Consistent Answers of Aggregation Queries using SAT Solvers

Akhil A. Dixit
University of California Santa Cruz
akadixit@ucsc.edu

Phokion G. Kolaitis
University of California Santa Cruz and IBM Research
kolaitis@ucsc.edu

ABSTRACT
The framework of database repairs and consistent answers to queries is a principled approach to managing inconsistent databases. We describe the first system to compute the consistent answers of general aggregation queries with the COUNT(A), COUNT(*), SUM(A), MIN(A), and MAX(A) operators, and with or without grouping constructs. Our system uses reductions to optimization versions of Boolean satisfiability (SAT) and then leverages powerful SAT solvers. We carry out an extensive set of experiments on both synthetic and real-world data that demonstrate the usefulness and scalability of this approach.

1 INTRODUCTION
The framework of database repairs and consistent query answering, introduced by Arenas, Bertossi, and Chomicki [6], is a principled approach to managing inconsistent databases, i.e., databases that violate one or more integrity constraints on their schema. In this framework, inconsistencies are handled at query time by considering all possible repairs of the inconsistent database, where a repair of an inconsistent database $I$ is a consistent database $J$ that differs from $I$ in a "minimal" way. The consistent answers to a query $q$ on a given database $I$ is the intersection of the results of $q$ applied on each repair of $I$. Thus, a consistent answer provides the guarantee that it will be found no matter on what repair the query has been evaluated. Computing the consistent answers can be an intractable problem, because an inconsistent database may have exponentially many repairs. In particular, computing the consistent answers of a fixed Select-Project-Join (SPJ) query can be a coNP-complete problem. By now, there is an extensive body of work on the complexity of consistent answers for SPJ queries (see Section 2).

Range Semantics: Concept and Motivation. Aggregation queries are the most frequently asked queries; they are of the form

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

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 SPJ query $q$ expressed in SQL. A scalar aggregation query is an aggregation query without a GROUP BY clause.

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, Arenas et al. [7] introduced the range consistent answers.

Let $Q$ be a scalar aggregation query and let $\Sigma$ be a set of integrity constraints. The set of possible answers to $Q$ on an inconsistent instance $I$ w.r.t. $\Sigma$ is the set of the answers to $Q$ over all repairs of $I$ w.r.t. $\Sigma$, i.e., $\text{Poss}(Q, \Sigma) = \{ Q(J) | J \text{ is a repair of } I \text{ w.r.t. } \Sigma \}$. By definition, the range consistent answers to $Q$ on $I$ is the interval $[\text{glb}(Q, I), \text{lub}(Q, I)]$, 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 $I$.

Summary of Contributions. In this paper, we report on and evaluate the performance of the AggCAvSAT (Aggregate Consistent Answers via Satisfiability Testing), which is an enhanced version of CAvSAT and is also the first system that is capable to compute the range consistent answers to all aggregation queries involving the operators SUM(A), COUNT(A), or COUNT(*) with or without grouping.

We first corroborate the need for a system that goes well beyond ConQuer by showing that there is an aggregation query $Q$ involving $\text{SUM}(A)$ such that the consistent answers of the underlying SPJ query

$\text{SUM}(\text{ACCOUNTS.BAL}) \text{ FROM } \text{ACCOUNTS, CUSTACC WHERE ACCOUNTS.ACCID} = \text{CUSTACC.ACCID AND CUSTACC.CID} = 'C2'\text{;$\}$

on the instance in Table 1 is the interval $[900, 2200]$. The meaning is that no matter how the database $I$ is repaired, the answer to the query is guaranteed to be in the range between 900 and 2200.

Arenas et al. [8] focused on scalar aggregation queries only. Fuxman, Fazli, and Miller [22] extended the notion of range consistent answers to aggregation queries with grouping (see Section 3).

Range semantics have become the standard semantics of aggregation queries in the framework of database repairs (see [12, Section 5.6]). Furthermore, range semantics have been adapted to give semantics to aggregation queries in several other contexts, including data exchange [3] and ontologies [36]. Finally, range semantics have been suggested as an alternative way to overcome some of the issues arising from SQL’s handling of null values [30].

Earlier Systems for Consistent Query Answering. Several academic prototype systems for consistent query answering have been developed [8, 11, 15, 20, 22, 23, 28, 35, 42, 43]. These systems use different approaches, including logic programming [11, 28], compact representations of repairs [14], or reductions to solvers [20, 35, 42]. In particular, in [20], we reported on CAvSAT, a system that at that time was able to compute the consistent answers of unions of SPJ queries w.r.t. denial constraints (which include functional dependencies as a special case) via reductions to SAT solvers. Among all these systems, however, only the ConQuer system by Fuxman et al. [22, 23] 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 SPJ query belongs to the class called $C_{\text{forest}}$. 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 $I$ can be obtained by directly evaluating $Q'$ on $I$. 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.
There is a large body of work on managing inconsistent databases via data cleaning. There are natural but are much more sophisticated than the reductions used in [20] to reduce the consistent answers of (SP) queries to SAT. After the reductions have been carried out, AggCAvSAT deploys powerful SAT solvers, such as the MaxHS solver [18], to compute the range consistent answers of aggregation queries. Furthermore, AggCAvSAT can handle databases that are inconsistent not only w.r.t. key constraints, but also w.r.t. arbitrary denial constraints, a much broader class of constraints.

An extensive experimental evaluation of AggCAvSAT is reported in Section 6. We carried out a suite of experiments on both synthetic and real-world databases, and for a variety of aggregation queries with and without grouping. The synthetic databases were generated using two different methods: (a) the DBGen tool of TPC-H was used to generate consistent data and then inconsistencies were injected artificially; (b) the PDBench inconsistent database generator from the probabilistic database management system MayBMS [5] was used. The experiments demonstrated the scalability of AggCAvSAT along both the size of the data and the degree of inconsistency in the data. Note that AggCAvSAT was also competitive in comparison to ConQuer (especially when the degree of inconsistency was not excessive), even though the latter is tailored to only handle a restricted class of aggregation queries whose range consistent answers are SQL-rewritable.

**Consistent Answers vs. Data Cleaning.** There is a large body of work on managing inconsistent databases via data cleaning. There are fundamental differences between the framework of the consistent answers and the framework of data cleaning (see [12, 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). Recent data cleaning systems, such as HoloClean [45] and Daisy [26, 27], produce a probabilistic database instance as the output (which again need not be a repair).

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 [45] and Daisy [26, 27] papers. At the performance level, the data cleaning approaches remove inconsistencies in the data offline, hence the time-consuming tasks are done prior to answering the queries; in contrast, systems for consistent query answering work online. It is an interesting project, 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.

## 2 PRELIMINARIES

**Integrity Constraints and Database Queries.** A relational 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; they can be identified with their positions, starting with 1 for the first attribute, $n$ for the last attribute, and $a_i$ for the $i$th attribute, for all $i \in \{ 1, ..., n \}$. An $\mathcal{R}$-instance is a collection $I$ of finite relations $R^I$, one for each relation symbol $R$ in $\mathcal{R}$. An expression of the form $R^I(a_1, ..., a_n)$ is a fact of the instance $I$ if $(a_1, ..., a_n) \in R^I$. A key is a minimal subset $X$ of $\mathcal{R}(R)$ such that the functional dependency $X \rightarrow \mathcal{R}(R)$ holds.

Starting with Codd’s seminal work [16, 17], first-order logic has been successfully used as a database query language; in fact, it forms the core of SQL. A conjunctive query is expressible by a first-order formula of the form $q(z) := \exists w \ (R_1(x_1) \land ... \land R_m(x_m))$, where each $x_i$ is a tuple of variables and constants, $z$ and $w$ are tuples of variables with no variable in common, and the variables in $x_1, \cdots, x_m$ appear in exactly one of the tuples $z$ and $w$. A conjunctive query with no free variables (i.e., all variables are existentially quantified) is a boolean query, while a conjunctive query with $k$ free variables in $z$ is a $k$-ary query. Conjunctive queries are also known as select-project-join (SPJ) queries with equi-joins, and are among the most frequently asked queries. For example, on the instance $I$ from Table 1, the binary conjunctive query $q(z, x) := \exists w \ (\text{CUST}(w, x, y) \land \text{CUSTACC}(w, z))$ returns the set of all pairs $(z, x)$ such that $z$ is an account ID of an account owned by customer named $x$.

Equivalently, this query can be expressed in SQL as

```sql
SELECT CUSTACC.ACCID, CUST.CNAME
FROM CUST, CUSTACC
WHERE CUST.CID = CUSTACC.CID
```

A union of conjunctive queries is expressible by a disjunction $q(z) := q_1 \lor \cdots \lor q_n$ of conjunctive queries, where all conjunctive queries $q_i$ have the same arity. Unions of conjunctive queries are strictly more expressive than conjunctive queries.

**Database Repairs and Consistent Answers.** Let $\Sigma$ be a set of integrity constraints on a database schema $\mathcal{R}$. An $\mathcal{R}$-instance $I$ is consistent if $I \models \Sigma$, i.e., $I$ satisfies every constraint in $\Sigma$; otherwise, $I$ is inconsistent. For example, let $I$ be the instance depicted in Table 1.

### Table 1: Running example – an inconsistent database instance $I$ (primary key attributes are underlined)

| CUSTOMER | CID | CNAME | CITY | ACCOUNTS | ACCID | TYPE | CITY | BAL |
|----------|-----|-------|------|----------|-------|------|------|-----|
| C1       | John LA $f_1$ |
| C2       | Mary LA $f_2$ |
| C2       | Mary SF $f_3$ |
| C3       | Don SF $f_4$ |
| C4       | Jen LA $f_5$ |
| A1       | Checking LA $f_6$ |
| A2       | Checking LA $f_7$ |
| A3       | Saving SJ $f_8$ |
| A4       | Saving SJ $f_9$ |
| C1       | A1 $f_{11}$ |
| C2       | A2 $f_{12}$ |
| C2       | A3 $f_{13}$ |
| C3       | A4 $f_{14}$ |
There are two key constraints, namely, CUST(CID) and ACC(ACCID).

Cons

Clearly, is inconsistent since the facts f₂, f₃ of CUST and facts f₄, f₅ of ACC violate these key constraints.

A repair of an inconsistent instance I w.r.t. Σ is a consistent instance I’ that differs from I in a “minimal” way. Different notions of minimality give rise to different types of repairs (see [12] for a comprehensive survey). Here, we focus on subset repairs, the most extensively studied type of repairs. An instance I’ is a subset repair of an instance I if I’ is a maximal consistent subinstance of I, that is, I’ ⊆ I (where I and I’ are viewed as sets of facts), I’ |= Σ, and there exists no instance I’’ such that I’’ |= Σ and I’ ⊂ I’’ ⊂ I.

Arens et al. [6] used repairs to give rigorous semantics to query SAT book [13]). In particular, significant progress has been made on research on different aspects of boolean satisfiability (see the hand-

formula

arguably the prototypical and the most widely studied NP-complete Boolean Satisfiability and SAT Solvers.

then

Certainty

rewritable [49]. Third, if Certainty is rewritable, or coNP-complete. It is an open problem whether or not

I

result to date is a consistent answer

poss

true on every repair I’ of I. We write Cons(q, I, Σ) to denote the set of all consistent answers to q on I w.r.t. Σ, i.e.,

Cons(q, I, Σ) = ∩{q(I’) : I’ is a repair of I w.r.t. Σ}.

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 is: given an instance I, compute Cons(q, I, Σ); we write Cons(q, Σ) to denote this problem. If q is a boolean query, then computing the consistent answers becomes the decision problem Certainty(q, Σ): given an instance I, is q true on every repair I’ of I w.r.t. Σ? When the constraints in Σ are understood from the context, we will write Cons(q) and Certainty(q) in place of Cons(q, Σ) and Certainty(q, Σ), respectively.

Complexity of Consistent Answers. There has been an extensive study of the consistent answers of conjunctive queries [12, 23, 24, 34, 37, 38, 47–49]. If Σ is a fixed set of key constraints and q is a boolean conjunctive query, then Certainty(q, Σ) is always in coNP, but, depending on the query and the constraints, Certainty(q, Σ) exhibits a variety of behaviors within coNP. The most definitive result to date is a trichotomy theorem by Koutris and Wijsen [37, 38]; it asserts that if q is a self-join-free (no repeated relation symbols) boolean conjunctive query with one key constraint per relation, then Certainty(q) is either SQL-rewritable, or in P but not SQL-rewritable, or coNP-complete. It is an open problem whether or not this trichotomy extends to arbitrary boolean conjunctive queries and to broader classes of constraints (e.g., denial constraints).

We illustrate the trichotomy theorem with three examples. In what follows, the underlined attributes constitute the keys to the relations. First, if q₁ is the query \( \exists x, y, z (R(x, z) \land S(z, y)) \), then Certainty(q₁) is SQL-rewritable [24]. Second, if q₂ is the query \( \exists x, y (R(x, y) \land S(y, x)) \), then Certainty(q₂) is in P, but is not SQL-rewritable [49]. Third, if q₃ is the query \( \exists x, y, z (R(x, y) \land S(x, z)) \), then Certainty(q₃) is coNP-complete [24].

Boolean Satisfiability and SAT Solvers. Boolean Satisfiability (SAT) is arguably the prototypical and the most widely studied NP-complete problem. SAT is the following decision problem: given a boolean formula \( \phi \), is \( \phi \) satisfiable? There has been an extensive body of research on different aspects of boolean satisfiability (see the handbook [13]). In particular, 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” [46]. Typically, a SAT-solver takes a boolean formula \( \phi \) in conjunctive normal form (CNF) as an input and outputs a satisfying assignment for \( \phi \) (if one exists) or tells that the formula \( \phi \) is unsatisfiable. Recall that a formula \( \phi \) is in CNF if it is a conjunction of clauses, where each clause is a disjunction of literals. For example, the formula \((x₁ \lor x₂ \land \lnot x₃) \land (\lnot x₂ \lor x₃) \land (\lnot x₁ \lor x₃)\) has a satisfying assignment \(x₁ = 1, x₂ = 0, x₃ = 0\), and \(x₄ = 1\).

At present, SAT-solvers are capable of solving quickly SAT-instances with millions of clauses and variables. SAT-solvers have been widely used in both academia and industry as general-purpose 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 [10] and CaDiCal [1]. Furthermore, SAT-solvers have been used in solving open problems in mathematics [31, 44]. In [20], we used SAT-solvers to build a prototypical system for consistent query answering, which we called CArSAT. This system can compute consistent answers to unions of conjunctive queries over relational databases that are inconsistent w.r.t. a fixed set of arbitrary denial constraints.

3 RANGE CONSISTENT ANSWERS

Frequently asked database queries often involve one of the standard aggregate operators COUNT(A), COUNT(*), SUM(A), AVG(A), MIN(A), MAX(A), and, possibly, a GROUP BY clause. 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

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

where \( f(A) \) is one of the aforementioned aggregate 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 \( w \) is called the aggregation attribute, and the attributes represented by \( Z \) are called the grouping attributes. A scalar aggregation query is one without a GROUP BY clause.

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 the definition 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. [7] proposed the range consistent answers, as an alternative notion of consistent answers. Let \( Q \) be a scalar aggregation query. The set of possible answers to \( Q \) on an inconsistent instance \( I \) consists of the answers to \( Q \) over all repairs of \( I \), i.e., Poss(\( Q, Σ \)) = \{\( Q(I’) | I’ \text{ is a repair of } I \text{ w.r.t. } Σ \)\}. By definition, the range consistent answers to \( Q \) on \( I \) is the interval [\( \text{glb}(Q, I), \text{lub}(Q, I) \)], where the endpoints of this interval are, respectively, the greatest lower bound (glb) and the least upper bound (lub) of the set Poss(Q, Σ) of possible answers to \( Q \) on \( I \).

For example, the range consistent answers of the query

SELECT SUM(ACCOUNTS.BAL) FROM ACCOUNTS, CUSTACC
WHERE ACCOUNTS.ACCID = CUSTACC.ACCID AND CUSTACC.CID = ’C2’
on the instance in Table 1 is the interval [900, 2200]. The guarantee is that no matter how the database $I$ is repaired, the answer to the query is guaranteed to be in the range between 900 and 2200. Note that, the glb-answer comes from a repair of $I$ that contains the fact $f_9$, while the lub-answer is from a repair that contains the fact $f_8$.

Arenas et al. [8] focused on scalar aggregation queries only. Fuxman, Fazli, and Miller [22] extended the notion of range consistent answers to aggregation queries with grouping, i.e., to queries

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

For such queries, a tuple $(T, [glb, lub])$ is a range consistent answer to $Q$ on $I$, if the following conditions hold:

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

If $Q$ is an aggregation query, Cons($Q$) denotes the problem: given an instance $I$, compute the range semantics of $Q$ on $I$.

Complexity of Range Consistent Answers. Arenas et al. [7] investigated the computational complexity of the range consistent answers for scalar aggregation queries of the form

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

where $f(A)$ is one of the standard aggregation operators and $R(U, A)$ is a relational schema with functional dependencies. The main findings in Arenas et al. [7] can be summarized as follows.

- If the relational schema $R(U, A)$ has at most one functional dependency and $f(A)$ is one of the aggregation operators $\text{MIN}(A)$, $\text{MAX}(A)$, $\text{SUM}(A)$, $\text{COUNT}(*)$, $\text{AVG}(A)$, then the range consistent answers of the query $\text{SELECT } f(A) \text{ FROM } R(U, A)$ is in P.
- There is a relational schema $R(U, A)$ with one key dependency such that computing the range consistent answers of the query $\text{SELECT COUNT}(A) \text{ FROM } R(U, A)$ is an NP-complete problem.
- There is a relational schema $R(U, A)$ with two functional dependencies, such that computing the range consistent answers of the query $\text{SELECT } f(A) \text{ FROM } R(U, A)$ is an 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

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

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 Cons($q$) of the underlying query $q$ is a hard problem, then computing the range consistent answers Cons($Q$) of the aggregation query $Q$ is a hard problem as well. This gives rise to the following question: what can we say about the complexity of the range consistent answers Cons($Q$) of the underlying query $q$ if computing the consistent answers Cons($q$) of the underlying query $q$ is an easy problem?

Fuxman and Miller [24] identified a class, called $C_{\text{forest}}$, of self-join free conjunctive queries whose consistent answers are SQL-rewritable. In his PhD thesis, Fuxman [21] introduced the class $C_{\text{agforest}}$ consisting of all aggregation queries such that the aggregation operator is one of $\text{MIN}(A)$, $\text{MAX}(A)$, $\text{SUM}(A)$, $\text{COUNT}(*)$, the underlying query $q$ is a conjunctive query in $C_{\text{forest}}$, and there is one key constraint for each relation in the underlying query $q$. Fuxman [21] showed that the range consistent answers of every query in $C_{\text{agforest}}$ are SQL-rewritable (earlier, similar results for a proper subclass of $C_{\text{agforest}}$ were obtained by Fuxman, Fazli, and Miller).

It is known that there are self-join free conjunctive queries outside the class $C_{\text{forest}}$ whose consistent answers are SQL-rewritable. In fact, Koutris and Wijsen [38] have characterized the self-join free conjunctive queries whose consistent answers are SQL rewritable. However, the SQL rewritability of aggregation queries beyond those in $C_{\text{agforest}}$ has not been investigated. In the sequel, we show that there exist a self-join free conjunctive query whose consistent answers are SQL-rewritable, but this property is not preserved 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 involving SUM and whose underlying conjunctive query has SQL-rewritable consistent answers. We begin by recalling the definition of the Maximum Cut problem, a fundamental NP-complete problem [32]. We state and prove a helping lemma (Lemma 3.3) before stating the main result in Theorem 3.2.

**Definition 3.1. Maximum Cut.** For an undirected graph $G = (V, E)$, a cut of $G$ is a partition $(S, \bar{S})$ of $V$, where $S \subseteq V$ and $\bar{S} = V \setminus S$. The set of edges with one vertex in $S$ and one vertex in $\bar{S}$ is denoted by $E(S, \bar{S})$, and the size of the cut $(S, \bar{S})$ is $|E(S, \bar{S})|$.

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 3.2.** Let $R$ be a schema with three relations $R_1(A_1, B_1)$, $R_2(A_2, B_2)$, and $R_3(A_1, B_1, A_2, B_2, C)$. Let $Q$ be the following aggregation query:

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

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

$$q(A) := \exists x \exists y \; R_1(x, 'red') \land R_2(y, 'blue') \land R_3(x, 'red', y, 'blue', A).$$

Then the following two statements hold.

1. Cons($q$) is SQL-rewritable.
2. Cons($Q$) is NP-hard.

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

$$q'(A) := \exists x \exists y \exists z \; R_1(x, 'red') \land R_2(y, 'blue') \land R_3(x, 'red', y, 'blue', A)$$

$$\land \forall z (R_1(x, z) \rightarrow z = 'red') \land \forall w (R_2(y, w) \rightarrow w = 'blue')).$$

We will show that for every instance $I$ and every value $a$, we have that $a \in q'(I) \iff a \in \text{Cons}(q, I)$. 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 $a \in q'(I)$, then $a \in q(J)$, for every repair $J$ of $I$, which means that $a \in \text{Cons}(q, I)$. In the other direction, we claim that if $a \in \text{Cons}(q, I)$, then $a \in q'(I)$. Indeed, if $a \notin q'(I)$, then for all $x$ and $y$ such that $R_1(x, 'red') \land R_2(y, 'blue') \land R_3(x, 'red', y, 'blue', a)$, 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 $J$ of $I$ as follows. First, for every $x$, if $'red'$ is the only value $z$ such that $R_1(x, z)$ is a fact of $I$, then put $R_1(x, 'red')$ in $J$; otherwise, pick an element $z'$ of $'red'$ such that $R_1(x, z')$ is a fact of $I$ and put $R_1(x, z')$ in $J$. Second, for every $y$, if $'blue'$ is the only value such that $R_2(y, w)$ is a fact of $I$,
then put $R_2(y,'blue')$ in $J'$; otherwise, pick an element $w' \neq 'blue' such that $R_1(y,w')$ is a fact of $I$ and put $R_2(x,w')$ in $J'$. Third, put every tuple of the relation $R_3$ of $I$ into $J'$. Clearly, $J'$ is a repair of $I$. Moreover, $a \notin q(J')$. Indeed, if $a \in q(J')$, then there are elements $x$ and $y$ such that $J' \models R_1(x, 'red') \land R_2(y, 'blue') \land R_3(x, 'red', y, 'blue', a)$. Since $a \notin q(J')$, 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 $J'$ implies that $R_1(x,'red')$ is not a fact of $J'$, while in the second case, the construction of $J'$ implies that $R_2(y,'blue')$ is not a fact of $J'$; in either case, we have arrived at a contradiction.

To show that $\text{Cons}(Q)$ is NP-hard, consider the following reduction from undirected graphs to $\mathcal{R}$-instances, where $\mathcal{R}$ is the schema with relations $R_1(A_1,B_1)$, $R_2(A_2,B_2)$, and $R_3(A_3,B_1,A_2,B_2,C)$.

**Reduction 3.1.** Let $G$ be an undirected graph of $Q(V,E)$, construct an $\mathcal{R}$-instance $I$ as follows. Let $m = |E| - 1$.

- 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 $I$.
- For each edge $(u,v) \in E$, add tuples $R_3(u,'red',v,'blue',m)$ to $I$.
- 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 $I$.

We will show that the preceding Reduction 3.1 reduces $\text{MAXIMUM CUT}$ to computing the range semantics of the aggregation query $Q$.

For the rest of this section, let $G$ be an undirected graph and $J$ be the database instance constructed from $G$ using Reduction 3.1. We say that a repair $J'$ of $I$ 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 $J'$ or both absent in $J'$. We now prove a useful lemma.

**Lemma 3.3.** For every repair $J$ of $I$, there exists a repair $J'$ of $I$ (not necessarily different from $J$) such that $J'$ produces a red-blue coloring of $G$ and $Q(J') \geq Q(J)$.

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

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

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

By Lemma 3.3, there exists a repair $J$ of $I$ such that $J'$ produces a red-blue coloring of $G$ and $Q(J')$ is the lub-answer in $\text{Cons}(Q,I)$. We will show, that for a non-negative integer $k$, there is a cut $(S,\bar{S})$ of $G$ such that $|E(S,\bar{S})| \geq k$ if and only if there exists a repair $J'$ of $I$ such that $J'$ produces a red-blue coloring of $G$ and $Q(J') \geq k$. Once this is shown, it will follow that $Q(J')$ is NP-hard to even compute the lub-answer in $\text{Cons}(Q,I)$.

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

**4 CONSISTENT ANSWERS VIA SAT SOLVING**

In this section, we give polynomial-time reductions from computing the range consistent answers of aggregation queries to variants of SAT. The reductions in this section assume that the database schema has one key constraint per relation; in Section 5, we show how these reductions can be extended to schemata with arbitrary denial constraints. Our reductions rely on several well-known optimization variants of SAT that we describe next.

- **Weighted MaxSAT** (or WMaxSAT) 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 will write $(l_1 \lor \cdots \lor l_k, w)$ to denote a clause $(l_1 \lor \cdots \lor l_k)$ with weight $w$. 

• Partial MaxSAT (or PMaxSAT) is the maximization variant of SAT in which some clauses of the formula are assigned 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 PMaxSAT instance are not simultaneously satisfiable, then we say that the instance is unsatisfiable. For simplicity, a hard clause \((l_1 \lor \cdots \lor l_k)\) is denoted as \((l_1 \lor \cdots \lor l_k)\).

• Weighted Partial MaxSAT (or WPMaxSAT) is the maximization variant of SAT in which some of the clauses of the formula are assigned 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, WPMaxSAT is a common generalization of both WMaxSAT (no hard clauses) and PMaxSAT (each soft clause has weight one).

Modern solvers, such as MaxHS [18], can efficiently solve large instances of these maximization variants of SAT. Note that these maximization problems have dual minimization problems, called PMinSAT, PMinSAT, and WPMinSAT, respectively. For example, the CNF-formula \(\neg l_1 \land (l_1 \lor \neg l_2) \land \cdots \land (l_1 \lor \cdots \lor l_{k-1} \lor \neg l_k)\) is easy to verify if \(l_1, \ldots, l_k\) are all false. A set \(S\) is key-equal to some fact in \(I\) if and only if \(\forall \alpha \in G\), \(\exists q \in \mathcal{Q} \colon \alpha \cdot \neg q \rightarrow \neg s(S)\).

Let \(R\) be a database schema with \(\text{SUM}\) and \(\text{COUNT}\) operators, and \(Q\) be the aggregation query

\[
Q := \text{SELECT} \ f \ \text{FROM} \ T(U, A)
\]

where \(f\) is one of the operators \(\text{COUNT}(*), \text{COUNT}(A), \text{SUM}(A),\) and \(T(U, A)\) is a relation expressed as a union of conjunctive queries over \(R\). We now reduce the range consistent answers to aggregation queries without grouping, and in Section 4.3, we describe an iterative algorithm that uses these reductions to handle aggregation queries with grouping.

### 4.1 Answering Queries with \(\text{SUM}\) and \(\text{COUNT}\)

Let \(R\) be a database schema with one key constraint per relation, and \(Q\) be the aggregation query

\[
Q := \text{SELECT} \ f \ \text{FROM} \ T(U, A)
\]

where \(f\) is one of the operators \(\text{COUNT}(\cdot), \text{COUNT}(A), \text{SUM}(A),\) and \(T(U, A)\) is a relation expressed as a union of conjunctive queries over \(R\). We now reduce the range consistent answers to aggregation queries without grouping, and in Section 4.3, we describe an iterative algorithm that uses these reductions to handle aggregation queries with grouping.

#### 4.1.1 Reductions to PMaxSAT and WPMaxSAT

**Reduction 4.1.** Let \(Q := \text{SELECT} \ f \ \text{FROM} \ T(U, A)\) be an aggregation query, where \(f\) is one of the operators \(\text{COUNT}(\cdot), \text{COUNT}(A), \) or \(\text{SUM}(A)\). Let \(I\) be an \(R\)-instance and \(G\) be the set of key-equal groups of facts in \(I\). For each fact \(f_i\) of \(I\), introduce a boolean variable \(x_i\). Let \(W\) be the bag of witnesses to the query \(q^*\) on \(I\), where

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

Construct a partial CNF-formula \(\phi\) (if \(f\) is \(\text{COUNT}(\cdot)\) or \(\text{COUNT}(A)\) ) or a weighted partial CNF-formula \(\phi\) (if \(f\) is \(\text{SUM}(A)\)) as follows:

1. For each \(G_j \in G\),
   - construct a hard clause \(\alpha_j := \lor x_i \) if \(f_i \in G_j\).
   - for each pair \((f_m, f_n)\) of facts in \(G_j\) such that \(m \neq n\), construct a hard clause \(\alpha_j := \neg x_m \lor \neg x_n\).

2. If \(f\) is \(\text{COUNT}(\cdot)\) or \(\text{COUNT}(A)\), then for each witness \(W_j \in W\), construct a soft clause \(\beta_j\), where

\[
\beta_j = \left( \lor_{f_i \in W_j} \neg x_i, m_j \right).
\]

Construct a partial CNF-instance

\[
\phi = \left( \bigwedge_{j=1}^{G_j} \alpha_j \right) \land \left( \bigwedge_{j=1}^{G_j} \left( \bigwedge_{f_i \in G_j} a_{j}^{m} \right) \right) \land \left( \bigwedge_{j=1}^{W} \beta_j \right).
\]

Note that two distinct witnessing assignments to an answer may give rise to the same witness. Thus, we consider the bag of witnesses to an answer, i.e., the bag consisting of witnesses arising from all witnessing assignments to that answer, where each witness \(s\) is accompanied by its multiplicity, an integer denoting the number of witnessing assignments that gave rise to \(s\). Finally, we define the bag of witnesses to a conjunctive query as the bag union of the bags of witnesses over all answers to \(q\) on \(I\) (in the bag union the multiplicities of the same set are added). The bag of witnesses to a union \(q := q_1 \cup \cdots \cup q_k\) of conjunctive queries is the bag union of the bags of witnesses to each conjunctive query \(q_i\) in \(q\). The bag of witnesses will be used in computing the range consistent answers to aggregation queries. In effect, the bag of witnesses corresponds to the provenance polynomials of conjunctive queries and their unions [29, 33].

It is easy to verify that both the key equal groups and the bag of the witnesses can be computed using SQL queries.
(2b) If \( f \) is \( \text{SUM}(A) \), let \( W_P \) and \( W_N \) be the subsets of \( W \) such that for each \( W_j \in W \), we have \( W_j \in W_P \) iff \( q^*(W_j) > 0 \), and \( W_j \in W_N \) iff \( q^*(W_j) < 0 \). Let also \( m_j = ||q^*(W_j)|| \), where \( ||q^*(W_j)|| \) is the absolute value of \( q^*(W_j) \). Construct a weighted soft clause \( \beta_j \) and a conjunction \( y_j \) of hard clauses as follows. If \( W_j \in W_N \), introduce a new variable \( y_j \) and let

\[
\beta_j = (y_j, W_j)
\]

and

\[
y_j = (\vee_{f_i \in W_j} \neg x_i) \land (\land_{f_i \in W_j} (\neg y_j \lor x_i))
\]

otherwise, let \( \beta_j = (\vee_{f_i \in W_j} \neg x_i, W_j) \) and do not construct \( y_j \).

Construct a weighted partial CNF-instance

\[
\phi = \left( \bigwedge_{j=1}^{||Q||} \alpha_j \right) \land \left( \bigwedge_{j=1}^{||Q||} \bigwedge_{m \in \mathcal{G}_j} \bigwedge_{f \in \mathcal{G}_j} \alpha_{m}^{mn} \right) \land \left( \bigwedge_{j=1}^{||W||} \bigwedge_{W_j \in W_N} \beta_j \right) \land \left( \bigwedge_{W_j \in W_P} y_j \right).
\]

**Purpose of the components of \( \phi \) in Reduction 4.1**

- Each \( \alpha_j \)-clause encodes the "at-least-one" constraint for each key-equal group \( G_j \) in the sense that satisfying \( \alpha_j \) requires setting at least one variable corresponding to a fact in \( G_j \) to true. Similarly, each \( \alpha_{m}^{mn} \)-clause encodes the "at-most-one" constraint for \( G_j \). In effect, every assignment that satisfies all \( \alpha \)-clauses sets exactly one variable corresponding to the facts from each key-equal group to true, and thus uniquely corresponds to a repair of \( I \).
- Satisfying a \( \beta_j \)-clause constructed in Step 2a requires setting at least one variable corresponding to the facts of a witness \( W_j \) to \( q^* \) on \( J \) to false. Thus, if \( s \) is an assignment that satisfies all \( \alpha \)-clauses, then \( \beta_j \) is satisfied by \( s \) if and only if \( W_j \not\in J \), where \( J \) is a repair corresponding to \( s \).
- The \( \beta_j \)-clauses constructed in Step 2b serve the same purpose as the ones from Step 2a, but here they are constructed only for the witnesses in \( W_P \). In \( W_N \), the \( \beta_j \)-clauses encode the condition that \( \beta_j \) is satisfied if and only if all variables corresponding to the facts in \( W_j \) are set to true. The hard \( y_j \)-clauses are used solely to express the equivalence \( y_j \leftrightarrow (\land_{f \in W_j} x_i) \) in conjunctive normal form.

The number of \( \alpha \)-clauses is \( O(n) \), where \( n \) is the size of the database; the number of \( \beta \)-clauses and \( y \)-clauses combined is \( O(n^2) \), where \( k \) is the number of relation symbols in \( Q \).

**Proposition 4.1.** Let \( Q \) be an aggregation query, where \( Q \) is one of the operators \( \text{COUNT}(*) \), \( \text{COUNT}(A) \), and \( \text{SUM}(A) \). In a maximum (a minimum) satisfying assignment of the \( \text{WPMaxSAT} \)-instance \( \phi \) constructed using Reduction 4.1, the sum of weights of the falsified clauses is the \( glb \)-answer (\( lub \)-answer) in the range consistent answers \( \text{Cons}(Q) \) on \( I \).

**Proof.** Let \( Q \) be an aggregation query, where \( Q \) is one of the operators \( \text{COUNT}(*) \), \( \text{COUNT}(A) \), and \( \text{SUM}(A) \). In a maximum (a minimum) satisfying assignment of the \( \text{WPMaxSAT} \)-instance \( \phi \) constructed using Reduction 4.1, the sum of weights of the falsified clauses is the \( glb \)-answer (\( lub \)-answer) in the range consistent answers \( \text{Cons}(Q) \) on \( I \).

**Example 4.2.** Let \( I \) be a database instance from Table 1, and \( Q \) be the following aggregation query which counts the number of customers who have an account in their own city:

\[
\text{SELECT COUNT(*)}
\]

\[
\text{FROM CUST, ACC, CUSTACC}
\]

\[
\text{WHERE CUST.CID = CUSTACC.CID}
\]

\[
\text{AND ACC.ACCID = CUSTACC.ACCID}
\]

\[
\text{AND CUST.CITY = ACC.CITY}
\]

From Reduction 4.1, we construct the following clauses:

\[
\alpha \text{-clauses: } x_1(x_2 \lor x_3), x_4, x_5, x_6, x_7, (x_8 \lor x_9), x_{10}:
\]

\[
\alpha_{m}^{mn} \text{-clauses: } (\neg x_2 \lor \neg x_3), (\neg x_4 \lor \neg x_5), (\neg x_6 \lor \neg x_7), (\neg x_8 \lor \neg x_9), (\neg x_{10} \lor \neg x_{11}).
\]

Observe that it is okay to omit the variables corresponding to the facts in \( \text{CUSTACC} \) since \( \text{CUSTACC} \) does not violate \( \Sigma \). A maximum satisfying assignment to the \( \text{PMaxSAT} \)-instance \( \phi \) constructed from above clauses is \( x_i = 0 \) for \( i \in \{2, 9\} \), and \( x_1 = 1 \) otherwise. It falsifies one clause, namely, \( (\neg x_1 \lor \neg x_6, 1) \). Similarly, an assignment \( x_1 = 0 \) for \( i \in \{2, 9\} \), and \( x_1 = 1 \) otherwise is a minimum satisfying assignment to the \( \text{PMinSAT} \)-instance \( \phi \), and it falsifies two clauses, namely, \( (\neg x_1 \lor \neg x_6, 1) \) and \( (\neg x_3 \lor \neg x_9, 1) \). Thus, \( \text{Cons}(Q, I) \) w.r.t. range semantics is \( [1, 2] \) by Proposition 4.1.

**Example 4.3.** Let us again consider the database instance \( I \) from Table 1, and the following aggregation query \( Q \):

\[
\text{SELECT SUM(ACC.BAL)}
\]

\[
\text{FROM CUST, ACC, CUSTACC}
\]

\[
\text{WHERE CUST.CID = CUSTACC.CID}
\]

\[
\text{AND ACC.ACCID = CUSTACC.ACCID}
\]

\[
\text{AND CUST.CNAME = 'Mary'}
\]

The hard clauses constructed using Reduction 4.1 are same as the ones from Example 4.2. The rest of the clauses are as follows:
\(\beta\)-clauses: \((\neg x_2 \lor \neg x_7, 1000), (\neg x_3 \lor \neg x_7, 1000), (\neg x_2 \lor \neg x_8, 1200), (\neg x_3 \lor \neg x_8, 1200), (y_1, 100), (y_2, 100)\)

\(\gamma\)-clauses: \((\neg x_2 \lor \neg x_3 \lor y_1), (\neg y_1 \lor x_2), (\neg y_1 \lor x_3), (\neg x_3 \lor \neg x_9 \lor y_2), (\neg y_2 \lor x_3), (\neg y_2 \lor x_9)\)

The witnesses \(\{f_2, f_3, f_5\}\) and \(\{f_3, f_4, f_5, f_6\}\) belong to \(\mathcal{W}_b\) since the account balance is \(-100\) in both cases, so we introduce new variables \(y_1\) and \(y_2\) respectively, and construct hard \(\beta\)-clauses as described above. The \(\beta\)-clauses corresponding to these witnesses are \((y_1, 100)\) and \((y_2, 100)\). We omit \(x_3\) in all of these clauses since CUST-ACC does not violate \(\Sigma\). Note that \(Q(I_F) = 4400\). An assignment in which \(x_3 = 0\) and \(y_3 = 1\) is a maximum satisfying assignment to the PMaxSAT instance \(\phi\) 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 4.1, we have that \(Q(J) = 4400 - 3500 = 900\) where \(J\) is a repair corresponding to the assignment in consideration. Similarly, setting \(x_3 = 1\) 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 \(1000\), indicating that \(\text{Cons}(Q, J) = \{900, 2200\}\).

### 4.1.2 Handling \textsc{Distinct}

Let \(Q := \text{SELECT } f \text{ FROM } T(U, A)\) be an aggregation query, where \(f\) is either \textsc{COUNT(DISTINCT } \(A)\text{) or \text{SUM(DISTINCT } A\text{).} Solving a PMaxSAT or a WPMaxSAT instance constructed using Reduction 4.1 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 the database instance \(I\) from Table 1, and a query \(Q := \text{SELECT COUNT(DISTINCT ACC.TYPE) FROM ACC}\).

The correct \(glb\) and \(lub\)-answers in \(\text{Cons}(Q, I)\) are both 2, but solutions to the PMaxSAT and PMinSAT instances constructed using Reduction 4.1 yield both answers as 4. The reason behind this is that the soft clauses \(\neg x_6\) and \(\neg x_7\) correspond to the account type Checking, and similarly \(\neg x_8\), \(\neg x_9\), and \(\neg x_{10}\) 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 \(\beta\)-clauses in Reduction 4.1 as follows.

Let \(A\) denote a set of distinct answers to the query \(q'(A) := \exists U T(U, A)\). For each answer \(b \in A\), let \(\mathcal{W}^b\) denote a subset of \(\mathcal{W}\) such that for every 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 \(b \in A\), 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 \(W^b \in \mathcal{W}^b\), we introduce an auxiliary variable \(x^b_j\) that is true if and only if \(W^b\) is not present in the repair. Then, we introduce an auxiliary variable \(y^b\) which is true if and only if all \(x^b\)-variables are true. These constraints are encoded in the set \(H^b\) returned by Algorithm 1, and are forced by making clauses in \(H^b\) hard. For every answer \(b \in A\), Algorithm 1 also returns one \(\beta^b\)-clause, which serves the same purpose as the \(\beta\)-clauses in Reduction 4.1. Now, a PMaxSAT or a WPMaxSAT instance can be constructed by taking in conjunction all \(\alpha\)-clauses from the key-equal groups, the hard \(\gamma\)-clauses if any, the hard clauses from all \(H^b\)-sets, and all soft \(\beta^b\)-clauses. With this, it is easy to see that a maximum (or minimum) satisfying assignment to PMaxSAT or WPMaxSAT instance give us the \(glb\)-answer (or \(lub\)-answer) in \(\text{Cons}(Q)\). This is illustrated in Example 4.4.

### Algorithm 1 Handling \textsc{Distinct}

1. procedure \text{handle} \textsc{Distinct} \((W^b)\)
2. let \(H^b = \emptyset\) //Empty set of clauses
3. for \(W^b \in \mathcal{W}^b\) do
4. \(H^b = H^b \cup \{\neg x^b_j \lor (\lor \in W^b)\}\)
5. for \(f_1 \in W^b\) do
6. \(H^b = H^b \cup \{\neg f_1 \lor (\lor \in W^b)\}\)
7. for \(W^b \in \mathcal{W}^b\) do
8. \(H^b = H^b \cup \{\lor \in W^b\}\)
9. \(H^b = H^b \cup \{\lor \in W^b\}\)
10. let \(\beta^b = (b^b, 1)\)
11. if \((f \text{ is SUM(DISTINCT } A))\) then
12. \(\beta^b = (\lor, ||b||)\)
13. if \(b < 0\) then \(\beta^b = (\lor, ||b||)\)
14. return \(H^b, \beta^b\)

### Example 4.4

Consider the following aggregation query \(Q\) on the database instance \(I\) from Table 1:

\(Q := \text{SELECT COUNT(DISTINCT ACC.TYPE) FROM ACC}\)

We have that \(A = \{\text{Checking}, \text{Saving}\}\). 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 \(y^b\)-variable is equivalent to a single literal, for example, \(y^b_1 \leftrightarrow \neg x_6\) and \(y^b_2 \leftrightarrow \neg x_7\). As a result, it is unnecessary to introduce any \(x^b\)-variables at all. Thus, we construct the following clauses from Reduction 4.1 and Algorithm 1:

- \(\alpha\)-clauses: \(x_6, x_7, (x_8 \lor x_9), x_{10}\)
- \(\alpha^{mn}\)-clauses: \((\neg x_8 \lor \neg x_6)\)
- \(H^b_1 : (x_6 \lor x_7 \lor \lor)\)
- \(H^b_2 : (x_8 \lor x_9 \lor x_{10})\)
- \(\beta\)-clauses: \((\lor, 1), (\lor, 1)\)

The maximum and minimum satisfying assignments to the PMaxSAT and PMinSAT instances constructed using these clauses falsify both \(\beta\)-clauses, since \(\text{Cons}(Q, I)\) w.r.t. range semantics is \([2, 2]\).

### 4.2 Answering Queries with \textsc{Min} and \textsc{Max}

Let \(R\) be a database schema with one key constraint per relation, and \(Q\) be the aggregation query

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

where \(f\) is one of the operators \textsc{Min}(\(A\)) and \textsc{Max}(\(A\)), and \(T(U, A)\) is a relation expressed as a union of conjunctive queries over \(R\). The semantics of the range consistent answers to aggregation queries with \textsc{Min} and \textsc{Max} operators are similar to aggregation queries with \textsc{Sum} or \textsc{Count} operators, but here we need to address one additional special case. We illustrate this special case using Example 4.5.

### Example 4.5

Consider the database instance \(I\) from Table 1 and two aggregation queries \(Q_1\) and \(Q_2\) as follows.
$Q_1 := \text{SELECT SUM(Accounts.BAL) FROM Accounts WHERE Accounts.CITY = 'SF'}$

$Q_2 := \text{SELECT MIN(Accounts.BAL) FROM Accounts WHERE Accounts.CITY = 'SF'}$

It is clear that the range consistent answers to $\text{Cons}(Q_1, I) = [-100, 0]$. The lub-answer of 0 in $\text{Cons}(Q_1, I)$ 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 for empty sets.

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 $[\text{glb}, \text{lub}]$ 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 lub-answer of an aggregation query with the MIN($A$) operator can be computed in polynomial time in the size of the original inconsistent database instance $I$ (Proposition 4.6). We then give an iterative SAT-based approach to compute the lub-answer to an aggregation query with the MIN($A$) operator. We do not explicitly state methods to compute the range consistent answers 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 and then computing the glb-answer for the MAX($A$) operator.

**Proposition 4.6.** Let $R$ be a database schema, $I$ an $R$-instance, and $Q$ the aggregation query $\text{SELECT MIN}(A)$ FROM $T(U, A)$ be the witness to $q_1$ on $I$ such that no two facts in $W_{gb}$ are key-equal, and there is no $W'$ such that $q(W') < q(W_{gb})$ and no two facts in $W'$ are key-equal. Then, $q_1(W_{gb})$ is the glb-answer in $\text{Cons}(Q, I)$.

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

To compute the range consistent answers to aggregation queries with MIN($A$) and MAX($A$) operators, we opt for an iterative SAT solving approach. In what follows, we formalize the construction of the SAT instance for the first iteration (Construction 4.1) and give Algorithm 2 that computes the lub-answer in $\text{Cons}(Q, I)$ by constructing and solving SAT instances in subsequent iterations.

**Construction 4.1.** Given an $R$-instance $I$, construct a CNF formula $\phi$ as follows. For each fact $f_i$ of $I$, introduce a boolean variable $x_i$. Let $G$ be the set of key-equal groups of facts of $I$, and $W = \{W_1, \ldots, W_m\}$ denote the set of minimal witnesses to a conjunctive query $q$ on $I$, where $q(w) := \exists \bar{u} T(\bar{u}, w)$. Assume that the set $W$ 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 $G_j \in G$, construct a clause $\alpha_j = \bigwedge f_i \in G_j \neg x_i$.
- Construct a CNF formula $\phi = \bigwedge_{j=1}^{\left|G\right|} \alpha_j$.

**Algorithm 2 Computing the lub-answer in Cons$(Q, I)$ for MIN via Iterative SAT**

1. **procedure** LUBAnswer-IterativeSAT($\phi, W$)
2. let $v = q(W_1), j = 1$
3. while $j \leq |W|$ do
4. if $v = q(W_j)$ then
5. let $\phi = \phi \land \bigvee f_i \in W_j \neg x_i$
6. $j = j + 1$
7. else
8. if UNSAT($\phi$) then
9. return $q(W_{j-1})$
10. else
11. let $v = q(W_j)$
12. return $q(W_j)$

**Proposition 4.7.** Let $Q := \text{SELECT MIN}(A)$ FROM $T(U, A)$ be an aggregation query, and $I$ be a database instance. Algorithm 2 returns the lub-answer in $\text{Cons}(Q, I)$.

**Proof.** The $\alpha$-clauses of $\phi$ make sure that a repair $\mathcal{J}$ of $I$ can be constructed from every assignment $s$ of $\phi$ that satisfies the $\alpha$-clauses, by arbitrarily choosing exactly one fact $f_i$ from each key-equal group of $I$ such that $s(x_i) = 1$. Let $\mathcal{A} = \{A_1, \ldots, A_{\text{lab}}, \ldots, A_{\text{lab}}\}$ denote the set of distinct answers to a conjunctive query $q(w) := \exists \bar{u} T(\bar{u}, w)$ on $I$, where $A_{\text{lab}}$ is the lub-answer to $Q$ on $I$. For a witness $W$ to $q$, a clause $(\forall i \in W \neg x_i)$ 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 $\phi$ is checked for satisfiability (line 8 of Algorithm 2), the formula contains the $\alpha$-clauses corresponding to the key-equal groups of $I$ 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 $\phi$ is satisfiable, then there exists a repair $\mathcal{J}$ of $I$ 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 $I$ 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 $\phi$ satisfiable at iteration $j$ if and only if $q(W_{j-1}) < A_{\text{lab}}$. Therefore, if $\phi$ becomes unsatisfiable for the first time at iteration $j$, it must be the case that $q(W_{j-1})$ is the lub-answer in $\text{Cons}(Q, I)$. $\square$
With grouping, we first compute the consistent answers to a
instances of SAT. The problem
approach is the following. Observe that, on an average, half of the SAT instances that the solver needs to solve in the binary
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, in practice, the linear search often works better than the binary search.

4.3 Answering Queries with Grouping

Let \( Q \) be the aggregation query
\[
Q := \text{SELECT } Z, f \text{ FROM } T(U, Z, A) \text{ GROUP BY } Z,
\]
where \( f \) is one of \( \text{COUNT}(\cdot), \text{COUNT}(A), \text{SUM}(A), \text{MIN}(A), \) or \( \text{MAX}(A) \), and \( T(U, A) \) is a relation expressed by a union of conjunctive queries on \( R \). We refer to the attributes in \( Z \) as the grouping attributes. For aggregation queries with grouping, it does not seem feasible to reduce \( \text{Cons}(Q) \) to a single PMaxSAT or WPMaxSAT instance because for each group of consistent answers, the GLB-answer and the LUB-answer may realize in different repairs of the inconsistent database. To illustrate this, consider the database from Table 1 and a query \( Q := \text{SELECT COUNT(\cdot)} \) from CUST GROUP BY CUST.CITY. Notice that, the GLB-answers (LA, 2) and (SF, 1) in \( \text{Cons}(Q) \) come from two different repairs of relation CUST, namely, \( \{f_1, f_2, f_3, f_5\} \) and \( \{f_1, f_2, f_3, f_6\} \) respectively. However, the reductions from the preceding section 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 an underlying conjunctive query \( q(Z) = \exists U, A \ T(U, Z, A) \). Then, for each answer \( b \) in \( \text{Cons}(q) \), we compute the GLB and LUB-answers to the query \( Q' := \text{SELECT } f \text{ FROM } T(U, Z, A) \land (Z = b) \) via PMaxSAT or WPMaxSAT solving as shown in Algorithm 3.

As noted earlier, the bags of witnessed uses in the preceding reductions capture the provenance of unions of conjunctive queries in the provenance polynomials model of [29, 33]. In [4], it was shown that a stronger provenance model is needed to express the provenance of aggregation queries, a model that uses a tensor product combining annotations with values. A future direction of research is to investigate whether this stronger provenance model can be used to produce more direct reductions of the range consistent answers to SAT.

5 BEYOND KEY CONSTRAINTS

Key constraints and functional dependencies are important special cases of denial constraints (DCs), which are expressible by first-order formulas of the form \( \forall x_1, ..., x_n (\phi(x_1, ..., x_n) \land \neg \psi(x_1, ..., x_n)) \), or, equivalently, \( \forall x_1, ..., x_n (\phi(x_1, ..., x_n) \rightarrow \neg \psi(x_1, ..., x_n)) \), where \( \phi(x_1, ..., x_n) \) is a conjunction of atomic formulas and \( \psi(x_1, ..., x_n) \) is a conjunction of expressions of the form \( (x_j \text{ op } x_k) \) with each op a built-in predicate, such as \( \neq, <, >, \leq, \geq \). In words, a denial constraint prohibits a set of tuples that satisfy certain conditions from appearing together in a database instance. If \( \Sigma \) 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 \( I \) and a number \( t \), is \( t \) the lub-answer (or the glb-answer) in \( \text{Cons}(Q, I) \) w.r.t. \( \Sigma \)? This is so because to check that \( t \) is not the lub-answer (or the glb-answer), we guess a repairs \( f \) of \( I \) and verify that \( t > Q(f) \) (or \( t < Q(f) \)). In all preceding reductions, the \( \alpha \)-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 \( \alpha \)-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 \( \alpha \)-clauses in the reductions from Section 4 by the ones provided below allows us to compute consistent answers over databases with a fixed finite set of arbitrary denial constraints. The reduction relies on the notions of minimal violations and near-violations to the set of denial constraints that we introduce next:

- Assume that \( \Sigma \) is a set of denial constraints, \( I \) is an \( R \)-instance, and \( S \) is a sub-instance of \( I \). We say that \( S \) is a minimal violation to \( \Sigma \), if \( S \not\models \Sigma \) and for every set \( S' \subseteq S \), we have that \( S' \models \Sigma \).

- Let \( \Sigma \) be a set of denial constraints, \( I \) an \( R \)-instance, \( S \) a sub-instance of \( I \), and \( f \) a fact of \( I \). We say that \( S \) is a near-violation w.r.t. \( \Sigma \) and \( f \) if \( S \models \Sigma \) and \( S \cup \{f\} \) is a minimal violation to \( \Sigma \). As a special case, if \( \{f\} \) itself is a minimal violation to \( \Sigma \), we say that there is exactly one near-violation w.r.t. \( f \), and it is the singleton \( \{f_{\text{true}}\} \), where \( f_{\text{true}} \) is an auxiliary fact.

Let \( R \) be a database schema, \( \Sigma \) be a fixed finite set of denial constraints on \( R \), \( Q \) be an aggregation query without grouping, and \( I \) be an \( R \)-instance.

**Reduction 5.1.** Given an \( R \)-instance \( I \), compute the sets:

1. \( \forall \cdot \): the set of minimal violations to \( \Sigma \) on \( I \).
2. \( N^1 \cdot \): the set of near-violations to \( \Sigma \), on \( I \), w.r.t. each fact \( f_i \) in \( I \).

For each fact \( f_i \) of \( I \), introduce a boolean variable \( x_i \), \( 1 \leq i \leq n \). For the auxiliary fact \( f_{\text{true}} \), introduce a constant \( x_{\text{true}} = \text{true} \), and for each \( N^j_i \in N^1 \cdot \), introduce a boolean variable \( p^j_i \).
We evaluate the performance of AggCAvSAT over both synthetic and real-world databases. The first set of experiments includes a comparison of AggCAvSAT 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 [5] (see Section 6.1.1 for details). Next, we assess the scalability of AggCAvSAT 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 5, we use a real-world Medigap [2] dataset 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 [18] for solving the WPMaxSAT instances. The AggCAvSAT system is implemented in Java 9.04 and its code is open-sourced at a GitHub repository https://github.com/ucccross/cvatsat via a BSD-style license. Various features of AggCAvSAT, including its graphical user interface, are presented in a short paper in the demonstration track of the 2021 SIGMOD conference [19].

### 6.1 Experiments with TPC-H Data and Queries

#### 6.1.1 Datasets

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 AggCAvSAT against ConQuer [22, 24] (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-equivalent 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 varied 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 [5] to generate four inconsistent database instances with varying degrees of inconsistency (see Table 2). In all four instances, the data is generated in such a way that every repair is of size 1 GB.

#### 6.1.2 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 3. In Section 6.1.3, 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 answers without grouping. We refer to these queries as $Q'_1, Q'_2, \ldots, Q'_9$. The definitions of these queries are given in Table 3.

#### 6.1.3 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 1 shows that much of the evaluation time used by AggCAvSAT 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, $Q'_q$ is not in the class $Aggforest$ and thus ConQuer cannot compute its range consistent answers. AggCAvSAT performs better than ConQuer on seven out of the remaining eight queries.

Next, we compared the performance of AggCAvSAT and ConQuer on database instances generated using PDBech. Figure 2 shows that AggCAvSAT performs better than ConQuer on PDBech instances with low inconsistency. As the inconsistency increases, the WPMaxSAT solver requires considerably large time to compute the optimal solutions (especially for $Q'_q$, $Q'_q$, and $Q'_q$). One reason is that the sizes of key-equally groups in PDBech instances with higher inconsistency percentage are large, which translates into clauses of large sizes in the WPMaxSAT instances, hence, the solver works hard to solve them. Also, Kiguel's reduction [39] from WPMinSAT to WPMaxSAT significantly increases the size of the CNF formula, resulting in higher time for the lub-answers to the queries.

Next, we varied the inconsistency in the database instances created using the DBGen-based data generator while keeping the size of
We then evaluated AggCAvSAT’s scalability by increasing the size of the databases while keeping the inconsistency to a constant 10%. Figure 4 shows that the evaluation time of AggCAvSAT 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 AggCAvSAT spending more time on computing the bags of witnesses to these queries as the size of the database grows.

Figure 3: AggCAvSAT on TPC-H data generated using the DBGen-based tool (varying inconsistency, 1 GB repairs)

Figure 4: AggCAvSAT on TPC-H data generated using the DBGen-based tool (varying database sizes, 10% inconsistency)

6.1.4 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 5), as the queries 6, 14, and 19 did not contain grouping. We evaluated the performance of AggCAvSAT and compared it to ConQuer on a database with 10% inconsistency w.r.t. primary keys (Figure 5). The repairs are of size 1 GB. AggCAvSAT computes the consistent answers to the underlying conjunctive query using the reductions from [20] which are, precisely, the consistent groups in the range consistent answers to the aggregation query. For each of these groups, it computes the $\text{glb}$-answer and the $\text{lub}$-answer using reductions to WPMAXSAT.

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, AggCAvSAT needs to construct and solve twice as many WPMAXSAT instances as there are consistent groups, i.e., one for the $\text{lub}$-answer and one for the $\text{glb}$-answer per consistent...
Table 5: TPC-H-inspired Aggregation Queries w/ Grouping

| # | Query                                                                 | Operator                     |
|---|----------------------------------------------------------------------|------------------------------|
| 1 | SELECT LINEDATE, LINENAME, SUM(LINEXTENDEDPRICE) FROM LINEITEM WHERE LINESKIPDATE <= dateadd(dd, -20, cast('1998-12-01' as datetime)) GROUP BY LINEDATE, LINENAME | SUM(LINEXTENDEDPRICE)        |
| 2 | SELECT LINEDATE, LINENAME, SUM(LINEXTENDEDPRICE) FROM LINEITEM WHERE LINESKIPDATE <= dateadd(dd, -20, cast('1998-12-01' as datetime)) GROUP BY LINEDATE, LINENAME | SUM(LINEXTENDEDPRICE)        |
| 3 | SELECT LINEDATE, LINENAME, SUM(LINEXTENDEDPRICE) FROM LINEITEM WHERE LINESKIPDATE <= dateadd(dd, -20, cast('1998-12-01' as datetime)) GROUP BY LINEDATE, LINENAME | SUM(LINEXTENDEDPRICE)        |
| 4 | SELECT LINEDATE, LINENAME, SUM(LINEXTENDEDPRICE) FROM LINEITEM WHERE LINESKIPDATE <= dateadd(dd, -20, cast('1998-12-01' as datetime)) GROUP BY LINEDATE, LINENAME | SUM(LINEXTENDEDPRICE)        |
| 5 | SELECT LINEDATE, LINENAME, SUM(LINEXTENDEDPRICE) FROM LINEITEM WHERE LINESKIPDATE <= dateadd(dd, -20, cast('1998-12-01' as datetime)) GROUP BY LINEDATE, LINENAME | SUM(LINEXTENDEDPRICE)        |

Figure 6: AggCAvSAT vs. ConQuer on PDBench instances

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 shows the comparison of AggCAvSAT and ConQuer for aggregation queries with grouping on PDBench instances. For the database instance with the lowest amount of inconsistency, AggCAvSAT beats ConQuer on all queries, but as the inconsistency grows, AggCAvSAT takes longer time to encode and solve for the consistent groups of the queries $Q_1$ and $Q_{10}$.

In Figure 7, we first plot the evaluation time of AggCAvSAT 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 AggCAvSAT constructs and solves many WPMaxSAT 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 7. Note that the Y-axis has logarithmic scaling...
in the second plot of Figure 7. 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 \( Q_5 \) is high but the number of SAT calls is comparatively less, while AggCAvSAT makes more SAT calls for \( Q_5 \) and \( Q_{12} \), even though their consistent answers are computed much faster. The query \( Q_{10} \) requires long time to construct and solve the WPMaxSAT instances for its consistent groups due to its high selectivity and the 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 8). The second plot in Figure 8 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.

6.1.5 Discussion. The experiments show that AggCAvSAT performed well across a broad range of queries and databases; it performed worse on queries with high selectivity because, in such cases, very large CNF formulas were generated. AggCAvSAT 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 Experiments with Real-world Data

6.2.1 Dataset. For this set experiments, we use the schema and the data from Medigap [2], an openly available real-world database about Medicare supplement insurance in the United States. We combine the data from 2019 and 2020 to obtain a database with over 61K tuples (Table 6a). We evaluated the performance of Reduction 5.1, since we consider two functional dependencies and one denial constraint on the Medigap schema, as shown in Table 6b. The actual data was inconsistent so no additional inconsistency was injected.

6.2.2 Queries. We use twelve natural aggregation queries on the Medigap database that involve the aggregation operators COUNT(\( \* \)), COUNT(\( \cdot \)), and SUM(\( \cdot \)). We refer to these as \( \{ Q_1^m, \ldots, Q_{12}^m \} \). The first six queries contain no grouping, while the rest of them do. The definitions of these queries are given in Table 7.

6.2.3 Results on Real-world Database. In Figure 9, we plot the overall time taken by AggCAvSAT 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 WPMaxSAT instances is based on Reduction 5.1. Consequently, the size of the CNF formulas is much larger compared to that of the ones produced by Reduction 4.1, 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 \( \gamma \)-clauses. This part of the encoding time is equal for all queries, but the computation time...
Table 7: Aggregation Queries on Medigap database

| # | Query                                                                 |
|---|----------------------------------------------------------------------|
| Q1 | SELECT COUNT(*) FROM OBS WHERE OBS.Name = 'Continental General Insurance Company' |
| Q2 | SELECT COUNT(*) FROM PBZ, SPT WHERE PBZ.Description = SPT.Simple_plantype_name AND SPT.Contract_year = 2020 AND SPT.Simple_plantype = 'B' |
| Q3 | SELECT SUM(PBZ.Over65) FROM PBZ WHERE PBZ.State_name = 'New York' |
| Q4 | SELECT PBZ.State_name, COUNT(*) FROM PBZ GROUP BY PBZ.State_name |
| Q5 | SELECT SUM(PBZ.Community) FROM PBZ WHERE PBZ.State_name = 'New York' |
| Q6 | SELECT COUNT(PR.Premium_range) FROM PR GROUP BY PR.Age_category, COUNT(*) |
| Q7 | SELECT SUM(PBZ.Over65) FROM PBZ WHERE PBZ.State_name = 'Wisconsin' AND SPT.Contract_year = 2020 AND SPT.Simple_plantype = 'K' |
| Q8 | SELECT PBS.State_name, SPT.Contract_year, SUM(PBS.Under65) FROM PBS, SPT WHERE SPT.Simple_plantype_name = PBS.Description AND SPT.Contract_year = 2020 AND PBS.State_name = 'New York' GROUP BY PBS.State_name, SPT.Contract_year ORDER BY PBS.State_name |
| Q9 | SELECT PBS.State_name, COUNT(*) FROM PBS GROUP BY PBS.State_name |
| Q10 | SELECT PBZ.State_name, COUNT(*) FROM PBZ GROUP BY PBZ.State_name |
| Q11 | SELECT PBZ.State_name, COUNT(*) FROM PBZ GROUP BY PBZ.State_name |
| Q12 | SELECT PBZ.State_name, COUNT(*) FROM PBZ GROUP BY PBZ.State_name |

Figure 10: Number of clauses in a CNF formula capturing the consistent answers to the underlying conjunctive query

even if the consistent answers to the underlying conjunctive query are SQL-rewritable. We then designed, implemented, and evaluated AggCAvSAT, a SAT-based system for computing range consistent answers to aggregation queries involving COUNT (A), COUNT (*), SUM(A), and grouping. It is the first system able to handle aggregation queries whose range consistent answers are not SQL-rewritable. Our experimental evaluation showed that AggCAvSAT is not only competitive with systems such as ConQuer but it is also scalable. The experiments on the Medigap data showed that AggCAvSAT can handle real-world databases having integrity constraints beyond primary keys. The next step in this investigation is to first delineate the complexity of the range consistent answers to aggregation queries with the operator \( \text{AVG}(A) \) and then enhance the capabilities of AggCAvSAT to compute the range consistent answers of such aggregation queries. Finally, 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 considered, such as universal constraints, then the consistent answers of even conjunctive queries become \( \Pi_2 \)-hard to compute [9], hence SAT-based methods are not applicable. In that case, Answer Set Programming solvers (for example, DLV [40] or Potassco [25]) have to be used, instead of SAT solvers.

Acknowledgments Dixit was supported by a Baskin School of Engineering Dissertation-Year Fellowship and by the Center for Research in Open Source Software (CROSS) at the UC Santa Cruz. Kolaitis was partially supported by NSF Award IIS: 1814152.

REFERENCES

1. [n.d.]. CadDiCal: Simplified Satisfiability Solver. http://fmv.jku.at/cadical/.
2. [n.d.]. Medigap database for Medicare health and drug plans. https://www.medicare.gov/download/downloaddb.asp.
3. Foto N. Afrati and Phokion G. Kolaitis. 2008. 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. ACM. 129–138. https://doi.org/10.1145/1376916.1376936
4. Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for aggregate queries. In Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece; Maurizio Lenzerini and Thomas Schwentick (Eds.). ACM, 153–164. https://doi.org/10.1145/1999284.1999302
5. Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. 2008. Fast and Simple Relational Processing of Uncertain Data. In 2008 IEEE 24th International Conference on Data Engineering, 983–992. https://doi.org/10.1109/ICDE.2008.4497507
https://doi.org/10.1145/1807085.1807111

[49] Jef Wijsen. 2010. A remark on the complexity of consistent conjunctive query answering under primary key violations. *Inform. Process. Lett.* 110, 21 (2010), 950 – 955. https://doi.org/10.1016/j.ipl.2010.07.021

[50] Zhu Zhu, Chu-Min Li, Felip Manyà, and Josep Argelich. 2012. A New Encoding from MinSAT into MaxSAT. In *Principles and Practice of Constraint Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 455–463.