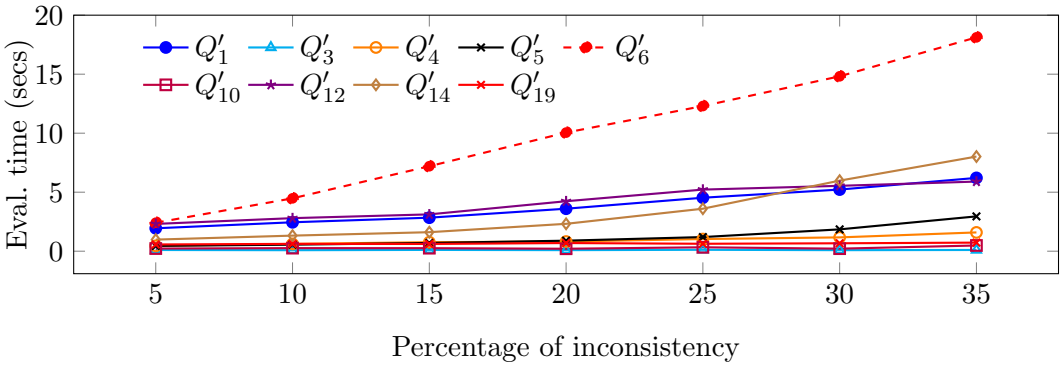


Figure 6.7: CAvSAT on TPC-H data generated using the `DBGen`-based tool (varying inconsistency, 1 GB repairs)

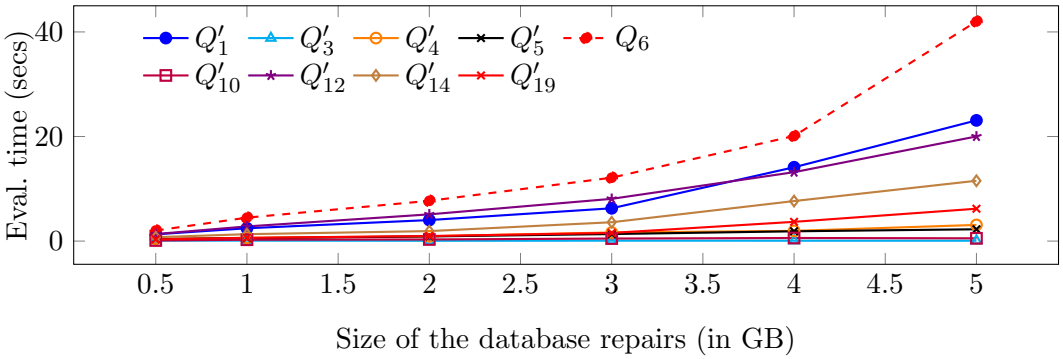


Figure 6.8: CAvSAT on TPC-H data generated using the `DBGen`-based tool (varying database sizes, 10% inconsistency)

largest CNF formulas. The size of the formulas grows nearly linearly as the inconsistency present in the data grows. The CNF formulas for Q'_6 are significantly larger than the ones corresponding to the other queries since Q'_6 has high selectivity and it is posed against the single largest relation `LINEITEM` which has over 8.2 million tuples in an instance with 35% inconsistency. This also explains why CAvSAT takes more time for computing its range consistent answers (Figure 6.7). In database instances with low

inconsistency, the consistent answers to the queries having low selectivity (e.g., Q'_3, Q'_{10}) are sometimes contained in the consistent part of the data, and CAvSAT does not need to construct a Weighted Partial MaxSAT instance at all.

We then evaluated CAvSAT's scalability by increasing the sizes of the databases while keeping the inconsistency to a constant 10%. Figure 6.8 shows that the evaluation time of CAvSAT for queries Q'_1, Q'_6, and Q'_{12} increases faster than that for the other queries. This is because the queries Q'_1 and Q'_6 are posed against `LINEITEM` while Q'_{12} involves a join between `LINEITEM` and `ORDERS` resulting in CAvSAT spending more time on computing the minimal witnesses to these queries as the size of the database grows. Table 6.4c and 6.4d show that the size of the CNF formulas grows almost linearly w.r.t. the size of the database. The largest CNF formula consisted of over 304 thousand variables and 823 thousand clauses and was exhibited by Q'_6 on a database of size 5 GB (47 million tuples). The low selectivity of queries Q'_3, Q'_{10}, and Q'_{19} resulted in very small CNF formulas, even on large databases.

6.2.4 Experimental Results on Queries with Grouping

In this set of experiments, we focus on TPC-H queries 1, 3, 4, 5, 10, and 12 (see Table A.2 in the Appendix), as the queries 6, 14, and 19 did not contain grouping. We evaluated the performance of CAvSAT and compared it to ConQuer on a database with 10% inconsistency w.r.t. primary keys (Figure 6.9). The repairs are of size 1 GB. CAvSAT computes the consistent answers to the underlying conjunctive query using the reductions from [33] which are, precisely, the consistent groups in the range consistent

answers to the aggregation query. For each of these groups, it computes the *glb*-answer and the *lub*-answer using reductions to Weighted Partial MaxSAT.

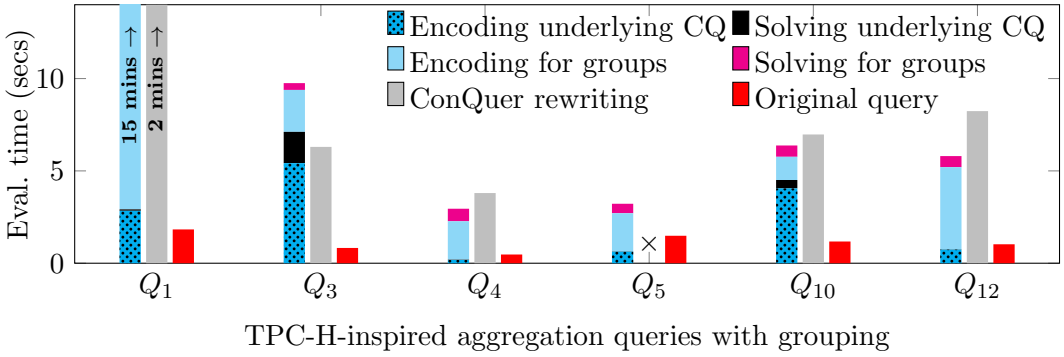


Figure 6.9: CAvSAT vs. ConQuer on TPC-H data generated using the `DBGen`-based tool (10% inconsistency, 1 GB repairs)

The overhead of computing the range consistent answers to aggregation queries with grouping is higher than that for the aggregation queries without grouping because for an aggregation query with grouping, CAvSAT needs to construct and solve twice as many Weighted Partial MaxSAT instances as there are consistent groups, i.e., one for the *lub*-answer and one for the *glb*-answer per consistent group. For queries that involved the `SELECT TOP` k construct of SQL, we chose top k consistent groups ordered by one or more grouping attributes present in the `ORDER BY` clause of the query. CAvSAT computes the range consistent answers to each query under ten seconds except for Q_1. It took under three seconds to compute the consistent groups of Q_1, but took over forty seconds to encode the range consistent answers of the groups and over fifteen minutes to solve the corresponding Weighted Partial MaxSAT instances. This is because some consistent groups have over 3 million tuples and so the Weighted Partial MaxSAT

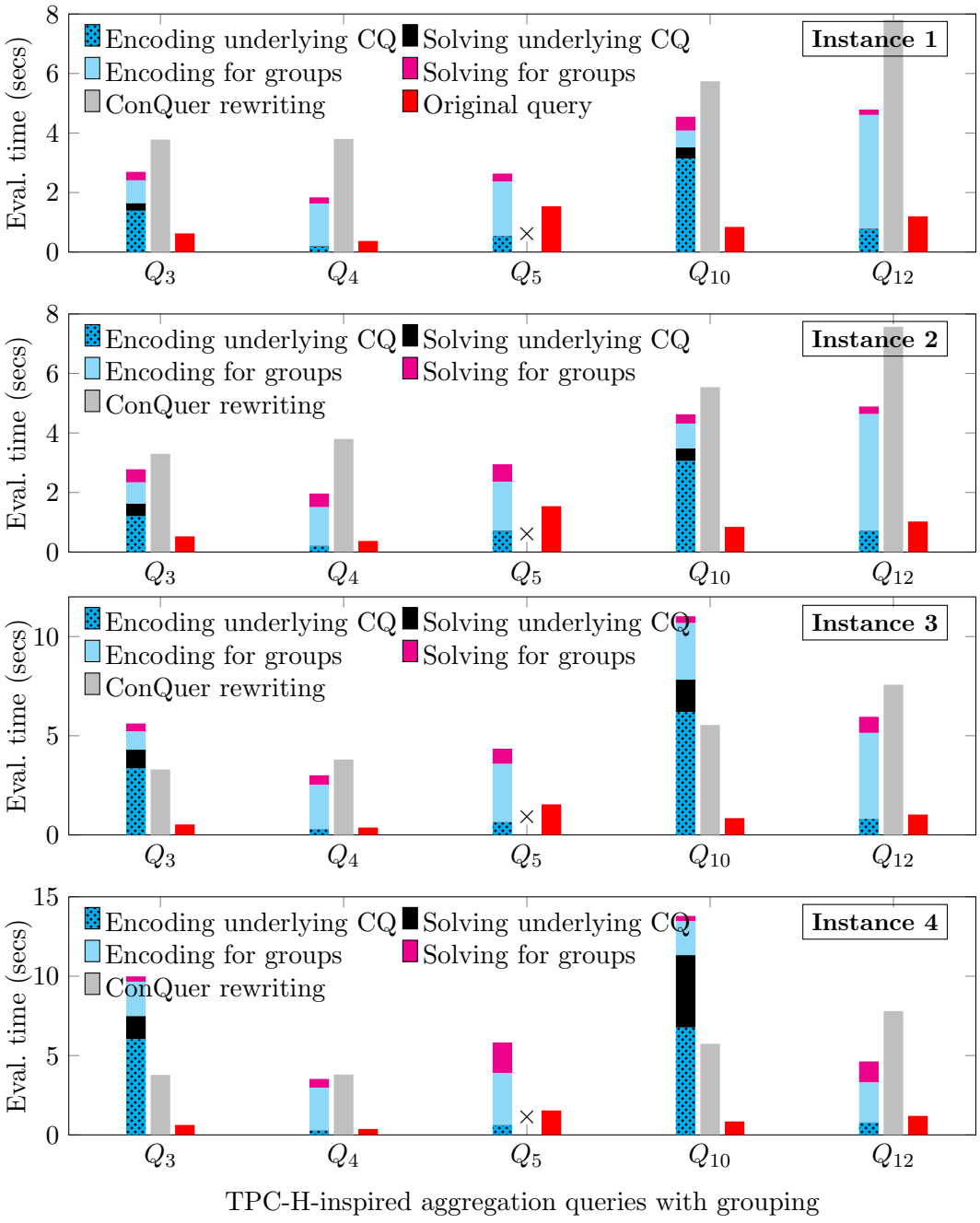


Figure 6.10: CAvSAT vs. ConQuer on PDBench instances

instances have over 600 thousand variables and over 1.3 million clauses. ConQuer took slightly over two minutes to compute the range consistent answers to Q_1. We did not include Q_1 in experiments with larger databases and higher inconsistency.

Figure 6.10 shows the comparison of CAvSAT and ConQuer for aggregation queries with grouping on PDBench instances. For the database instance with the lowest amount of inconsistency, CAvSAT beats ConQuer on all queries, but as the inconsistency grows, CAvSAT takes longer time to encode and solve for the consistent groups of the queries Q_3 and Q_{10}. In Figure 6.11, we first plot the evaluation time of CAvSAT as the percentage of inconsistency in the data grows from 5% to 35% in the instances generated using the `DBGen`-based data generator. The size of the database repairs is kept constant at 1 GB (8 million tuples).

Since CAvSAT constructs and solves many Weighted Partial MaxSAT instances having varying sizes for an aggregation query involving grouping, we also plot the overall number of SAT calls made by the solver in Figure 6.11. Note that the Y-axis has logarithmic scaling in the second plot of Figure 6.11. There are ten consistent groups in the answers to Q_3, and just five and two consistent groups in the answers to Q_5 and Q_{12} respectively. In each consistent group, the aggregation operator is applied over a much larger set of tuples in Q_5 and Q_{12} than in Q_3. As a result, the evaluation time for Q_3 is high but the number of SAT calls is comparatively less, while CAvSAT makes more SAT calls for Q_5 and Q_{12}, even though their consistent answers are computed much faster. The query Q_{10} requires a long time to construct and solve the Weighted Partial MaxSAT instances for its consistent groups due to its high selectivity and the

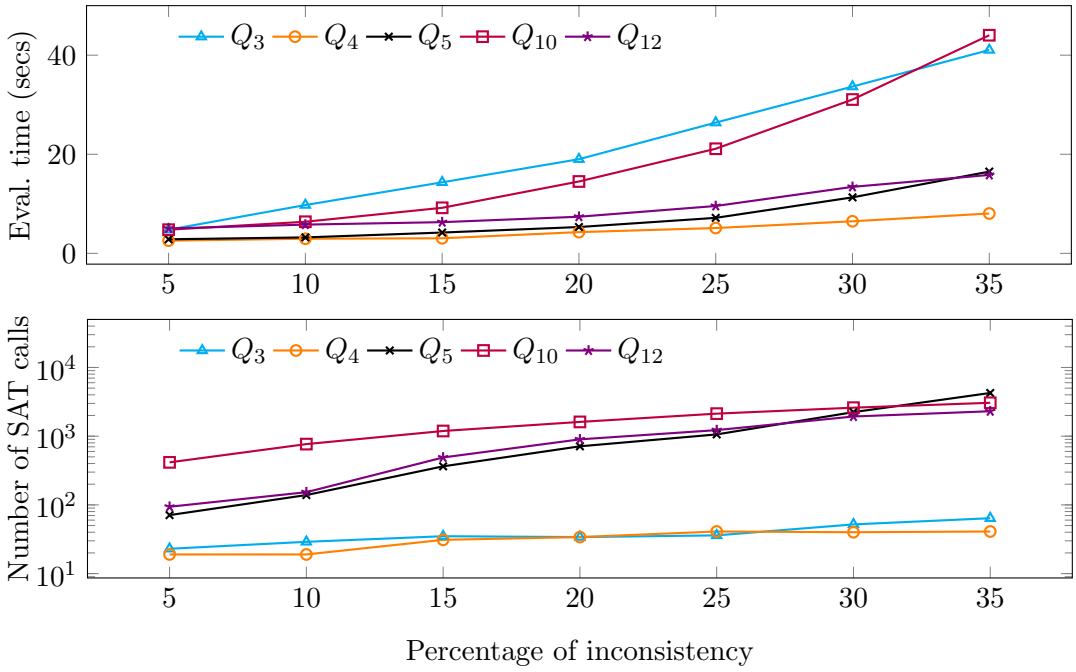


Figure 6.11: CAvSAT on TPC-H data generated using the `DBGen`-based tool (varying inconsistency, 1 GB repairs)

presence of joins between four relations. The evaluation time of computing the range consistent answers to aggregation queries with grouping increases almost linearly w.r.t. the size of the database when the percentage of inconsistency is constant (Figure 6.12). The second plot in Figure 6.12 depicts the number of SAT calls made by the solver as the size of the database grows. Due to low selectivity, the answers to Q_4 are encoded into small CNF formulas even on databases with high inconsistency or large sizes, resulting in fast evaluations.

The experiments show that CAvSAT performed well across a broad range of queries and databases; it performed worse on queries with high selectivity because,

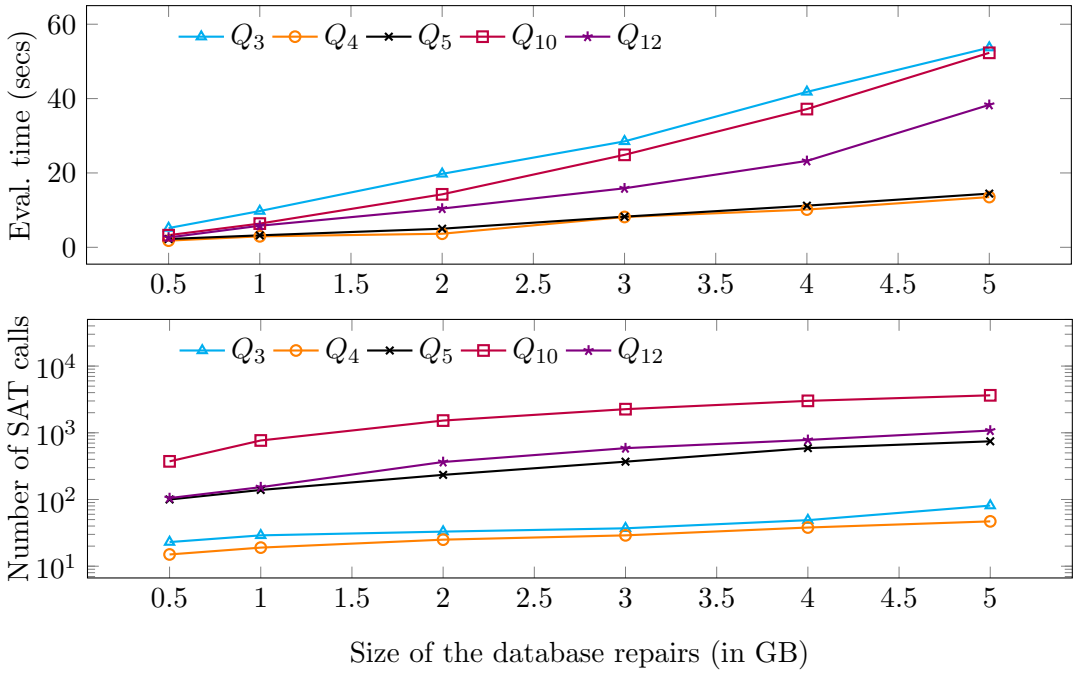


Figure 6.12: CAvSAT on TPC-H data generated using the `DBGen`-based tool (varying database sizes, 10% inconsistency)

in such cases, very large CNF formulas were generated. CAvSAT slowed down on databases with high degree of inconsistency ($> 30\%$) and with key-equal groups of large sizes (> 15). These are rather corner cases that should not be encountered in real-world databases.

6.2.5 Experiments with Real-world Data

Real-world Database and Queries

For this set experiments, we use the schema and the data from Medigap [3], an openly available real-world database about Medicare supplement insurance in the

Table 6.5: Details of the Medigap real-world database

(a) Medigap schema and the size of the instance

Relation	Acronym	# of attributes	# of tuples
OrgsByState	OBS	5	3872
PlansByState	PBS	18	21002
PlansByZip	PBZ	20	4748
PlanType	PT	4	2434
Premiums	PR	7	29148
SimplePlanType	SPT	4	70

(b) Integrity constraints and inconsistency

Type	Constraint Definition	Inconsistency
FD	OBS (orgID $\rightarrow$ orgName)	2.58%
FD	PBS (addr, city, abbrev $\rightarrow$ zip)	1.5%
DC	$\forall\, t \in$ PBS (t.webAddr $\neq$ ‘’)	0.15%

United States. We combine the data from 2019 and 2020 to obtain a database with over 61K tuples (Table 6.5a). We evaluated the performance of Reduction 3.2.1, since we consider two functional dependencies and one denial constraint on the Medigap schema, as shown in Table 6.5b. The actual data was inconsistent so no additional inconsistency was injected. We use twelve natural aggregation queries on the Medigap database that involve the aggregation operators `COUNT(*)`, `COUNT(`A`)`, and `SUM(`A`)`. We refer to these as $(Q_1^m, \cdots, Q_{12}^m)$. The first six queries contain no grouping, while the rest of them do. The definitions of these queries are given in Table 6.6.

Table 6.6: Aggregation Queries on Medigap database

#	Query
Q_1^m	SELECT COUNT(*) FROM OBS WHERE OBS.Name = 'Continental General Insurance Company'
Q_2^m	SELECT COUNT(*) FROM PBZ, SPT WHERE PBZ.Description = SPT.Simple_plantype_name AND SPT.Contract_year = 2020 AND SPT.Simple_plantype = 'B'
Q_3^m	SELECT SUM(PBZ.Over65) FROM PBZ WHERE PBZ.State_name = 'Wisconsin' AND PBZ.County_name = 'GREEN LAKE'
Q_4^m	SELECT SUM(PBZ.Community) FROM PBZ WHERE PBZ.State_name = 'New York'
Q_5^m	SELECT COUNT(PR.Premium_range) FROM PR
Q_6^m	SELECT COUNT(PR.Premium_range) FROM PT, PR WHERE PT.State_abbrev = PR.State_abbrev AND PT.Plan_type = PR.Plan_type AND PT.Contract_year = PR.Contract_year AND PT.Contract_year = 2020 AND PT.Simple_plantype = 'K'
Q_7^m	SELECT SPT.Contract_year, COUNT(*) FROM SPT GROUP BY SPT.Contract_year ORDER BY SPT.Contract_year DESC
Q_8^m	SELECT PBZ.State_name, COUNT(*) FROM PBZ GROUP BY PBZ.State_name
Q_9^m	SELECT PBZ.Zip, SUM(PBZ.Community) FROM PBZ WHERE PBZ.State_name = 'New York' GROUP BY PBZ.Zip
Q_{10}^m	SELECT TOP 10 PBS.State_name, SPT.Contract_year, SUM(PBS.Under65) FROM PBS, SPT WHERE SPT.Simple_plantype_name = PBS.Description AND SPT.Simple_plantype = 'A' AND SPT.Language_id = 1 GROUP BY PBS.State_name, SPT.Contract_year ORDER BY PBS.State_name
Q_{11}^m	SELECT PR.Age_category, COUNT(PR.Premium_range) FROM PR GROUP BY PR.Age_category ORDER BY PR.Age_category
Q_{12}^m	SELECT TOP 10 PT.Simple_plantype, COUNT(PR.Premium_range) FROM PT, PR WHERE PT.State_abbrev = PR.State_abbrev AND PT.Plan_type = PR.Plan_type AND PT.Contract_year = PR.Contract_year and PT.Contract_year = 2020 GROUP BY PT.Simple_plantype ORDER BY PT.Simple_plantype

6.2.6 Results on Real-world Database

In Figure 6.13, we plot the overall time taken by CAvSAT to compute the range consistent answers to the twelve aggregation queries on the Medigap database. Since the Medigap schema has functional dependencies and a denial constraint, the encoding of CQA into Weighted Partial MaxSAT instances is based on Reduction 3.2.1. Consequently, the size of the CNF formulas is much larger compared to that of the ones produced by Reduction 3.1.1, resulting in longer encoding times. For all twelve

queries, the encoding time is dominated by the time required to compute the near-violations and hence the γ-clauses. This part of the encoding time is equal for all queries, but the computation time for minimal witnesses depends on the query. The solver takes comparatively minuscule amount of time to compute the consistent answers to the underlying conjunctive query. For the queries $Q_7^m, \cdots, Q_{12}^m$, the *glb*-answer and the *lub*-answer are encoded and then solved for for each consistent group, causing high overhead. The longest evaluation time is taken by queries Q_{10}^m, Q_{12}^m, and Q_6^m since they consist of 10, 10, and 6 consistent groups, respectively.

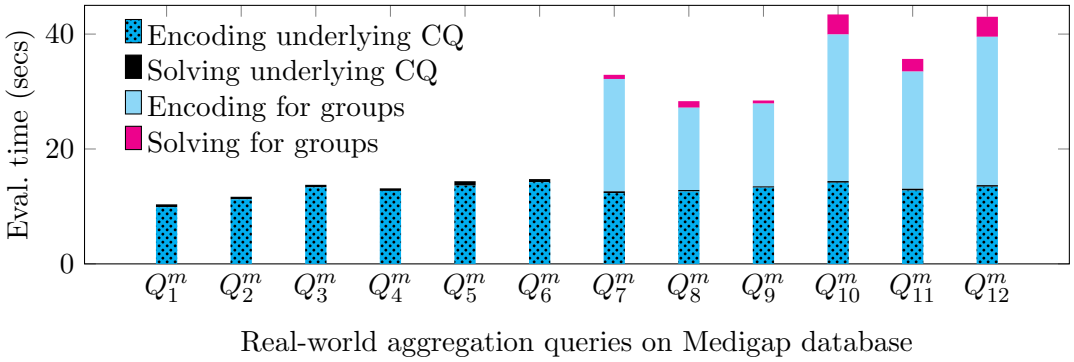


Figure 6.13: Performance of CAvSAT on computing the range consistent answers on a real-world database

For these experiments, we did not compute the range consistent answers from the consistent part of the data first. Thus, for all CNF formulas, the number of variables is equal to the number of tuples in the data (about 61K). The number of clauses, however, varies depending on the query, as shown in Figure 6.14. The query Q_5^m has the highest number of clauses since all tuples in the vwPremiums table are the minimal

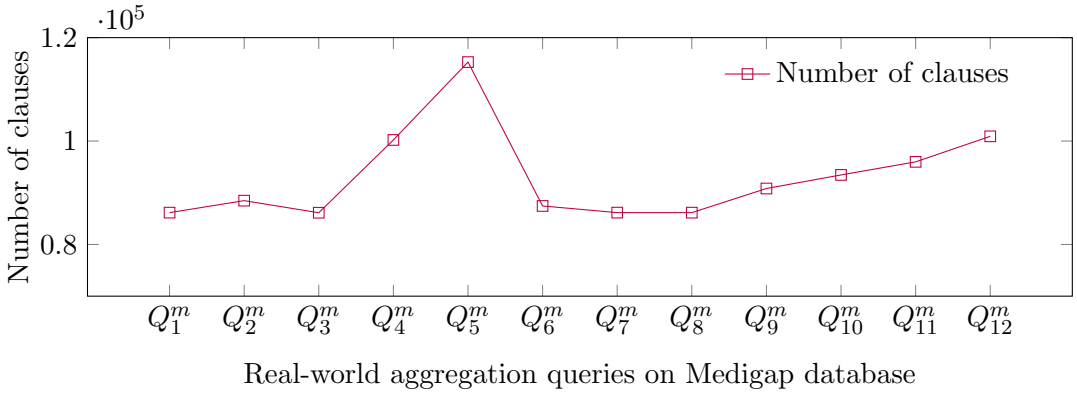


Figure 6.14: Number of clauses in a CNF formula capturing the consistent answers to the underlying conjunctive query

witnesses to its underlying conjunctive query.

The clauses arising from the inconsistency in the data can be constructed independently from the clauses arising from the witnesses to the queries. In the near future, we plan to parallelize their computation to improve CAvSAT's performance.

Chapter 7

Discussion and Concluding Remarks

We designed and implemented CAvSAT, the first SAT-based system for consistent query answering that leverages natural polynomial-time reductions from the problem of computing consistent answers to SAT and its variants. We note that, on non-aggregation queries with first-order rewritable consistent answers, CAvSAT had comparable or even better performance to evaluating the first-order rewritings using a database engine. This finding suggests a potential difference between theory and practice since the study of the first-order rewritability of the consistent answers was motivated from having an efficient evaluation of consistent answers using the database engine alone. We acknowledge that the first-order rewritings used in our experiments were fed directly to the database engine without any further optimizations. Moreover, the recent result from Koutris and Wijsen [61] calls for further experiments that would compare reduction-based approaches such as SAT solving with the theoretically more efficient method of using a Datalog engine for self-join-free conjunctive queries that have

LOGSPACE-computable consistent answers.

We gave natural reductions from range consistent answers to range consistent answers to aggregation queries involving `COUNT(`A`)`, `COUNT(*)`, `SUM(`A`)`, `MIN(`A`)`, and `MAX(`A`)` operators to various optimization variants of SAT. We implemented the SAT solving module in CAvSAT that is tailored for aggregation queries with or without grouping and evaluated its performance using both synthetically generated databases and real-world databases. With this, CAvSAT became the first system that can handle arbitrary aggregation queries with or without grouping whose range consistent answers are not SQL-rewritable. Our experimental evaluation showed that CAvSAT is not only competitive with systems such as ConQuer but is also scalable. The experiments with the Medigap database showed that CAvSAT is capable of computing the range consistent answers to the queries over real-world databases having integrity constraints beyond primary key constraints.

The modular architecture of CAvSAT leverages (some of) the theoretical results in consistent query answering and allows CAvSAT to easily incorporate new efficient approaches for queries that have consistent answers in low complexity classes, e.g., new query-rewriting methods or special polynomial-time algorithms, if developed in the future. Moreover, since CAvSAT uses the SAT solvers as black-boxes, they can be easily replaced with newer more powerful ones in the future. If new Weighted Partial MinSAT solvers are developed, CAvSAT would not have to reduce Weighted Partial MinSAT to Weighted Partial MaxSAT using Kügel's technique resulting in significant improvements in the performance. Similarly, if hierarchical Partial MaxSAT solvers are

built, the range consistent answers to aggregation queries with `MIN(`A`)` and `MAX(`A`)` operators can be computed much faster.

In what follows, we discuss some of the directions for future research that arose during our work on consistent query answering via SAT solving.

Incompatibility of the Reductions for Keys and Denial Constraints

In Chapters 3 and 4, we presented reductions from computing the consistent answers or the range consistent answers of queries to several optimization variants of SAT. We gave different reductions to handle different classes of integrity constraints. Specifically, the reductions based on the key-equal groups of facts can handle only primary keys, while the ones based on minimal violations and near-violations are more general and they support arbitrary denial constraints. The general reductions are significantly more complex and less efficient (as they produce bigger SAT instances) compared to the ones tailored to only primary keys. Moreover, these reductions are not compatible with each other in the following sense. If the set of integrity constraints on the database schema has a mix of primary keys and denial constraints, CAvSAT is forced to use the more general reduction for all constraints, i.e., CAvSAT cannot encode the inconsistency arising due to the primary key violations based on the key-equal groups of facts and use the more general encoding just for the denial constraints, because doing this simply would not work. In other words, CAvSAT is forced to treat all primary keys like denial constraints if there is even a single denial constraint (which is not a primary key constraint) set on the schema. In fact, we do not have special reductions for handling

functional dependencies or equality-generating dependencies. Therefore, the presence of even a single functional dependency compels CAvSAT to use the general reduction for all constraints, affecting the overall performance.

We call this the incompatibility of the reductions with each other and it comes from the following fact. The primary keys are special compared to the rest of the integrity constraints in the sense that all subset repairs w.r.t. a set of primary keys have the same cardinality. This is not true even for functional dependencies, let alone the denial constraints. This property naturally allows us to encode the inconsistency w.r.t. the primary key violations based on the key-equal groups of facts (observe that the size of each repair is, precisely, the total number of key-equal groups of facts). It is an interesting project to work on more efficient reductions designed for specific classes of integrity constraints and investigate their compatibility with each other.

Difficulties with `MIN(`A`)`, `MAX(`A`)`, and `AVG(`A`)`

On the foundational level, the next step in the investigation is to delineate the complexity of the range consistent answers to the queries involving aggregation operators such as `MIN(`A`)`, `MAX(`A`)`, and `AVG(`A`)`. It was shown by Arenas et al. [17] in 2003 that the range consistent answers to an aggregation query with the `COUNT(`A`)` operator can be NP-hard even if the underlying conjunctive query has FO-rewritable consistent answers. In this thesis, we proved a similar result for the `SUM(`A`)` operator, but we note that such a result is still unknown for the other standard aggregation operators. Our reductions for the `MIN(`A`)` and `MAX(`A`)` operators are fundamentally different from the ones for

SUM(A), COUNT(A), and COUNT(*), and on the practical level they are relatively less efficient due to an iterative approach. For MIN(A) and MAX(A), a more efficient natural reduction to a SAT-variant would be worth investigating.

The AVG(A) aggregation operator is tough to handle due to its non-linear behavior. The difficulty appears to be in picking a SAT-like NP-complete problem to which one can naturally reduce the problem of computing the range consistent answers to an aggregation query with the AVG(A) operator. Recall that, the AVG(A) operator is left out in the ConQuer system too, as there is no known result about the SQL-rewritability of the range consistent answers to aggregation queries involving AVG(A).

Consistent Query Answering Beyond Denial Constraints

We note that the SAT-based methods used here are applicable to broader classes of SQL queries, such as queries with nested subqueries, as long as denial constraints are considered. If broader classes of constraints are taken into account, such as universal constraints, then the consistent answers of even conjunctive queries become Π_2^p-hard to compute [18], hence SAT-based methods are not applicable. In that case, Answer Set Programming solvers (for example, DLV [66] or Potassco [43]) have to be used, instead of SAT solvers.

Difficulties in Comparing Consistent Query Answering with Data Cleaning

There is a large body of work on managing inconsistent databases via data cleaning. There are fundamental differences between the framework of repairs and

the consistent answers, and the framework of data cleaning (see [22, Section 6]). In particular, the consistent answers provide the guarantee that each such answer will be found no matter on which repair the query at hand is evaluated, while data cleaning provides no similar guarantee. Data cleaning has the attraction that it produces a single consistent instance but the process need not be deterministic and the instance produced need not even be a repair (i.e., it need not be a maximally consistent instance). This is due to the fact that data cleaning systems are not limited to tuple-deletions as the single allowed database operation to clean the data but they often rely on modifying the tuples or adding new tuples to the database during the cleaning process. Recent data cleaning systems, such as HoloClean [74] and Daisy [45, 44], produce a probabilistic database instance as the output (which again need not be a repair).

As a result, it is not clear how to compare query answers over the database returned by a data cleaning system and the (range) consistent answers computed by a consistent query answering system. In fact, no such comparison is given in the HoloClean [74] and Daisy [45, 44] papers. Comparing the quality of the query answers on a database cleaned using a data cleaning system to the consistent answers using measures such as the F-score may not be fair, as it would require treating one of the two approaches as the gold standard and comparing the other one with it. A comparison based on the evaluation time for answering queries is also not fair as the data cleaning systems such as HoloClean rely on cleaning the data offline, i.e, completely independent of the queries, and thus the time-consuming tasks are done prior to answering the queries, but systems such as CAvSAT, however, need to do the majority of the work after receiving

the input queries. The Daisy system does have an online component in its workflow as the idea of Daisy is to clean only a part of the data that is relevant to the input query, but again, the online work that a consistent query answering system is required to do is typically much higher compared to what a data cleaning system such as Daisy needs to do. Another challenge is that the data cleaning systems may use a variety of signals to detect the errors in the data; for example, HoloClean uses external information such as dictionaries and knowledge bases and quantitative statistics of the input database such as co-occurrences of attribute values in addition to the set of integrity constraints. Consistent Query Answering, however, is a framework designed specifically to deal with only integrity constraint violations. One way to integrate additional signals into the framework of consistent query answering could be to use the notions of *preferred repairs* and *preferred consistent answers*, introduced in [77] and further discussed in [75]. One could let the signals such as external information and/or quantitative statistics on the database dictate the preferences on the facts of the database and then consider computing preferred consistent answers to the queries. However, for certain families of preferred repairs, e.g., globally optimal preferred repairs, the problem of computing the preferred consistent answers can be Π_2^p-hard [75] and thus the SAT-based approaches will not be applicable. As a result, it is an interesting project, however, left for future research, to develop a methodology and carry out a fair comparison on a level playing field between systems for data cleaning and systems for consistent query answering.

Bibliography

[1] CaDiCaL simplified satisfiability solver. `http://fmv.jku.at/cadical/`.

[2] The four V's of big data. `https://www.ibmbigdatahub.com/infographic/four-vs-big-data`.

[3] Medigap database for medicare health and drug plans. `https://www.medicare.gov/download/downloaddb.asp`.

[4] *Proc. of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*. ACM, 2008.

[5] *Scalable Uncertainty Management - 4th International Conference, SUM 2010, Toulouse, France, September 27-29, 2010. Proc.*, volume 6379 of *Lecture Notes in Computer Science*. Springer, 2010.

[6] Food inspections, city of Chicago. https://data.cityofchicago.org/Health-Human-Services/Food-Inspections/4ijn-s7e5, Aug 2011.

[7] New york city restaurant inspection results, department of health and mental hy-

giene (DOHMH). https://data.cityofnewyork.us/Health/DOHMH-New-York-City-Restaurant-Inspection-Results/43nn-pn8j, Aug 2014.

[8] *Proc. of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 2016.

[9] *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proc.*, volume 9710 of *Lecture Notes in Computer Science*. Springer, 2016.

[10] *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020.

[11] *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*. EasyChair, 2020.

[12] *Proc. of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 2020.

[13] Foto N. Afrati and Phokion G. Kolaitis. Answering aggregate queries in data exchange. In *Proc. of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada* [4], pages 129–138.

[14] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. Fast and simple relational processing of uncertain data. In *2008 IEEE 24th International Conference on Data Engineering*, pages 983–992, 2008.

[15] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proc. of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, pages 68–79, New York, NY, USA, 1999. ACM.

[16] Marcelo Arenas, Leopoldo Bertossi, Jan Chomicki, Xin He, Vijay Raghavan, and Jeremy Spinrad. Scalar aggregation in inconsistent databases. *Theoretical Computer Science*, 296(3):405–434, 2003. Database Theory.

[17] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *TPLP*, 3(4-5):393–424, 2003.

[18] Sebastian Arming, Reinhard Pichler, and Emanuel Sallinger. Complexity of repair checking and consistent query answering. In Martens and Zeume [72], pages 21:1–21:18.

[19] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proc. of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[20] Pablo Barceló and Leopoldo E. Bertossi. Logic programs for querying inconsistent

databases. In *Practical Aspects of Declarative Languages, 5th International Symposium, PADL 2003, New Orleans, LA, USA, January 13-14, 2003, Proc.*, pages 208–222, 2003.

[21] Philip A. Bernstein and Laura M. Haas. Information integration in the enterprise. *Commun. ACM*, 51(9):72–79, September 2008.

[22] Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[23] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[24] Balder Cate, Gaëlle Fontaine, and Phokion G. Kolaitis. On the data complexity of consistent query answering. *Theor. Comp. Sys.*, 57(4):843–891, November 2015.

[25] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Computing consistent query answers using conflict hypergraphs. In *Proc. of the Thirteenth ACM International Conference on Information and Knowledge Management*, CIKM '04, pages 417–426, New York, NY, USA, 2004. ACM.

[26] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Hippo: A system for computing consistent answers to a class of sql queries. In *Advances in Database Technology - EDBT 2004*, pages 841–844, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[27] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. Hippo: A system for computing consistent answers to a class of SQL queries. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proc.*, pages 841–844, 2004.

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

[29] E. F. Codd. Relational completeness of data base sublanguages. *Research Report / RJ / IBM / San Jose, California*, RJ987, 1972.

[30] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[31] Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Principles and Practice of Constraint Programming – CP 2011*, pages 225–239, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[32] Akhil A. Dixit. CAvSAT: A system for query answering over inconsistent databases. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1823–1825, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Akhil A. Dixit and Phokion G. Kolaitis. A SAT-based system for consistent query

answering. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proc.*, volume 11628 of *Lecture Notes in Computer Science*, pages 117–135. Springer, 2019.

[34] Akhil A. Dixit and Phokion G. Kolaitis. A SAT-based system for consistent query answering. *CoRR*, abs/1905.02828, 2019.

[35] Akhil A. Dixit and Phokion G. Kolaitis. CAvSAT: Answering aggregation queries over inconsistent databases via SAT solving. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2701–2705. ACM, 2021.

[36] Akhil A. Dixit and Phokion G. Kolaitis. Consistent answers of aggregation queries using SAT solvers. *CoRR*, abs/2103.03314, 2021.

[37] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. Querying and repairing inconsistent numerical databases. *ACM Trans. Database Syst.*, 35(2):14:1–14:50, 2010.

[38] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. Range-consistent answers of aggregate queries under aggregate constraints. In *Scalable Uncertainty Management - 4th International Conference, SUM 2010, Toulouse, France, September 27-29, 2010. Proc.* [5], pages 163–176.

[39] Ariel Fuxman. *Efficient Query Processing over Inconsistent Databases*. PhD thesis, Department of Computer Science, University of Toronto, 2007.

[40] Ariel Fuxman, Elham Fazli, and Renée J. Miller. ConQuer: Efficient management of inconsistent databases. In *Proc. of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 155–166, New York, NY, USA, 2005. ACM.

[41] Ariel Fuxman, Diego Fuxman, and Renée J. Miller. ConQuer: A system for efficient querying over inconsistent databases. In *Proc. of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 1354–1357. VLDB Endowment, 2005.

[42] Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.*, 73(4):610–635, June 2007.

[43] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[44] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. Cleaning denial constraint violations through relaxation. In *Proc. of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020* [12], pages 805–815.

[45] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. Query-driven repair of functional dependency violations. In *36th IEEE International*

Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020 [10], pages 1886–1889.

[46] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Trans. on Knowl. and Data Eng.*, 15(6):1389–1408, November 2003.

[47] Luca Grieco, Domenico Lembo, Riccardo Rosati, and Marco Ruzzi. Consistent query answering under key and exclusion dependencies: Algorithms and experiments. In *Proc. of the 14th ACM International Conference on Information and Knowledge Management*, CIKM '05, pages 792–799, New York, NY, USA, 2005. ACM.

[48] Paolo Guagliardo and Leonid Libkin. Making SQL queries correct on incomplete databases: A feasibility study. In *Proc. of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016* [8], pages 211–223.

[49] J. Hellerstein. Quantitative data cleaning for large databases. 2008.

[50] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th Intern. Conference, Bordeaux, France, July 5-8, 2016, Proc.* [9], pages 228–245.

[51] Ihab F. Ilyas and Xu Chu. Trends in cleaning relational data: Consistency and deduplication. *Found. Trends databases*, 5(4):281–393, October 2015.

[52] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[53] Rajeev Kohli, Ramesh Krishnamurti, and Prakash Mirchandani. The minimum satisfiability problem. *SIAM J. Discret. Math.*, 7(2):275–283, May 1994.

[54] Phokion G. Kolaitis and Enela Pema. A dichotomy in the complexity of consistent query answering for queries with two atoms. *Inf. Process. Lett.*, 112(3):77–85, January 2012.

[55] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. Efficient querying of inconsistent databases with binary integer programming. *PVLDB*, 6(6):397–408, 2013.

[56] Egor V. Kostylev and Juan L. Reutter. Complexity of answering counting aggregate queries over dl-lite. *J. Web Semant.*, 33:94–111, 2015.

[57] Paraschos Koutris and Jef Wijsen. The data complexity of consistent query answering for self-join-free conjunctive queries under primary key constraints. In *Proc. of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '15, pages 17–29, New York, NY, USA, 2015. ACM.

[58] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys. *SIGMOD Rec.*, 45(1):15–22, June 2016.

[59] Paraschos Koutris and Jef Wijsen. Consistent query answering for self-join-free conjunctive queries under primary key constraints. *ACM Trans. Database Syst.*, 42(2):9:1–9:45, June 2017.

[60] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with negated atoms. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, SIGMOD/PODS '18, page 209–224, New York, NY, USA, 2018. Association for Computing Machinery.

[61] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in logspace. In Pablo Barceló and Marco Calautti, editors, *22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal*, volume 127 of *LIPIcs*, pages 23:1–23:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[62] Paraschos Koutris and Jef Wijsen. First-order rewritability in consistent query answering with respect to multiple keys. PODS'20, page 113–129, New York, NY, USA, 2020. Association for Computing Machinery.

[63] Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in datalog. *Theory Comput. Syst.*, 65(1):122–178, 2021.

[64] Adrian Kügel. Natural max-sat encoding of min-sat. In *Revised Selected Papers*

of the 6th International Conference on Learning and Intelligent Optimization - Volume 7219, LION 6, page 431–436, Berlin, Heidelberg, 2012. Springer-Verlag.

[65] Domenico Lembo, Riccardo Rosati, and Marco Ruzzi. On the first-order reducibility of unions of conjunctive queries over inconsistent databases. In *Proc. of the 2006 International Conference on Current Trends in Database Technology*, EDBT'06, pages 358–374, Berlin, Heidelberg, 2006. Springer-Verlag.

[66] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[67] Chu-Min Li, Zhu Zhu, Felip Manyà, and Laurent Simon. Minimum satisfiability and its applications. In *Proc. of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume One*, IJCAI'11, page 605–610. AAAI Press, 2011.

[68] Marco Manna, Francesco Ricca, and Giorgio Terracina. Consistent query answering via ASP from different perspectives: Theory and practice. *CoRR*, abs/1107.4570, 2011.

[69] Marco Manna, Francesco Ricca, and Giorgio Terracina. Taming primary key violations to query large inconsistent data. *CoRR*, abs/1507.06103, 2015.

[70] Mónica Caniupán Marileo and Leopoldo E. Bertossi. The consistency extractor system: Querying inconsistent databases using answer set programs. In *Scalable*

Uncertainty Management, First International Conference, SUM 2007, Washington, DC, USA, October 10-12, 2007, Proc., pages 74–88, 2007.

[71] Mónica Caniupán Marileo and Leopoldo E. Bertossi. The consistency extractor system: Answer set programs for consistent query answering in databases. *Data Knowl. Eng.*, 69(6):545–572, 2010.

[72] Wim Martens and Thomas Zeume, editors. *19th International Conference on Database Theory, ICDT 2016, Bordeaux, France, March 15-18, 2016*, volume 48 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[73] Peter Oostema, Ruben Martins, and Marijn Heule. Coloring unit-distance strips using SAT. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020* [11], pages 373–389.

[74] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, August 2017.

[75] Slawek Staworko, Jan Chomicki, and Jerzy Marcinkowski. Prioritized repairing and consistent query answering in relational databases. *Ann. Math. Artif. Intell.*, 64(2-3):209–246, 2012.

[76] Slawomir Staworko and Jan Chomicki. Priority-based conflict resolution in inconsistent relational databases. *CoRR*, abs/cs/0506063, 2005.

[77] Slawomir Staworko and Jan Chomicki. Consistent query answers in the presence of universal constraints. *Inf. Syst.*, 35(1):1–22, January 2010.

[78] Michael Stonebraker, Daniel Bruckner, Ihab F. Ilyas, George Beskales, Mitch Cherniack, Stanley B. Zdonik, Alexander Pagan, and Shan Xu. Data curation at scale: The data tamer system. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[79] Balder ten Cate, Gaëlle Fontaine, and Phokion G. Kolaitis. On the data complexity of consistent query answering. In *Proc. of the 15th International Conference on Database Theory*, ICDT '12, pages 22–33, New York, NY, USA, 2012. ACM.

[80] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 137–146, New York, NY, USA, 1982. Association for Computing Machinery.

[81] Moshe Y. Vardi. Symbolic techniques in propositional satisfiability solving. In *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 2–3, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[82] Jef Wijsen. Consistent query answering under primary keys: A characterization of tractable queries. In *Proc. of the 12th International Conference on Database Theory*, ICDT '09, pages 42–52, New York, NY, USA, 2009. ACM.

[83] Jef Wijsen. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In *Proc. of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, pages 179–190, New York, NY, USA, 2010. ACM.

[84] Jef Wijsen. A remark on the complexity of consistent conjunctive query answering under primary key violations. *Inf. Process. Lett.*, 110(21):950–955, October 2010.

[85] Jef Wijsen. A remark on the complexity of consistent conjunctive query answering under primary key violations. *Information Processing Letters*, 110(21):950 – 955, 2010.

[86] Jef Wijsen. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.*, 37(2):9:1–9:35, June 2012.

[87] Jef Wijsen. Charting the tractability frontier of certain conjunctive query answering. In *Proc. of the 32Nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '13, pages 189–200, New York, NY, USA, 2013. ACM.

[88] Zhu Zhu, Chu-Min Li, Felip Manyà, and Josep Argelich. A new encoding from minsat into maxsat. In *Principles and Practice of Constraint Programming*, pages 455–463, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

Appendix A

Definitions of TPC-H Queries used in the Experiments

Table A.1: TPC-H-inspired aggregation queries without grouping

#	Query
Q'_1	SELECT SUM(LINEITEM.L_QUANTITY) FROM LINEITEM WHERE LINEITEM.L_SHIPDATE <= dateadd(dd, -90, cast('1998-12-01' as datetime)) AND LINEITEM.L_RETURNFLAG = 'N' AND LINEITEM.L_LINESTATUS = 'F'
Q'_3	SELECT SUM(LINEITEM.L_EXTENDEDPRICE*(1-LINEITEM.L_DISCOUNT)) FROM CUSTOMER, ORDERS, LINEITEM WHERE CUSTOMER.C_MKTSEGMENT = 'BUILDING' AND CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY AND ORDERS.O_ORDERDATE < '1995-03-15' AND LINEITEM.L_SHIPDATE > '1995-03-15' AND LINEITEM.L_ORDERKEY = 988226 AND ORDERS.O_ORDERDATE = '1995-02-01' AND ORDERS.O_SHIPPRIORITY = 0
Q'_4	SELECT COUNT(*) FROM ORDERS WHERE ORDERS.O_ORDERDATE >= '1993-07-01' AND ORDERS.O_ORDERDATE < dateadd(mm,3, cast('1993-07-01' as datetime)) AND ORDERS.O_ORDERPRIORITY = '1-URGENT'
Q'_5	SELECT SUM(LINEITEM.L_EXTENDEDPRICE*(1-LINEITEM.L_DISCOUNT)) FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION WHERE CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY AND LINEITEM.L_SUPPKEY = SUPPLIER.S_SUPPKEY AND CUSTOMER.C_NATIONKEY = SUPPLIER.S_NATIONKEY AND SUPPLIER.S_NATIONKEY = NATION.N_NATIONKEY AND NATION.N_REGIONKEY = REGION.R_REGIONKEY AND REGION.R_NAME = 'ASIA' AND ORDERS.O_ORDERDATE >= '1994-01-01' AND ORDERS.O_ORDERDATE < DATEADD(YY, 1, cast('1994-01-01' as datetime)) AND NATION.N_NAME = 'INDIA'
Q'_6	SELECT SUM(LINEITEM.L_EXTENDEDPRICE*LINEITEM.L_DISCOUNT) FROM LINEITEM WHERE LINEITEM.L_SHIPDATE >= '1994-01-01' AND LINEITEM.L_SHIPDATE < dateadd(yy, 1, cast('1994-01-01' as datetime)) AND LINEITEM.L_DISCOUNT BETWEEN .06 - 0.01 AND .06 + 0.01 AND LINEITEM.L_QUANTITY < 24
Q'_{10}	SELECT SUM(LINEITEM.L_EXTENDEDPRICE*(1-LINEITEM.L_DISCOUNT)) FROM CUSTOMER, ORDERS, LINEITEM, NATION WHERE CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY AND ORDERS.O_ORDERDATE >= '1993-10-01' AND ORDERS.O_ORDERDATE < dateadd(mm, 3, cast('1993-10-01' as datetime)) AND LINEITEM.L_RETURNFLAG = 'R' AND CUSTOMER.C_NATIONKEY = NATION.N_NATIONKEY AND CUSTOMER.C_CUSTKEY = 77296 AND CUSTOMER.C_NAME = 'Customer#000077296' AND CUSTOMER.C_ACCTBAL = 1250.65 AND CUSTOMER.C_PHONE = '12-248-307-9719' AND NATION.N_NAME = 'BRAZIL'
Q'_{12}	SELECT COUNT(*) FROM ORDERS, LINEITEM WHERE ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY AND LINEITEM.L_SHIPMODE = 'MAIL' AND (ORDERS.O_ORDERPRIORITY = '1-URGENT' OR ORDERS.O_ORDERPRIORITY = '2-HIGH') AND LINEITEM.L_COMMITDATE < LINEITEM.L_RECEIPTDATE AND LINEITEM.L_SHIPDATE < LINEITEM.L_COMMITDATE AND LINEITEM.L_RECEIPTDATE >= '1994-01-01' AND LINEITEM.L_RECEIPTDATE < dateadd(mm, 1, cast('1995-09-01' as datetime))

Continued on next page ⟶

⟵ Continued from previous page

Q'_{14}	SELECT SUM(LINEITEM.L_EXTENDEDPRICE*(1-LINEITEM.L_DISCOUNT)) FROM LINEITEM, PART WHERE LINEITEM.L_PARTKEY = PART.P_PARTKEY AND LINEITEM.L_SHIPDATE >= '1995-09-01' AND LINEITEM.L_SHIPDATE < dateadd(mm, 1, '1995-09-01') AND PART.P_TYPE LIKE 'PROMO%%'
Q'_{19}	SELECT SUM(LINEITEM.L_EXTENDEDPRICE*(1-LINEITEM.L_DISCOUNT)) FROM LINEITEM, PART WHERE (PART.P_PARTKEY = LINEITEM.L_PARTKEY AND PART.P_BRAND = 'Brand#12' AND PART.P_CONTAINER IN ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG') AND LINEITEM.L_QUANTITY >= 1 AND LINEITEM.L_QUANTITY <= 1 + 10 AND PART.P_SIZE BETWEEN 1 AND 5 AND LINEITEM.L_SHIPMODE IN ('AIR', 'AIR REG') AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN PERSON') OR (PART.P_PARTKEY = LINEITEM.L_PARTKEY AND PART.P_BRAND ='Brand#23' AND PART.P_CONTAINER IN ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK') AND LINEITEM.L_QUANTITY >= 10 AND LINEITEM.L_QUANTITY <= 10 + 10 AND PART.P_SIZE BETWEEN 1 AND 10 AND LINEITEM.L_SHIPMODE IN ('AIR', 'AIR REG') AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN PERSON') OR (PART.P_PARTKEY = LINEITEM.L_PARTKEY AND PART.P_BRAND = 'Brand#34' AND PART.P_CONTAINER IN ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG') AND LINEITEM.L_QUANTITY >= 20 AND LINEITEM.L_QUANTITY <= 20 + 10 AND PART.P_SIZE BETWEEN 1 AND 15 AND LINEITEM.L_SHIPMODE IN ('AIR', 'AIR REG') AND LINEITEM.L_SHIPINSTRUCT = 'DELIVER IN PERSON')

Table A.2: TPC-H-inspired aggregation queries with grouping

#	Query
Q_1	SELECT LINEITEM.L_RETURNFLAG, LINEITEM.L_LINESTATUS, SUM(LINEITEM.L_QUANTITY) FROM LINEITEM WHERE LINEITEM.L_SHIPDATE <= dateadd(dd, -90, cast('1998-12-01' as datetime)) GROUP BY LINEITEM.L_RETURNFLAG, LINEITEM.L_LINESTATUS
Q_3	SELECT TOP 10 LINEITEM.L_ORDERKEY, SUM(LINEITEM.L_EXTENDEDPRICE), ORDERS.O_ORDERDATE, ORDERS.O_SHIPPRIORITY FROM CUSTOMER, ORDERS, LINEITEM WHERE CUSTOMER.C_MKTSEGMENT = 'BUILDING' AND CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY AND ORDERS.O_ORDERDATE < '1995-03-15' AND LINEITEM.L_SHIPDATE > '1995-03-15' GROUP BY LINEITEM.L_ORDERKEY, ORDERS.O_ORDERDATE, ORDERS.O_SHIPPRIORITY
Q_4	SELECT ORDERS.O_ORDERPRIORITY, COUNT(*) AS O_COUNT FROM ORDERS WHERE ORDERS.O_ORDERDATE >= '1993-07-01' AND ORDERS.O_ORDERDATE < dateadd(mm,3, cast('1993-07-01' as datetime)) GROUP BY ORDERS.O_ORDERPRIORITY
Q_5	SELECT NATION.N_NAME, SUM(LINEITEM.L_EXTENDEDPRICE*(1-LINEITEM.L_DISCOUNT)) AS REVENUE FROM CUSTOMER, ORDERS, LINEITEM, SUPPLIER, NATION, REGION WHERE CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY AND LINEITEM.L_SUPPKEY = SUPPLIER.S_SUPPKEY AND CUSTOMER.C_NATIONKEY = SUPPLIER.S_NATIONKEY AND SUPPLIER.S_NATIONKEY = NATION.N_NATIONKEY AND NATION.N_REGIONKEY = REGION.R_REGIONKEY AND REGION.R_NAME = 'ASIA' AND ORDERS.O_ORDERDATE >= '1994-01-01' AND ORDERS.O_ORDERDATE < DATEADD(YY, 1, cast('1994-01-01' as datetime)) GROUP BY NATION.N_NAME
Q_{10}	SELECT TOP 20 CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME, SUM(LINEITEM.L_EXTENDEDPRICE * (1 - LINEITEM.L_DISCOUNT)) AS REVENUE, CUSTOMER.C_ACCTBAL, NATION.N_NAME, CUSTOMER.C_ADDRESS, CUSTOMER.C_PHONE FROM CUSTOMER, ORDERS, LINEITEM, NATION WHERE CUSTOMER.C_CUSTKEY = ORDERS.O_CUSTKEY AND LINEITEM.L_ORDERKEY = ORDERS.O_ORDERKEY AND ORDERS.O_ORDERDATE>= '1993-10-01' AND ORDERS.O_ORDERDATE < dateadd(mm, 3, cast('1993-10-01' as datetime)) AND LINEITEM.L_RETURNFLAG = 'R' AND CUSTOMER.C_NATIONKEY = NATION.N_NATIONKEY GROUP BY CUSTOMER.C_CUSTKEY, CUSTOMER.C_NAME, CUSTOMER.C_ACCTBAL, CUSTOMER.C_PHONE, NATION.N_NAME, CUSTOMER.C_ADDRESS
Q_{12}	SELECT LINEITEM.L_SHIPMODE, COUNT(*) AS HIGH_LINE_COUNT FROM ORDERS, LINEITEM WHERE ORDERS.O_ORDERKEY = LINEITEM.L_ORDERKEY AND LINEITEM.L_SHIPMODE IN ('MAIL','SHIP') AND (ORDERS.O_ORDERPRIORITY = '1-URGENT' OR ORDERS.O_ORDERPRIORITY = '2-HIGH') AND LINEITEM.L_COMMITDATE < LINEITEM.L_RECEIPTDATE AND LINEITEM.L_SHIPDATE < LINEITEM.L_COMMITDATE AND LINEITEM.L_RECEIPTDATE >= '1994-01-01' AND LINEITEM.L_RECEIPTDATE < dateadd(mm, 1, cast('1995-09-01' as datetime)) GROUP BY LINEITEM.L_SHIPMODE