Faster Rewritings for Consistent Query Answering

Zhiwei Fan
University of Wisconsin–Madison
zhiwei@cs.wisc.edu

Paraschos Koutris
University of Wisconsin–Madison
paris@cs.wisc.edu

Xiating Ouyang
University of Wisconsin–Madison
xouyang@cs.wisc.edu

Jef Wijsen
University of Mons
jef.wijsen@umons.ac.be

ABSTRACT

Recent research efforts have focused on building systems for consistent query answering (CQA) on inconsistent databases that violate one or more integrity constraints. Given a query q, these systems return only the tuples that occur in the output of q across all possible repairs of the inconsistent database. Some systems only treat queries for which CQA is known to be **FO**-rewritable (meaning we can express the problem as a query), but their rewriting does not have any formal guarantees. Other systems treat all queries by a generic exponential time algorithm, even if the problem has a more efficient solution. This work identifies a class of join queries that admit an efficient rewriting with a formal linear-time guarantee, capturing a large fragment of queries seen in practice. Our rewriting method can be viewed as a generalization of Yannakakis's algorithm for acyclic joins to the inconsistent setting. We implement our rewriting as a system (LinCQA) that produces both a SQL query and a Datalog program as outputs. We demonstrate that LinCQA outperforms the state-of-the-art CQA systems across various queries and datasets and under different inconsistency parameters.

PVLDB Reference Format:
Zhiwei Fan, Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. Faster Rewritings for Consistent Query Answering. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/xiatingouyang/LinCQA.

1 INTRODUCTION

A database is *inconsistent* if it violates one or more integrity constraints that are supposed to be satisfied. Database inconsistency occurs due to various reasons. For example, inconsistency can naturally occur when the dataset results from an integration of heterogeneous sources, or because of noise during data collection.

Data cleaning [58] is the most widely used approach to manage inconsistent data in practice. It first *repairs* the inconsistent database by removing or modifying the inconsistent records to obey the integrity constraints. Then, users can run queries on a *clean* database. There has been a long line of research on data cleaning. Several frameworks have been proposed [1, 29–31, 60], using techniques such as knowledge bases and machine learning [7, 10, 16, 17, 26, 32, 50, 51, 59, 62]. Data cleaning has also been studied under different contexts [8, 11, 13, 37, 38, 57]. However, the process of data cleaning is often ad hoc and arbitrary choices are frequently made regarding which data to keep to restore database consistency. This comes at the price of losing important information since the number of cleaned versions of the database can be exponential in the database size. Moreover, data cleaning is commonly seen as a laborious and time-intensive process in data analysis. There have been efforts to accelerate the data cleaning process [17, 18, 59, 60], but in most applications users need to wait until the data is clean before being able to query the database.

Consistent query answering (CQA) is an alternative approach to data cleaning for managing inconsistent data [2]. Instead of singling out the "best" repair, CQA considers *all* possible repairs of the inconsistent database, returning the intersection of the query answers over all repairs, called the *consistent answers*. CQA serves as an ideal complementary procedure to data cleaning for multiple reasons. First, it deals with inconsistent data at query time without needing an expensive offline cleaning process during which the users cannot query the database. Thus, users can quickly perform preliminary data analysis to obtain the consistent answers while waiting for the cleaned version of the database. Second, consistent answers can also be returned alongside the answers obtained after data cleaning, by marking which answers are certainly/reliably correct and which are not. This information may provide further guidance in critical decision-making data analysis tasks.

In this paper, we will focus on CQA for the most common kind of integrity constraint: *primary key constraints*. A primary key constraint enforces that no two distinct tuples in the same table agree on all primary key attributes. Efficient algorithms for CQA under primary key constraints have been extensively pursued over the last two decades.

From a theoretical perspective, solving CQA for join queries is in general computationally hard (**coNP**-complete). However, for some queries, the consistent answers can be computed in polynomial time, and for some other queries, CQA is *First-Order rewritable* (**FO**-rewritable): we can construct another query such that executing it directly on the inconsistent database will return the consistent answers of the original query. After a long line of research [39, 43, 44, 47, 49], it was proven that for any self-join-free join query q, the problem is either **FO**-rewritable, polynomial-time solvable but not **FO**-rewritable, or **coNP**-complete [45].

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Table 1: A summary of systems for consistent query answering

System	Target class of CQ	Intermediate output	Backend
EQUIP [40]	all CQs	Big Integer Program (BIP)	DBMS & BIP solver
CAvSAT [24, 25]	all CQs	SAT formula	DBMS & SAT solver
Conquer [27]	C_{forest}	SQL rewriting	DBMS
Conquesto [36]	self-join-free CQ in **FO**	Datalog rewriting	Datalog engine
LinCQA (this paper)	PPJT	SQL rewriting / Datalog rewriting	DBMS or Datalog engine

From a systems standpoint, most CQA systems fall into two categories (summarized in Table 1). First, systems that can compute the consistent answers of join queries with arbitrary denial constraints but require solvers for computationally hard problems. EQUIP [40] utilizes Integer Programming solvers, and CAvSAT [24, 25] requires SAT solvers. Second, systems that output the **FO**-rewriting of the input query, but only target a specific class of queries that occurs frequently in practice. Fuxman and Miller [27] identified a class of **FO**-rewritable queries called C_{forest} and implemented their rewriting in ConQuer, which outputs a SQL query. Conquesto [36] is the most recent system targeting **FO**-rewritable join queries by producing the rewriting in Datalog.

We can identify several drawbacks with all systems above. Both EQUIP and CAvSAT rely on solvers for **NP**-complete problems, which does not guarantee efficient termination, even if the input query is **FO**-rewritable. Even though C_{forest} captures most common join queries seen in practice, it excludes queries that involve key-to-key and non-key-to-non-key joins simultaneously. Conquesto, on the other hand, implements the generic **FO**-rewriting algorithm without strong performance guarantees. Moreover, neither ConQuer nor Conquesto have guarantees on the running time of their produced rewritings.

To address these issues, we identify a subclass of acyclic join queries for which we can produce efficient **FO**-rewritings with a linear running time guarantee. We implement the algorithm to produce these rewritings into LinCQA (Linear Consistent Query Answering), a system that produces the **FO**-rewriting in both SQL and Datalog. For consistent databases, Yannakakis's algorithm [6] evaluates acyclic join queries in linear time in the size of the database. Our result shows that even when inconsistency is introduced w.r.t. primary key constraints, the consistent answers of many acyclic join queries can still be computed in linear time, exhibiting no overhead to Yannakakis's algorithm. Our technical treatment also follows Yannakakis's algorithm by considering a rooted join tree with an additional annotation of the **FO**-rewritability property, called a *pair-pruning join tree*. Our algorithm recursively follows the pair-pruning join tree to compute the consistent answers. Interestingly, our algorithm degenerates to Yannakakis's algorithm if the database has no inconsistencies.

Contributions. In summary, we make the following contributions:

- We introduce the novel notion of a *pair-pruning join tree (PPJT)* that captures a wide range of queries commonly seen in practice and properly subsumes all acyclic queries in C_{forest} (Section 4.1). We show that for any join query that admits a PPJT, we can compute its consistent answers in linear time (Section 4).
- We implement our algorithm in LinCQA, a system prototype that produces an efficient and optimized rewriting in both SQL and non-recursive Datalog with negation (Section 5).[1]
- We perform an extensive experimental evaluation comparing LinCQA to the other state-of-the-art systems for CQA. Our findings show that *(i)* a properly implemented rewriting can significantly outperform a generic CQA system (e.g., CAvSAT), *(ii)* LinCQA achieves the best performance throughout all our experiments under different inconsistency scenarios, and *(iii)* the strong theoretical guarantees of LinCQA translate to a significant performance gap for worst-case database instances.

2 RELATED WORK

Inconsistency in databases have been studied in different contexts [4, 5, 12, 14, 33, 35, 53, 61]. The notion of Consistent query answering (CQA) was introduced in the seminal work by Arenas, Bertossi, and Chomicki [2]. After twenty years, their contribution was acknowledged in a *Gems of PODS session* [9]. An overview of complexity classification results in CQA appeared recently in the *Database Principles* column of SIGMOD Record [65].

The term CERTAINTY(q) was coined in [63] to refer to CQA for Boolean queries q on databases that violate primary keys, one per relation, which are fixed by q's schema. The complexity classification of CERTAINTY(q) for the class of self-join-free Boolean conjunctive queries started with the work by Fuxman and Miller [28], and was further pursued in [39, 43–45, 47, 49], which eventually revealed that the complexity of CERTAINTY(q) for self-join-free conjunctive queries displays a trichotomy between **FO**, **L**-complete, and **coNP**-complete. A recent result also extends the complexity classification of CERTAINTY(q) to path queries that may contain self-joins [42]. The complexity of CERTAINTY(q) for self-join-free Boolean conjunctive queries with negated atoms was studied in [46]. For self-join-free Boolean conjunctive queries w.r.t. multiple keys, it remains decidable whether or not CERTAINTY(q) is in **FO** [48].

Several systems for CQA that are used for comparison in our study have already been described in the introduction: ConQuer [27], Conquesto [36], CAvSAT [24, 25], and EQUIP [40]. Most early systems for CQA used efficient solvers for Disjunctive Logic Programming and Answer Set Programming (ASP) [3, 15, 31, 53–55].

Similar notions to CQA are also emerging in machine learning with the goal of computing the consistent classification result of certain machine learning model over inconsistent training data [34].

[1] Full version of this paper and the LinCQA source code are available in [52].

3 PRELIMINARIES

We assume disjoint sets of *variables* and *constants*. We will use $x, y, z, \ldots$ to denote variables, and $a, b, c, \ldots$ for constants.

Atoms and key-equal facts. Let $\vec{x}$ be a sequence containing variables and constants. We denote $\mathsf{vars}(\vec{x})$ as the set of variables that appear in $\vec{x}$. Let R be the relation name, then $R(\underline{\vec{x}}, \vec{y})$ is an atom, where the primary key $\vec{x}$ of R is underlined and we denote $\mathsf{key}(R) = \vec{x}$. An *$R$-fact* (or simply fact or tuple) is an R-atom in which no variable occurs. Two tuples $R_1(\underline{\vec{a}_1}, \vec{b}_1), R_2(\underline{\vec{a}_2}, \vec{b}_2)$ are said to be *key-equal* if $R_1 = R_2$ and $\vec{a}_1 = \vec{a}_2$.

Database instances, blocks, and repairs. A *database schema* is a finite set of relation names. All constructs that follow are defined relative to a fixed database schema.

A *database instance* **db** is a finite set of tuples using only the relation names of the schema. A *block* of **db** is a maximal set of tuples of **db** with the same key. Whenever a database instance **db** is understood, we denote $R(\underline{\vec{c}}, *)$ as the block containing all tuples with primary-key value $\vec{c}$ in relation R. A database instance **db** is *consistent* if it does not contain distinct tuples that are key-equal (i.e., if no block of **db** contains more than one tuple). A *repair* of **db** is an inclusion-maximal consistent subset of **db**.

Conjunctive Queries. We define a *Conjunctive Query (CQ)* q as a rule of the following form:

$$q(\vec{u}) \text{ :- } R_1(\underline{\vec{x}_1}, \vec{y}_1), \ldots, R_n(\underline{\vec{x}_n}, \vec{y}_n) \tag{1}$$

where each $R_i(\underline{\vec{x}_i}, \vec{y}_i)$ is an atom for $1 \leq i \leq n$. We denote by $\mathsf{vars}(q)$ the set of variables that occur in q and $\vec{u}$ is said to be the *free variables* of q. The atom $q(\vec{u})$ is the *head* of the rule, and the remaining atoms are called the *body* of the rule.

A CQ q is Boolean (BCQ) if it has no free variables, and it is *full* if all its variables are free. We say that q has a *self-join* if some relation name occurs more than once in q. A CQ without self-joins is called *self-join-free*. A CQ q is *connected* if there are no two CQs q_1, q_2 such that $q = q_1 \cup q_2$ and $\mathsf{vars}(q_1) \cap \mathsf{vars}(q_2) = \emptyset$.

For a CQ q, let $\vec{x} = \langle x_1, \ldots, x_\ell \rangle$ be a sequence of distinct variables that occur in q and $\vec{a} = \langle a_1, \ldots, a_\ell \rangle$ be a sequence of constants, then $q_{[\vec{x} \to \vec{a}]}$ denotes the query obtained from q by replacing all occurrences of x_i with a_i for all $1 \leq i \leq \ell$.

Datalog. A Datalog program P is a finite set of rules of the form (1), with the extension that negated atoms can be used in rule bodies. A rule can be interpreted as a logical implication: if the body is true, then so is the head of the rule. We assume that rules are always safe: this means that every variable that occurs in the rule, must also occur in a non-negated atom of the rule body. A relation is an IDB relation if it is used in the head of some rule; otherwise it is an EDB relation (i.e., input relation).

Our rewriting uses non-recursive Datalog with negation. This means that the rules of a Datalog program P can be partitioned into $(P_1, P_2, \ldots, P_n)$ such that the rule body of a rule in P_i uses only IDB predicates defined by rules in some P_j with $j < i$. Here, it is understood that all rules with the same head predicate belong in the same partition.

Consistent query answering. For every BCQ q, the decision problem $\mathsf{CERTAINTY}(q)$ takes as input a database instance **db**, and asks whether q is satisfied by every repair of **db**. It is straightforward that for every BCQ q, $\mathsf{CERTAINTY}(q)$ is in **coNP**.

Attack graph. Let q be a self-join-free BCQ. We define $\mathcal{K}(q)$ as the following set of functional dependencies:

$$\mathcal{K}(q) := \{\mathsf{key}(F) \to \mathsf{vars}(F) \mid F \in q\}.$$

For every atom $F \in q$, we define $F^{+,q}$ as follows:

$$F^{+,q} := \{x \in \mathsf{vars}(q) \mid \mathcal{K}(q \setminus \{F\}) \models \mathsf{key}(F) \to x\}.$$

The attack graph of q is a directed graph whose vertices are the atoms of q. There is a directed edge from F to G ($F \neq G$) if there exists a sequence $F_0, F_1, \ldots, F_n$ of (not necessarily distinct) atoms of q such that

- $F_0 = F$ and $F_n = G$; and
- for all $i \in \{0, \ldots, n-1\}$, $\mathsf{vars}(F_i) \cap \mathsf{vars}(F_{i+1}) \not\subseteq F^{+,q}$.

A directed edge from F to G in the attack graph of q is also called an attack from F to G, denoted by $F \overset{q}{\rightsquigarrow} G$. The attack graph of q is used to determine the data complexity of $\mathsf{CERTAINTY}(q)$.

Theorem 3.1 ([45]). *Let q be a self-join-free BCQ. Then the attack graph of q is acyclic if and only if $\mathsf{CERTAINTY}(q)$ is in **FO**.*

4 A LINEAR-TIME REWRITING

In this section, we discuss an algorithm for $\mathsf{CERTAINTY}(q)$ that runs in linear time. We first provide some intuition behind our approach. We use the **Company** database shown in Table 2 as a running example, where the primary key attribute of each table is highlighted in bold and the blocks in each relation are separated by dashed lines. Consider the following query:

> *Is there an office whose contact person manages the office since 2020 and, moreover, works from home in the same city as the office?*

This query can be expressed as the following SQL query:

```
SELECT DISTINCT 1
FROM Employee E, Manager M, Contact C
WHERE E.employee_id = M.manager_id
  AND M.manager_id = C.contact_id
  AND E.office_city = E.wfh_city
  AND E.office_city = M.office_city
  AND M.office_city = C.office_city
  AND M.start_year = 2020 ;
```

We can also express the query using a non-recursive Datalog rule, where the underlined attributes denote the primary key positions.

$$q^{\mathsf{ex}}() \text{ :- } \mathsf{Employee}(\underline{x}, z, z), \mathsf{Manager}(\underline{z}, x, \mathtt{2020}), \mathsf{Contact}(\underline{z}, x)$$

We will shortly present a linear-time algorithm for computing the consistent answer to q^{ex}. To the best of our knowledge, for this query, the fastest running time guaranteed by existing systems is quadratic in the input size N. The only attacks are $\mathsf{Employee} \rightsquigarrow \mathsf{Manager}$ and $\mathsf{Employee} \rightsquigarrow \mathsf{Contact}$, and so the attack graph of q^{ex} is acyclic. Thus, by Theorem 3.1, $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ can be solved via a FO rewriting. Both EQUIP and CAvSAT solve the problem through Integer Programming or SAT solvers, which do not have any runtime guarantees. Also, since q^{ex} is not in C_{forest}, ConQuer cannot produce a FO rewriting. Finally, the non-recursive

Table 2: An example inconsistent database Company.

Employee			Manager			Contact	
employee_id	office_city	wfh_city	**office_city**	manager_id	start_year	**office_city**	contact_id
0011	Boston	Boston	Boston	0011	2020	Boston	0011
0011	Chicago	New York	Boston	0011	2021	Boston	0022
0011	Chicago	Chicago	Chicago	0022	2020	Chicago	0022
0022	New York	New York	LA	0034	2020	LA	0034
0022	Chicago	Chicago	LA	0037	2020	LA	0037
0034	Boston	New York	New York	0022	2020	New York	0022

Datalog rewriting of CERTAINTY(q^{ex}) produced by Conquesto contains cartesian products between two tables, which means that it runs in $\Omega(N^2)$ time in the worst case. One key observation is that q^{ex} requires a primary-key to primary-key join and a non-key to non-key join at the same time. As will become apparent in our technical treatment in Section 4.2, this property allows us to solve CERTAINTY(q^{ex}) in $O(N)$ time.

The remainder of this section is organized as follows. In Section 4.1, we introduce the pair-pruning join tree (PPJT). In Section 4.2, we consider every Boolean query q having a PPJT and present a novel linear time non-recursive Datalog program for CERTAINTY(q) (Theorem 4.6). Finally, we extend our result to all acyclic self-join-free CQs in Section 4.3 (Theorem 4.14) .

4.1 Pair-pruning Join Tree

Here we introduce the notion of a *pair-pruning join tree* (PPJT). We first assume that the query q is connected, and then discuss how to handle disconnected queries at the end of the section.

A *join tree* of q is an undirected tree whose nodes are the atoms of q such that for every variable x that occurs in q, the node-induced subgraph induced by the set of nodes in which x occurs, is connected. We define that q is *acyclic*[2] if it has a join tree. If τ is a subtree of a join tree of a query q, we will denote by q_τ the query whose atoms are the nodes of τ. Whenever R is a node in an undirected tree τ, then (τ, R) denotes the rooted tree obtained by choosing R as the root of the tree.

Whenever a self-join-free query is understood, we can use a relation name wherever an atom is expected. For example, we may use Employee as a shorthand for the atom Employee$(\underline{x}, y, y)$.

Definition 1 (PPJT). *Let q be an acyclic self-join-free BCQ. Let τ be a join tree of q and R a node in τ. The tree (τ, R) is a pair-pruning join tree (PPJT) of q if for any rooted subtree (τ', R') of (τ, R), the atom R' is not attacked in $q_{\tau'}$.*

Example 4.1. For the join tree τ in Figure 1, the rooted tree $(\tau, \text{Employee})$ is a PPJT for q^{ex}. The atom Employee$(\underline{x}, z, z)$ is not attacked in the query q. For the child subtree $(\tau_M, \text{Manager})$ of $(\tau, \text{Employee})$, the atom Manager$(\underline{z}, x, \mathtt{2020})$ is also not attacked in the corresponding subquery

$$q^{\text{ex}}_{\tau_M}() \text{ :- } \text{Manager}(\underline{z}, x, \mathtt{2020}), \text{Contact}(\underline{z}, x).$$

[2]Throughout this paper, whenever we say that a CQ is acyclic, we mean acyclicity as defined in [6], a notion that today is also known a α-acyclicity, to distinguish it from other notions of acyclicity.

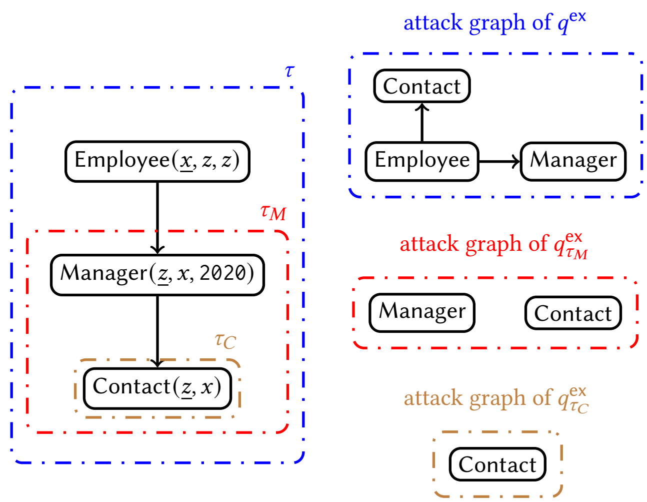


Figure 1: A pair-pruning join tree (PPJT) of the query q^{ex}, along with the attack graphs of its subtrees.

Finally, for the subtree $(\tau_C, \text{Contact})$, the atom Contact$(\underline{z}, x)$ is also not attacked in the corresponding subquery:

$$q^{\text{ex}}_{\tau_C}() \text{ :- } \text{Contact}(\underline{z}, x).$$

Hence $(\tau, \text{Employee})$ is a PPJT of q^{ex}. □

Which queries admit a PPJT? As the next proposition shows, having a PPJT is a sufficient condition for the existence of an **FO**-rewriting.

Proposition 4.2. *Let q be an acyclic self-join-free BCQ. If q has a PPJT, then the attack graph of q is acyclic.*

However, as we demonstrate in the next example, not all acyclic self-join-free BCQs with an acyclic attack graph have a PPJT.

Example 4.3. Let $q() \text{ :- } R(\underline{x, w}, y), S(\underline{y, w}, z), T(\underline{w}, z)$. The attack graph of q is acyclic since $T \rightsquigarrow R$, $T \rightsquigarrow S$, and $R \rightsquigarrow S$. The only join tree τ of q is the path $R - S - T$. However, neither (τ, R) nor (τ, S) is a PPJT for q since R and S are attacked in q; and (τ, T) is not a PPJT since in its subtree (τ', S), S is attacked in the subquery that contains R and S.

Fuxman and Miller [28] identified a large class of queries called C_{forest} that includes queries with primary-key-foreign-key joins, path queries, and queries on a star schema. This class covers most of the queries seen in practical settings. It can be shown that any acyclic query in the class C_{forest} has a PPJT.

Proposition 4.4. *If an acyclic self-join-free BCQ q is in C_{forest}, then q has a pair-pruning join tree.*

How to find a PPJT. For any acyclic self-join-free BCQ q, we can check whether q admits a PPJT via a brute-force search over all possible join trees and roots. If q involves n relations, then there are at most n^{n-1} candidate rooted join trees for PPJT (n^{n-2} join trees and for each join tree, n choices for the root). This search cost is acceptable for most join queries that do not involve too many tables, especially since in many cases the number of possible join trees is much smaller than the worst case.

In certain cases, a PPJT can be constructed efficiently without the need of an exhaustive search.

Proposition 4.5. *Let q be an acyclic self-join-free BCQ whose attack graph is acyclic. If for all two distinct atoms $F, G \in q$, neither of $\mathsf{key}(F)$ or $\mathsf{key}(G)$ is included in the other, then q has a PPJT that can be constructed in quadratic time in the number of atoms in q.*

Main Result. We previously showed that the existence of a PPJT implies a FO rewriting that computes the consistent answers. Our main result shows that it also leads to an efficient algorithm that runs in linear time.

Theorem 4.6. *Let q be an acyclic self-join-free Boolean CQ that admits a PPJT, and $\mathbf{db}$ be a database instance of size N. Then, there exists an algorithm for $\mathsf{CERTAINTY}(q)$ that runs in time $O(N)$.*

It is worth to contrast our result with computing the answers for an acyclic CQ [66]. Yannakakis' algorithm computes the result of any acyclic Boolean CQ also in linear time $O(N)$. Hence, the existence of a PPJT implies that computing $\mathsf{CERTAINTY}(q)$ will have the same asymptotic complexity.

From PPJT to Rewriting. We now present the construction of the FO rewriting starting from a PPJT. The main idea of our approach is inspired from the reification Lemma [45].

Lemma 4.7 ([45]). *Let q be a self-join-free BCQ whose attack graph is acyclic and $R(\underline{\vec{x}}, \vec{y})$ an unattacked atom in q. Let $\mathbf{db}$ be an instance for $\mathsf{CERTAINTY}(q)$. Then the following statements are equivalent:*

(1) $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$; and
(2) $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{x}\to\vec{c}]})$ for some sequence of constants $\vec{c}$.

If $R(\underline{\vec{x}}, \vec{y})$ is an atom of a query q with a PPJT (τ, R), then R is not attacked in q by definition. By Lemma 4.7, it suffices to compute *all* keys $\vec{c}$ in the relation R such that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{x}\to\vec{c}]})$, called the *good keys* of R with respect to the query q and the database $\mathbf{db}$.

Definition 2. *Let $\mathbf{db}$ be a database instance for $\mathsf{CERTAINTY}(q)$ and $R(\underline{\vec{x}}, \vec{y})$ an atom in q. We define the good keys of R with respect to query q and $\mathbf{db}$, denoted by $R_{\mathsf{gkey}}(q, \mathbf{db})$, as follows:*

$$R_{\mathsf{gkey}}(q, \mathbf{db}) := \{\vec{c} \mid \mathbf{db} \text{ a "yes"-instance for } \mathsf{CERTAINTY}(q_{\vec{x}\to\vec{c}})\}.$$

Example 4.8. For the atom $\mathsf{Employee}(\underline{x}, y, y)$ in q^{ex} and the **Company** database, we have that

$$\mathsf{Employee}_{\mathsf{gkey}}(q^{\mathsf{ex}}, \mathbf{Company}) = \{0022\}.$$

Indeed, no matter whether we choose Employee(0022, New York, New York) or Employee(0022, Chicago, Chicago) in a repair, the chosen tuple will join with some corresponding tuple in the Manager and Contact table. Therefore the query $q^{\mathsf{ex}}_{[x\to 0022]}$ will return `True` for all repairs of database **Company**. No other employee_id satisfies this condition. Since $\mathsf{Employee}_{\mathsf{gkey}}(q^{\mathsf{ex}}, \mathbf{Company}) \neq \emptyset$ and Employee is not attacked in q^{ex}, the consistent answer of q^{ex} on the **Company** database is `True`.

Lemma 4.9. *Let $R(\underline{\vec{x}}, \vec{y})$ be an atom in an acyclic self-join-free BCQ q with a PPJT (τ, R). Let $\mathbf{db}$ be an instance for $\mathsf{CERTAINTY}(q)$. For every sequence $\vec{c}$ of constants, of the same length as $\vec{x}$, the following are equivalent:*

(1) $\vec{c} \in R_{\mathsf{gkey}}(q, \mathbf{db})$; and
(2) the block $R(\underline{\vec{c}}, *)$ of $\mathbf{db}$ is non-empty and for every fact $R(\underline{\vec{c}}, \vec{d})$ in $\mathbf{db}$, the following hold:
   (a) $\{R(\underline{\vec{c}}, d)\}$ satisfies the BCQ $() \text{ :- } R(\underline{\vec{x}}, \vec{y})$; and
   (b) for every child subtree (τ_S, S) of (τ, R), there exists $\vec{s} \in S_{\mathsf{gkey}}(q_{\tau_S}, \mathbf{db})$ such that for every fact $S(\underline{\vec{s}}, \vec{t})$ in $\mathbf{db}$, the pair $\{R(\underline{\vec{c}}, \vec{d}), S(\underline{\vec{s}}, \vec{t})\}$ satisfies the BCQ

$$() \text{ :- } R(\underline{\vec{x}}, \vec{y}), S(\underline{\vec{x}'}, \vec{y}'),$$

   where $S(\underline{\vec{x}'}, \vec{y}')$ is the S-atom of q.

The term "pair-pruning" is motivated by the latter item. To test $() \text{ :- } R(\underline{\vec{x}}, \vec{y}), S(\underline{\vec{x}'}, \vec{y}')$, one can use that $S(\underline{\vec{s}}, \vec{t})$ is known to satisfy already $() \text{ :- } S(\underline{\vec{x}'}, \vec{y}')$. Therefore, it is sufficient to test that $R(\underline{\vec{c}}, \vec{d})$ and $S(\underline{\vec{s}}, \vec{t})\}$ agree on the variables shared by $R(\underline{\vec{x}}, \vec{y})$ and $S(\underline{\vec{x}'}, \vec{y}')$.

Lemma 4.9 immediately implies that if q has a PPJT (τ, R), the set $R_{\mathsf{gkey}}(q, \mathbf{db})$ can be recursively computed by items 2a and 2b, which can be naïvely implemented in $O(N^2)$ time. In the next section, we show that items 2a and 2b can be expressed in a non-recursive Datalog program that can be executed in linear time.

Disconnected CQs. If the BCQ q is not connected, then q can be written as $q = q_1, q_2, \dots, q_n$ where each q_i is connected and $\mathsf{vars}(q_i) \cap \mathsf{vars}(q_j) = \emptyset$ for $1 \leq i < j \leq n$. If every q_i has a PPJT, then $\mathsf{CERTAINTY}(q)$ can be solved by checking whether the input database is a "yes"-instance for each $\mathsf{CERTAINTY}(q_i)$, by Lemma B.1 of [42].

4.2 The Rewriting Rules

We now show how to use items 2a and 2b in Lemma 4.9 to produce an efficient rewriting. We will use Datalog syntax to describe the rewriting, since it will simplify the exposition. In the next section, we will discuss how to translate the Datalog program to SQL.

Fix a query q with a PPJT (τ, R). To describe the rewriting, we will need two predicates for every atom S in the tree (let T be the unique parent of S in τ):

- the predicate S_{fkey} has arity equal to $|\mathsf{key}(S)|$ and collects the primary-key values of the S-table that cannot contribute to a consistent answer for q; and
- the predicate S_{join} has arity equal to $|\mathsf{vars}(S) \cap \mathsf{vars}(T)|$ and collects the values for these variables in the S-table that may contribute to a consistent answer.

The rewriting algorithm traverses the PPJT recursively starting from the root of the tree:

We now describe how each step is implemented in detail. Figure 2 depicts how each step generates the rewriting rules for q^{ex}.

Exit-Rule $\{$ $\text{Employee}_{\text{join}}()$:- $\text{Employee}(x, y, y), \neg\text{Employee}_{\text{fkey}}(x)$.

Self-Pruning $\{$ $\text{Employee}_{\text{fkey}}(z_1)$:- $\text{Employee}(z_1, z_2, z_3), z_3 \neq z_2$.

$\text{Employee}(\underline{x}, z, z)$

$\text{Employee}_{\text{fkey}}(x)$:- $\text{Employee}(x, z, z), \neg\text{Manager}_{\text{join}}(z, x)$. $\}$ Pair-Pruning

Exit-Rule $\{$ $\text{Manager}_{\text{join}}(z, x)$:- $\text{Manager}(z, x, 2020), \neg\text{Manager}_{\text{fkey}}(z)$.

Self-Pruning $\{$ $\text{Manager}_{\text{fkey}}(z_1)$:- $\text{Manager}(z_1, z_2, z_3), z_3 \neq 2020$.
$\text{Manager}_{\text{fkey}}(z)$:- $\text{Manager}(z, x, 2020), \text{Manager}(z, z_1, z_2), z_1 \neq x$.

$\text{Manager}(\underline{z}, x, 2020)$

$\text{Manager}_{\text{fkey}}(z)$:- $\text{Manager}(z, x, 2020), \neg\text{Contact}_{\text{join}}(z, x)$. $\}$ Pair-Pruning

Exit-Rule $\{$ $\text{Contact}_{\text{join}}(z, x)$:- $\text{Contact}(z, x), \neg\text{Contact}_{\text{fkey}}(z)$.

Self-Pruning $\{$ $\text{Contact}_{\text{fkey}}(z)$:- $\text{Contact}(z, x), \text{Contact}(z, z_1), z_1 \neq x$.

$\text{Contact}(\underline{z}, x)$

Figure 2: The non-recursive Datalog program produced for CERTAINTY(q^{ex}).

Algorithm 1: PPJT-Rewriting(τ, R)

Input: PPJT (τ, R) of q
Output: a Datalog program computing $R_{\text{gkey}}(q, \mathbf{db})$
Self-Pruning(R)
foreach *child node S of R in τ* **do**
    PPJT-Rewriting(τ, S)
    Pair-Pruning(R, S)
Exit-Rule(R)

Self-Pruning(R): Let $R(\underline{x_1, \dots x_k}, x_{k+1}, \dots, x_n)$, where x_i can be a variable or a constant. The first rule finds the primary-key values of the R-table that can be pruned because of the local selection conditions imposed on R.

Rule 1. *If $x_i = c$ for some constant c, we add the rule*

$$R_{\text{fkey}}(z_1, \dots, z_k) \text{ :- } R(z_1, \dots, z_n), z_i \neq c.$$

If for some variable x_i there exists $j < i$ with $x_i = x_j$, we add the rule

$$R_{\text{fkey}}(z_1, \dots, z_k) \text{ :- } R(z_1, \dots, z_n), z_i \neq z_j.$$

Here, $z_1, \dots, z_n$ are fresh distinct variables.

The second rule finds the primary-key values of the R-table that can be pruned because R joins with its (unique) parent T in the tree. The underlying intuition is that if some R-block of the input database contains two tuples that disagree on a non-key position that is used in an equality-join with T, then for every given T-tuple t, we can pick an R-tuple in that block that does not join with t.

Rule 2. *For each variable x_i with $i > k$ (so in a non-key position) such that $x_i \in \text{vars}(T)$, we produce a rule*

$$R_{\text{fkey}}(x_1, \dots, x_k) \text{ :- } R(x_1, \dots, x_k, x_{k+1}, \dots, x_n),$$
$$R(x_1, \dots, x_k, z_{k+1}, \dots, z_k), z_i \neq x_i.$$

where $z_{k+1}, \dots, z_n$ are fresh variables.

Example 4.10. The self-pruning step at the PPJT (τ, Employee) produces one rule (from Rule 1) since the atom $\text{Employee}(\underline{x}, z, z)$ has one variable that occurs twice. When executed on the **Company** database, keys 0011 and 0034 would be added to $\text{Employee}_{\text{fkey}}$.

The self-pruning phase on(τ_M, Manager) produces two rules, one using Rule 1 and one using Rule 2. When executed on the **Company** database, the keys $\{Boston, LA\}$ are added to $\text{Manager}_{\text{fkey}}$.

Finally, the self-pruning phase on the PPJT (τ_C, Contact) produces one rule using Rule 2 (here x is the non-key join variable). Hence, the keys Boston and LA will be added to $\text{Contact}_{\text{fkey}}$. ◻

Pair-Pruning(R, S): Suppose that q contains the atoms $R(\underline{\vec{x}}, \vec{y})$ and $S(\underline{\vec{u}}, \vec{v})$. Let $\vec{w}$ be a sequence of distinct variables containing all (and only) variables in $\text{vars}(R) \cap \text{vars}(S)$.

The pair-pruning step introduces one rule, which uses the fact that, by item 2b, a key $\vec{c} \in R_{\text{key}}(q, \mathbf{db})$ can be pruned if there is a fact $R(\underline{\vec{c}}, \vec{d})$ such that for every "good" key $\vec{s}$ from the S-table, $R(\underline{\vec{c}}, \vec{d})$ does not join with any fact in $S(\underline{\vec{s}}, *)$.

Rule 3. *Add the rule*

$$R_{\text{fkey}}(\vec{x}) \text{ :- } R(\vec{x}, \vec{y}), \neg S_{\text{join}}(\vec{w}).$$

The rule is safe because every variable in $\vec{w}$ occurs in $R(\vec{x}, \vec{y})$.

Example 4.11. Figure 2 shows the two pair-pruning rules generated (in general, there will be one pair-pruning rule for each edge in the PPJT). In both cases, the join variables are $\{z, x\}$. ◻

Exit-Rule(R): Suppose that q contains $R(\underline{\vec{x}}, \vec{y})$. Let $\vec{w}$ be a sequence of distinct variables containing all (and only) variables in R and its parent node in τ (so the join variables). If R is the root node, then $\vec{w}$ is the empty vector. The exit rule uses all the pruned blocks to compute the join tuples that will be used for pair pruning.

Rule 4. *If R_{fkey} exists in the head of a rule, we produce the rule*

$$R_{\text{join}}(\vec{w}) \text{ :- } R(\vec{x}, \vec{y}), \neg R_{\text{fkey}}(\vec{x}).$$

Otherwise, we produce the rule

$$R_{\mathsf{join}}(\vec{w}) \text{ :- } R(\vec{x}, \vec{y}).$$

Example 4.12. Figure 2 shows the three exit rules for q^{ex}—one rule for each node in the tree. We will use the boolean predicate $\mathsf{Employee}_{\mathsf{join}}$ to determine whether True is the consistent answer to the running query. □

Runtime Analysis It is easy to see that Rules 1, 3, and 4 can be evaluated in linear time. We now argue how to evaluate Rule 2 in linear time as well. Indeed, instead of performing the self-join on the key, it suffices to create a hash table using the primary key as the key (which can be constructed in linear time). Then, for every value of the key, we can easily check whether all tuples in the block have the same value at the i-th attribute.

4.3 Extension to Non-Boolean Queries

Let $q(\vec{u})$ be an acyclic self-join-free CQ with free variables $\vec{u}$, and **db** be a database instance. If $\vec{c}$ is a sequence of constants of the same length as $\vec{u}$, we say that $\vec{c}$ is a *consistent answer* to q on **db** if $\vec{c} \in q(I)$ in every repair I of **db**. Furthermore, we say that $\vec{c}$ is a *possible answer* to q on **db** if $\vec{c} \in q(\mathbf{db})$. It can be easily seen that for CQs every consistent answer is a possible answer.

Lemma 4.13 reduces computing the consistent answers of non-Boolean queries to that of Boolean queries.

Lemma 4.13. *Let q be a CQ with free variables $\vec{u}$, and let $\vec{c}$ be a sequence of constants of the same length as $\vec{u}$. Let* **db** *be an database instance. Then $\vec{c}$ is a consistent answer to q on* **db** *if and only if* **db** *is a "yes"-instance for* $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$.

If q has free variables $\vec{u} = (u_1, u_2, \ldots, u_n)$, we say that q admits a PPJT if the Boolean query $q_{[\vec{u}\to\vec{c}]}$ admits a PPJT, where $\vec{c} = (c_1, c_2, \ldots, c_n)$ is a sequence of distinct constants. We can now state our main result for non-Boolean CQs.

Theorem 4.14. *Let q be an acyclic self-join-free Conjunctive Query that admits a PPJT, and* **db** *be a database instance of size N. Let OUT_p be the set of possible answers to q on* **db**, *and OUT_c the set of consistent answers to q on* **db**. *Then:*

(1) The set of consistent answers can be computed in time $O(N \cdot |\mathsf{OUT}_p|)$; and
(2) Moreover, if q is full, the set of consistent answers can be computed in time $O(N + |\mathsf{OUT}_c|)$.

To contrast this with Yannakakis result, for acyclic full CQs we have a running time of $O(N + |\mathsf{OUT}|)$, and a running time of $O(N \cdot |\mathsf{OUT}|)$ for general CQs.

Proof. By Lemma 4.13, we can first compute $q(\mathbf{db})$ to obtain a set of *possible* answers, which must contain all the consistent answers of q on **db**. We then return all answers $\vec{c} \in q(\mathbf{db})$ such that **db** is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$. This approach gives an algorithm with running time $O(N \cdot |\mathsf{OUT}_i|)$.

If q is full, there is an algorithm that computes the set of consistent answers even faster. The algorithm proceeds by first computing the consistent part R^c for any relation R, that is, the set of facts that do not conflict with any other fact in R. Next, it evaluates q on just the consistent part of the database. The algorithm is correct, since by Lemma 4.13, all constants $\vec{c}$ such that the block $R(\vec{c}, *)$ is inconsistent would be pruned, and thus R^c is precisely the relation obtained by filtering the keys of R on R_{gkey}. □

Rewriting for non-Boolean Queries Let $\vec{c} = (c_1, c_2, \ldots, c_n)$ be any sequence of distinct constants that do not appear in **db**. If $q_{[\vec{u}\to\vec{c}]}$ has a PPJT, the Datalog rewriting for $\mathsf{CERTAINTY}(q)$ can be obtained as follows:

(1) Produce the program P for $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$ using the rewriting algorithm for Boolean queries (Subsection 4.2).
(2) Replace each occurrence of the constant c_i in P with the free variable u_i.
(3) Add the rule: $\mathsf{ground}(\vec{u})$:- $\mathsf{body}(q)$.
(4) For a relation T, let $\vec{u}_T$ be a sequence of all free variables that occur in the subtree rooted at T. Then, append $\vec{u}_T$ to every occurrence of T_{join} and T_{fkey}.
(5) For any rule of P that has a free variable u_i that is unsafe, add the atom $\mathsf{ground}(\vec{u})$ to the rule.

Example 4.15. Consider the non-Boolean query

$$q^{\mathsf{nex}}(w) : -\mathsf{Employee}(\underline{x}, z, z), \mathsf{Manager}(\underline{z}, x, w), \mathsf{Contact}(\underline{z}, x).$$

Note that this query is the same as q^{ex}, with the only difference that the constant 2020 is replaced by the free variable w. Hence, the program P for $\mathsf{CERTAINTY}(q^{\mathsf{nex}}_{[w\to c]})$ is the same as Figure 2, with the only difference that 2020 is replaced by the constant c. The ground rule produced is:

$$\mathsf{ground}(w) : -\mathsf{Employee}(x, z, z), \mathsf{Manager}(z, x, w), \mathsf{Contact}(z, x).$$

To see how the rule of P would change for the non-Boolean case, consider the self-pruning rule for Contact. This rule would remain as is, because it contains no free variable and the predicate $\mathsf{Contact}_{\mathsf{fkey}}$ remains unchanged. In contrast, consider the first self-pruning rule for Manager, which in P would be:

$$\mathsf{Manager}_{\mathsf{fkey}}(z_1) \text{ :- } \mathsf{Manager}(z_1, z_2, z_3), z_3 \neq w.$$

Here, w is unsafe, so we would need to add the atom $\mathsf{ground}(w)$. Additionally, w is now a free variable in the subtree rooted at Manager, so the predicate $\mathsf{Manager}_{\mathsf{fkey}}(z_1)$ becomes $\mathsf{Manager}_{\mathsf{fkey}}(z_1, w)$. The transformed rule will be:

$$\mathsf{Manager}_{\mathsf{fkey}}(z_1, w) \text{ :- } \mathsf{Manager}(z_1, z_2, z_3), z_3 \neq w, \mathsf{ground}(w).$$

The full rewriting for q^{nex} can be seen in Figure 3. □

The above rewriting process may introduce cartesian products in the rules. In the next section, we will see how we can tweak the rules in order to avoid this inefficiency.

5 IMPLEMENTATION

In this section, we first present LinCQA, a system that produces the consistent **FO**rewriting of a query q in both Datalog and SQL formats if q has a PPJT. Having a rewriting in both formats allows us to use both Datalog and SQL engines as a backend. We then discuss the flaws of two existing CQA systems, namely ConQuer and Conquesto, and present our improvements upon both.

GROUND-RULE $\left\{ \text{ground}(w) :- \text{Employee}(x,z,z), \text{Manager}(z,x,w), \text{Contact}(z,x). \right.$

EXIT-RULE $\left\{ \text{Employee}_{\text{join}}(w) \text{ :- } \text{Employee}(x,y,y), \neg\text{Employee}_{\text{fkey}}(x,w), \text{ground}(w). \right.$

SELF-PRUNING $\left\{ \text{Employee}_{\text{fkey}}(z_1,w) \text{ :- } \text{Employee}(z_1,z_2,z_3), z_3 \neq z_2, \text{ground}(w). \right.$

Employee($\underline{x}, z, z$)

$\text{Employee}_{\text{fkey}}(x,w) \text{ :- } \text{Employee}(x,z,z), \neg\text{Manager}_{\text{join}}(z,x,w), \text{ground}(w).$ $\}$ PAIR-PRUNING

EXIT-RULE $\left\{ \text{Manager}_{\text{join}}(z,x,w) \text{ :- } \text{Manager}(z,x,w), \neg\text{Manager}_{\text{fkey}}(z). \right.$

SELF-PRUNING $\left\{ \begin{array}{l} \text{Manager}_{\text{fkey}}(z_1,w) \text{ :- } \text{Manager}(z_1,z_2,z_3), z_3 \neq w, \text{ground}(w). \\ \text{Manager}_{\text{fkey}}(z,w) \text{ :- } \text{Manager}(z,x,w), \text{Manager}(z,z_1,z_2), z_1 \neq x. \end{array} \right.$

Manager($\underline{z}, x, w$)

$\text{Manager}_{\text{fkey}}(z,w) \text{ :- } \text{Manager}(z,x,w), \neg\text{Contact}_{\text{join}}(z,x).$ $\}$ PAIR-PRUNING

EXIT-RULE $\left\{ \text{Contact}_{\text{join}}(z,x) \text{ :- } \text{Contact}(z,x), \neg\text{Contact}_{\text{fkey}}(z). \right.$

SELF-PRUNING $\left\{ \text{Contact}_{\text{fkey}}(z) \text{ :- } \text{Contact}(z,x), \text{Contact}(z,z_1), z_1 \neq x. \right.$

Contact($\underline{z}, x$)

Figure 3: The non-recursive Datalog program produced for CERTAINTY(q^{nex})**.**

5.1 LinCQA: Rewriting in Datalog/SQL

Our implementation takes as input a self-join-free CQ q written in either Datalog or SQL. LinCQA first checks whether the query q admits a PPJT, and if so, it proceeds to produce the consistent **FO**rewriting of q in either Datalog or SQL.

5.1.1 Datalog rewriting. LinCQA implements all rules introduced in Subsection 4.2, with one modification to the ground rule atom. Let the input query be

$$q(\vec{u}) \text{ :- } R_1(\underline{\vec{x_1}}, \vec{y_1}), R_2(\underline{\vec{x_2}}, \vec{y_2}), \ldots, R_k(\underline{\vec{x_k}}, \vec{y_k}).$$

In Subsection 4.3, the head of the ground rule is $\text{ground}(\vec{u})$. In the implementation, we replace that rule with

$$\text{ground}^*(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_k, \vec{u}) \text{ :- } \text{body}(q),$$

keeping the key variables of all atoms. For each unsafe rule with head $R_{i,\text{label}}$ where label $\in \{\text{fkey}, \text{join}\}$, let $\vec{v}$ be the key in the occurrence of R_i in the body of the rule (if the unsafe rule is produced by Rule 2, both occurrences of R_i share the same key). Then, we add to the rule body the atom

$$\text{ground}^*(\vec{z}_1, \ldots, \vec{z}_{i-1}, \vec{v}, \vec{z}_{i+1}, \ldots, \vec{z}_k, \vec{u})$$

where $\vec{z}_i$ is a sequence of fresh variables of the same length as $\vec{x}_i$.

The rationale is that appending $\text{ground}(\vec{u})$ to all unsafe rules could potentially introduce a Cartesian product between $\text{ground}(\vec{u})$ and some existing atom $R(\vec{v}, \vec{w})$ in the rule. The Cartesian product has size $O(N \cdot |\text{OUT}_p|)$ and would take $\Omega(N \cdot |\text{OUT}_p|)$ time to compute, often resulting in inefficient evaluations or even out-of-memory errors. On the other hand, adding ground* guarantees a join with an existing atom in the rule. Hence the revised rules would take $O(N + |\text{ground}^*|)$ time to compute. Note that the size of ground* can be as large as $N^k \cdot |\text{OUT}_p|$ in the worst case; but as we observe in the experiments, the size of ground* is small in practice.

5.1.2 SQL rewriting. We now describe how to translate the Datalog rules in Subsection 4.2 to SQL queries. Given a query q, we first denote the following:

(1) **KeyAttri**(R): the primary key attributes of relation R;
(2) **JoinAttri**(R, T): the attributes of R that join with T;
(3) **Comp**(R): the conjunction of comparison predicates imposed entirely on R, excluding all join predicates (e.g., R.A = 42 and R.A = R.B); and
(4) **NegComp**(R): the negation of **Comp**(R) (e.g., R.$A \neq 42$ or R.$A \neq$ R.B).

Translation of Rule 1. We translate Rule 1 of Subsection 4.2 into the following SQL query computing the keys of R.

```
SELECT KeyAttri(R) FROM R WHERE NegComp(R)
```

Translation of Rule 2. We first produce the projection on all key attributes and the joining attributes of R with its parent T (if it exists), and then compute all blocks containing at least two facts that disagree on the joining attributes. This can be effectively implemented in SQL with `GROUP BY` and `HAVING`.

```
SELECT KeyAttri(R)
FROM (SELECT DISTINCT KeyAttri(R), JoinAttri(R,T) FROM R) t
GROUP BY KeyAttri(R)
HAVING COUNT(*) > 1
```

Translation of Rule 3. For Rule 3 in the pair-pruning phase, we need to compute all blocks of R containing some fact that does not join with some fact in S_{join} for some child node S of R. This can be achieved through a *left outer join* between R and each of its child node $\text{S}^1_{\text{join}}, \text{S}^2_{\text{join}}, \ldots, \text{S}^k_{\text{join}}$, which are readily computed in the recursive steps. For each $1 \leq i \leq k$, let the attributes of S^i be

$B^i_1, B^i_2, \ldots, B^i_{m_i}$, joining with attributes $A_{\alpha^i_1}, A_{\alpha^i_2}, \ldots, A_{\alpha^i_{m_i}}$ in R respectively. We produce the following rule:

$$\begin{array}{l}
\texttt{SELECT } \mathbf{KeyAttri}(\mathsf{R}) \texttt{ FROM R} \\
\texttt{LEFT OUTER JOIN } \mathsf{S}^1_{\text{join}} \texttt{ ON} \\
\quad \mathsf{R}.A_{\alpha^1_1} = \mathsf{S}^1_{\text{join}}.B^1_1 \texttt{ AND } \ldots \texttt{ AND } \mathsf{R}.A_{\alpha^1_{m_1}} = \mathsf{S}^1_{\text{join}}.B^1_{m_1} \\
\ldots \\
\texttt{LEFT OUTER JOIN } \mathsf{S}^k_{\text{join}} \texttt{ ON} \\
\quad \mathsf{R}.A_{\alpha^k_1} = \mathsf{S}^k_{\text{join}}.B^k_1 \texttt{ AND } \ldots \texttt{ AND } \mathsf{R}.A_{\alpha^k_{m_k}} = \mathsf{S}^n_{\text{join}}.B^k_{m_k} \\
\texttt{WHERE } \mathsf{S}^1_{\text{join}}.A_{\alpha^1_1} \texttt{ IS NULL OR } \ldots \; \mathsf{S}^1_{\text{join}}.A_{\alpha^1_{m_1}} \texttt{ IS NULL OR} \\
\qquad \mathsf{S}^2_{\text{join}}.A_{\alpha^2_1} \texttt{ IS NULL OR } \ldots \; \mathsf{S}^2_{\text{join}}.A_{\alpha^2_{m_1}} \texttt{ IS NULL OR} \\
\qquad \ldots \\
\qquad \mathsf{S}^k_{\text{join}}.A_{\alpha^k_1} \texttt{ IS NULL OR } \ldots \; \mathsf{S}^k_{\text{join}}.A_{\alpha^k_{m_1}} \texttt{ IS NULL}
\end{array}$$

The *inconsistent blocks* represented by the keys found by the above three queries are *combined* using `UNION` and stored in the intermediate relation (e.g., R_{fkey} in Rule 1, 2, 3) via a `WITH` clause.

Translation of Rule 4. Finally, we translate Rule 4 computing the values on join attributes between *good blocks* in R and its unique parent T if it exists. Let $A_1, A_2, \ldots, A_k$ be the key attributes of R.

$$\begin{array}{l}
\texttt{SELECT } \mathbf{JoinAttri}(\mathsf{R}, \mathsf{T}) \texttt{ FROM R} \\
\texttt{WHERE NOT EXISTS (} \\
\quad \texttt{SELECT * FROM } \mathsf{R}_{\text{fkey}} \\
\quad \texttt{WHERE } \mathsf{R}.A_1 = \mathsf{R}_{\text{fkey}}.A_1 \texttt{ AND } \ldots \texttt{ AND } \mathsf{R}.A_k = \mathsf{R}_{\text{fkey}}.A_k \texttt{)}
\end{array}$$

If R is the root relation of the PPJT, we replace **JoinAttri**(R, T) with `DISTINCT 1` (i.e. a Boolean query). Otherwise, the results returned from the above query are stored in the intermediate relation R_{join} via a `WITH` clause and the recursive process continues as described in Algorithm 1.

Extention to non-Boolean queries. Let q be a non-Boolean query. We use **ProjAttri**(q) to denote a sequence of projection attributes of q and let **CompPredicate**(q) be the comparison expression in the `WHERE` clause of q. We first produce the SQL query that computes the ground* predicate and stores the result as a view called `Ground`.

$$\begin{array}{l}
\texttt{SELECT } \mathbf{KeyAttri}(\mathsf{R}_1)\texttt{,}\mathbf{KeyAttri}(\mathsf{R}_2)\texttt{,}\ldots\texttt{,}\mathbf{KeyAttri}(\mathsf{R}_k)\texttt{,}\mathbf{ProjAttri}(q) \\
\texttt{FROM } \mathsf{R}_1\texttt{, } \mathsf{R}_2\texttt{, } \ldots\texttt{, } \mathsf{R}_k \\
\texttt{WHERE } \mathbf{CompPredicate}(q)
\end{array}$$

We then modify each SQL statement as follows. Consider a SQL statement whose corresponding Datalog rule is unsafe and let $T(\vec{v}, \vec{w})$ be an atom in the rule body. Let $\vec{u}_T$ be a sequence of free variables in q_{τ_T} and let **FreeAttri**(T) be a sequence of projection attributes in q_{τ_T} that corresponds to the variables in $\vec{u}_T$. Recall that $T_{\text{join}}(\vec{v})$ and $T_{\text{fkey}}(\vec{v})$ would be replaced with $T_{\text{join}}(\vec{v}, \vec{u}_T)$ and $T_{\text{fkey}}(\vec{v}, \vec{u}_T)$ respectively. We thus first append **FreeAttri**(T) to the `SELECT` clause and then add a `JOIN` between table T and ground on all attributes in KeyAttri(T). Finally, for each projection attribute ground.A whose corresponding free variable appears in some negative IDB in the rule body,

- if the rule is produced by Rule 3, in each `LEFT OUTER JOIN` with S^i_{join} we add the expression `ground.A = ` $\mathsf{S}^i_{\text{join}}.B$ connected by the `AND` operator, where B is a projection variable in $\mathsf{S}^i_{\text{join}}$. In the `WHERE` clause we also add an expression `ground.A IS NULL`, connected by the `OR` operator.
- if the rule is produced by Rule 4, in the `WHERE` clause of the subquery we add an expression `ground.A = ` R_{fkey}`.A`.

5.2 Improvements upon existing CQA systems

ConQuer [27] and Conquesto [36] are two other CQA systems targeting their own subclasses of **FO**-rewritable queries. In order to provide a fair comparison with LinCQA, we implemented our own optimized version of both rewriting methods.

ConQuer. Fuxman and Miller [28] identified C_{forest}, a class of CQs whose consistent answers can be computed via a **FO**rewriting. However, their accompanying system can only handle queries in C_{forest} whose join graph is a tree, unable to handle the query in C_{forest} whose join graph consists of at least 2 connected components [27]. Since we were unable to find the original ConQuer implementation, we re-implemented ConQuer and added an efficient implementation of the method `RewriteConsistent` in Figure 2 of [28], enabling us to produce the consistent SQL rewriting for every query in C_{forest}.

FastFO: an optimized implementation of Conquesto. Conquesto [36] produces a non-recursive Datalog program that implements the algorithm in [45], targetting all **FO**-rewritable self-join-free CQs. Despite being able to handle the largest class of **FO**-rewritable queries among all CQA systems, it suffers from repeated computation and unnecessary cartesian products. For example, consider the CQ $q(z)$:- $R_1(\underline{x}, y, z), R_2(\underline{y}, v, w)$. The Conquesto rewriting is presented below. Note that rule (5) involves a Cartesian product, and that the bodies for the rules computing $\mathsf{Sr}_{\mathsf{R}_1}(y)$ and $\mathsf{Sr}_{\mathsf{R}_2}(z)$ overlap, resulting in re-computation.

$$\mathsf{Sr}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{R}_2(y, v, w). \tag{2}$$
$$\mathsf{Yes}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{Sr}_{\mathsf{R}_2}(y), \mathsf{R}_2(y, v, w). \tag{3}$$
$$\mathsf{Sr}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{R}_1(x, y, z), \mathsf{R}_2(y, v, w). \tag{4}$$
$$\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, v_2), \mathsf{Yes}_{\mathsf{R}_2}(y), v_2 = z. \tag{5}$$
$$\mathsf{Bb}_{\mathsf{R}_1}(x, z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, v_2), \neg\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z). \tag{6}$$
$$\mathsf{Yes}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, z), \neg\mathsf{Bb}_{\mathsf{R}_1}(x, z). \tag{7}$$

We thus implement FastFO to address the aforementioned issues, incorporating our ideas in Subsection 5.1.1. Instead of re-computing the *local safe ranges* such as $\mathsf{Sr}_{\mathsf{R}_1}(y)$ and $\mathsf{Sr}_{\mathsf{R}_2}(z)$, we compute a *global safe range* $\mathsf{Sr}(x, y, z)$, which includes all key variables from all atoms and the free variables. This removes all undesired Cartesian products and the recomputations of the local safe ranges at once. The FastFO rewriting for q is presented below.

$$\mathsf{Sr}(x, y, z) \text{ :- } \mathsf{R}_1(x, y, z), \mathsf{R}_2(y, v, w). \tag{8}$$
$$\mathsf{Yes}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{Sr}(x, y, z), \mathsf{R}_2(y, v, w). \tag{9}$$
$$\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z) \text{ :- } \mathsf{Sr}(x, \_, z), \mathsf{R}_1(x, y, v_2), \mathsf{Yes}_{\mathsf{R}_2}(y), v_2 = z. \tag{10}$$
$$\mathsf{Bb}_{\mathsf{R}_1}(x, z) \text{ :- } \mathsf{Sr}(x, \_, z), \mathsf{R}_1(x, y, v_2), \neg\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z). \tag{11}$$
$$\mathsf{Yes}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{Sr}(x, y, z), \mathsf{R}_1(x, y, z), \neg\mathsf{Bb}_{\mathsf{R}_1}(x, z). \tag{12}$$

For evaluation, the computation of each intermediate relation (i.e. all rules except for the one computing $\mathsf{Yes}_{\mathsf{R}_1}(z)$) is then translated to a SQL subquery via a `WITH` clause.

6 EXPERIMENTS

In this section, we present our experimental evaluation focusing on answering the following questions:

(1) Do consistent first-order rewriting techniques have performance benefits when compared to a generic state-of-the-art CQA system (e.g., CAvSAT)?
(2) How does LinCQA perform compared to other existing CQA techniques?
(3) How do different CQA techniques behave on inconsistent databases with different properties (e.g., varying inconsistent block sizes, inconsistency)?
(4) Are there instances where we can observe the worst-case guarantee of LinCQA that other CQA techniques lack?

To answer these questions, we perform a set of experiments using both synthetic workloads and the inconsistent databases generated from the TPC-H benchmark as being used in previous work. We compare LinCQA against several state-of-the-art CQA systems. To the best of our knowledge, this is the most comprehensive performance evaluation of existing CQA techniques.

6.1 Experimental Setup

We next briefly describe the setup of our experiments.

System configuration. All of our experiments are conducted on a bare-metal server in Cloudlab [19], a large cloud infrastructure. The server runs Ubuntu 18.04.1 LTS and has two Intel Xeon E5-2660 v3 2.60 GHz (Haswell EP) processors. Each processor has 10 cores, and 20 hyper-threading hardware threads. The server has 160GB memory and each NUMA node is directly attached to 80GB of memory. We run Microsoft SQL Server 2019 Developer Edition (64-bit) on Linux as the relational backend for all CQA systems. For CAvSAT, MaxHS v3.2.1 [20] is used as the solver for the output WPMaxSAT instances.

Other CQA systems. We compare the performance of LinCQA with several state-of-the-art CQA methods.

ConQuer: a CQA system that outputs a SQL rewriting for queries that are in C_{forest} [27]. We implement the complete version of ConQuer as described in Section 5.2.

FastFO: our own implementation of the general method that can handle any query where CQA is **FO**-rewritable. It improves upon Conquesto [36] by addressing a few of its inefficiencies as described in Section 5.2.

CAvSAT: a recent SAT-based system. It reduces the complement of CQA with arbitrary denial constraints to a SAT problem, which is solved with an efficient SAT solver [24].

We note that CAvSAT requires preprocessing the input relations into consistent and inconsistent parts while all **FO**-based systems do not. Here we only report the running time of computing consistent answers for all systems. For each query and database instance shown in the experimental results, we run each CQA system five times (unless timed out), discarding the first run and report the average time of the last four runs.

6.2 Databases and Queries

Synthetic workload. We consider the synthetic workload used in previous work [23, 24, 41]. Specifically, we take the seven queries that are consistent first-order rewritable in [23, 24, 41]. These queries feature joins between primary-key attributes to foreign-key attributes and projection on non-key attributes:

$$\begin{aligned}
q_1(z) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_3(\underline{y}, v, w).\\
q_2(z, w) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_2(\underline{y}, v, w).\\
q_3(z) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_2(\underline{y}, v, w)\mathrm{R}_7(\underline{v}, u, d).\\
q_4(z, d) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_2(\underline{y}, v, w), \mathrm{R}_7(\underline{v}, u, d).\\
q_5(z) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_8(\underline{y, v}, w).\\
q_6(z) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_6(\underline{t}, y, w), \mathrm{R}_9(\underline{x}, y, d).\\
q_7(z) &\text{ :- } \mathrm{R}_3(\underline{x}, y, z), \mathrm{R}_4(\underline{y}, x, w), \mathrm{R}_{10}(\underline{x}, y, d).
\end{aligned}$$

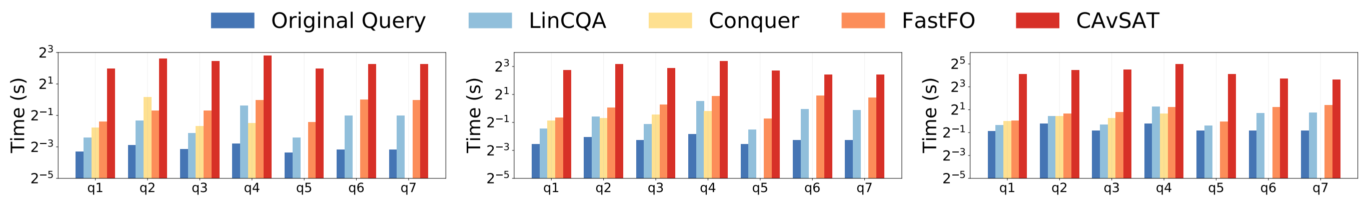
The synthetic instances are generated in two phases. In the first phase, we generate the consistent instance, while in the second phase we inject inconsistency. We use the following parameters for data generation: (i) rSize: the number of tuples per relation, (ii) inRatio: the ratio of the number of tuples that violate primary key constraint (i.e., number of tuples that are in inconsistent blocks) to the total number of tuples of the database, and (iii) bSize: the number of inconsistent tuples in each inconsistent block.

Consistent data generation. Each relation in the consistent database has the same number of tuples, so that injecting inconsistency with specified bSize and inRatio makes the total number of tuples in the relation equal to rSize. The data generation is *query-specific*: for each of the seven queries, the data is generated in a way to ensure the output size of the original query on the consistent database is reasonably large. To achieve this purpose, when generating the database instance for one of the seven queries, we ensure that for any two relations that join on some attributes, the number of matching tuples in each relation is approximately 25%; for the third attribute in each ternary relation that does not participate in a join but is sometimes projected out, the values are chosen uniformly from the range $[1, \text{rSize}/10]$.

Inconsistency injection. In each relation, we first select a number of primary keys (or number of inconsistent blocks inBlockNum) from the generated consistent instance. Then, for each selected primary key, the inconsistency is injected by inserting the *same number of additional tuples* ($\text{bSize}-1$) into each block. The parameter inBlockNum is calculated by the given rSize, inRatio and bSize: $\text{inBlockNum} = (\text{inRatio} \cdot \text{rSize})/\text{bSize}$.

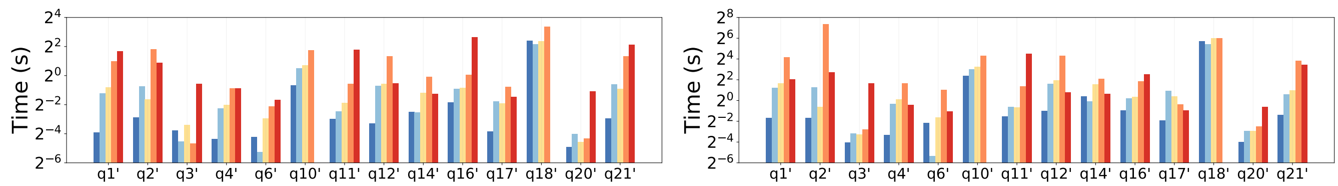
TPC-H benchmark. We also altered the 22 queries from the original TPC-H benchmark [56] by removing aggregation, nested subqueries and selection predicates other than constant constraints, yielding 14 simplified conjunctive queries, namely query q_1', q_2', q_3', q_4', q_6', q_{10}', q_{11}', q_{12}', q_{14}', q_{16}', q_{17}', q_{18}', q_{20}', q_{21}'. All of the 14 queries are in C_{forest} and hence each query has a PPJT, suggesting that they can be handled by both ConQuer and LinCQA.

We generate the inconsistent instances by injecting inconsistency into the TPC-H databases of scale factor (SF) 1 and 10 in the same way as described for the synthetic data. The only difference is that for a given consistent database instance, instead of fixing rSize for the inconsistent database, we determine the number of inconsistent tuples to be injected based on the size of the consistent database instance, the specified inRatio and bSize.



(a) rSize = 500K **(b)** rSize = 1M **(c)** rSize = 5M

Figure 4: Performance comparison of different CQA systems on a synthetic workload with varying relation sizes.



(a) SF = 1 **(b)** SF = 10

Figure 5: Performance comparison of different CQA systems on the TPC-H benchmark with varying scale factor (SF)**.**

6.3 Experimental Results

In this section, we report our results over the synthetic workload and the TPC-H benchmark for LinCQA and the other CQA systems.

Fixed inconsistency with varying relation sizes. To compare LinCQA with other CQA systems, we evaluate all systems using both the synthetic workload and the altered TPC-H benchmark with fixed inconsistency (inRatio $=$ 10%, bSize $=$ 2) as in previous work [23, 24, 41]. We vary the size of each relation (rSize $\in \{500K, 1M, 5M\}$) in the synthetic data (Figure 4) and we evaluate on TPC-H database instances of scale factors 1 and 10 (Figure 5). Both figures include the time for running the original query on the inconsistent database (which obtains the possible answers).

In the synthetic dataset, all three systems based on **FO**-rewriting techniques outperform CAvSAT, often by an order of magnitude. This observation suggests that if CERTAINTY(q) is **FO**-rewritable, a properly implemented rewriting is more efficient than the generic algorithm in practice, contradicting to what has been observed in [23, 24, 41]. Compared to ConQuer, LinCQA performs better or comparably on the first four queries. For queries q_5, q_6 and q_7 that are not in C_{forest}, ConQuer cannot produce their SQL-rewritings. In summary, LinCQA presents significant performance benefits on all seven queries.

In the TPC-H benchmark, the CQA systems are much closer in terms of performance. In this experiment, we observe that LinCQA almost always produces the fastest rewriting, and even when it is not, its performance is comparable to the other baselines. It is also worth observing that for most queries in the TPC-H benchmark, the overhead over running the SQL query directly is much smaller compared to the synthetic benchmark. Note that CAvSAT times out after 1 hour for queries q'_{10} and q'_{18} for both scale 1 and 10, while the systems based on **FO**-rewriting techniques terminate. We also remark that for Boolean queries, CAvSAT will terminate intermediately without processing the inconsistent part of the database using SAT solvers if the consistent part of the database already satisfies the query (e.g., q'_6, q'_{14}, q'_{17} in TPC-H). Overall, both LinCQA and ConQuer perform better than FastFO, since they both are better at exploiting the structure of the join tree.

Fixed relation size with varying inconsistency. We perform experiments to observe how different CQA systems react when the inconsistency of the instance changes. Using synthetic data, we first fix rSize $=$ 1M, bSize $=$ 2 and run all CQA systems on databases instances of varying inconsistent ratio from inRatio $=$ 10% to inRatio $=$ 100%. The results are depicted in Figure 6. We observe that the running time of CAvSAT increases when the inconsistent ratio of the database instance becomes larger. This happens because the SAT formula grows with larger inconsistency, and hence the SAT solver becomes slower. In contrast, the running time of all **FO**-rewriting techniques is relatively stable across database instances of different inconsistent ratios. More interestingly, the running time of LinCQA decreases when the inconsistency ratio becomes larger. This behavior occurs because of the early pruning on the relations at lower levels of the PPJT, which shrinks the size of candidate space being considered at higher levels of the PPJT and thus reduces the overall computation time. The overall performance trends of different systems are similar for all queries and thus we present only figures of q_1, q_3, q_5, q_7 here due to the space limit. Figure 8 in Appendix D of [52] presents the additional results.

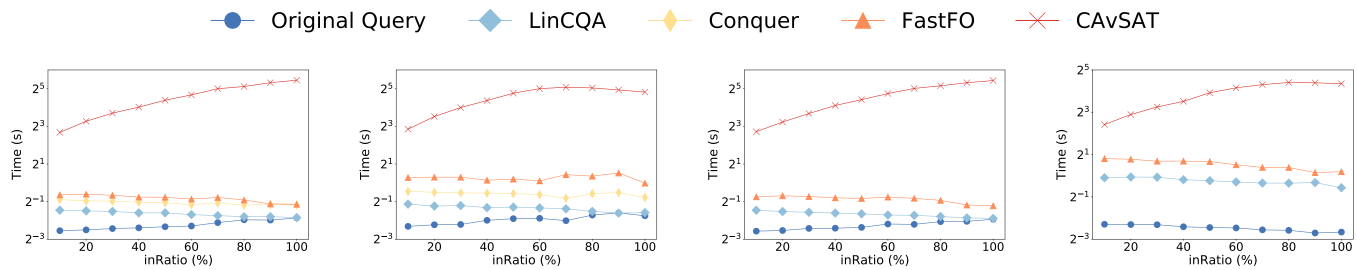
In our next experiment, we fix the the database instance size with rSize $=$ 1M and inconsistent ratio with inRatio $=$ 10%, running all CQA systems on databases of varying inconsistent block size bSize from 2 to 10. We observe that the performance of all CQA systems is not very sensitive to the change of inconsistent block sizes and thus we omit the results here. Figure 9 in Appendix D of [52] presents the full results.

6.4 Worst-Case Study

To demonstrate the robustness and efficiency of LinCQA, we generate synthetic *worst-case* inconsistent database instances for the 2-path query $Q_{2-\text{path}}$ and the 3-path query $Q_{3-\text{path}}$:

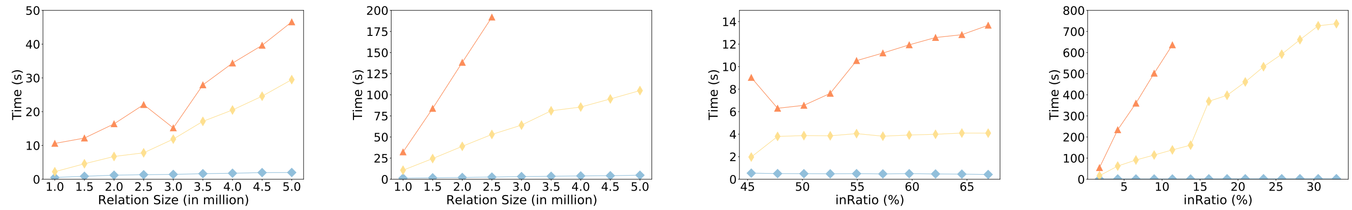
$$Q_{2-\text{path}}(x) \text{ :- } \mathsf{R}(\underline{x}, y), \mathsf{S}(\underline{y}, z).$$

$$Q_{3-\text{path}}(x) \text{ :- } \mathsf{R}(\underline{x}, y), \mathsf{S}(\underline{y}, z), \mathsf{T}(\underline{z}, w).$$



(a) q_1 **(b)** q_3 **(c)** q_5 **(d)** q_7

Figure 6: Performance of different systems on inconsistent database of varying inconsistent ratio



(a) $Q_{2-\text{path}}$ **(b)** $Q_{3-\text{path}}$ **(c)** $Q_{2-\text{path}}$ **(d)** $Q_{3-\text{path}}$

Figure 7: Performance comparison between different systems on varying relation sizes/inconsistent ratios

We compare the performance of LinCQA with ConQuer and FastFO on both queries. CAvSAT does not finish its execution on any instance within an hour, due to the long time it requires to solve the SAT formula. Thus, we do not report the time of CAvSAT.

To generate the input instances for $Q_{2-\text{path}}$, we generate the relations R and S with integer parameters a, b, c and N:

$$R = ([a] \times [b]) \cup \{(u,u) \mid ab + 1 \leq u \leq N, u \in \mathbb{Z}^+\},$$
$$S = ([b] \times [c]) \cup \{(u,u) \mid bc + 1 \leq u \leq N, u \in \mathbb{Z}^+\},$$

where $[n] = \{1, 2, \ldots, n\}$ and $[a] \times [b]$ denotes the cartesian product between $[a]$ and $[b]$. For $Q_{3-\text{path}}$, we additionally generate the relation T with an additional integer parameter d:

$$T = ([c] \times [d]) \cup \{(u,u) \mid cd + 1 \leq u \leq N, u \in \mathbb{Z}^+\}$$

Intuitively, for R, $[a] \times [b]$ is the set of inconsistent tuples and $\{(u,u) \mid ab+1 \leq u \leq N, u \in \mathbb{Z}^+\}$ is the set of consistent tuples. The values of a and b control both the number of inconsistent tuples (i.e. ab) and the size of inconsistent blocks (i.e. b). We note that $[a] \times [b]$ and $\{(u,u) \mid ab + 1 \leq u \leq N, u \in \mathbb{Z}^+\}$ are disjoint.

Fixed database inconsistency with varying size. We perform experiments to see how robust different CQA systems are when running queries on an instance of increasing size. For $Q_{2-\text{path}}$, we fix $b = c = 800$, and for each $k = 0, 1, \ldots, 8$, we construct a database instance with $a = 120 + 460k$ and $N = (1 + k/2) \cdot 10^6$. By construction, each database instance has inconsistent block size bSize $= b = c = 800$ in both relations R and S, and inRatio $= (ab + bc)/2N = 36.8\%$, with varying relation size rSize $= N$ ranging from 1M to 5M. Similarly for $Q_{3-\text{path}}$, we fix $b = c = d = 120$, and for each $k = 0, 1, \ldots, 8$, we construct a database instance with $a = 120 + 180k$ and $N = (1 + k/2) \cdot 10^6$. Here the constructed database instances have inRatio $= (ab + bc + cd)/3N = 1.44\%$. As shown in Figures 7a and 7b, the performance of LinCQA is much less sensitive to the changes of the relation sizes when compared to other CQA systems. We omit reporting the running time of FastFO for $Q_{3-\text{path}}$ on relatively larger database instances in Figure 7b for better contrast with ConQuer and LinCQA.

Fixed database sizes with varying inconsistency. Next, we experiment on instances of varying inconsistent ratio inRatio in which the joining mainly happens between inconsistent blocks of different relations. For $Q_{2-\text{path}}$, we fix $b = c = 800$ and $N = 10^6$ and generate a database instances for each $a = 100, 190, 280, \ldots, 1000$. All generated database instances have inconsistent block size bSize $= b = c = 800$ for both relations R and S, and the size of each relation rSize $= N = 10^6$ by construction. The database inconsistent ratio inRatio varies from 36% to 72%. For $Q_{3-\text{path}}$, we fix $b = c = d = 120$ and $N = 10^6$ and generate a database instances with $a = 200, 800, 1400, \ldots, 8000$. The inconsistent ratio of the generated database instances varies from 1.76% to 32.96%. Figures 7c and 7d show that LinCQA is the only system whose performance is agnostic to the change of the inconsistency ratio. The running time of FastFO and Conquer increases when the input database inconsistency increases. Similar to the experiments varying relation sizes, the running times of FastFO for $Q_{3-\text{path}}$ are omitted on relatively larger database instances in Figure 7d for better contrast with ConQuer and LinCQA.

7 CONCLUSION

In this paper, we introduce the notion of a pair-pruning join tree (PPJT) and show that if a BCQ has a PPJT, then CERTAINTY(q) is in **FO** and solvable in linear time in the size of the inconsistent database. We implement this idea in a system called LinCQA that produces a SQL query to compute the consistent answers of q. Our experiments show that LinCQA produces extremely efficient rewritings, is scalable, and robust on worst case instances.

We now identify some open questions. From a theoretical perspective, it remains open whether for any acyclic query with an acyclic attack graph there exists a linear time algorithm for CQA. A second open question is whether a PPJT can be constructed efficiently, without exploring all possible join trees.

REFERENCES

[1] Arvind Arasu and Raghav Kaushik. 2009. A grammar-based entity representation framework for data cleaning. In *SIGMOD Conference*. ACM, 233–244.

[2] Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. 1999. Consistent Query Answers in Inconsistent Databases. In *PODS*. ACM Press, 68–79.

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

[4] Pablo Barceló and Gaëlle Fontaine. 2015. On the Data Complexity of Consistent Query Answering over Graph Databases. In *ICDT (LIPIcs)*, Vol. 31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 380–397.

[5] Pablo Barceló and Gaëlle Fontaine. 2017. On the data complexity of consistent query answering over graph databases. *J. Comput. Syst. Sci.* 88 (2017), 164–194.

[6] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. 1983. On the Desirability of Acyclic Database Schemes. *J. ACM* 30, 3 (1983), 479–513. https://doi.org/10.1145/2402.322389

[7] Moria Bergman, Tova Milo, Slava Novgorodov, and Wang Chiew Tan. 2015. Query-Oriented Data Cleaning with Oracles. In *SIGMOD Conference*. ACM, 1199–1214.

[8] Leopoldo Bertossi, Solmaz Kolahi, and Laks VS Lakshmanan. 2013. Data cleaning and query answering with matching dependencies and matching functions. *Theory of Computing Systems* 52, 3 (2013), 441–482.

[9] Leopoldo E. Bertossi. 2019. Database Repairs and Consistent Query Answering: Origins and Further Developments. In *PODS*. ACM, 48–58.

[10] Leopoldo E. Bertossi, Solmaz Kolahi, and Laks V. S. Lakshmanan. 2013. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. *Theory Comput. Syst.* 52, 3 (2013), 441–482.

[11] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional Functional Dependencies for Data Cleaning. In *ICDE*. IEEE Computer Society, 746–755.

[12] Marco Calautti, Marco Console, and Andreas Pieris. 2021. Benchmarking Approximate Consistent Query Answering. In *PODS*. ACM, 233–246.

[13] Reynold Cheng, Jinchuan Chen, and Xike Xie. 2008. Cleaning uncertain data with quality guarantees. *Proc. VLDB Endow.* 1, 1 (2008), 722–735.

[14] Jan Chomicki and Jerzy Marcinkowski. 2005. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.* 197, 1-2 (2005), 90–121. https://doi.org/10.1016/j.ic.2004.04.007

[15] Jan Chomicki, Jerzy Marcinkowski, and Slawomir Staworko. 2004. Hippo: A System for Computing Consistent Answers to a Class of SQL Queries. In *EDBT (Lecture Notes in Computer Science)*, Vol. 2992. Springer, 841–844.

[16] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. 2016. Data Cleaning: Overview and Emerging Challenges. In *SIGMOD Conference*. ACM, 2201–2206.

[17] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *ICDE*. IEEE Computer Society, 458–469.

[18] Xu Chu, John Morcos, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. KATARA: A Data Cleaning System Powered by Knowledge Bases and Crowdsourcing. In *SIGMOD Conference*. ACM, 1247–1261.

[19] CloudLab 2018. https://www.cloudlab.us/.

[20] Jessica Davies and Fahiem Bacchus. 2011. Solving MAXSAT by Solving a Sequence of Simpler SAT Instances. In *CP (Lecture Notes in Computer Science)*, Vol. 6876. Springer, 225–239.

[21] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast Join Project Query Evaluation using Matrix Multiplication. In *SIGMOD Conference*. ACM, 1213–1223.

[22] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2021. Enumeration Algorithms for Conjunctive Queries with Projection. In *ICDT (LIPIcs)*, Vol. 186. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:17.

[23] Akhil Anand Dixit. 2021. *Answering Queries Over Inconsistent Databases Using SAT Solvers*. Ph.D. Dissertation. UC Santa Cruz.

[24] Akhil A. Dixit and Phokion G. Kolaitis. 2019. A SAT-Based System for Consistent Query Answering. In *SAT (Lecture Notes in Computer Science)*, Vol. 11628. Springer, 117–135.

[25] Akhil A. Dixit and Phokion G. Kolaitis. 2021. Consistent Answers of Aggregation Queries using SAT Solvers. *CoRR* abs/2103.03314 (2021).

[26] Amr Ebaid, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2013. NADEEF: A Generalized Data Cleaning System. *Proc. VLDB Endow.* 6, 12 (2013), 1218–1221.

[27] Ariel Fuxman, Elham Fazli, and Renée J Miller. 2005. Conquer: Efficient management of inconsistent databases. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 155–166.

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

[29] Congcong Ge, Yunjun Gao, Xiaoye Miao, Bin Yao, and Haobo Wang. 2021. A Hybrid Data Cleaning Framework Using Markov Logic Networks (Extended Abstract). In *ICDE*. IEEE, 2344–2345.

[30] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC Data-Cleaning Framework. *Proc. VLDB Endow.* 6, 9 (2013), 625–636.

[31] Gianluigi Greco, Sergio Greco, and Ester Zumpano. 2003. A Logical Framework for Querying and Repairing Inconsistent Databases. *IEEE Trans. Knowl. Data Eng.* 15, 6 (2003), 1389–1408.

[32] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek R. Narasayya, and Surajit Chaudhuri. 2018. Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (2018), 1165–1177.

[33] Lara A Kahale, Assem M Khamis, Batoul Diab, Yaping Chang, Luciane Cruz Lopes, Arnav Agarwal, Ling Li, Reem A Mustafa, Serge Koujanian, Reem Waziry, et al. 2020. Meta-Analyses Proved Inconsistent in How Missing Data Were Handled Across Their Included Primary Trials: A Methodological Survey. *Clinical Epidemiology* 12 (2020), 527–535.

[34] Bojan Karlas, Peng Li, Renzhi Wu, Nezihe Merve Gürel, Xu Chu, Wentao Wu, and Ce Zhang. 2020. Nearest Neighbor Classifiers over Incomplete Information: From Certain Answers to Certain Predictions. *Proc. VLDB Endow.* 14, 3 (2020), 255–267.

[35] Yannis Katsis, Alin Deutsch, Yannis Papakonstantinou, and Vasilis Vassalos. 2010. Inconsistency resolution in online databases. In *ICDE*. IEEE Computer Society, 1205–1208.

[36] Aziz Amezian El Khalfioui, Jonathan Joertz, Dorian Labeeuw, Gaëtan Staquet, and Jef Wijsen. 2020. Optimization of Answer Set Programs for Consistent Query Answering by Means of First-Order Rewriting. In *CIKM*. ACM, 25–34.

[37] Zuhair Khayyat, Ihab F. Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. BigDansing: A System for Big Data Cleansing. In *SIGMOD Conference*. ACM, 1215–1230.

[38] Henning Kohler and Sebastian Link. 2021. Possibilistic data cleaning. *IEEE Transactions on Knowledge and Data Engineering* (2021).

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

[40] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. 2013. Efficient Querying of Inconsistent Databases with Binary Integer Programming. *Proc. VLDB Endow.* 6, 6 (2013), 397–408.

[41] Phokion G. Kolaitis, Enela Pema, and Wang-Chiew Tan. 2013. Efficient Querying of Inconsistent Databases with Binary Integer Programming. *Proc. VLDB Endow.* 6, 6 (2013), 397–408.

[42] Paraschos Koutris, Xiating Ouyang, and Jef Wijsen. 2021. Consistent Query Answering for Primary Keys on Path Queries. In *PODS*. ACM, 215–232.

[43] Paraschos Koutris and Dan Suciu. 2014. A Dichotomy on the Complexity of Consistent Query Answering for Atoms with Simple Keys. In *ICDT*. OpenProceedings.org, 165–176.

[44] Paraschos Koutris and Jef Wijsen. 2015. The Data Complexity of Consistent Query Answering for Self-Join-Free Conjunctive Queries Under Primary Key Constraints. In *PODS*. ACM, 17–29.

[45] Paraschos Koutris and Jef Wijsen. 2017. Consistent Query Answering for Self-Join-Free Conjunctive Queries Under Primary Key Constraints. *ACM Trans. Database Syst.* 42, 2 (2017), 9:1–9:45.

[46] Paraschos Koutris and Jef Wijsen. 2018. Consistent Query Answering for Primary Keys and Conjunctive Queries with Negated Atoms. In *PODS*. ACM, 209–224.

[47] Paraschos Koutris and Jef Wijsen. 2019. Consistent Query Answering for Primary Keys in Logspace. In *ICDT (LIPIcs)*, Vol. 127. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:19.

[48] Paraschos Koutris and Jef Wijsen. 2020. First-Order Rewritability in Consistent Query Answering with Respect to Multiple Keys. In *PODS*. ACM, 113–129.

[49] Paraschos Koutris and Jef Wijsen. 2021. Consistent Query Answering for Primary Keys in Datalog. *Theory Comput. Syst.* 65, 1 (2021), 122–178.

[50] Sanjay Krishnan, Jiannan Wang, Eugene Wu, Michael J. Franklin, and Ken Goldberg. 2016. ActiveClean: Interactive Data Cleaning For Statistical Modeling. *Proc. VLDB Endow.* 9, 12 (2016), 948–959.

[51] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: A Study for Evaluating the Impact of Data Cleaning on ML Classification Tasks. In *ICDE*. IEEE, 13–24.

[52] LinCQA 2021. https://github.com/xiatingouyang/LinCQA.

[53] Andrei Lopatenko and Leopoldo E. Bertossi. 2007. Complexity of Consistent Query Answering in Databases Under Cardinality-Based and Incremental Repair Semantics. In *ICDT (Lecture Notes in Computer Science)*, Vol. 4353. Springer, 179–193.

[54] Marco Manna, Francesco Ricca, and Giorgio Terracina. 2015. Taming primary key violations to query large inconsistent data via ASP. *Theory Pract. Log. Program.* 15, 4-5 (2015), 696–710.

[55] Mónica Caniupán Marileo and Leopoldo E. Bertossi. 2005. Optimizing repair programs for consistent query answering. In *SCCC*. IEEE Computer Society, 3–12.

[56] Meikel Poess and Chris Floyd. 2000. New TPC benchmarks for decision support and web commerce. *ACM Sigmod Record* 29, 4 (2000), 64–71.

[57] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. 2015. Combining Quantitative and Logical Data Cleaning. *Proc. VLDB Endow.* 9, 4 (2015), 300–311.

[58] Erhard Rahm and Hong Hai Do. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.* 23, 4 (2000), 3–13.
[59] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201.
[60] El Kindi Rezig, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Ahmed R. Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable Dependency-driven Data Cleaning. *Proc. VLDB Endow.* 14, 11 (2021), 2546–2554.
[61] M. Andrea Rodríguez, Leopoldo E. Bertossi, and Mónica Caniupán Marileo. 2013. Consistent query answering under spatial semantic constraints. *Inf. Syst.* 38, 2 (2013), 244–263.
[62] Yongxin Tong, Caleb Chen Cao, Chen Jason Zhang, Yatao Li, and Lei Chen. 2014. CrowdCleaner: Data cleaning for multi-version data on the web via crowdsourcing. In *ICDE*. IEEE Computer Society, 1182–1185.
[63] Jef Wijsen. 2010. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In *PODS*. ACM, 179–190.
[64] Jef Wijsen. 2012. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.* 37, 2 (2012), 9:1–9:35.
[65] Jef Wijsen. 2019. Foundations of Query Answering on Inconsistent Databases. *SIGMOD Rec.* 48, 3 (2019), 6–16.
[66] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.

A CONQUESTO REWRITING FOR CERTAINTY(q)

The rewriting of CERTAINTY(q) produced by Conquesto for the query shown in Rule 2 comprises Rules (13–21), where Emp and Mgr are shorthands for tables Employee and Manager respectively.

$$\mathsf{yes}_1 \text{ :- } \mathsf{Emp}(X, Y, Y), \neg \mathsf{bb}_1(X). \tag{13}$$

$$\mathsf{bb}_1(X) \text{ :- } \mathsf{Emp}(X, Y, Z_1), Y \neq Z_1. \tag{14}$$

$$\mathsf{bb}_1(X) \text{ :- } \mathsf{Emp}(X, Y, Z_1), \neg \mathsf{yes}_2(Y, X). \tag{15}$$

$$\mathsf{yes}_2(Y, X) \text{ :- } \mathsf{Mgr}(Y, X, c), \neg \mathsf{bb}_2(Y, X). \tag{16}$$

$$\mathsf{bb}_2(Y, X) \text{ :- } \mathsf{Emp}(X, Y_1, Y_2), \mathsf{Mgr}(Y, Z_0, Z_1), Z_1 \neq 2020. \tag{17}$$

$$\mathsf{bb}_2(Y, X) \text{ :- } \mathsf{Emp}(X, Y_1, Y_2), \mathsf{Mgr}(Y, Z_0, Z_1), Z_0 \neq X. \tag{18}$$

$$\mathsf{bb}_2(Y, X) \text{ :- } \mathsf{Emp}(X, Y_1, Y_2), \mathsf{Mgr}(Y, Z_0, Z_1), \neg \mathsf{yes}_3(Y, X). \tag{19}$$

$$\mathsf{yes}_3(Y, X) \text{ :- } \mathsf{Mgr}(Y, X), \neg \mathsf{bb}_3(Y, X). \tag{20}$$

$$\mathsf{bb}_3(Y, X) \text{ :- } \mathsf{Emp}(X, Y_1, Y_2), \mathsf{Contact}(Y, Z_0), Z_0 \neq X. \tag{21}$$

B EFFICIENT CONSTRUCTION OF PPJT

Proof of Proposition 4.5. Let q be a self-join-free Boolean conjunctive query with an acyclic attack graph. Let τ be a join tree for q (thus q is α-acyclic). Assume the following hypothesis:

> *Hypothesis of Disjoint Keys:* for all atoms $G, H \in q$, $G \neq H$, we have that $\mathsf{key}(G)$ and $\mathsf{key}(H)$ are not comparable by set inclusion.

We show, by induction on $|q|$, that CERTAINTY(q) is in linear time. For the basis of the induction, $|q| = \emptyset$, it is trivial that CERTAINTY(q) is in linear time. For the induction step, let $|q| \geq 1$. Let F be an unattacked atom of q. Let (τ, F) be a join tree of q with root F. Let $F_1, \ldots, F_n$ be the children of F in (τ, F) with subtrees $\tau_1, \tau_2, \ldots, \tau_n$.

Let $i \in \{1, \ldots, n\}$. We claim that q_{τ_i} has an acyclic attack graph. Assume for the sake of contradiction that the attack graph of q_{τ_i} has a cycle, and therefore has a cycle of size 2. Then there are $G, H \in q_{\tau_i}$ such that $G \overset{q_{\tau_i}}{\rightsquigarrow} H \overset{q_{\tau_i}}{\rightsquigarrow} G$. From the *Hypothesis of Disjoint Keys*, it follows $G \overset{q}{\rightsquigarrow} H \overset{q}{\rightsquigarrow} G$, contradicting the acyclicity of q's attack graph.

We claim the following:

$$\text{for every } G \in q_{\tau_i}, \mathbf{vars}(G) \cap \mathbf{vars}(F) \subseteq \mathsf{key}(G). \tag{22}$$

This claim follows from the *Hypothesis of Disjoint Keys* and the assumption that F is unattacked in q's attack graph.

It suffices to show that there is an atom $F_i' \in q_{\tau_i}$ (possibly $F_i' = F_i$) such that

(1) F_i' is unattacked in the attack graph of q_{τ_i}; and
(2) $\mathsf{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \mathsf{key}(F_i')$.

We distinguish two cases:

Case that F_i is unattacked in the attack graph of q_{τ_i}. Then we can pick $F_i' := F_i$.

Case that F_i is attacked in the attack graph of q_{τ_i}. We can assume an atom G such that $G \overset{q_{\tau_i}}{\rightsquigarrow} F_i$. Since $G \overset{q}{\not\rightsquigarrow} F$, by the *Hypothesis of Disjoint Keys*, it must be that $\mathbf{vars}(F_i) \cap \mathbf{vars}(F) \subseteq \mathsf{key}(G)$. Then from $\mathsf{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \mathbf{vars}(F_i)$, it follows $\mathsf{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \mathsf{key}(G)$. If G is unattacked in the attack graph of $q_{\tau_i i}$, then we can pick $F_i' := G$. Otherwise we repeat the same reasoning (with G playing the role previously played by F_i). This repetition cannot go on forever since the attack graph of q_{τ_i} is acyclic.

The proof is now complete. □

We remark that CERTAINTY(q) remains solvable in linear time for certain acyclic self-join-free CQ that is **FO**-rewritable but does not have a PPJT. It uses techniques from efficient query result enumeration algorithms [21, 22].

Proposition B.1. *Let $q()$:- $R(\underline{c}, x)$, $S(\underline{c}, y)$, $T(\underline{x, y})$. Then there exists a linear time algorithm for* CERTAINTY(q).

C MISSING PROOFS

Proof of Proposition 4.2. Suppose, for the sake of contradiction, that the attack graph is not acyclic. Then there must be two atoms R, S such that $R \overset{q}{\rightsquigarrow} S$ and $S \overset{q}{\rightsquigarrow} R$ by Lemma 3.6 of [45]. Let (τ, T) be the PPJT for q, and let (τ', U) be the smallest subtree of (τ, T) that contains both S and R (it may be that $U = R$ or $U = S$). The first observation is that in the subquery $q_{\tau'}$ it also holds that R attacks S and vice versa. Moreover, since (τ', U) is the smallest possible subtree, the unique path that connects R and S must go through the root U. We now distinguish two cases:

- If $U = R$, then S attacks the root of the subtree q', a contradiction to the PPJT definition.
- If $U \neq R$, then the unique path from R to S goes through U. Since R must attack every atom in that path by Lemma 4.9 of [64], it must also attack U, a contradiction as well.

This completes the proof of the proposition. □

Proof of Proposition 4.4. Let q be a query in $\mathcal{C}_{\text{forest}}$ and let G be the join graph of q as in Definition 6 of [28]. In particular, (i) the vertices of G are the atoms of q, and (ii) there is an arc from R to S if $R \neq S$ and there is some variable $w \in \mathsf{vars}(S)$ such that $w \in \mathsf{vars}(R) \setminus \mathsf{key}(R)$. By the definition of $\mathcal{C}_{\text{forest}}$, G is a directed forest with connected components $\tau_1, \tau_2, \ldots, \tau_n$, where the root atoms are $R_1, R_2, \ldots, R_n$ respectively.

Claim 1: each τ_i is a join tree. Suppose for the sake of contradiction that τ_i is not a join tree. Then there exists a variable w and two non-adjacent atoms R and S in τ_i such that $w \in \mathsf{vars}(R)$, $w \in \mathsf{vars}(S)$, and for any atom T_i in the unique path $R - T_1 - \cdots - T_k - S$, we have $w \notin \mathsf{vars}(T_i)$. We must have $w \in \mathsf{key}(R)$ and $w \in \mathsf{key}(S)$, or otherwise there would be an arc between R and S, a contradiction. From the property $\mathcal{C}_{\text{forest}}$, it also holds that no atom in the tree receives arcs from two different nodes. Hence, there is either an arc (T_1, R) or (T_k, S). Without loss of generality, assume there is an arc from T_1 to R. Then, since all nonkey-to-key joins are full, $w \in \mathsf{vars}(T_1)$, a contradiction to our assumption.

Claim 2: the forest $\tau_1 \cup \cdots \cup \tau_n$ can be extended to a join tree τ of q. To show this, we will show that $\tau_1 \cup \cdots \cup \tau_n$ corresponds to a partial join tree as constructed by the GYO ear-removal algorithm. Indeed, suppose that atom T is a child of atom T' in τ_i. Then, T was an ear while constructing τ_i for q_{τ_i}, with T' as its witness. Recall that this means that if a variable x is not exclusive in T, then $x \in T'$. We will show that this is a valid ear removal step for q as well. Indeed, consider an exclusive variable x in T for q_{τ_i} that does not remain exclusive in q. Then, x occurs in some other tree τ_j. We will now use the fact that, by Lemma 2 of [28], if τ_i and τ_j share a variable x, then x can only appear in the root atoms R_i and R_j. This implies that x appears at the root of τ_i, and hence at T' as well, a contradiction.

We finally claim that (τ, R_1) is a PPJT for q. By construction, τ is a join tree. Next, consider any two adjacent atoms R and S in τ such that R is a parent of S in (τ, R_1). Let p be any connected subquery of q containing R and S. It suffices to show that S does not attack R in p. If R and S are both root nodes of some τ_i and τ_j, we must have $\mathsf{vars}(R) \cap \mathsf{vars}(S) \subseteq \mathsf{key}(S) \subseteq S^{+,p}$, and thus S does not attack R in p. If R and S are in the same join tree τ_i, since there is no arc from S to R, all nonkeys of S are not present in R, and thus $\mathsf{vars}(R) \cap \mathsf{vars}(S) = \mathsf{vars}(R) \cap \mathsf{key}(S) \subseteq \mathsf{key}(S) \subseteq S^{+,p}$. Hence, there is no attack from S to R as well. □

Proof of Lemma 4.9. We consider two directions.

2 ⟹ 1 Here we must have $S_{\mathsf{gkey}}(q_S, \mathbf{db}) \neq \emptyset$ for all child node S of R in τ. Let r be any repair of $\mathbf{db}$ and let $R(\vec{c}, \vec{d}) \in r$. Since 2 holds, for every child node S of R, there exists a fact $S(\vec{s}, \vec{d}) \in r$ with $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$ and a valuation μ_S such that $R(\mu_S(\vec{x}), \mu_S(\vec{y})) = R(\vec{c}, \vec{d})$ and $S(\mu_S(\vec{u}), \mu_S(\vec{v})) = S(\vec{s}, \vec{t})$. Since r is a repair of $\mathbf{db}$ and $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, there exists a valuation ξ_S such that $\xi_S(q_S) \subseteq r$ with $\xi_S(\vec{u}) = \vec{s} = \mu_S(\vec{u})$. Note that all μ_S agree on the valuation of $\mathsf{vars}(\vec{x}) \cup \mathsf{vars}(\vec{y})$, let μ be the valuation such that $R(\mu(\vec{x}), \mu(\vec{y})) = R(\mu(\vec{x_S}), \mu(\vec{y_S}))$ for all child node S of R.
Next we show that for all q_S and any variable $z \in \mathsf{vars}(R) \cap \mathsf{vars}(q_S)$, $\mu(z) = \xi_S(z)$. Since r is consistent, we must have $S(\mu_S(\vec{u}), \mu_S(\vec{v})) = S(\xi_S(\vec{u_S}), \xi_S(\vec{v_S})) \in r$. Since T is a join tree, we must have $z \in \mathsf{vars}(R) \cap \mathsf{vars}(S)$, and it follows that $\xi_S(z) = \mu_S(z) = \mu(z)$, as desired.
Then, the following valuation

$$\mu(z) = \begin{cases} \mu(z) & z \in \mathsf{vars}(R) \\ \xi_i(z) & z \in \mathsf{vars}(q_S) \setminus \mathsf{vars}(R) \\ d & z = d \text{ is constant} \end{cases}$$

is well-defined and satisfies that $\mu(q_{\vec{x} \to \vec{c}}) \subseteq r$, as desired.

1 ⟹ 2 By contraposition. Assume that 2 does not hold, and we show that there exists a repair r of $\mathbf{db}$ that does not satisfy $q_{[\vec{x} \to \vec{c}]}$.
If 2a does not hold, then there exists some fact $f = R(\vec{c}, \vec{d})$ that does not satisfy $R(\underline{\vec{x}}, \vec{y})$, and any repair containing the fact f does not satisfy $q_{[\vec{x} \to \vec{c}]}$. Next we assume that 2a holds but 2b does not.
If $S_{\mathsf{gkey}}(q_S, \mathbf{db}) = \emptyset$ for some child node S of R in τ, then by monotonicity of conjunctive queries and Lemma 4.7, $\mathbf{db}$ is a "no"-instance for $\mathsf{CERTAINTY}(q_S)$, $\mathsf{CERTAINTY}(q)$ and thus $\mathsf{CERTAINTY}(q_{[\vec{x} \to \vec{c}]})$. In what follows we assume that $S_{\mathsf{gkey}}(q_S, \mathbf{db}) \neq \emptyset$ for all child node S of R.
Since 2b does not hold, there exist a fact $R(\vec{c}, \vec{d})$ and some child node S of R in τ and query q_S such that for any block $S(\vec{s}, *)$ with $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, there exists a fact $S(\vec{s}, \vec{t})$ that does not join with $R(\vec{c}, \vec{d})$.
Let $\mathbf{db}' = \mathbf{db} \setminus R \setminus \{S(\vec{s}, *) \mid \vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})\}$. We show that $\mathbf{db}'$ is a "no"-instance for $\mathsf{CERTAINTY}(q_S)$. Indeed, suppose otherwise that $\mathbf{db}'$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_S)$, then there exists some $\vec{s}$ such that $\mathbf{db}'$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{S,[\vec{u} \to \vec{s}]})$. Note that by construction, $\vec{s} \notin S_{\mathsf{gkey}}(q_S, \mathbf{db})$. Since $\mathbf{db}' \subseteq \mathbf{db}$, we have $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{S,[\vec{u} \to \vec{s}]})$, implying that $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, a contradiction.
Consider the following repair r of $\mathbf{db}$ that contains
- $R(\vec{c}, \vec{d})$ and an arbitrary fact from all blocks $R(\vec{b}, *)$ with $\vec{b} \neq \vec{c}$;
- for each $\vec{s} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, any fact $S(\vec{s}, \vec{t})$ that does not join with $R(\vec{c}, \vec{d})$; and
- any falsifying repair r' of $\mathbf{db}'$ for $\mathsf{CERTAINTY}(q_S)$.

We show that r does not satisfy $q_{[\vec{x} \to \vec{c}]}$. Suppose for contradiction that there exists a valuation μ with $\mu(q_{[\vec{x} \to \vec{c}]}) \subseteq r$ and $R(\mu(\vec{x}), \mu(\vec{y})) = R(\vec{c}, \vec{d}) \in r$. Let $S(\vec{s^*}, \vec{t^*}) = S(\mu(\vec{u}), \mu(\vec{v}))$, then we must have $\vec{s^*} \notin S_{\mathsf{gkey}}(q_S, \mathbf{db})$, since otherwise we would have $S(\vec{s^*}, \vec{t^*})$ joining with $R(\vec{c}, \vec{d})$ where we have $\vec{s^*} \in S_{\mathsf{gkey}}(q_S, \mathbf{db})$, a contradiction to the construction of r. Since $\vec{s^*} \notin S_{\mathsf{gkey}}(q_S, \mathbf{db})$, we would then have $\mu(q_S) \subseteq r'$, a contradiction to that r' is a falsifying repair of $\mathbf{db}'$ for $\mathsf{CERTAINTY}(q_S)$. Finally, if 2b holds, then all facts $S(\vec{s}, \vec{t})$ must agree on $\vec{w}$ since they all join with the same fact $R(\vec{c}, \vec{d})$.

The proof is now complete. □

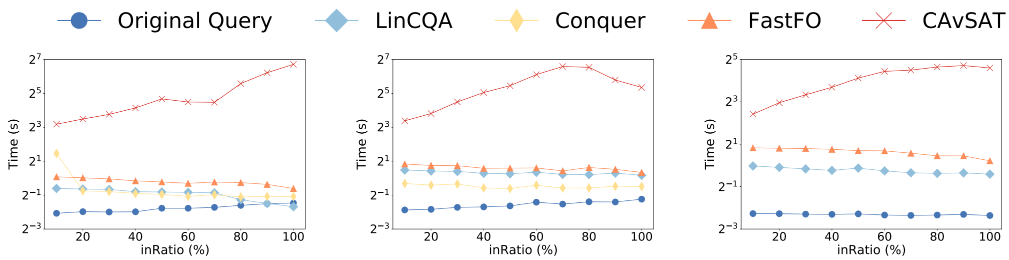
Proof of Lemma 4.13. Consider both directions. First we assume that $\vec{c}$ is a consistent answer of q on $\mathbf{db}$. Let r be any repair of $\mathbf{db}$. Then there exists a valuation μ with $\mu(q) \subseteq r$ with $\mu(\vec{u}) = \vec{c}$, and hence $\mu(q_{[\vec{u} \to \vec{c}]}) = \mu(q) \subseteq r$. That is, $q_{[\vec{u} \to \vec{c}]}(r)$ is true. Hence $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u} \to \vec{c}]})$. For the other direction, we assume that $\mathbf{db}$ is "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u} \to \vec{c}]})$. Let r be any repair of $\mathbf{db}$. Then there is a valuation μ with $\mu(q_{[\vec{u} \to \vec{c}]}) \subseteq r$. Let θ be the valuation with $\theta(\vec{u}) = \vec{c}$. Consider the valuation

$$\mu^+(z) = \begin{cases} \theta(z) & z \in \mathsf{vars}(\vec{u}) \\ \mu(z) & \text{otherwise,} \end{cases}$$

and we have $\mu^+(q) = \mu(q_{[\vec{u} \to \vec{c}]}) \subseteq r$ with $\mu^+(\vec{u}) = \theta(\vec{u}) = \vec{c}$, as desired. □

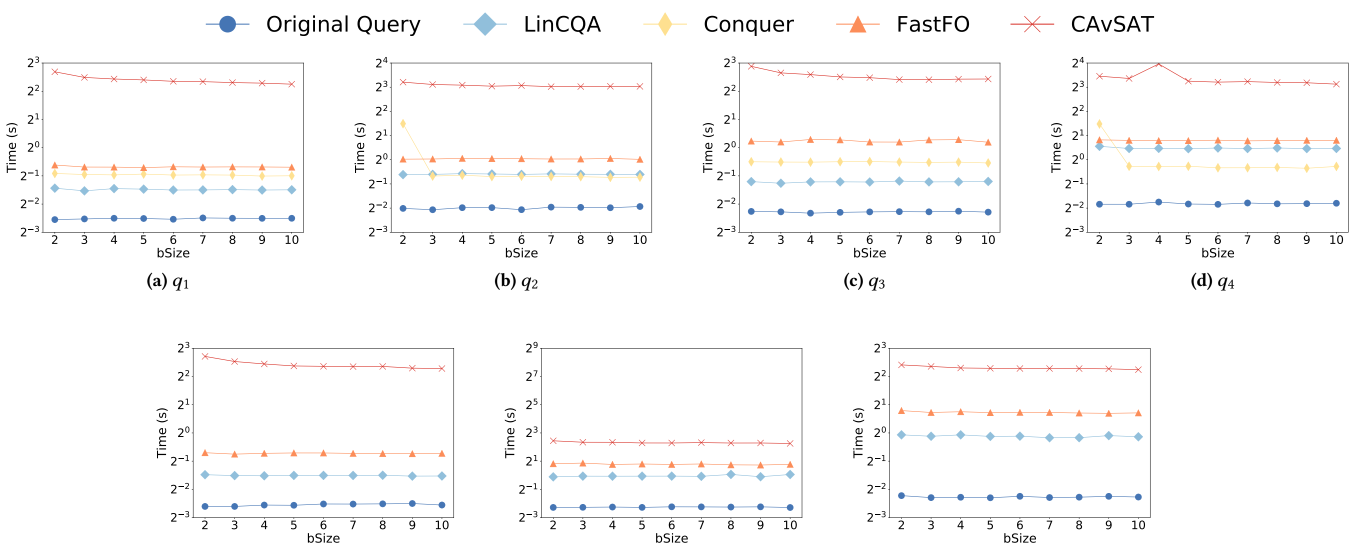
D ADDITIONAL FIGURES FOR EXPERIMENTS

Figure 8 presents the performance of queries q_2, q_4 and q_6 on varying inconsistency ratios, supplementing Figure 6. Figure 9 summarizes the performance of all seven synthetic queries on varying block sizes.



(a) q_2 **(b)** q_4 **(c)** q_6

Figure 8: Performance of different systems on inconsistent databsae of varying inconsistency ratio



(a) q_1 **(b)** q_2 **(c)** q_3 **(d)** q_4

(e) q_5 **(f)** q_6 **(g)** q_7

Figure 9: Performance of different systems on inconsistent database of varying block size