Computing Consistent Least Upper Bounds in Aggregate Logic

Aziz Amezian El Khalfioui
Research fellow FNRS, University of Mons, Mons, Belgium

Jef Wijsen
University of Mons, Mons, Belgium

Abstract

We consider the problem of answering conjunctive queries with aggregation on database instances that may violate primary key constraints. In SQL, these queries follow the SELECT-FROM-WHERE-GROUP BY format, where the WHERE clause involves a conjunction of equalities, and the SELECT clause can incorporate aggregate operators like MAX, MIN, SUM, AVG, or COUNT. Repairs of a database instance are defined as inclusion-maximal subsets that satisfy all primary keys. The range-consistent answer to a numerical query over an inconsistent database is a pair [glb, lub], where glb and lub are, respectively, the smallest and the greatest results returned by the query over all possible repairs. While previous work has focused on the computation of the glb, the current paper studies the computation of the lub for a numerical domain of non-negative rational numbers. We introduce the notion of κ-acyclicity for self-join-free conjunctive queries. We show that if the body of a SUM-query is κ-acyclic, then the lub can be computed through a rewriting in first-order aggregate logic. Moreover, we show that this result extends to all aggregate operators that are monotone and associative. Importantly, we also prove the inverse: if the body of a SUM-query is not κ-acyclic, then the lub cannot be computed in first-order aggregate logic.

2012 ACM Subject Classification Information systems → Relational database query languages; Theory of computation → Incomplete, inconsistent, and uncertain databases; Theory of computation → Logic and databases

Keywords and phrases Consistent query answering, primary key, conjunctive query, aggregate logic

Digital Object Identifier 10.4230/LIPIcs.ICDT.2026.4

1 Introduction

Consistent query answering (CQA) was introduced in [3] as a principled approach to answering queries on databases that are inconsistent with respect to a given set of integrity constraints. The only integrity constraints we consider in the current work are primary keys. A *block* in a database instance is a $\subseteq$-maximal set of tuples of a same relation R that agree on the primary key of R. A *repair* of a database instance picks exactly one tuple from each block. Given a Boolean query q, CERTAINTY(q) is then defined as the decision problem that takes a database instance **db** as input, and asks whether q holds true in every repair of **db**. This problem, while commonly studied for Boolean queries, can be readily extended to queries with free variables $\vec{x}$: a *consistent answer* to a query $q(\vec{x})$ is any sequence $\vec{c}$ of constants, of length $|\vec{x}|$, such that $q(\vec{c})$ holds true in every repair. The computational complexity of CERTAINTY(q) is well understood for all queries q in sjfBCQ, the class of self-join-free Boolean conjunctive queries [29, 32]. This understanding readily extends to queries with free variables.

It is significant to generalize CQA from Boolean queries, which return either true or false, to *numerical queries*, which return a numeric result. Aggregation queries constitute an important class of numerical queries. However, numerical aggregation queries are likely to return different results on different repairs, and therefore lack a single consistent answer that holds true across all repairs. For this reason, Arenas et al. [4] have proposed *range*

semantics, which provides the greatest lower bound (glb) and the least upper bound (lub) of query answers across all repairs. Specifically, for a *numerical aggregation query* $g()$, the function problems GLB-CQA($g()$) and LUB-CQA($g()$) take a database instance **db** as input, and return, respectively, the glb and the lub of the set that contains each number returned by $g()$ on some repair.

In the current paper, we consider numerical aggregation queries $g()$ that take the following form in the extended datalog syntax of [10, 11]:

$$\texttt{AGG}(r) \leftarrow q(\vec{u}), \tag{1}$$

where the *body* $q(\vec{u})$ is a conjunction of atoms (a.k.a. subgoals), r is either a numeric variable occurring in $\vec{u}$ or a non-negative number, and AGG is an aggregate symbol (like MAX, MIN, SUM, AVG, COUNT). Such a query will be called an AGG-query, and is interpreted as follows. Every aggregate symbol AGG in our query language is associated with an aggregate operator, denoted $\mathcal{F}_{\texttt{AGG}}$, which is a function that takes a multiset of numbers, and returns a number. The semantics on a given database instance **db** is standard: let $\theta_1, \theta_2, \ldots, \theta_n$ enumerate all homomorphisms (a.k.a. embeddings) of the body into **db**, then the query returns $\mathcal{F}_{\texttt{AGG}}(\{\{\theta_1(r), \theta_2(r), \ldots, \theta_n(r)\}\})$. Note that the argument of $\mathcal{F}_{\texttt{AGG}}$ is a multiset, because it is possible that $\theta_i(r) = \theta_j(r)$ for $i \neq j$. For this semantics to be well-defined, each $\theta_i(r)$ must necessarily belong to some numerical domain D. In this case, the aggregate query is said to be *over* D. In this paper, we take D to be $\mathbb{Q}_{\geq 0}$.

In [24], progress has been made in studying the complexity of GLB-CQA($g()$) for queries $g()$ of the form (1), particularly regarding the conditions under which GLB-CQA($g()$) can be expressed in an extension of first-order logic with aggregate operators. However, the analogous question for LUB-CQA($g()$) is not systematically addressed in that work and is the focus of the present paper. Thus, we study the following function problem for any AGG-query $g()$ of the form (1):

Problem LUB-CQA($g()$).
Input: A database instance **db** that may violate its primary key constraints.
Output: $\max\{[\![g()]\!]^{\mathbf{r}}) \mid \mathbf{r}$ is a repair of **db**$\}$.

Here, $[\![g()]\!]^{\mathbf{r}}$ denotes the result of $g()$ on $\mathbf{r}$. The function problem GLB-CQA(q) is obtained by replacing max with min in the above problem. Note that since every database has a finite number of repairs, the glb and lub are equal to the min and max, respectively. In [24], we considered the case where an aggregate operator is undefined over the empty multiset. In contrast, in the current paper, we assume that $\mathcal{F}_{\texttt{AGG}}(\emptyset)$ is well-defined and equals 0, which, as explained in Section 4.1, is a reasonable assumption for the monotone and associative aggregate operators we consider.

We write AGGR[sjfBCQ] for the class of all queries $\texttt{AGG}(r) \leftarrow q(\vec{u})$ such that $\exists\vec{u}(q(\vec{u}))$ is a self-join-free Boolean conjunctive query (i.e., not containing two distinct atoms with the same relation name). We are interested in the following complexity classification problem: Given a query $g()$ in AGGR[sjfBCQ], determine the computational complexity of the function problem LUB-CQA($g()$).

The restriction to self-join-free queries is due to the following reason. The computational complexity of CERTAINTY(q) is well understood whenever q is self-join-free [29] – an understanding we build upon in the current paper – but it remains largely unexplored for queries with self-joins. Only recently has it been established for self-joins of size two [35]. In particular, the concept of an attack graph, which is a key tool used in [29] and in the current paper, loses its meaning and usefulness when self-joins are present. Given the limited understanding of CQA for self-joins, we exclude self-joins in the current paper.

Dealers	*Name*	*Town*
†	Smith	Boston
	Smith	New York
	Smith	Philadelphia
†	James	Boston

Stock	*Product*	*Town*	*Qty*
	Tesla X	Boston	35
†	Tesla X	Boston	40
†	Tesla Y	Boston	35
	Tesla Y	New York	72
†	Tesla Y	New York	73

Figure 1 Database instance $\mathbf{db}_{\mathsf{Stock}}$. Blocks are separated by dashed lines.

The following example illustrates the foregoing with a concrete case. It also demonstrates that, for queries $g() = \mathtt{SUM}(r) \leftarrow q(\vec{u})$, GLB-CQA($g()$) agrees with CERTAINTY($\exists \vec{u}\,(q(\vec{u}))$) insofar as it seeks to minimize the set of embeddings, in contrast to LUB-CQA($g()$), which maximizes the set of embeddings. This difference explains why the two problems require different techniques.

▶ **Example 1.** The database of Fig. 1 records the quantity of products in stock in various towns (relation *Stock*) and the town of operation for each dealer (relation *Dealers*). The primary keys are underlined, and blocks are separated by dashed lines. The inconsistencies concern Smith's town of operation, and the stock levels of Tesla X and Tesla Y in, respectively, Boston and New York. For the example database of Fig. 1, the following query returns the total quantity of cars in stock in Smith's town of operation:

$$\mathtt{SUM}(y) \leftarrow \mathit{Dealers}(\text{“}\underline{\text{Smith}}\text{”}, t), \mathit{Stock}(\underline{p, t}, y).$$

In Fig. 1, the repair composed of the tuples preceded by † yields the answer 75 ($= 40 + 35$), which is the greatest result achievable among all repairs. Notably, the smallest result among all repairs is 0, obtained by a repair that picks the tuple ⟨“Smith”, “Philadelphia”⟩ from *Dealers*, thereby falsifying the body of this SUM-query. ⌟

In our complexity study, we specifically aim to understand under which conditions LUB-CQA($g()$) is solvable through rewriting in an aggregate logic, denoted AGGR[FOL], which extends first-order logic with aggregate operators along the lines of [22]. So our central problem takes as input a numerical query $g()$ in AGGR[sjfBCQ], and asks whether or not there is a numerical query $\varphi()$ in AGGR[FOL] that solves LUB-CQA($g()$); moreover, when such $\varphi()$ exists, we are interested in constructing it, a task loosely referred to as “(consistent) lub rewriting of $g()$ in AGGR[FOL].” A practical motivation for focusing on AGGR[FOL] is that formulas in this logic are well-suited for implementation in SQL, allowing them to benefit from existing DBMS technology. Before presenting the major contributions of the current paper, we recall a notable consequence from [24] which holds true under the assumption $\mathcal{F}_{\mathtt{SUM}}(\emptyset) = 0$:

▶ **Theorem 2** ([24]). *Let r be a numerical variable or a positive number. For each query $g() := \mathtt{SUM}(r) \leftarrow q(\vec{u})$ in* AGGR[sjfBCQ] *over $\mathbb{Q}_{\geq 0}$,* GLB-CQA($g()$) *is expressible in* AGGR[FOL] *if and only if the attack graph of $\exists \vec{u}(q(\vec{u}))$ is acyclic.*

Our results show that the right-to-left implication in Theorem 2 fails if GLB-CQA($g()$) is replaced with LUB-CQA($g()$), while keeping all other conditions the same. That is, acyclicity of the attack graph is not a sufficient condition for LUB-CQA($g()$) to be expressible in AGGR[FOL]. To address this, we will introduce *κ-acyclicity*, a polynomial-time decidable syntactic property of sjfBCQ queries. Our first main result can then be stated as follows:

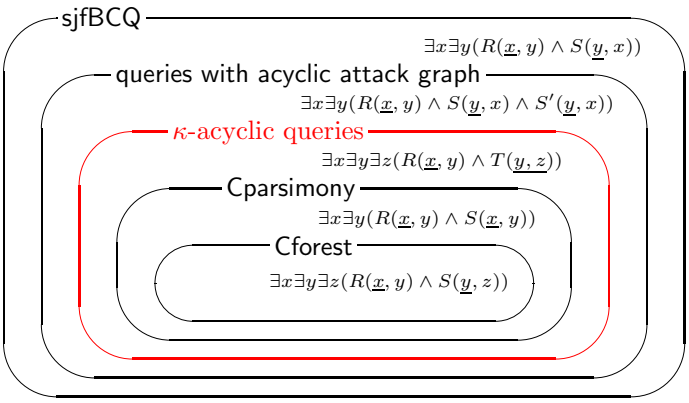

Figure 2 Subclasses of self-join-free Boolean conjunctive queries, with example queries. The class of κ-acyclic queries is the largest class that allows lub rewriting for SUM-queries. The class of queries with an acyclic attack graph is the largest class that allows glb rewriting.

▶ **Theorem 3** (Separation Theorem for SUM). *Let r be a numerical variable or a positive number. For each query $g() := \texttt{SUM}(r) \leftarrow q(\vec{u})$ in* AGGR[sjfBCQ] *over $\mathbb{Q}_{\geq 0}$,* LUB-CQA$(g())$ *is expressible in* AGGR[FOL] *if and only if $\exists \vec{u}(q(\vec{u}))$ is κ-acyclic.*

COUNT-queries of the form $\texttt{COUNT}(r) \leftarrow q(\vec{u})$ can be expressed as $\texttt{SUM}(1) \leftarrow q(\vec{u})$, and are therefore also covered by Theorem 3. Our second main result is that the right-to-left implication in Theorem 3 extends from SUM-queries to all AGG-queries in AGGR[sjfBCQ] whose aggregate operator is monotone and associative, two properties that will be precisely defined in Section 4.1.

▶ **Theorem 4.** *Let $g() := \texttt{AGG}(r) \leftarrow q(\vec{u})$ be a query in* AGGR[sjfBCQ] *whose aggregate operator is monotone and associative. If $\exists \vec{u}(q(\vec{u}))$ is κ-acyclic, then* LUB-CQA$(g())$ *can be expressed by a query in* AGGR[FOL] *(and such a query can be effectively constructed in polynomial time in the size of $q(\vec{u})$).*

The converse of Theorem 4 does not hold in general. For example, LUB-CQA$(g())$ is expressible in AGGR[FOL] for all MAX-queries in AGGR[sjfBCQ], since the maximum over all repairs coincides with the result of the MAX-query on the original database.

Our results add κ-acyclic queries to the query classes in Fig. 2 and prove the correctness of the inclusions shown. It was already known that lub rewriting of SUM-queries is possible for the classes Cforest [18] and Cparsimony [23]. The notion of κ-acyclicity is new and identifies the largest class of queries that allow lub rewriting of SUM-queries. Interestingly, Theorem 2 and our results show that there are SUM-queries that allow glb rewriting but not lub rewriting, specifically those that are not κ-acyclic but have an acyclic attack graph.

As will be argued in Section 8, all our results naturally extend to queries with a GROUP BY feature. For example, the following SQL query returns, for each dealer, the total quantity of products in stock in their town of operation:

```
SELECT   D.Name, SUM(S.Qty)
FROM     Dealers AS D, Stock AS S
WHERE    D.Town = S.Town
GROUP BY D.Name
```

This query is stated as follows in the extended datalog syntax of [10]:

$$(x, \texttt{SUM}(y)) \leftarrow \mathit{Dealers}(\underline{x}, t), \mathit{Stock}(\underline{p, t}, y).$$

Range semantics are obtained by replacing x with every possible dealer name ("Smith" and "James," in our example), and calculating the glb and lub as before.

This paper is organized as follows. Section 2 discusses related work, and Section 3 introduces preliminaries. In Section 4, we formally define AGGR[sjfBCQ], the class of queries for which we want to compute range consistent query answers, along with the aggregate logic AGGR[FOL], which serves as the target language for our rewritings. We also introduce a key auxiliary result, the *Reduction Lemma*. Section 5 introduces the new notion of κ-acyclicity and establishes some of its properties. The expressibility result of Theorem 4 is discussed in Section 6. Section 7 establishes the inexpressibility direction of Theorem 3, showing that lub rewriting in AGGR[FOL] is impossible for SUM-queries that lack κ-acyclicity. Section 8 shows that all queries in Cparsimony [23] are κ-acyclic. That section also explains that, while our results are proved for closed queries, they can be readily extended to accommodate free variables. Finally, Section 9 concludes the paper.

This paper is largely based on [2, Chapter 7], in which the proofs are developed. Nevertheless, Section 6.2 introduces a novel approach that departs from that of [2].

2 Related Work

Consistent query answering (CQA) originated in 1999 with a seminal paper by Arenas, Bertossi, and Chomicki [3], which introduced the notions of repair and consistent answer. Two years later, the same authors introduced the *range semantics* (with lower and upper bounds) for queries with aggregation [4, 5][6, Chapter 5], which has been commonly adopted ever since.

Ariel Fuxman defined in his PhD thesis [18] a syntactic class of self-join-free conjunctive queries, called Cforest, whose extension with aggregate operators (MAX, MIN, SUM, COUNT) yields Caggforest. Range semantics for queries in Caggforest can be obtained by executing two first-order queries (one for lower bounds, and one for upper bounds) followed by simple aggregation steps; this technique was subsequently implemented in the ConQuer system [19] through rewriting in SQL. The class Cparsimony [23] is an extension of Cforest that contains all (and only) self-join-free conjunctive queries for which Fuxman's technique applies for COUNT. In [24], Fuxman's technique was extended to a richer rewriting in the first-order aggregate logic AGGR[FOL], but only for greatest lower bounds (see Theorem 2). The current paper extends this work to least upper bounds. Query rewriting in AGGR[FOL] is different from the AggCAvSAT approach [12], which uses powerful SAT solvers for computing range semantics, and thus can solve queries that are beyond the computational power of AGGR[FOL].

CQA for queries q without aggregation, with respect to primary keys, has been studied in considerable depth. Its decision variant, which was coined CERTAINTY(q) in 2010 [36], asks whether a Boolean query q holds in every repair of a given database instance. A systematic study of its complexity for sjfBCQ queries had started already in 2005 [20], and was eventually solved in two journal articles by Koutris and Wijsen [29, 32], as follows: for every sjfBCQ query q, CERTAINTY(q) is either in FO, L-complete, or coNP-complete, and it is decidable, given q, which case applies. This complexity classification extends to non-Boolean queries by treating free variables as constants. Other extensions beyond this trichotomy deal with foreign keys [21], more than one key per relation [31], negated atoms [30], restricted self-joins [27, 28, 35], or naturally ordered positive semirings [26]. For unions of conjunctive

queries q, Fontaine [17] established interesting relationships between $\mathsf{CERTAINTY}(q)$ and Bulatov's dichotomy theorem for conservative CSP [8]. Recently, Figueira et al. [15, 16] proposed a polynomial-time algorithm for solving $\mathsf{CERTAINTY}(q)$ for conjunctive queries q, including those with self-joins, and showed that it can replace all polynomial-time algorithms found in [27, 29]. Overviews of two decades of theoretical research in CQA can be found in [7, 25, 38].

3 Preliminaries

We assume denumerable sets **var** and **dom** of variables and constants respectively. The set **dom** includes $\mathbb{Q}_{\geq 0}$, the set of non-negative rational numbers. The set of *numerical variables* is a subset of **var**.

We assume denumerably many *relation names*. Every relation name is associated with a *signature*, which is a triple (n, k, J) where n is the *arity*, $\{1, \ldots, k\}$ is called the *primary key*, and $J \subseteq \{1, \ldots, n\}$ is the set of *numerical positions* (also called *numerical columns*). This relation name is *full-key* if $n = k$. Note that each relation name R is associated with exactly one key constraint, which is determined by the signature of R. For an n-tuple $\vec{x} = (x_1, \ldots, x_n)$, we write $|\vec{x}|$ to denote its *arity* n. We often blur the distinction between a sequence $(x_1, \ldots, x_n)$ of distinct variables and the set $\{x_1, \ldots, x_n\}$, which is also denoted $\mathsf{vars}(\vec{x})$. If $\vec{x} = (x_1, \ldots, x_n)$ and $\vec{y} = (y_1, \ldots, y_m)$, then we define their *concatenation* $\vec{x} \cdot \vec{y}$ as the tuple $(x_1, \ldots, x_n, y_1, \ldots, y_m)$.

Atoms, facts, and database instances. Let R be a relation name of signature (n, k, J). An *atom* is an expression $R(u_1, \ldots, u_n)$ where each u_i is either a constant or a variable, and for every $j \in J$, u_j is a numerical variable or a number in $\mathbb{Q}_{\geq 0}$. It is common to underline positions of the primary key. An atom is said to be *full-key* if its relation name is full-key. If F is an atom, then $\mathsf{vars}(F)$ is the set of variables that occur in F, and $\mathsf{Key}(F)$ is the set of variables that occur in F at a position of the primary key. Further, we define $\mathsf{notKey}(F) := \mathsf{vars}(F) \setminus \mathsf{Key}(F)$. A *fact* is an atom without variables. A fact with relation name R is also called an R-fact. Two facts $R_1(\underline{\vec{a}_1}, \vec{c}_1)$ and $R_2(\underline{\vec{a}_2}, \vec{c}_2)$ are said to be *key-equal* if $R_1 = R_2$ and $\vec{a}_1 = \vec{a}_2$.

A *database instance* is a finite set of facts. If R is a relation name, then the *R-relation of* **db** is the set of all R-facts in **db**. A *database instance* is *consistent* if it does not contain two distinct facts that are key-equal. Primary keys need not be explicitly specified, as they are determined by the predefined relation signatures. The *block* of a fact $R(\underline{\vec{a}}, \vec{c})$ in **db**, denoted $R(\underline{\vec{a}}, *)$, is the set of all facts in **db** that are key-equal to that fact.

Valuation. A *valuation* over a finite set U of variables is a total mapping θ from U to **dom** such that $\theta(r) \in \mathbb{Q}_{\geq 0}$ for every numerical variable r. For a valuation θ over U, we write $\mathsf{dom}(\theta)$ to denote its *domain* U. A valuation θ over U is extended to every element u in $\mathbf{var} \cup \mathbf{dom}$ by letting $\theta(u) = u$ for every $u \notin U$.

Let θ be a valuation. If F is the atom $R(u_1, \ldots, u_n)$, then $\theta(F) := R(\theta(u_1), \ldots, \theta(u_n))$. If q is a set of atoms, then $\theta(q) := \{\theta(F) \mid F \in q\}$. Notably, every variable in $\mathsf{vars}(q) \setminus \mathsf{dom}(\theta)$ remains a variable in $\theta(q)$, where $\mathsf{vars}(q) = \bigcup_{F \in q} \mathsf{vars}(F)$.

Partial valuation. Let **db** be a database instance, and $\varphi(\vec{x})$ a first-order formula with free variables $\vec{x}$. Let θ be a valuation. Then we write $(\mathbf{db}, \theta) \models \varphi(\vec{x})$ to denote that θ can be extended to a valuation θ' over $\mathsf{dom}(\theta) \cup \mathsf{vars}(\vec{x})$ such that for $\vec{a} := \theta'(\vec{x})$, we have

Figure 3 Attack graphs for two queries in sjfBCQ. The query q_0' on the right is derived from the query on the left by replacing $T(\underline{z},x)$ with the full-key atom $T'(\underline{z,x})$.

$\mathbf{db} \models \varphi(\vec{a})$ using standard semantics (see, e.g., [33, p. 15]). Typically, but not necessarily, $\mathsf{dom}(\theta) \subseteq \mathsf{vars}(\vec{x})$. If φ has no free variables, then we write $\mathbf{db} \models \varphi$ instead of $(\mathbf{db}, \varnothing) \models \varphi$, where $\varnothing$ is the empty valuation.

Repairs and CERTAINTY(φ). A *repair* of a database instance is a $\subseteq$-maximal consistent subset of it. We write $\mathsf{rset}(\mathbf{db})$ for the set of repairs of a database instance $\mathbf{db}$. For a closed formula φ, $\mathsf{CERTAINTY}(\varphi)$ is the decision problem that takes a database instance $\mathbf{db}$ as input and determines whether or not every repair of $\mathbf{db}$ satisfies φ.

Boolean Conjunctive Queries. A *self-join-free Boolean conjunctive query* q is a closed first-order formula $\exists\vec{u}(R_1(\vec{v}_1) \wedge \cdots \wedge R_n(\vec{v}_n))$, where each $R_i(\vec{v}_i)$ is an atom, $\vec{u}$ is a sequence containing every variable occurring in some $\vec{v}_i$, and $i \neq j$ implies $R_i \neq R_j$. The conjunction $R_1(\vec{v}_1) \wedge \cdots \wedge R_n(\vec{v}_n)$, whose free variables are $\vec{u}$, is called the *body* of the query. We write sjfBCQ for the set of self-join-free Boolean conjunctive queries. We often blur the distinction between the Boolean query q, its body, and the set $\{R_1(\vec{v}_1), \ldots, R_n(\vec{v}_n)\}$. For example, if F is an atom of (the body of) q, then $q \setminus \{F\}$ is the query obtained from q by deleting F from its body.

Gaifman graph. The *Gaifman graph of* q, denoted $\mathcal{G}aifman(q)$, is an undirected simple graph whose vertex-set is $\mathsf{vars}(q)$. There is an edge between x and y if $x \neq y$ and some atom of q contains both x and y.

Guardedness. Let q_1 and q_2 be two queries in sjfBCQ. We say that q_1 *is guarded by* q_2, denoted $q_1 \preccurlyeq_{\mathsf{g}} q_2$, if for every atom F in q_1, there exists an atom G in q_2 such that $\mathsf{vars}(F) \subseteq \mathsf{vars}(G)$. We write $q_1 \sim_{\mathsf{g}} q_2$ if both $q_1 \preccurlyeq_{\mathsf{g}} q_2$ and $q_2 \preccurlyeq_{\mathsf{g}} q_1$. It is straightforward that $q_1 \sim_{\mathsf{g}} q_2$ implies $\mathcal{G}aifman(q_1) = \mathcal{G}aifman(q_2)$ (but the converse does not hold).

Attack graph. Attack graphs for queries q in sjfBCQ were first introduced in [37], later generalized in [29], and have since been used in several studies, e.g., [16, 24, 26]. We write $\mathcal{K}(q)$ for the set of functional dependencies that contains $\mathsf{Key}(F) \to \mathsf{vars}(F)$ whenever $F \in q$. For $F \in q$, we define $F^{+,q} := \{x \in \mathsf{vars}(q) \mid \mathcal{K}(q \setminus \{F\}) \models \mathsf{Key}(F) \to x\}$, where $\models$ is the standard notion of logical implication. An atom F of q is said to *attack* a variable x, denoted $F \overset{q}{\rightsquigarrow} x$, if $\mathcal{G}aifman(q)$ contains a (possibly empty) path between some variable in $\mathsf{notKey}(F)$ and x such that no variable on the path belongs to $F^{+,q}$. The *attack graph* of q is a directed simple graph whose vertices are the atoms of q. There is a directed edge from F to G, denoted $F \overset{q}{\rightsquigarrow} G$, if F attacks some variable of $\mathsf{vars}(G)$.

Whenever a query in sjfBCQ is clear from the context, we can use a relation name as a shorthand for the unique atom with that relation name in the query. For example, in the following example, R is used as a shorthand for the atom $R(\underline{x, y}, z)$.

▶ **Example 5.** Consider the query q_0 from sjfBCQ shown on the left side of Fig. 3. The directed edges represent attacks. We have $R^{+,q_0} = \{x, y\}$, $S^{+,q_0} = \{z, x\}$, and $T^{+,q_0} = \{z, x\}$. The sequence (z) entails $R \overset{q_0}{\rightsquigarrow} S$ and $R \overset{q_0}{\rightsquigarrow} T$. The query q_0' on the right side is obtained from q_0 by replacing the T-atom with a full-key atom, which introduces a cycle in the attack graph. ⌟

Let $q(\vec{u})$ now be a self-join-free conjunction of n atoms such that the attack graph of $\exists\vec{u}(q(\vec{u}))$ is acyclic. The following definitions are relative to a fixed topological sort $(R_1, \ldots, R_n)$ of q's attack graph. We define the following sequences of variables for $\ell \in \{1, \ldots, n\}$:

- $\vec{u}_\ell$ contains all (and only) variables of $\bigcup_{i=1}^{\ell} \mathsf{vars}(R_i)$. Thus, $\vec{u}_n = \vec{u}$;
- $\vec{x}_\ell$ contains the variables of $\mathsf{Key}(R_\ell)$ that do not already occur in $\bigcup_{i=1}^{\ell-1} \mathsf{vars}(R_i)$; and
- $\vec{y}_\ell$ contains the variables of $\mathsf{notKey}(R_\ell)$ that do not already occur in $\bigcup_{i=1}^{\ell-1} \mathsf{vars}(R_i)$.

Moreover, we define $\vec{u}_0 = ()$, the empty sequence. With this notation, we have that for every $\ell \in \{1, \ldots, n\}$, $\vec{u}_\ell = (\vec{u}_{\ell-1}, \vec{x}_\ell, \vec{y}_\ell)$.

4 Aggregate Logic and CQA

In this section, we first define the notion of *aggregate operator* and then introduce the logic AGGR[FOL] which will serve as the target language for our rewritings. We formally define consistent least upper bounds, denoted LUB-CQA($g(\vec{x})$), for arbitrary numerical terms $g(\vec{x})$ in AGGR[FOL] with free variables $\vec{x}$. Our study will subsequently focus on closed numerical terms $g()$ in AGGR[sjfBCQ], a subclass of AGGR[FOL] introduced in Definition 9. Finally, this section introduces an important auxiliary lemma, called the *Reduction Lemma*.

4.1 Aggregating Non-Negative Numbers

A *(positive) aggregate operator* is a function $\mathcal{F}$ that maps each finite multiset of non-negative rational numbers to a non-negative rational number.

In the following definitions, all multisets are understood to be multisets of non-negative rational numbers. An aggregate operator $\mathcal{F}$ is *associative* if for all multisets X and Y, we have:

$$\mathcal{F}(X \uplus Y) = \mathcal{F}(\{\{\mathcal{F}(X)\}\} \uplus Y), \tag{2}$$

where $\uplus$ denotes union of multisets. By letting $X = \emptyset$ in Eq. 2, we obtain that if $\mathcal{F}$ is associative, then for all multisets Y, we have $\mathcal{F}(Y) = \mathcal{F}(\{\{\mathcal{F}(\emptyset)\}\} \uplus Y)$. This condition holds for $\mathcal{F}_{\texttt{SUM}}$ and $\mathcal{F}_{\texttt{MAX}}$ over $\mathbb{Q}_{\geq 0}$ if and only if we define $\mathcal{F}_{\texttt{SUM}}(\emptyset) := 0$ and $\mathcal{F}_{\texttt{MAX}}(\emptyset) := 0$.

▶ **Example 6.** Examples of associative aggregate operators are $\mathcal{F}_{\texttt{SUM}}$ and $\mathcal{F}_{\texttt{MAX}}$. On the other hand, $\mathcal{F}_{\texttt{AVG}}$, $\mathcal{F}_{\texttt{COUNT}}$, and $\mathcal{F}_{\texttt{SUM-DISTINCT}}$ are not associative. For instance, $\mathcal{F}_{\texttt{COUNT}}(\{\{5, 6, 7, 8\}\}) = 4$ and $\mathcal{F}_{\texttt{COUNT}}(\{\{\mathcal{F}_{\texttt{COUNT}}(\{\{5, 6, 7\}\}), 8\}\}) = \mathcal{F}_{\texttt{COUNT}}(\{\{3, 8\}\}) = 2$. Also, $\mathcal{F}_{\texttt{MIN}}$ lacks associativity over $\mathbb{Q}_{\geq 0}$ because any choice for $\mathcal{F}_{\texttt{MIN}}(\emptyset)$ disrupts associativity. In particular, if we define $\mathcal{F}_{\texttt{MIN}}(\emptyset) := 0$, then $\mathcal{F}_{\texttt{MIN}}(\{\{1\}\}) \neq \mathcal{F}_{\texttt{MIN}}(\{\{\mathcal{F}_{\texttt{MIN}}(\emptyset)\}\} \uplus \{\{1\}\}) = 0$. ⌟

An aggregate operator $\mathcal{F}$ is *monotone* if for all $m \geq 0$ and every (possibly empty) multiset Y, we have:

$$\mathcal{F}(\{\{x_1, \ldots, x_m\}\}) \leq \mathcal{F}(\{\{x_1', \ldots, x_m'\}\} \uplus Y) \text{ whenever } x_i \leq x_i' \text{ for every } i. \tag{3}$$

By letting $m = 0$ in Eq. 3, we obtain that if $\mathcal{F}$ is monotone, then for all multisets Y, we have $\mathcal{F}(\emptyset) \leq \mathcal{F}(Y)$. Again, this condition holds for $\mathcal{F}_{\texttt{SUM}}$ and $\mathcal{F}_{\texttt{MAX}}$ over $\mathbb{Q}_{\geq 0}$ if and only if we define $\mathcal{F}_{\texttt{SUM}}(\emptyset) := 0$ and $\mathcal{F}_{\texttt{MAX}}(\emptyset) := 0$.

▶ **Example 7.** Examples of monotone aggregate operators are $\mathcal{F}_{\texttt{SUM}}$, $\mathcal{F}_{\texttt{MAX}}$, and $\mathcal{F}_{\texttt{COUNT}}$. Note that $\mathcal{F}_{\texttt{MIN}}$ is not monotone since $\mathcal{F}_{\texttt{MIN}}(\{\{3\}\}) > \mathcal{F}_{\texttt{MIN}}(\{\{2,3\}\})$. `COUNT-DISTINCT` also lacks monotonicity: if we increase 3 to 4 in the multiset $\{\{3,4\}\}$, the number returned by `COUNT-DISTINCT` drops from 2 to 1. ⌟

Significantly, an aggregate operator $\mathcal{F}$ that is not monotone in general may become monotone under a restriction of its underlying domain. For example, $\mathcal{F}_{\texttt{PRODUCT}}$, defined by $\mathcal{F}_{\texttt{PRODUCT}}(\{\{x_1, \ldots, x_m\}\}) := \Pi_{i=1}^{m} x_i$, is not monotone over $\mathbb{Q}_{\geq 0}$ (because $\mathcal{F}_{\texttt{PRODUCT}}(\{\{1\}\}) > \mathcal{F}_{\texttt{PRODUCT}}(\{\{1, \frac{1}{2}\}\})$), but is monotone over $\mathbb{Q}_{\geq 1}$.

4.2 The Logic AGGR[FOL]

Our treatment of aggregate logic follows the approach in [22, 33]. We write AGGR[FOL] for the extension of predicate calculus with numerical terms introduced next.

Every formula in classical predicate calculus is also a formula in AGGR[FOL]. In AGGR[FOL], terms are not restricted to variables and constants; they also include numerical terms, as defined below. Let $q(\vec{x}, \vec{y})$ be a formula in AGGR[FOL], where $\vec{x}$, $\vec{y}$ are disjoint sequences that together contain each free variable of q exactly once. A *primitive numerical term* is either a non-negative rational number or a numerical variable in $\vec{x} \cdot \vec{y}$. For every possible aggregate operator $\mathcal{F}$, a primitive numerical term r, and a formula $q(\vec{x}, \vec{y})$, we have a new *numerical term*

$$g(\vec{x}) := \mathsf{Aggr}_{\mathcal{F}} \vec{y} \, [r, q(\vec{x}, \vec{y})] \, .$$

Variables $\vec{y}$ that are free in $q(\vec{x}, \vec{y})$ become bound in $\mathsf{Aggr}_{\mathcal{F}} \vec{y} \, [r, q(\vec{x}, \vec{y})]$; in this respect $\mathsf{Aggr}_{\mathcal{F}} \vec{y}$ behaves like a sort of quantification over $\vec{y}$. Next, we define the semantics.

Let $\vec{a}$ be a sequence of constants of length $|\vec{x}|$. The value $g(\vec{a})$ on a database instance $\mathbf{db}$, denoted $[\![g(\vec{a})]\!]^{\mathbf{db}}$, is calculated as follows. Let $\{\theta_1, \theta_2, \ldots, \theta_m\}$ be a (possibly empty) $\subseteq$-maximal set of valuations over $\vec{x} \cdot \vec{y}$ such that $\theta_i(\vec{x}) = \vec{a}$ and $(\mathbf{db}, \theta_i) \models q(\vec{x}, \vec{y})$ for $1 \leq i \leq m$. Then $[\![g(\vec{a})]\!]^{\mathbf{db}} = \mathcal{F}(\{\{\theta_1(r), \ldots, \theta_m(r)\}\})$. Note that the argument of $\mathcal{F}$ is in general a multiset, since $\theta_i(r)$ may be equal to $\theta_j(r)$ for $i \neq j$. A numerical term without free variables is also called a *numerical query*, denoted $g()$.

▶ **Example 8.** If $g_0(t) := \mathsf{Aggr}_{\mathcal{F}_{\texttt{SUM}}} (p, z) \, [z, \mathit{Stock}(p, t, z)]$, then on the example database of Fig. 1, $g_0(\text{“Boston”}) = 110$, the sum of quantities in Boston. This numerical term can be used in other formulas. For example, $q_0(t, y) := \exists p \exists z (\mathit{Stock}(p, t, z)) \wedge y = g_0(t)$ returns, for each town t, the total quantity y of products stored in t. ⌟

4.3 Least Upper Bounds Across All Repairs

For each numerical term $g(\vec{x}) := \mathsf{Aggr}_{\mathcal{F}} \vec{y} \, [r, q(\vec{x}, \vec{y})]$, we define $\mathsf{LUB\text{-}CQA}(g(\vec{x}))$ relative to a database instance $\mathbf{db}$. Let $\vec{a}$ be a sequence of constants of length $|\vec{x}|$. Then,

$$[\![\mathsf{LUB\text{-}CQA}(g(\vec{a}))]\!]^{\mathbf{db}} := \max \{ [\![g(\vec{a})]\!]^{\mathbf{r}} \mid \mathbf{r} \in \mathsf{rset}(\mathbf{db}) \} \, .$$

A general research question is: under which conditions can $\mathsf{LUB\text{-}CQA}(g(\vec{x}))$ be expressed in AGGR[FOL]? In this paper, we study this question for the case of numerical terms

$g(\vec{x}) := \mathsf{Aggr}_{\mathcal{F}\vec{y}}[r, q(\vec{x},\vec{y})]$ where $q(\vec{x},\vec{y})$ is a self-join-free conjunction of atoms. For readability, we often express such a numerical term $g(\vec{x})$ using the following datalog-like syntax:

$$(\vec{x}, \mathtt{AGG}(r)) \leftarrow q(\vec{x},\vec{y}),$$

where the aggregate symbol AGG is interpreted by the aggregate operator $\mathcal{F}$, which is often made explicit by writing $\mathcal{F}_{\mathtt{AGG}}$. For instance, SUM is interpreted by $\mathcal{F}_{\mathtt{SUM}}$, and MAX by $\mathcal{F}_{\mathtt{MAX}}$.

In the initial technical treatment, we assume that $\vec{x}$ is empty, i.e., we are dealing with numerical terms $g() := \mathsf{Aggr}_{\mathcal{F}\vec{y}}[r, q(\vec{y})]$ without free variables, which are conveniently expressed as $\mathtt{AGG}(r) \leftarrow q(\vec{y})$. In Section 8, we treat the extension to free variables. The following definition pinpoints the class of queries we are interested in.

▶ **Definition 9.** AGGR[sjfBCQ] *is defined as the class of numerical queries* $\mathsf{Aggr}_{\mathcal{F}\vec{y}}[r, q(\vec{y})]$ *where* $q(\vec{y})$ *is a self-join-free conjunction of atoms, and* r *is either a numerical variable or a non-negative number. An alternative syntax is* $\mathtt{AGG}(r) \leftarrow q(\vec{y})$*, where the aggregate symbol* AGG *is interpreted by* $\mathcal{F}$ *(which is often made explicit by writing* $\mathcal{F}_{\mathtt{AGG}}$ *instead of* $\mathcal{F}$*). We call* $\mathtt{AGG}(r)$ *the* head*, and* $q(\vec{y})$ *the* body*.*

4.4 The Reduction Lemma

Let P_1 and P_2 be function problems whose output is a single number. If I is an instance of P_j, we write $P_j(I)$ to denote the output of P_j on input I, where $j \in \{1, 2\}$. We say that P_1 is first-order reducible to P_2, denoted $P_1 \leq_{\mathsf{FO}} P_2$ if there exists a first-order definable function f such that for every instance I of P_1, $f(I)$ is an instance of P_2 such that $P_1(I) = P_2(f(I))$. We write $P_1 \leq_{\mathsf{nr\text{-}datalog}} P_2$ if the reduction can even be expressed by a nonrecursive datalog (nr-datalog) program, as defined in [1, page 62–63].

In the statement of the following lemma, referred to as the *Reduction Lemma*, we use q_1 as a shorthand for $\exists\vec{u}(q_1(\vec{u}))$ for readability.

▶ **Lemma 10** (Reduction Lemma)**.** *Let* $g_1() := \mathtt{AGG}(r) \leftarrow q_1(\vec{u})$ *and* $g_2() := \mathtt{AGG}(r) \leftarrow q_2(\vec{u})$ *be queries in* AGGR[sjfBCQ] *such that* $\mathcal{F}_{\mathtt{AGG}}$ *is monotone. If* $q_1 \preccurlyeq_{\mathsf{g}} q_2$ *and* $\mathcal{K}(q_1) \equiv \mathcal{K}(q_2)$*, then* $\mathsf{LUB\text{-}CQA}(g_1()) \leq_{\mathsf{nr\text{-}datalog}} \mathsf{LUB\text{-}CQA}(g_2())$.

We illustrate Lemma 10 below and show that it does not hold for greatest lower bounds.

▶ **Example 11.** Consider the queries $g_1() := \mathtt{SUM}(1) \leftarrow q_0(x, y, z)$ and $g_2() := \mathtt{SUM}(1) \leftarrow q_0'(x, y, z)$, where q_0 and q_0' are shown in Fig. 3. Their bodies are related by $\sim_{\mathsf{g}}$ and induce the same set of functional dependencies (notably, $\{xy \to z, z \to x\}$). The Reduction Lemma implies that $\mathsf{LUB\text{-}CQA}(g_1())$ and $\mathsf{LUB\text{-}CQA}(g_2())$ are equivalent under first-order reductions. However, this equivalence does not hold for glb: indeed, by Theorem 2, $\mathsf{GLB\text{-}CQA}(g_1())$ is in AGGR[FOL], whereas $\mathsf{GLB\text{-}CQA}(g_2())$ is not. ⌟

5 Kernels and κ-Acyclicity

We assume that the reader is familiar with the notion of a *minimal cover* of a set of functional dependencies (FDs), which can be computed in polynomial time [34][1, p. 257]. To recall briefly, a set Σ of FDs is a *minimal cover* (a.k.a. *irreducible*) if each functional dependency σ in Σ has the form $X \to A$, where A is an attribute, and σ is *left-reduced* and *non-redundant*. A set Σ of FDs is a minimal cover of another set Σ' of FDs if Σ is a minimal cover and $\Sigma \equiv \Sigma'$. We now extend this notion of irreducibility to sjfBCQ queries, introduce the notion of a *kernel*, and present some auxiliary lemmas.

▶ **Definition 12** (Irreducible query, kernel). *Let q be a query in* sjfBCQ*, and let $\widehat{q}$ denote the set of atoms in q that are not full-key. We say that q is* irreducible *if the following two conditions hold:*

1. *for every atom F of $\widehat{q}$, we have $\mathcal{K}(\widehat{q} \setminus \{F\}) \not\equiv \mathcal{K}(\widehat{q})$; and*
2. *the set $\{\mathsf{Key}(F) \to \mathsf{notKey}(F) \mid F \in \widehat{q}\}$ is irreducible.*

A kernel of q *is an irreducible query q' in* sjfBCQ *such that $\mathcal{K}(q') \equiv \mathcal{K}(q)$ and $q' \sim_{\mathsf{g}} q$.*

▶ **Example 13.** Consider the queries $q_0 = \{R(\underline{x,y},z), S(\underline{z},x), T(\underline{z},x)\}$ and $q_0' = \{R(\underline{x,y},z), S(\underline{z},x), T'(\underline{z,x})\}$ in Fig. 3. The query q_0 is not irreducible, because it does not satisfy the first condition in Definition 12. The query q_0' is irreducible. Since $\mathcal{K}(q_0) \equiv \mathcal{K}(q_0') \equiv \{xy \to z, z \to x\}$ and $q_0 \sim_{\mathsf{g}} q_0'$, it follows that q_0' is a kernel of q_0. Notably, the attack graph of q_0 is acyclic, whereas the attack graph of its kernel contains a cycle.

As a side remark, consider the query $g() := \mathtt{SUM}(1) \leftarrow q_0(x,y,z)$. By Theorem 2, GLB-CQA$(g())$ is expressible in AGGR[FOL], while, as we will see shortly, Theorem 3 implies that LUB-CQA$(g())$ is not expressible in AGGR[FOL]. This difference can be explained by the observation that T-blocks or S-blocks of size greater than 1 can be omitted when computing GLB-CQA$(g())$. For example, assume that **db** contains $S(\underline{a},c)$, $T(\underline{a},b_1)$, and $T(\underline{a},b_2)$ with $b_1 \neq b_2$. Then a repair **r** can select $S(\underline{a},c)$ and a fact $T(\underline{a},b_i)$, $i \in \{1,2\}$, such that $b_i \neq c$. Consequently, no homomorphism from $q_0(x,y,z)$ to **r** can map z to a. Preventing homomorphisms in this way minimizes the value of $g()$. On the other hand, computing LUB-CQA$(g())$ requires maximizing the number of embeddings. In that case, one would prefer selecting $S(\underline{a},d)$ and $T(\underline{a},d)$ if such a common d exists. ⌟

▶ **Lemma 14.** *Let q be a query in* sjfBCQ *that is irreducible. Let G be an atom in q that is not full-key. Then, there is a variable x such that $\mathsf{notKey}(G) = \{x\}$ and $G \overset{q}{\rightsquigarrow} x$.*

Proof. By the second item in Definition 12, $|\mathsf{notKey}(G)| = 1$. Hence, we can assume x such that $\mathsf{notKey}(G) = \{x\}$. Let $\widehat{q}$ denote the set of atoms in q that are not full-key. By the first item in Definition 12, $\mathcal{K}(\widehat{q} \setminus \{G\}) \not\equiv \mathcal{K}(\widehat{q})$. It must obviously be the case that $\mathcal{K}(\widehat{q} \setminus \{G\}) \not\models \mathsf{Key}(G) \to x$, which implies $G \overset{q}{\rightsquigarrow} x$. ◀

▶ **Lemma 15.** *Let q be a query in* sjfBCQ*. A kernel of q can be constructed in polynomial time in the size of q.*

Example 13 illustrates that an sjfBCQ query with an acyclic attack graph can have a kernel whose attack graph contains a cycle. Informally, this means that cycles in attack graphs can emerge during the construction of a kernel but, as stated in Proposition 18, they cannot disappear. Moreover, the following lemma states that all kernels of the same query either all have an acyclic attack graph or all have a cyclic attack graph.

▶ **Lemma 16.** *Let q be a query in* sjfBCQ*. If some kernel of q has an acyclic attack graph, then every kernel of q has an acyclic attack graph.*

The proof of Lemma 16 in [2, p. 133] also establishes the following result: if some kernel of q has an acyclic attack graph, then all kernels agree on their atoms that are not full-key, up to the choice of relation names. Lemma 16 allows the following definition.

▶ **Definition 17** (κ-acyclic). *A query q in* sjfBCQ *is κ-acyclic if it has a kernel with an acyclic attack graph (and hence, by Lemma 16, all of its kernels have acyclic attack graphs).*

As illustrated in the Venn diagram in Fig. 2, κ-acyclic queries have acyclic attack graphs:

▶ **Proposition 18.** *Every κ-acyclic query in* sjfBCQ *has an acyclic attack graph.*

6 Expressibility for κ-Acyclic Queries (Theorem 4)

We will demonstrate a slightly stronger result than Theorem 4: instead of proving expressibility in AGGR[FOL], we use a weaker logic – introduced next – that yields more readable rewritings.

6.1 The Logic AGGR[nr-datalog]

We assume that the reader is familiar with the concept of a *nonrecursive datalog program (*nr-datalog *program)*, whose definition can be found in [1, pp. 62–63]. We now extend such programs by incorporating aggregation. Assume that an nr-datalog program contains the following rule, where r is either a number or a numerical variable:

$$P(\vec{x}, r) \leftarrow R_1(\vec{v}_1), \ldots, R_n(\vec{v}_n). \tag{4}$$

Note that by the safety requirement of nr-datalog, if r is a variable, it also occurs in the body of the rule. We allow such a rule to be replaced with:

$$P(\vec{x}, \mathtt{AGG}(r)) \leftarrow R_1(\vec{v}_1), \ldots, R_n(\vec{v}_n),$$

where AGG is an aggregate symbol. The semantics on a given database **db** is as follows. Let $B(\vec{u})$ be the conjunction of all atoms in the body of the rule, where $\vec{u}$ is a shortest sequence containing every variable that occurs in the body of the rule. Let $\vec{a}$ be a sequence of constants of length $|\vec{x}|$. Let $\{\theta_1, \theta_2, \ldots, \theta_m\}$ be a $\subseteq$-maximal set of valuations over $\vec{u}$ such that $\theta_i(\vec{x}) = \vec{a}$ and $(\mathbf{db}, \theta_i) \models B(\vec{u})$ for $1 \leq i \leq m$. If $m \geq 1$, then the rule derives $P(\vec{a}, p)$ with $p = \mathcal{F}_{\mathtt{AGG}}(\{\{\theta_1(r), \ldots, \theta_m(r)\}\})$. Note that the argument of $\mathcal{F}_{\mathtt{AGG}}$ is in general a multiset, since $\theta_i(r)$ may be equal to $\theta_j(r)$ for $i \neq j$. In case $m = 0$, no such fact is derived.

We write AGGR[nr-datalog] for the language consisting of all programs obtained from nr-datalog programs by allowing aggregate operators in the head (but not in the body). Examples 19 and 21 show programs in AGGR[nr-datalog].

▶ **Example 19.** Retrieve the town(s) with the largest total quantity of stored products.

$$\begin{aligned} \mathit{TotalStockPerCity}(t, \mathtt{SUM}(z)) &\leftarrow \mathit{Stock}(p, t, z) \\ \mathit{MaxStock}(\mathtt{MAX}(s)) &\leftarrow \mathit{TotalStockPerCity}(t, s) \\ \mathit{Answer}(t) &\leftarrow \mathit{TotalStockPerCity}(t, s), \mathit{MaxStock}(s) \end{aligned}$$

Replacing $\mathtt{SUM}(z)$ and $\mathtt{MAX}(s)$ with z and s, respectively, in the above program yields a standard nr-datalog program, as required by AGGR[nr-datalog]. ⌟

6.2 Computing Consistent Least Upper Bounds in AGGR[nr-datalog]

We first show a weaker version of Theorem 4 for irreducible queries.

▶ **Lemma 20.** *Let $g() := \mathtt{AGG}(r) \leftarrow q(\vec{u})$ be a query in* AGGR[sjfBCQ] *whose aggregate operator is monotone and associative. If $\exists\vec{u}(q(\vec{u}))$ is irreducible and has an acyclic attack graph, then an* AGGR[nr-datalog] *program solving* LUB-CQA$(g())$ *can be constructed in polynomial time in the size of $q(\vec{u})$.*

We will now explain the AGGR[nr-datalog] program in the statement of Lemma 20, and explain how its correctness is proved. Let $\exists\vec{u}(q(\vec{u}))$ be a query in sjfBCQ with an acyclic attack graph. Let $(R_1, \ldots, R_n)$ be a topological sort of the attack graph. We assume that r is a variable in $\vec{u}$; our treatment can be easily specialized to the simpler case in which r is a

$$E_n(r, \texttt{AGG}(r)) \leftarrow S \tag{5}$$

$$K_\ell(\langle\!\langle \vec{u}_{\ell-1} \rangle\!\rangle, \vec{x}_\ell, \texttt{MAX}(r_\ell)) \leftarrow R_\ell, E_\ell(\langle\!\langle \vec{u}_\ell \rangle\!\rangle, r_\ell) \qquad \text{for } 1 \leq \ell \leq n \tag{6}$$

$$E_{\ell-1}(\langle\!\langle \vec{u}_{\ell-1} \rangle\!\rangle, \texttt{AGG}(m_\ell)) \leftarrow K_\ell(\langle\!\langle \vec{u}_{\ell-1} \rangle\!\rangle, \vec{x}_\ell, m_\ell) \qquad \text{for } 1 \leq \ell \leq n \tag{7}$$

Figure 4 Rules of an AGGR[nr-datalog] program for computing the $\mathcal{F}_{\texttt{AGG}}$ value of an AMCS, relative to a topological sort $(R_1, \ldots, R_n)$ of the attack graph of the query. S denotes an atom with $r \in \mathsf{vars}(S)$.

constant. For $i \in \{1, \ldots, n\}$, the sequences $\vec{u}_i$, $\vec{x}_i$, and $\vec{y}_i$ are defined in Section 3. Moreover, $\vec{u}_0 = ()$, the empty sequence. For $i \in \{0, 1, \ldots, n\}$, we define $\langle\!\langle \vec{u}_i \rangle\!\rangle$ as the restriction of $\vec{u}_i$ to those variables that belong to $\{r\} \cup \mathsf{vars}(\{R_{i+1}, R_{i+2}, \ldots, R_n\})$. In particular, $\langle\!\langle \vec{u}_0 \rangle\!\rangle = ()$ and $\langle\!\langle \vec{u}_n \rangle\!\rangle = (r)$.

Our program contains the rules shown in Fig. 4, for $\ell \in \{n, n-1, \ldots, 1\}$, where S is an arbitrarily chosen atom such that $r \in \mathsf{vars}(S)$. The IDB predicates are computed in the order E_n, K_n, E_{n-1}, K_{n-1}, …, E_1, K_1, E_0. Rule (6) is easily verified to be safe, and it is instructive to note that all variables of $\mathsf{Key}(R_\ell)$ occur in its head.

▶ **Example 21.** Let $g_0() := \texttt{SUM}(r) \leftarrow R_1(\underline{x}, y), R_2(\underline{y, z}, r)$. Then, $\vec{u}_1 = xy$, $\langle\!\langle \vec{u}_1 \rangle\!\rangle = y$, and $\vec{x}_2 = z$. We obtain a program with the following rules:

$$E_2(r, \texttt{SUM}(r)) \leftarrow R_2(\underline{y, z}, r)$$
$$K_2(y, z, \texttt{MAX}(r_2)) \leftarrow R_2(\underline{y, z}, r), E_2(r, r_2)$$
$$E_1(y, \texttt{SUM}(m_2)) \leftarrow K_2(y, z, m_2)$$
$$K_1(x, \texttt{MAX}(r_1)) \leftarrow R_1(\underline{x}, y), E_1(y, r_1)$$
$$E_0(\texttt{SUM}(m_1)) \leftarrow K_1(x, m_1)$$

Let **db** be the following database instance:

	R_1	$\underline{x}$	y
†		a_1	b_1
		a_1	b_2
†		a_2	b_2

	R_2	$\underline{y}$	$\underline{z}$	r
		b_1	c_1	1
†		b_1	c_1	4
†		b_1	c_2	3
		b_2	c_3	5
†		b_2	c_3	6

Note that $[\![\mathsf{LUB\text{-}CQA}(g())]\!]^{\mathbf{db}} = 13$, which is obtained by a repair that consists of the facts marked with †. Our program computes the following facts:

$E_2(1,1), E_2(4,4), E_2(3,3), E_2(5,5), E_2(6,6)$, $K_2(b_1, c_1, 4), K_2(b_1, c_2, 3), K_2(b_2, c_3, 6)$,
$E_1(b_1, 7), E_1(b_2, 6)$, $K_1(a_1, 7), K_1(a_2, 6)$,
$E_0(13)$.

We will show that, in general, the computed E_0-fact contains the value $\mathsf{LUB\text{-}CQA}(g())$. ⌟

▶ **Definition 22.** *If M is a set of valuations over a set U of variables that contains r, and $\mathcal{F}_{\texttt{AGG}}$ is an aggregation function, then $\mathcal{F}_{\texttt{AGG}}(M) = \mathcal{F}_{\texttt{AGG}}(\{\{\theta(r) \mid \theta \in M\}\})$.*

▶ **Definition 23** (AMCS). *Let $g() := \texttt{AGG}(r) \leftarrow q(\vec{u})$ be a query in* AGGR[sjfBCQ]. *Let* **db** *be a database instance. An* aggregation-maximal consistent set (AMCS) *(with respect to $g()$ and* **db***) is a set M of valuations over $\mathsf{vars}(\{R_1, \ldots, R_n\})$ such that*

(A) Homomorphic: *for every* $\theta \in M$, $(\mathbf{db}, \theta) \models \{R_1, \ldots, R_n\}$;

(B) Consistent: $M \models \mathcal{K}(\{R_1, \ldots, R_n\})$; *and*

(C) $\mathcal{F}_{\texttt{AGG}}$-Maximal: *among all sets of valuations over* $\mathsf{vars}(\{R_1, \ldots, R_n\})$ *satisfying the conditions in* (A) *and* (B), M *is one that maximizes* $\mathcal{F}_{\texttt{AGG}}$.

The following lemma has an easy proof.

▶ **Lemma 24.** *Let* $g() := \texttt{AGG}(r) \leftarrow q(\vec{u})$ *be a query in* AGGR[sjfBCQ] *such that* $\mathcal{F}_{\texttt{AGG}}$ *is monotone. Let* $\mathbf{db}$ *be a database instance. If* M *is an AMCS, then* $\mathcal{F}_{\texttt{AGG}}(M) = [\![\mathsf{LUB\text{-}CQA}(g())]\!]^{\mathbf{db}}$.

Using the refined notion of AMCS defined next, we will show that the AGGR[nr-datalog] program of Fig. 4 recursively computes the $\mathcal{F}_{\texttt{AGG}}$ value of an AMCS.

▶ **Definition 25** ((ℓ, λ)-AMCS). *Let* $g() := \texttt{AGG}(r) \leftarrow q(\vec{u})$ *be a query in* AGGR[sjfBCQ] *such that* $\exists\vec{u}(q(\vec{u}))$ *has an acyclic attack graph, with topological sort* $(R_1, R_2, \ldots, R_n)$. *Let* $\mathbf{db}$ *be a database instance. Let* $\ell \in \{1, \ldots, n\}$, *and let* λ *be a valuation over* $\mathsf{vars}(\langle\!\langle \vec{u}_{\ell-1} \rangle\!\rangle)$. *An* (ℓ, λ)-AMCS *is a set* M *of valuations over* $\mathsf{vars}(\{R_\ell, \ldots, R_n\}) \cup \{r\}$ *such that*

(A) Extending: *for every* $\theta \in M$, $\theta(\langle\!\langle \vec{u}_{\ell-1} \rangle\!\rangle) = \lambda(\langle\!\langle \vec{u}_{\ell-1} \rangle\!\rangle)$;

(B) Homomorphic: *for every* $\theta \in M$, $(\mathbf{db}, \theta) \models \{R_\ell, \ldots, R_n\}$;

(C) Consistent: $M \models \mathcal{K}(\{R_\ell, \ldots, R_n\})$; *and*

(D) $\mathcal{F}_{\texttt{AGG}}$-Maximal: *among all sets of valuations over* $\mathsf{vars}(\{R_\ell, \ldots, R_n\}) \cup \{r\}$ *satisfying the conditions in* (A), (B) *and* (C), M *is one that maximizes* $\mathcal{F}_{\texttt{AGG}}$.

Let Q *be the set of all numbers in the* r*-column of* S *used in rule* (5) *of Fig. 4. If* λ *is a valuation over* $\mathsf{vars}(\langle\!\langle \vec{u}_n \rangle\!\rangle) = \{r\}$ *such that* $\lambda(r) \in Q$, *then* $\{\lambda\}$ *is an* $(n+1, \lambda)$-AMCS.

Clearly, a $(1, \emptyset)$-AMCS is an AMCS. Appendix A provides a proof sketch of the following lemma.

▶ **Lemma 26.** *Let* $g() := \texttt{AGG}(r) \leftarrow q(\vec{u})$ *be a query in* AGGR[sjfBCQ] *whose aggregate operator is monotone and associative. Suppose that* $\exists\vec{u}(q(\vec{u}))$ *is irreducible and has an acyclic attack graph, with topological sort* $(R_1, R_2, \ldots, R_n)$. *Let* $\mathbf{db}$ *be a database instance, and* $\ell \in \{0, 1, 2, \ldots, n\}$. *Then, the following are equivalent:*

(A) *The program of Fig. 4 computes* $E_\ell(\vec{c}, o)$.

(B) *There is an* $(\ell+1, \lambda)$*-AMCS* M *with* $\lambda(\langle\!\langle \vec{u}_\ell \rangle\!\rangle) = \vec{c}$ *such that* $\mathcal{F}_{\texttt{AGG}}(M) = o$.

We can now provide the proofs of Lemma 20 and Theorem 4.

Proof of Lemma 20. By letting $\ell = 0$ in Lemma 26, we obtain that $E_0(() , o)$ is computed if and only if there is a $(1, \emptyset)$-AMCS M such that $\mathcal{F}_{\texttt{AGG}}(M) = o$. Since a $(1, \emptyset)$-AMCS is an AMCS, the desired result follows by Lemma 24. ◀

Proof of Theorem 4. Assume that $\exists\vec{u}(q(\vec{u}))$ is κ-acyclic. By Lemma 15, it is possible to construct, in polynomial time in the size of $q(\vec{u})$, a kernel $\exists\vec{u}(q'(\vec{u}))$ of $\exists\vec{u}(q(\vec{u}))$. Let $g'() := \texttt{AGG}(r) \leftarrow q'(\vec{u})$. By Lemma 10, $\mathsf{LUB\text{-}CQA}(g()) \leq_{\mathsf{nr\text{-}datalog}} \mathsf{LUB\text{-}CQA}(g'())$. By Definition 12, $\exists\vec{u}(q'(\vec{u}))$ is irreducible. Since $\exists\vec{u}(q(\vec{u}))$ is κ-acyclic, the attack graph of $\exists\vec{u}(q'(\vec{u}))$ is acyclic. By Lemma 20, $\mathsf{LUB\text{-}CQA}(g'())$ is expressible in AGGR[nr-datalog]. Consequently, $\mathsf{LUB\text{-}CQA}(g())$ can be solved in AGGR[nr-datalog] by first reducing it to $\mathsf{LUB\text{-}CQA}(g'())$ using the nr-datalog reduction, and then solving $\mathsf{LUB\text{-}CQA}(g'())$. ◀

7 Inexpressibility for Non-κ-Acyclic Queries

The right-to-left implication in Theorem 3 follows from Theorem 4. We now show the left-to-right implication, whose contraposition reads as follows: for each query $g() := \texttt{SUM}(r) \leftarrow q(\vec{u})$ in AGGR[sjfBCQ] over $\mathbb{Q}_{\geq 0}$, with $r \neq 0$, if $\exists\vec{u}(q(\vec{u}))$ is not κ-acyclic, then LUB-CQA$(g())$ is not expressible in AGGR[FOL]. The proof relies on a reduction from the 2DM problem (a.k.a. *Bipartite Perfect Matching*), which we recall next.

2-DIMENSIONAL MATCHING (2DM).

INSTANCE: A set $M \subseteq A \times B$, where A and B are disjoint sets having the same number n of elements.

QUESTION: Does M contain a matching, that is, a subset $M' \subseteq M$ such that $|M'| = n$ and no two elements of M' agree in any coordinate?

▶ **Lemma 27.** *Let r be a numerical variable or a positive number. Let $g() := \texttt{SUM}(r) \leftarrow q(\vec{u})$ be a query in* AGGR[sjfBCQ] *over $\mathbb{Q}_{\geq 0}$. If the attack graph of $\exists\vec{u}(q(\vec{u}))$ has a cycle, then* 2DM $\leq_{\mathsf{FO}}$ LUB-CQA$(g())$.

A key crux in the following proof of Theorem 3 is a deep result by Hella et al. [22], which implies that every query in AGGR[FOL] is Hanf-local.

Proof of Theorem 3. The right-to-left implication follows from Theorem 4. We show the left-to-right implication by contraposition. Assume that $\exists\vec{u}(q(\vec{u}))$ is not κ-acyclic. Let $\exists\vec{u}(q'(\vec{u}))$ be a kernel of $\exists\vec{u}(q(\vec{u}))$, and let $g'() := \texttt{SUM}(r) \leftarrow q'(\vec{u})$. Since $\exists\vec{u}(q(\vec{u}))$ is not κ-acyclic, the attack graph of $\exists\vec{u}(q'(\vec{u}))$ contains a cycle. By Lemma 27, 2DM $\leq_{\mathsf{FO}}$ LUB-CQA$(g'())$. By Lemma 10, LUB-CQA$(g'()) \leq_{\mathsf{FO}}$ LUB-CQA$(g())$. Since first-order reductions compose, 2DM $\leq_{\mathsf{FO}}$ LUB-CQA$(g())$. In [33, Corollary 8.26 and Exercise 8.16], it is established that every query in a logic called $\mathcal{L}_{\text{aggr}}$ is Hanf-local. It is easily verified that every query in AGGR[FOL] is expressible in $\mathcal{L}_{\text{aggr}}$, and therefore cannot express 2DM. It follows that LUB-CQA$(g())$ cannot be expressed in AGGR[FOL]. ◀

8 Special Cases and Free Variables

Consider rule (7) in the AGGR[nr-datalog] program of Fig. 4. If $\vec{x}_\ell = ()$, then this rule aggregates over a singleton containing the result m_ℓ of a prior aggregation, say $m_\ell = \mathcal{F}_{\texttt{AGG}}(X)$. Since $\mathcal{F}_{\texttt{AGG}}(\{\{\mathcal{F}_{\texttt{AGG}}(X)\}\}) = \mathcal{F}_{\texttt{AGG}}(X)$ by associativity, the rule is redundant if $\vec{x}_\ell = ()$. A special case arises when $\vec{x}_\ell$ is empty for each ℓ, in which case no intermediate aggregation is required. In [23], the term *parsimonious aggregation* was introduced to refer to aggregation queries that admit glb and lub rewritings in AGGR[FOL] where aggregation is applied only at the end, but not intermediately. The same work defined the query class Cparsimony, which captures exactly all COUNT-queries (i.e., SUM-queries with head SUM(1)) that support parsimonious counting. It was also shown that Cparsimony strictly includes Cforest, for which Fuxman [18] demonstrated the possibility of parsimonious counting. Since parsimonious aggregation is a special case of aggregation, our results imply that every query in Cparsimony must be κ-acyclic. A purely syntactic proof of the following result can be found in [2, p. 81].

▶ **Proposition 28.** *Every query in* Cparsimony *is κ-acyclic.*

So far, we have focused on numerical terms $g()$ without free variables. We now explain how our results extend to numerical terms $g(\vec{x}) = \mathsf{Aggr}_{\mathcal{F}_{\mathtt{SUM}}}\vec{y}\,[r, q(\vec{x},\vec{y})]$ with free variables $\vec{x} := (x_1,\ldots,x_k)$, where $q(\vec{x},\vec{y})$ is a self-join-free conjunction of atoms. Let $\vec{c} = (c_1,\ldots,c_k)$ be a sequence of distinct constants. Let $q_{\vec{c}}(\vec{y})$ be the conjunction obtained from $q(\vec{x},\vec{y})$ by replacing, for $i \in \{1,\ldots,k\}$, each occurrence of each x_i by c_i. Our results imply the following:

- if $\exists\vec{y}(q_{\vec{c}}(\vec{y}))$ is not κ-acyclic, then $\mathsf{LUB\text{-}CQA}(g(\vec{c}))$ is not expressible in $\mathsf{AGGR[FOL]}$; and
- if $\exists\vec{y}(q_{\vec{c}}(\vec{y}))$ is κ-acyclic, then there is a program $\varphi_{\vec{c}}$ in $\mathsf{AGGR[nr\text{-}datalog}]$ that solves $\mathsf{LUB\text{-}CQA}(g(\vec{c}))$. This expressibility holds not only for $\mathcal{F}_{\mathtt{SUM}}$, but for every aggregate operator that is monotone and associative.

Since $q(\vec{x},\vec{y})$ is self-join-free, different sequences $\vec{c}$ of constants yield the same program $\varphi_{\vec{c}}$ up to a renaming of the constants. Thus, we can compute the program once by treating the free variables in $\vec{x}$ as distinct constants. This approach is commonly used in CQA and logic in general [33, Lemma 2.3], but it breaks down in the presence of self-joins.

9 Conclusion and Future Research

The main finding of this paper is the new notion of κ-acyclicity, which allowed us (Theorem 3) to exactly characterize SUM-queries that enable the computation of consistent (with respect to primary keys) least upper bounds in aggregate logic. One direction of this characterization extends to all aggregate operators AGG that are monotone and associative (Theorem 4): if $\exists\vec{u}(q(\vec{u}))$ is κ-acyclic, then consistent least upper bounds for $\mathtt{AGG}(r) \leftarrow q(\vec{u})$ can be computed in aggregate logic. However, the inverse does not generally hold as some aggregate operators, such as MAX, allow a straightforward computation of consistent least upper bounds.

A challenge for further research is to pinpoint the complexity of $\mathsf{LUB\text{-}CQA}(g())$ for SUM-queries $g()$ when $\mathsf{LUB\text{-}CQA}(g())$ is not in $\mathsf{AGGR[FOL]}$. In this paper, we have only established 2DM-hardness for these problems, but we have not determined the complexity upper bounds. Note that for $g_0() := \mathtt{SUM}(1) \leftarrow R(\underline{x},y), S(\underline{y},x)$, $\mathsf{GLB\text{-}CQA}(g_0)$ is equivalent to 2DM under first-order reductions (one direction of this equivalence follows from Lemma 27, and the proof of the other direction is straightforward). This challenge is non-trivial: while 2DM is known to be NL-hard [9], its complexity upper bound is still under investigation [13, 14].

References

1 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

2 Aziz Amezian El Khalfioui. *Computing Range Consistent Answers to Conjunctive Queries with Aggregation*. PhD thesis, UMONS - University of Mons [Faculté des Sciences], Mons, Belgium, 18 September 2025. URL: https://hdl.handle.net/20.500.12907/53232.

3 Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS*, pages 68–79. ACM Press, 1999. doi:10.1145/303976.303983.

4 Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Scalar aggregation in FD-inconsistent databases. In *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2001. doi:10.1007/3-540-44503-X_3.

5 Marcelo Arenas, Leopoldo E. Bertossi, Jan Chomicki, Xin He, Vijay Raghavan, and Jeremy P. Spinrad. Scalar aggregation in inconsistent databases. *Theor. Comput. Sci.*, 296(3):405–434, 2003. doi:10.1016/S0304-3975(02)00737-5.

6 Leopoldo E. Bertossi. *Database Repairing and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011. doi:10.2200/S00379ED1V01Y201108DTM020.

7 Leopoldo E. Bertossi. Database repairs and consistent query answering: Origins and further developments. In *PODS*, pages 48–58. ACM, 2019. doi:10.1145/3294052.3322190.

8 Andrei A. Bulatov. Complexity of conservative constraint satisfaction problems. *ACM Trans. Comput. Log.*, 12(4):24:1–24:66, 2011. doi:10.1145/1970398.1970400.

9 Ashok K. Chandra, Larry J. Stockmeyer, and Uzi Vishkin. Constant depth reducibility. *SIAM J. Comput.*, 13(2):423–439, 1984. doi:10.1137/0213028.

10 Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Rewriting queries with arbitrary aggregation functions using views. *ACM Trans. Database Syst.*, 31(2):672–715, 2006. doi:10.1145/1138394.1138400.

11 Sara Cohen, Werner Nutt, and Alexander Serebrenik. Algorithms for rewriting aggregate queries using views. In *DMDW*, volume 19 of *CEUR Workshop Proceedings*, page 9. CEUR-WS.org, 1999. URL: https://ceur-ws.org/Vol-19/paper9.pdf.

12 Akhil A. Dixit and Phokion G. Kolaitis. Consistent answers of aggregation queries via SAT. In *ICDE*, pages 924–937. IEEE, 2022. doi:10.1109/ICDE53745.2022.00074.

13 Stephen A. Fenner, Rohit Gurjar, and Thomas Thierauf. A deterministic parallel algorithm for bipartite perfect matching. *Commun. ACM*, 62(3):109–115, 2019. doi:10.1145/3306208.

14 Stephen A. Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-NC. *SIAM J. Comput.*, 50(3), 2021. doi:10.1137/16M1097870.

15 Diego Figueira, Anantha Padmanabha, Luc Segoufin, and Cristina Sirangelo. A simple algorithm for consistent query answering under primary keys. In *ICDT*, volume 255 of *LIPIcs*, pages 24:1–24:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ICDT.2023.24.

16 Diego Figueira, Anantha Padmanabha, Luc Segoufin, and Cristina Sirangelo. A simple algorithm for consistent query answering under primary keys. *Logical Methods in Computer Science*, Volume 21, Issue 1, February 2025. doi:10.46298/lmcs-21(1:18)2025.

17 Gaëlle Fontaine. Why is it hard to obtain a dichotomy for consistent query answering? *ACM Trans. Comput. Log.*, 16(1):7:1–7:24, 2015. doi:10.1145/2699912.

18 Ariel Fuxman. *Efficient query processing over inconsistent databases*. PhD thesis, University of Toronto, 2007.

19 Ariel Fuxman, Elham Fazli, and Renée J. Miller. ConQuer: Efficient management of inconsistent databases. In *SIGMOD Conference*, pages 155–166. ACM, 2005. doi:10.1145/1066157.1066176.

20 Ariel Fuxman and Renée J. Miller. First-order query rewriting for inconsistent databases. In *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2005. doi:10.1007/978-3-540-30570-5_23.

21 Miika Hannula and Jef Wijsen. A dichotomy in consistent query answering for primary keys and unary foreign keys. In *PODS*, pages 437–449. ACM, 2022. doi:10.1145/3517804.3524157.

22 Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, 2001. doi:10.1145/502090.502100.

23 Aziz Amezian El Khalfioui and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with counting. In *ICDT*, volume 255 of *LIPIcs*, pages 23:1–23:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.ICDT.2023.23.

24 Aziz Amezian El Khalfioui and Jef Wijsen. Computing range consistent answers to aggregation queries via rewriting. *Proc. ACM Manag. Data*, 2(5):218:1–218:19, 2024. doi:10.1145/3695836.

25 Benny Kimelfeld and Phokion G. Kolaitis. A unifying framework for incompleteness, inconsistency, and uncertainty in databases. *Commun. ACM*, 67(3):74–83, 2024. doi:10.1145/3624717.

26 Phokion G. Kolaitis, Nina Pardal, Jonni Virtema, and Jef Wijsen. Rewriting consistent answers on annotated data. *Proc. ACM Manag. Data*, 3(2):110:1–110:26, 2025. doi:10.1145/3725247.

27 Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on path queries. In *PODS*, pages 215–232. ACM, 2021. doi:10.1145/3452021.3458334.

28 Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Consistent query answering for primary keys on rooted tree queries. *Proc. ACM Manag. Data*, 2(2):76, 2024. doi:10.1145/3651139.

29 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, 2017. doi:10.1145/3068334.

30 Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys and conjunctive queries with negated atoms. In *PODS*, pages 209–224. ACM, 2018. doi:10.1145/3196959.3196982.

31 Paraschos Koutris and Jef Wijsen. First-order rewritability in consistent query answering with respect to multiple keys. In *PODS*, pages 113–129. ACM, 2020. doi:10.1145/3375395.3387654.

32 Paraschos Koutris and Jef Wijsen. Consistent query answering for primary keys in datalog. *Theory Comput. Syst.*, 65(1):122–178, 2021. doi:10.1007/S00224-020-09985-6.

33 Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07003-1.

34 David Maier. Minimum covers in relational database model. *J. ACM*, 27(4):664–674, 1980. doi:10.1145/322217.322223.

35 Anantha Padmanabha, Luc Segoufin, and Cristina Sirangelo. A dichotomy in the complexity of consistent query answering for two atom queries with self-join. *Proc. ACM Manag. Data*, 2(2):74, 2024. doi:10.1145/3651137.

36 Jef Wijsen. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In *PODS*, pages 179–190. ACM, 2010. doi:10.1145/1807085.1807111.

37 Jef Wijsen. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.*, 37(2):9:1–9:35, 2012. doi:10.1145/2188349.2188351.

38 Jef Wijsen. Foundations of query answering on inconsistent databases. *SIGMOD Rec.*, 48(3):6–16, 2019. doi:10.1145/3377391.3377393.

A Proof Sketch of Lemma 26

▶ **Definition 29** (Pre-attacks). *Let q be a query in* sjfBCQ*, and $\leq$ be a linear order on the atoms of q. We write $F < G$ if $F \leq G$ and $F \neq G$. For every atom G in q, we define $G^{+,(q,<)}$ as the closure of $\mathsf{Key}(G)$ with respect to $\{\mathsf{Key}(F) \to \mathsf{vars}(F) \mid F < G\}$.*

Let G be an atom of q, and $u \in \mathsf{vars}(q)$. We say that G pre-attacks *u (with respect to $(q, \leq)$), denoted $G \overset{(q,\leq)}{\rightsquigarrow} u$, if $\mathcal{G}aifman(q)$ contains a path $(v_0, v_1, \ldots, v_n)$ with $n \geq 0$, $v_0 \in \mathsf{notKey}(G)$, and $v_n = u$ such that no v_i belongs to $G^{+,(q,<)}$.*

▶ **Lemma 30** (Lemma D.3.1 in [2]). *Let q be an irreducible query in* sjfBCQ *with an acyclic attack graph. Let $\leq$ be a topological sort of q's attack graph. For every atom G in q, for every variable $u \in \bigcup_{F<G} \mathsf{vars}(F)$, we have $G \overset{(q,\leq)}{\not\rightsquigarrow} u$.*

▶ **Definition 31.** *Let $\vec{x}$ and $\vec{a}$ be sequences of variables and constants, respectively, of the same length. If θ is a valuation such that $\mathsf{vars}(\vec{x}) \subseteq \mathsf{dom}(\theta)$ and $\theta(\vec{x}) = \vec{a}$, then we say that θ* satisfies *$\vec{x} = \vec{a}$, also denoted $\theta \models \vec{x} = \vec{a}$. For a set M of valuations, $M \models \vec{x} = \vec{a}$ denotes that each valuation in M satisfies $\vec{x} = \vec{a}$. We write $\vec{x} \mapsto \vec{a}$ for the valuation over $\mathsf{vars}(\vec{x})$ that satisfies $\vec{x} = \vec{a}$.*

Proof sketch of Lemma 26. The implication B $\implies$ A is the easier direction. We next outline the proof of A $\implies$ B, which proceeds by induction on decreasing $\ell = n, n-1, \ldots, 0$. For the induction basis, $\ell = n$, assume $E_n(o, \texttt{AGG}(o))$ is computed. If λ is a valuation over $\{r\}$

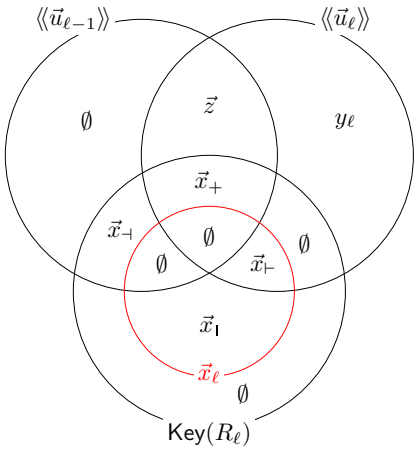

Figure 5 Notations.

satisfying $r = o$, then $\{\lambda\}$ is an $(n+1, \lambda)$-AMCS. Clearly, $\mathcal{F}_{\texttt{AGG}}(\{\lambda\}) = \mathcal{F}_{\texttt{AGG}}(\{\lambda(r)\}) = \texttt{AGG}(o)$. For the induction step from ℓ to $\ell - 1$, we assume $\mathsf{notKey}(R_\ell) = \{y_\ell\} \subseteq \mathsf{vars}(\langle\langle \vec{u}_\ell \rangle\rangle)$; the other case, where $\mathsf{notKey}(R_\ell) \not\subseteq \mathsf{vars}(\langle\langle \vec{u}_\ell \rangle\rangle)$, is simpler.

We use the notations shown in the Venn diagram of Fig. 5, so that the R_ℓ-atom has the form $R_\ell(\underline{\vec{x}_\dashv, \vec{x}_+, \vec{x}_\mathsf{I}, \vec{x}_\vdash}, y_\ell)$. Since $\vec{z} \cdot \vec{x}_+$ is shared by $\langle\langle \vec{u}_\ell \rangle\rangle$ and $\langle\langle \vec{u}_{\ell-1} \rangle\rangle$, we can assume from here on that all valuations satisfy $\vec{z} = \vec{d}$ and $\vec{x}_+ = \vec{a}_+$, for fixed sequences $\vec{d}$ and $\vec{a}_+$.

The remainder of the proof consists of two parts. First, we show how an (ℓ, λ)-AMCS can be constructed from $(\ell + 1, \lambda_i)$-AMCSs. Second, we argue that rules (6) and (7) of the AGGR[nr-datalog] program in Fig. 4 capture this construction.

Construction of an (ℓ, λ)-AMCS with $\lambda \models \vec{z} \cdot \vec{x}_\dashv \cdot \vec{x}_+ = \vec{d} \cdot \vec{a}_\dashv \cdot \vec{a}_+$, for fixed $\vec{a}_\dashv$. Consider an R_ℓ-block $\mathbf{b}$ of $\mathbf{db}$, $\mathbf{b} = \left\{R_\ell(\underline{\vec{a}_\dashv, \vec{a}_+, \vec{a}_\mathsf{I}, \vec{a}_\vdash}, b_i)\right\}_{i=1}^{|\mathbf{b}|}$. For $i \in \{1, \ldots, |\mathbf{b}|\}$, let λ_i be the valuation over $\langle\langle \vec{u}_\ell \rangle\rangle$ that satisfies $\vec{z} \cdot \vec{x}_+ \cdot \vec{x}_\vdash \cdot y_\ell = \vec{d} \cdot \vec{a}_+ \cdot \vec{a}_\vdash \cdot b_i$, and let M_i be an $(\ell + 1, \lambda_i)$-AMCS; possibly $M_i = \emptyset$.

Let $i^* \in \{1, \ldots, |\mathbf{b}|\}$ be such that for all $j \in \{1, \ldots, |\mathbf{b}|\}$, $\mathcal{F}_{\texttt{AGG}}(M_{i^*}) \geq \mathcal{F}_{\texttt{AGG}}(M_j)$; if this maximizing index is not unique, break the tie by picking the index i^* whose corresponding b_{i^*} comes first according to a fixed linear order on $\mathbf{dom}$. Significantly, the fact chosen from the block $\mathbf{b}$ may depend on λ, as illustrated next.

▶ **Example 32.** Consider the query $\texttt{SUM}(r) \leftarrow R_1(\underline{c}, r), R_2(\underline{c}, y), R_3(\underline{y, r})$ where c is a constant. Consider the database instance $\mathbf{db} = \{R_1(\underline{c}, 5), R_1(\underline{c}, 7), R_2(\underline{c}, a), R_2(\underline{c}, b), R_3(\underline{a}, 5), R_3(\underline{b}, 7)\}$. We have $\langle\langle \vec{u}_1 \rangle\rangle = r$. Then, $\{yr \mapsto a5\}$ is a $(2, r \mapsto 5)$-AMCS which uses $R_2(\underline{c}, a)$, and $\{yr \mapsto b7\}$ is a $(2, r \mapsto 7)$-AMCS which uses $R_2(\underline{c}, b)$. Finally, $\{yr \mapsto b7\}$ is a $(1, \emptyset)$-AMCS. ⌟

Let $M^{\vec{d} \cdot \vec{a}_\dashv \cdot \vec{a}_+}_{\vec{a}_\mathsf{I} \cdot \vec{a}_\vdash}$ denote the set of all valuations over $\mathsf{vars}(\{R_\ell, \ldots, R_n\}) \cup \{r\}$ that extend a valuation from M_{i^*} and satisfy $\vec{x}_\dashv \cdot \vec{x}_\mathsf{I} = \vec{a}_\dashv \cdot \vec{a}_\mathsf{I}$, hence $M^{\vec{d} \cdot \vec{a}_\dashv \cdot \vec{a}_+}_{\vec{a}_\mathsf{I} \cdot \vec{a}_\vdash} \models \vec{z} \cdot \vec{x}_\dashv \cdot \vec{x}_+ \cdot \vec{x}_\mathsf{I} \cdot \vec{x}_\vdash = \vec{d} \cdot \vec{a}_\dashv \cdot \vec{a}_+ \cdot \vec{a}_\mathsf{I} \cdot \vec{a}_\vdash$. Let $(\vec{a}_{\mathsf{I}1}, \vec{a}_{\vdash 1}), \ldots, (\vec{a}_{\mathsf{I}k}, \vec{a}_{\vdash k})$ enumerate all pairs $(\vec{a}_\mathsf{I}, \vec{a}_\vdash)$ from $\mathbf{dom}^{|\vec{x}_\mathsf{I}|} \times \mathbf{dom}^{|\vec{x}_\vdash|}$ such that $R_\ell(\underline{\vec{a}_\dashv, \vec{a}_+, \vec{a}_\mathsf{I}, \vec{a}_\vdash}, *)$ is an R_ℓ-block of $\mathbf{db}$. The crux is now to show that

$$\biguplus_{j=1}^{k} M^{\vec{d} \cdot \vec{a}_\dashv \cdot \vec{a}_+}_{\vec{a}_{\mathsf{I}j} \cdot \vec{a}_{\vdash j}} \models \mathcal{K}(\{R_\ell, \ldots, R_n\}). \tag{8}$$

By the induction hypothesis and our construction, for every $j \in \{1,\ldots,k\}$, $M^{\vec{d}\cdot\vec{a}_{\dashv}\cdot\vec{a}_{+}}_{\vec{a}_{\shortmid j}\cdot\vec{a}_{\vdash j}} \models \mathcal{K}(\{R_\ell,\ldots,R_n\})$. We next consider valuations θ_1,θ_2 drawn from two distinct components of the disjoint union in (8) (hence $k \geq 2$). We show by induction on increasing $h = \ell, \ell+1,\ldots,n$ that

$$\{\theta_1,\theta_2\} \models \mathsf{Key}(R_h) \rightarrow \mathsf{notKey}(R_h). \tag{9}$$

If $h = \ell$, the desired result (9) holds because θ_1 and θ_2 disagree on some variable of $\mathsf{Key}(R_\ell)$. For $h > \ell$, the induction hypothesis is that $\{\theta_1,\theta_2\} \models \mathcal{K}(\{R_\ell,\ldots,R_{h-1}\})$. If $\mathsf{notKey}(R_h) = \emptyset$, then (9) holds vacuously. So assume $\mathsf{notKey}(R_h) = \{y_h\}$ from here on. Assume that θ_1 and θ_2 agree on every variable of $\mathsf{Key}(R_h)$. It suffices to show $\theta_1(y_h) = \theta_2(y_h)$.

We can extend θ_1 and θ_2 to (arbitrary) valuations θ_1^+ and θ_2^+ over $\mathsf{vars}(q)$ such that $\{\theta_1^+,\theta_2^+\} \models \mathcal{K}(\{R_1,\ldots,R_{\ell-1}\})$. In particular, let $\theta_1^+(v) = \theta_2^+(v)$ for every $v \in \mathsf{vars}(\vec{u}_{\ell-1}) \setminus \mathsf{vars}(\langle\!\langle \vec{u}_{\ell-1} \rangle\!\rangle)$. We remark that $\mathsf{vars}(\vec{u}_{\ell-1})$ is disjoint from $\mathsf{vars}(\vec{x}_{\shortmid}\cdot\vec{x}_{\vdash})$, the latter containing variables on which θ_1 and θ_2 may disagree. Consider any path $(v_0,v_1,\ldots,v_m)$ in $\mathcal{G}aifman(q)$ between y_h and some variable from $\vec{x}_{\shortmid}\cdot\vec{x}_{\vdash}$, that is, $v_0 = y_h$ and $v_m \in \mathsf{vars}(\vec{x}_{\shortmid}\cdot\vec{x}_{\vdash})$. Since $R_h \overset{q}{\rightsquigarrow} y_h$ by Lemma 14, $R_h \overset{(q,\leq)}{\nmapsto} y_h$. Since $R_h \overset{(q,\leq)}{\not\mapsto} v_m$ by Lemma 30, there is a smallest index $i \in \{1,\ldots,m\}$ such that $R_h \overset{(q,\leq)}{\not\mapsto} v_i$, hence $\mathcal{K}(\{R_1,\ldots,R_{h-1}\}) \models \mathsf{Key}(R_h) \rightarrow v_i$, and thus $\theta_1^+(v_i) = \theta_2^+(v_i)$. Consequently, if the R_h-fact of $q(\vec{u})$ is $R_h(\underline{\vec{s}}, y_h)$, the repair of the block $R_h(\underline{\theta_1(\vec{s})}, *)$, which equals $R_h(\underline{\theta_2(\vec{s})}, *)$, is independent of values assigned to $\vec{x}_{\shortmid}\cdot\vec{x}_{\vdash}$, given $\vec{z}\cdot\vec{x}_{\dashv}\cdot\vec{x}_{+} = \vec{d}\cdot\vec{a}_{\dashv}\cdot\vec{a}_{+}$. Hence, we can conclude that $\theta_1(y_h) = \theta_2(y_h)$, as desired, assuming ties are broken according to a fixed linear order on $\mathbf{dom}$. It then follows that $\biguplus_{j=1}^{k} M^{\vec{d}\cdot\vec{a}_{\dashv}\cdot\vec{a}_{+}}_{\vec{a}_{\shortmid j}\cdot\vec{a}_{\vdash j}}$ is an (ℓ,λ)-AMCS with $\lambda \models \vec{z}\cdot x_{\dashv}\cdot\vec{x}_{+} = \vec{d}\cdot\vec{a}_{\dashv}\cdot\vec{a}_{+}$.

Computation of $\mathcal{F}_{\mathtt{AGG}}(M)$ for an (ℓ,λ)-AMCS M with $\lambda \models \vec{z}\cdot\vec{x}_{\dashv}\cdot\vec{x}_{+} = \vec{d}\cdot\vec{a}_{\dashv}\cdot\vec{a}_{+}$. Rules (6) and (7) of Fig. 4 are executed in moving from ℓ to $\ell-1$, which read as follows with the notation in the Venn diagram of Fig. 5.

$$K_\ell(\vec{z},\vec{x}_{\dashv},\vec{x}_{+},\vec{x}_{\shortmid},\vec{x}_{\vdash},\mathtt{MAX}(r_\ell)) \leftarrow R_\ell(\underline{\vec{x}_{\dashv},\vec{x}_{+},\vec{x}_{\shortmid},\vec{x}_{\vdash}},y_\ell), E_\ell(\vec{z},\vec{x}_{+},\vec{x}_{\vdash},y_\ell,r_\ell)$$
$$E_{\ell-1}(\vec{z},\vec{x}_{\dashv},\vec{x}_{+},\mathtt{AGG}(m_\ell)) \leftarrow K_\ell(\vec{z},\vec{x}_{\dashv},\vec{x}_{+},\vec{x}_{\shortmid},\vec{x}_{\vdash},m_\ell)$$

Assume that $E_{\ell-1}(\vec{d},\vec{a}_{\dashv},\vec{a}_{+},p)$ is computed by the program in Fig. 4. We need to show:

there is an (ℓ,λ)-AMCS M with $\lambda \models \vec{z}\cdot\vec{x}_{+}\cdot\vec{x}_{\dashv} = \vec{d}\cdot\vec{a}_{+}\cdot\vec{a}_{\dashv}$ such that $\mathcal{F}_{\mathtt{AGG}}(M) = p$.

The fact $E_{\ell-1}(\vec{d},\vec{a}_{\dashv},\vec{a}_{+},p)$ must be derived by the following rules:

$$K_\ell(\vec{d},\vec{a}_{\dashv},\vec{a}_{+},\vec{x}_{\shortmid},\vec{x}_{\vdash},\mathtt{MAX}(r_\ell)) \leftarrow R_\ell(\underline{\vec{a}_{\dashv},\vec{a}_{+},\vec{x}_{\shortmid},\vec{x}_{\vdash}},y_\ell), E_\ell(\vec{d},\vec{a}_{+},\vec{x}_{\vdash},y_\ell,r_\ell) \tag{10}$$
$$E_{\ell-1}(\vec{d},\vec{a}_{\dashv},\vec{a}_{+},\mathtt{AGG}(m_\ell)) \leftarrow K_\ell(\vec{d},\vec{a}_{\dashv},\vec{a}_{+},\vec{x}_{\shortmid},\vec{x}_{\vdash},m_\ell) \tag{11}$$

Let $(\vec{a}_{\shortmid 1},\vec{a}_{\vdash 1},p_1),\ldots,(\vec{a}_{\shortmid g},\vec{a}_{\vdash g},p_g)$ enumerate all triples in $\mathbf{dom}^{|\vec{x}_{\shortmid}|} \times \mathbf{dom}^{|\vec{x}_{\vdash}|} \times \mathbb{Q}_{\geq 0}$ such that, for each $j \in \{1,\ldots,g\}$, $K_\ell(\vec{d},\vec{a}_{\dashv},\vec{a}_{+},\vec{a}_{\shortmid j},\vec{a}_{\vdash j},p_j)$ is a computed K_ℓ-fact, hence $p = \mathcal{F}_{\mathtt{AGG}}(\{\{p_1,\ldots,p_g\}\})$. Each such K_ℓ-fact must be derived by the following partial grounding of rule (10):

$$K_\ell(\vec{d},\vec{a}_{\dashv},\vec{a}_{+},\vec{a}_{\shortmid j},\vec{a}_{\vdash j},\mathtt{MAX}(r_\ell)) \leftarrow R_\ell(\underline{\vec{a}_{\dashv},\vec{a}_{+},\vec{a}_{\shortmid j},\vec{a}_{\vdash j}},y_\ell), E_\ell(\vec{d},\vec{a}_{+},\vec{a}_{\vdash j},y_\ell,r_\ell).$$

For every $j \in \{1, \ldots, g\}$, there exists $b_j^* \in \mathbf{dom}$ such that $R_\ell(\underline{\vec{a}_{\dashv}, \vec{a}_{+}, \vec{a}_{\downarrow j}, \vec{a}_{\vdash j}}, b_j^*)$ is an R_ℓ-fact in $\mathbf{db}$, and $E_\ell(\vec{d}, \vec{a}_{+}, \vec{a}_{\vdash j}, b_j^*, p_j)$ is a computed E_ℓ-fact. If there is more than one such b_j^*, the smallest is chosen according to a fixed linear order on $\mathbf{dom}$. By the induction hypothesis, there is an $(\ell{+}1, \lambda_j)$-AMCS, denoted $M^{\vec{d}\cdot\vec{a}_{+}}_{\vec{a}_{\vdash j}}$, with $\lambda_j \models \vec{z}\cdot\vec{x}_{+}\cdot\vec{x}_{\vdash}\cdot y_\ell = \vec{d}\cdot\vec{a}_{+}\cdot\vec{a}_{\vdash j}\cdot b_j^*$ such that $\mathcal{F}_{\mathtt{AGG}}\left(M^{\vec{d}\cdot\vec{a}_{+}}_{\vec{a}_{\vdash j}}\right) = p_j$. For every $j \in \{1, \ldots, g\}$, let μ_j be a valuation over $\mathsf{vars}(\vec{x}_{\dashv}\cdot\vec{x}_{\downarrow})$ satisfying $\vec{x}_{\dashv}\cdot\vec{x}_{\downarrow} = \vec{a}_{\dashv}\cdot\vec{a}_{\downarrow j}$, and let $M^{\vec{d}\cdot\vec{a}_{\dashv}\cdot\vec{a}_{+}}_{\vec{a}_{\downarrow j}\cdot\vec{a}_{\vdash j}} = \left\{\mu_j \cup \theta \mid \theta \in M^{\vec{d}\cdot\vec{a}_{+}}_{\vec{a}_{\vdash j}}\right\}$. As argued in the first part of the induction step, $M := \biguplus_{j=1}^{g} M^{\vec{d}\cdot\vec{a}_{\dashv}\cdot\vec{a}_{+}}_{\vec{a}_{\downarrow j}\cdot\vec{a}_{\vdash j}}$ is an (ℓ, λ)-AMCS with $\lambda \models \vec{z}\cdot\vec{x}_{\dashv}\cdot\vec{x}_{\dashv} = \vec{d}\cdot\vec{a}_{\dashv}\cdot\vec{a}_{+}$ and $\mathcal{F}_{\mathtt{AGG}}(M) = p$. This concludes the proof of Lemma 26. ◀