LinCQA: Faster Consistent Query Answering with Linear Time Guarantees

Anonymous Author(s)

ABSTRACT

Most data analytical pipelines often encounter the problem of querying inconsistent data that violate pre-determined integrity constraints. Data cleaning is an extensively studied paradigm that singles out a consistent repair of the inconsistent data. Consistent query answering (CQA) is an alternative approach to data cleaning that asks for all tuples guaranteed to be returned by a given query on all (in most cases, exponentially many) repairs of the inconsistent data. This paper identifies a class of acyclic select-project-join (SPJ) queries for which CQA can be solved via SQL rewriting with a linear time guarantee. Our rewriting method can be viewed as a generalization of Yannakakis's algorithm for acyclic joins to the inconsistent setting. We present LinCQA, a system that can output rewriting in both SQL and non-recursive Datalog rules for every query in this class. We show that LinCQA often outperforms the existing CQA systems on both synthetic and real-world workloads, and in some cases, by orders of magnitude.

CCS CONCEPTS

• **Information systems → Relational database query languages**.

KEYWORDS

conjunctive queries, consistent query answering

ACM Reference Format:

Anonymous Author(s). 2018. LinCQA: Faster Consistent Query Answering with Linear Time Guarantees. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A database is *inconsistent* if it violates one or more integrity constraints that are supposed to be satisfied. Database 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 [2, 30–32, 60], using techniques such as knowledge bases and machine learning [8, 11, 17, 18, 27, 33, 51, 52, 59, 62]. Data cleaning has also been studied under different contexts [9, 12, 14, 38, 39, 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 [18, 19, 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 [3] that has recently received more attention [10, 65]. 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 a viable 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. Third, CQA can be used to design more efficient data cleaning algorithms [35].

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

From a theoretical perspective, CQA for select-project-join (SPJ) queries is computationally hard as it naturally requires inspecting potentially exponentially many repairs. However, for some SPJ queries the consistent answers can be computed in polynomial time, and for some other SPJ 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 [40, 44, 45, 48, 50], it was proven that given any self-join-free SPJ query, the problem is either **FO**-rewritable, polynomial-time solvable but not **FO**-rewritable, or **coNP**-complete [46].

From a systems standpoint, most CQA systems fall into two categories (summarized in Table 1): (1) systems that can compute the consistent answers of join queries with arbitrary denial constraints but require solvers for computationally hard problems (e.g., EQUIP [41] relies on Integer Programming solvers, and CAvSAT [25,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Table 1: A summary of systems for consistent query answering

System	Target class of queries	Intermediate output	Backend
EQUIP [41]	all SPJ Queries	Big Integer Program (BIP)	DBMS & BIP solver
CAvSAT [25, 26]	all SPJ Queries	SAT formula	DBMS & SAT solver
Conquer [28]	C_{forest}	SQL rewriting	DBMS
Conquesto [37]	self-join-free SPJ Queries in **FO**	Datalog rewriting	Datalog engine
LinCQA (this paper)	PPJT	SQL rewriting / Datalog rewriting	DBMS or Datalog engine

26] requires SAT solvers) and (2) 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 [28] identified a class of **FO**-rewritable queries called C_{forest} and implemented their rewriting in ConQuer, which outputs a single SQL query. Conquesto [37] is the most recent system targeting **FO**-rewritable join queries by producing the rewriting in Datalog.

We 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 many join queries seen in practice, it excludes queries that involve *(i)* joining with only part of a composite primary key, often appearing in snowflake schemas, and *(ii)* equality joins that involve both primary-key and non-primary-key attributes, which commonly occur in settings such as entity matching and cross-comparison scenarios. Conquesto, on the other hand, implements the generic **FO**-rewriting algorithm without strong performance guarantees. Moreover, neither ConQuer nor Conquesto have theoretical guarantees on the running time of their produced rewritings.

Contributions. To address the above observed issues, we make the following contributions:

Theory & Algorithms. We identify a subclass of acyclic join queries that captures a wide range of queries commonly seen in practice for which we can produce **FO**-rewritings with a *linear running time guarantee* (Section 4). This class subsumes all acyclic queries in C_{forest}. For consistent databases, Yannakakis's algorithm [7] evaluates acyclic join queries in linear time in the size of the database. Our algorithm 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 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 (PPJT)*. Our algorithm follows the pair-pruning join tree to compute the consistent answers and degenerates to Yannakakis's algorithm if the database has no inconsistencies.

Implementation. We implement our algorithm in LinCQA (Linear Consistent Query Answering), a system prototype that produces an efficient and optimized rewriting in both SQL and non-recursive Datalog rules with negation (Section 5).

Evaluation. We perform an extensive experimental evaluation comparing LinCQA to the other state-of-the-art CQA systems. Our findings show that *(i)* a properly implemented rewriting can significantly outperform a generic CQA system (e.g., CAvSAT); *(ii)* LinCQA achieves the best overall 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. LinCQA often outperforms other CQA systems, in several cases by orders-of-magnitude on both synthetic and real-world workloads. We also demonstrate that CQA can be an effective approach even for real-world datasets of very large scale (~400GB), which, to the best of our knowledge, have not been tested before.

2 RELATED WORK

Inconsistency in databases have been studied in different contexts [5, 6, 13, 15, 34, 36, 53, 61]. The notion of Consistent Query Answering (CQA) was introduced in the seminal work by Arenas, Bertossi, and Chomicki [3]. After twenty years, their contribution was acknowledged in a *Gems of PODS session* [10]. 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 [29], and was further pursued in [40, 44–46, 48, 50], 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 [43]. The complexity of CERTAINTY(q) for self-join-free Boolean conjunctive queries with negated atoms was studied in [47]. For self-join-free Boolean conjunctive queries w.r.t. multiple keys, it remains decidable whether or not CERTAINTY(q) is in **FO** [49].

Several systems for CQA that are used for comparison in our study have already been described in the introduction: ConQuer [28], Conquesto [37], CAvSAT [25, 26], and EQUIP [41]. Most early systems for CQA used efficient solvers for Disjunctive Logic Programming and Answer Set Programming (ASP) [4, 16, 32, 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 models over inconsistent training data [35].

3 BACKGROUND

In this section, we define some notations used in our paper. We use the example **Company** database shown in Table 2 to illustrate our constructs, where the primary key attribute of each table is highlighted in bold.

Database instances, blocks, and repairs. A *database schema* is a finite set of table names. Each table name is associated with a finite sequence of *attributes*. Some of these attributes are declared

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

as *primary-key attributes*, forming together the *primary key*. A *database instance* **db** associates to each table name a finite set of *tuples*, called a *relation*. A relation is *consistent* if it does not contain two distinct tuples that agree on all primary-key attributes. A *block* of a relation is a maximal set of tuples that agree on all primary-key attributes. Thus, a relation is consistent if and only if it has no block with two or more tuples. A *repair* of a (possibly inconsistent) relation is obtained by selecting exactly one tuple from each block. Clearly, a relation with n blocks of size 2 each has 2^n repairs, an exponential number. A database instance **db** is consistent if all relations in it are consistent. A repair of a (possibly inconsistent) database instance is obtained by selecting one repair for each relation. In the technical treatment, it will be convenient to view a database instance **db** as a set of facts: if the relation associated with table name R contains tuple $\vec{t}$, then we say that $R(\vec{t})$ is a fact of **db**.

Example 3.1. The **Company** database is inconsistent with respect to primary key constraints. For example, in the Employee table there are 3 distinct tuples sharing the same primary key employee_id 0011. The blocks in the **Company** database are already highlighted using dashed lines. An example repair of the **Company** database can be obtained by choosing exactly one tuple from each block, and there are in total $96 = 3 \times 2^5$ distinct repairs. □

Atoms and key-equal facts. Let $\vec{x}$ be a sequence of variables and constants. We write $\mathsf{vars}(\vec{x})$ for the set of variables that appear in $\vec{x}$. Let R be a table name, then $R(\underline{\vec{x}}, \vec{y})$ is an atom, where the primary key $\vec{x}$ of R is underlined and we denote $\mathsf{key}(R) = \vec{x}$. Whenever a database instance **db** is understood, we write $R(\underline{\vec{c}}, *)$ for the block containing all tuples with primary-key value $\vec{c}$ in relation R.

Example 3.2. For the **Company** database, we can have atoms $\mathsf{Employee}(\underline{x}, y, y)$, $\mathsf{Manager}(\underline{u}, v, \mathtt{2020})$, and $\mathsf{Contact}(\underline{\mathsf{LA}}, \mathtt{2020})$. The block $\mathsf{Manager}(\underline{\mathsf{Boston}}, *)$ contains two facts: Manager(<u>Boston</u>, 0011, 2020) and Manager(<u>Boston</u>, 0011, 2021). □

Conjunctive Queries. For select-project-join (SPJ) queries, we will also use the term *conjunctive queries (CQ)*. Each CQ q can be represented as a succinct 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, $\mathsf{body}(q)$.

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*. If the body of a CQ of the form (1) can be partitioned into two nonempty parts that have no variable in common, then we say that the query is *disconnected*; otherwise it is *connected*.

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 $1 \leq i \leq \ell$.

Example 3.3. Consider the query over the **Company** database that *returns the id's of all employees who work in some office city with a manager who started in year 2020*. It can be expressed by the following SQL query:

```
SELECT E.employee_id FROM Employee E, Manager M
WHERE E.office_city=M.office_city AND M.start_year=2020
```

and the following CQ:

$$q(x) \text{ :- } \mathsf{Employee}(\underline{x}, y, z), \mathsf{Manager}(\underline{y}, w, \mathtt{2020}).$$

The following CQ q' is a BCQ, since it merely asks whether 0011 is such an employee_id satisfying the conditions in q:

$$q'() \text{ :- } \mathsf{Employee}(\underline{\mathtt{0011}}, y, z), \mathsf{Manager}(\underline{y}, w, \mathtt{2020}).$$

It is easy to see that q' is equivalent to $q_{[x \to \mathtt{0011}]}$. □

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 [1]. 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 CQ q, given an input database instance **db**, the problem $\mathsf{CERTAINTY}(q)$ asks for the intersection of all query outputs $q(\mathbf{r})$ for all repairs $\mathbf{r}$ of **db**. If q is Boolean, the problem $\mathsf{CERTAINTY}(q)$ then asks whether q is satisfied by every repair of the input database instance **db**.

The problem $\mathsf{CERTAINTY}(q)$ has a first-order rewriting (**FO**-rewriting) if there is another first-order query q' (which, in most cases, uses the difference operator and hence is not a SPJ query) such that evaluating q' on the input database **db** would return the answers of $\mathsf{CERTAINTY}(q)$. In other words, executing q' directly on the inconsistent database *simulates* computing the original query q over all possible repairs.

Example 3.4. Recall that in Example 3.3, the query q returns $\{0011, 0022, 0034\}$ on the inconsistent database **Company**. For $\mathsf{CERTAINTY}(q)$ however, the only output is 0022: for any repair that contains the tuples Employee($\underline{0011}$, Boston, Boston) and Manager($\underline{\text{Boston}}$, 0011, 2021), neither 0011 nor 0034 would be returned by q; and in any repair, 0022 is returned by q with the following crucial observation: *Regardless of which tuple in Employee*($\underline{0022}$, $*$) *the repair contains, both offices are present in Manager table and the managers in both offices in Chicago or New York started in 2020.*

This observation is sufficient to solve $\mathsf{CERTAINTY}(q)$ and can be expressed by the following SQL query, called an **FO**-rewriting of $\mathsf{CERTAINTY}(q)$.

```
SELECT E.employee_id FROM Employee E EXCEPT
SELECT E.employee_id FROM Employee E
WHERE E.office_city NOT IN (
    SELECT M.office_city FROM Manager EXCEPT
    SELECT M.office_city FROM Manager
    WHERE M.start_year <> 2020)
```
□

4 A LINEAR-TIME REWRITING

Before presenting our linear time rewriting for $\mathsf{CERTAINTY}(q)$, we first provide a motivating example. Consider the following query on the **Company** database shown in Table 2:

> *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 by the following CQ:

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

To the best of our knowledge, the most efficient running time for $\mathsf{CERTAINTY}(q^{\text{ex}})$ guaranteed by existing systems is quadratic in the input database size, denoted N. The problem $\mathsf{CERTAINTY}(q^{\text{ex}})$ admits an **FO**-rewriting by the classification theorem in [43]. However, the non-recursive Datalog rewriting of $\mathsf{CERTAINTY}(q^{\text{ex}})$ produced by Conquesto contains cartesian products between two tables, which means that it runs in $\Omega(N^2)$ time in the worst case. Also, since q^{ex} is not in C_{forest}, ConQuer cannot produce an **FO**-rewriting. Both EQUIP and CAvSAT solve the problem through Integer Programming or SAT solvers, which can take exponential time. 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 $\mathsf{CERTAINTY}(q^{\text{ex}})$ in $O(N)$ time, while existing CQA systems will run in more than linear 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 $\mathsf{CERTAINTY}(q)$ (Theorem 4.6). Finally, we extend our result to all acyclic self-join-free CQs in Section 4.3 (Theorem 4.12) .

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*[1] 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($\underline{x}, y, y$) in q^{ex}.

For self-join-free BCQs, the existence of an **FO**-rewriting for q depends on the notion of a *reifiable* atom, which we define next.

Definition 1 (Reifiable Atom). *Let q be a self-join-free BCQ. Let $R(\underline{\vec{x}}, \vec{y})$ be an atom in q. We say that R is* reifiable *if for every instance* **db** *for* $\mathsf{CERTAINTY}(q)$*, the following statements are equivalent:*

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

Informally, this means that the primary key of a reifiable atom can be replaced by constants (which depend on **db**) to create subqueries that can be handled independently.

Example 4.1. The atom Employee($\underline{x}, z, z$) is reifiable in q^{ex}. The **Company** database is a "yes"-instance for $\mathsf{CERTAINTY}(q^{\text{ex}})$. Therefore, there must exist some employee_id c for which the **Company** database is a "yes"-instance for $\mathsf{CERTAINTY}(q^{\text{ex}}_{[x\to c]})$. Indeed for $c = 2022$, 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 $q^{\text{ex}}_{[x\to 0022]}$ will return `True` for all repairs of database **Company**. □

Checking whether an atom is reifiable in a self-join-free BCQ can be efficiently done via the syntactic notion of an *attack* from atom R to atom S, defined in [43]. In particular, an atom is reifiable if and only if it is not attacked by any other atom. Additionally, there exists an **FO**-rewriting for q if and only if the graph consisting of all attacks in q, called an *attack graph*, is acyclic.

We can now apply the notion of a reifiable atom to define a *pair-pruning join tree.*

Definition 2 (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 reifiable in $q_{\tau'}$.*

Example 4.2. For the join tree τ in Figure 1, the rooted tree (τ, Employee) is a PPJT for q^{ex}. The atom Employee($\underline{x}, z, z$) is reifiable in q. For the child subtree (τ_M, Manager) of (τ, Employee), the

[1] Throughout this paper, whenever we say that a CQ is acyclic, we mean acyclicity as defined in [7], a notion that today is also known a α-acyclicity, to distinguish it from other notions of acyclicity.

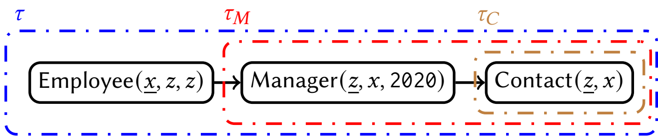


Figure 1: A pair-pruning join tree (PPJT) of the query q^{ex}.

atom Manager($\underline{z}$, x, 2020) is also reifiable in the following subquery

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

Finally, for the subtree (τ_C, Contact), the atom Contact($\underline{z}$, x) is also reifiable in the corresponding subquery $q^{\text{ex}}_{\tau_C}()$:- Contact($\underline{z}$, x). Hence (τ, Employee) is a PPJT of q^{ex}. □

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

Proposition 4.3. *Let q be an acyclic self-join-free BCQ. If q has a PPJT, then* CERTAINTY(q) *admits an* **FO**-*rewriting.*

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

Example 4.4. Let $q()$:- $R(\underline{x, w}, y), S(\underline{y}, w, z), T(\underline{w}, z)$. The attack graph of q is acyclic. 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 [29] 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 join query in the class C_{forest} has a PPJT.

Proposition 4.5. *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.

The extended version of this paper shows that the foregoing brute-force search for q can be optimized to run in polynomial time when q has an acyclic attack graph and, when expressed as rule, does not contain two distinct body atoms $R(\underline{\vec{x}}, \vec{y})$ and $S(\underline{\vec{u}}, \vec{w})$ such that every variable occurring in $\vec{x}$ also occurs in $\vec{u}$. Most queries we observe and used in our experiments fall under this category.

Main Result. We previously showed that the existence of a PPJT implies an **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* **db** *be a database instance of size N. Then, there exists an algorithm for* CERTAINTY(q) *that runs in time $O(N)$.*

It is worth contrasting our result with Yannakakis' algorithm, which computes the result of any acyclic Boolean CQ also in linear time $O(N)$ [66]. Hence, the existence of a PPJT implies that computing 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, which follows from our definition of a reifiable atom in Definition 1.

Definition 3. *Let* **db** *be a database instance for* 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* **db**, *denoted by $R_{\text{gkey}}(q, \textbf{db})$, as follows:*

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

To solve CERTAINTY(q) on some input database **db**, it suffices to compute $R_{\text{gkey}}(q, \textbf{db})$ for some reifiable atom R in q: By Definition 1, $R_{\text{gkey}}(q, \textbf{db}) \neq \emptyset$ if and only if **db** is a "yes"-instance for CERTAINTY(q).

Example 4.7. For the reifiable atom Employee($\underline{x}$, y, y) in q^{ex} and the **Company** database, $\text{Employee}_{\text{gkey}}(q^{\text{ex}}, \textbf{Company}) = \{0022\}$. **Company** is therefore a "yes"-instance for CERTAINTY(q^{ex}). □

The intuition behind our algorithm is as follows. Let (τ, R) be a PPJT of q and let (τ', S) be any of its child subtrees. By the recursive construction of a PPJT, S is reifiable in $q_{\tau'}$, and thus by Definition 1, the set $S_{\text{gkey}}(q_{\tau'}, \textbf{db})$ encodes the answer for CERTAINTY($q_{\tau'}$). To compute $R_{\text{gkey}}(q, \textbf{db})$, since R and S are adjacent in the join tree τ of q, it then suffices to consider only all child subtrees (τ', S) of (τ, R). In fact, we will use the set $S_{\text{gkey}}(q_{\tau'}, \textbf{db})$ that we recursively compute to "prune" the good keys of R. The term "pair-pruning" is motivated by this process of pruning the good keys by considering one pair of atoms at a time. This idea is formalized in Lemma C.1 in the extended version of this paper. In Section 4.2, we show that each pair-pruning step can be expressed in a non-recursive Datalog program that can be executed in linear time.

Disconnected CQs. Every disconnected BCQ q can be written as $q = q_1, q_2, \dots, q_n$ where $\text{vars}(q_i) \cap \text{vars}(q_j) = \emptyset$ for $1 \leq i < j \leq n$ and each q_i is connected. If each q_i has a PPJT, then CERTAINTY(q) can be solved by checking whether the input database is a "yes"-instance for each CERTAINTY(q_i), by Lemma B.1 of [43].

4.2 The Rewriting Rules

We now show how to produce an efficient rewriting. We will use Datalog syntax to describe the rewriting, since it will simplify the exposition. In Section 5, 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 $|\text{key}(S)|$ and collects the primary-key values of the S-table that cannot contribute to a consistent answer for q [2]; and
- the predicate S_{join} has arity equal to $|\text{vars}(S) \cap \text{vars}(T)|$ and collects the values for these variables in the S-table that may contribute to a consistent answer.

[2]The f in fkey is for "false key". We want to distinguish it from "bad key" because the latter is natural to be interpreted as "not good key", but in fact the set S_{fkey} computed by the rules (sometimes strictly) contains the "not good keys".

$$
\begin{array}{ll}
\textsc{Exit-Rule} \ \{ & \mathsf{Employee_{join}}() \text{ :- } \mathsf{Employee}(x, y, y), \neg\mathsf{Employee_{fkey}}(x). \\
\textsc{Self-Pruning} \ \{ & \mathsf{Employee_{fkey}}(z_1) \text{ :- } \mathsf{Employee}(z_1, z_2, z_3), z_3 \neq z_2.
\end{array}
\quad \mathsf{Employee}(\underline{x}, z, z)
$$

$$
\downarrow \ \mathsf{Employee_{fkey}}(x) \text{ :- } \mathsf{Employee}(x, z, z), \neg\mathsf{Manager_{join}}(z, x). \quad \} \ \textsc{Pair-Pruning}
$$

$$
\begin{array}{ll}
\textsc{Exit-Rule} \ \{ & \mathsf{Manager_{join}}(z, x) \text{ :- } \mathsf{Manager}(z, x, 2020), \neg\mathsf{Manager_{fkey}}(z). \\
\textsc{Self-Pruning} \ \{ & \mathsf{Manager_{fkey}}(z_1) \text{ :- } \mathsf{Manager}(z_1, z_2, z_3), z_3 \neq 2020. \\
 & \mathsf{Manager_{fkey}}(z) \text{ :- } \mathsf{Manager}(z, x, 2020), \mathsf{Manager}(z, z_1, z_2), z_1 \neq x.
\end{array}
\quad \mathsf{Manager}(\underline{z}, x, 2020)
$$

$$
\downarrow \ \mathsf{Manager_{fkey}}(z) \text{ :- } \mathsf{Manager}(z, x, 2020), \neg\mathsf{Contact_{join}}(z, x). \quad \} \ \textsc{Pair-Pruning}
$$

$$
\begin{array}{ll}
\textsc{Exit-Rule} \ \{ & \mathsf{Contact_{join}}(z, x) \text{ :- } \mathsf{Contact}(z, x), \neg\mathsf{Contact_{fkey}}(z). \\
\textsc{Self-Pruning} \ \{ & \mathsf{Contact_{fkey}}(z) \text{ :- } \mathsf{Contact}(z, x), \mathsf{Contact}(z, z_1), z_1 \neq x.
\end{array}
\quad \mathsf{Contact}(\underline{z}, x)
$$

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

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

Algorithm 1: PPJT-Rewriting(τ, R)

Input: PPJT (τ, R) of q
Output: a Datalog program P computing $R_{\mathsf{gkey}}(q, \mathbf{db})$

```
P := ∅
P := P ∪ Self-Pruning(R)
foreach child node S of R in τ do
    P := P ∪ PPJT-Rewriting(τ, S)
    P := P ∪ Pair-Pruning(R, S)
P := P ∪ Exit-Rule(R)
return P
```

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

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*

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

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

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

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 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 $x_i \in \mathsf{vars}(T)$, we produce a rule*

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

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

Example 4.8. The self-pruning step at the PPJT $(\tau, \mathsf{Employee})$ produces one rule (from Rule 1) since the atom $\mathsf{Employee}(\underline{x}, z, z)$ has one variable that occurs twice. When executed on the **Company** database, keys 0011 and 0034 would be added to $\mathsf{Employee_{fkey}}$.

The self-pruning phase on$(\tau_M, \mathsf{Manager})$ 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 $\mathsf{Manager_{fkey}}$.

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

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

The pair-pruning step introduces one rule, which uses the fact that a key $\vec{c} \in R_{\mathsf{key}}(q, \mathbf{db})$ can be pruned if there is a fact $R(\underline{\vec{c}}, \vec{d})$ that does not join with any fact in $S_{\mathsf{join}}(\vec{w})$. The intuition is that the fact $R(\underline{\vec{c}}, \vec{d})$ will not contribute to any results of q in any repair containing it. This idea is formalized in Lemma C.1 of the extended version of this paper.

Rule 3. *Add the rule*

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

where the rules for S_{join} will be defined in Rule 4.

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

Example 4.9. 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*

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

Otherwise, we produce the rule

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

Example 4.10. Figure 2 shows the three exit rules for q^{ex}—one rule for each node in the PPJT in Figure 1. 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. This proves Theorem 4.6.

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

Lemma 4.11. *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 $\vec{u} = (u_1, u_2, \dots, u_n)$, we say that q admits a PPJT if the Boolean query $q_{[\vec{u}\to\vec{c}]}$ admits a PPJT, where $\vec{c} = (c_1, c_2, \dots, c_n)$ is a sequence of distinct constants. We can now state our main result for non-Boolean CQs.

Theorem 4.12. *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 4.11, 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 $O(N \cdot |\mathsf{OUT}_p|)$.

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 4.11, 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 $\vec{c} = (c_1, c_2, \dots, c_n)$ be any sequence of distinct constants that do not appear in q. 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 4.13. 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 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.

GROUND-RULE { ground(w) : −Employee(x, z, z), Manager(z, x, w), Contact(z, x).

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

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

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

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

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

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

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

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

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

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

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

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

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 briefly discuss how we address the flaws of Conquer and Conquesto that impair their actual runtime performance.

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

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

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

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

keeping the key variables of all atoms. For each unsafe rule with head $R_{i,\text{label}}$ where label $\in$ {fkey, 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

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

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

The rationale is that appending ground$(\vec{u})$ to all unsafe rules could potentially introduce a Cartesian product between 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 $S^1_{\text{join}}, S^2_{\text{join}}, \ldots, S^k_{\text{join}}$, 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:

```
SELECT KeyAttri(R) FROM R
LEFT OUTER JOIN S^1_join ON
  R.A_{α^1_1} = S^1_join.B^1_1 AND ... AND R.A_{α^1_{m_1}} = S^1_join.B^1_{m_1}
...
LEFT OUTER JOIN S^k_join ON
  R.A_{α^k_1} = S^k_join.B^k_1 AND ... AND R.A_{α^k_{m_k}} = S^n_join.B^k_{m_k}
WHERE S^1_join.A_{α^1_1} IS NULL OR ... S^1_join.A_{α^1_{m_1}} IS NULL OR
      S^2_join.A_{α^2_1} IS NULL OR ... S^2_join.A_{α^2_{m_1}} IS NULL OR
      ...
      S^k_join.A_{α^k_1} IS NULL OR ... S^k_join.A_{α^k_{m_1}} IS NULL
```

The *inconsistent blocks* represented by the keys found by the above three queries are *combined* using UNION ALL and could be stored in a view (e.g., R_{fkey} in Rule 1, 2, 3).

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.

```
SELECT JoinAttri(R, T) FROM R WHERE NOT EXISTS (
  SELECT * FROM R_fkey
  WHERE R.A_1 = R_fkey.A_1 AND ... AND R.A_k = R_fkey.A_k)
```

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 view R_{join} 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.

```
SELECT KeyAttri(R_1),KeyAttri(R_2),...,KeyAttri(R_k),ProjAttri(q)
FROM R_1, R_2, ..., R_k WHERE CompPredicate(q)
```

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 = $S^i_{\text{join}}.B$ connected by the AND operator, where B is a projection variable in S^i_{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 [28] and Conquesto [37] are two other CQA systems targeting their own subclasses of **FO**-rewritable queries, both with noticeable performance issues. For a more fair comparison with LinCQA, we implemented our own optimized version of both systems. Specifically, we complement Conquer presented in [28] which was only able to handle tree queries (a subclass of C_{forest}), allowing us to handle all queries in C_{forest}. Additionally, we optimized Conquesto[37] to get rid of the unnecessarily repeated computation and the undesired cartesian products produced due to its original formulation. The optimized system has significant performance gain over the original implementation and is named FastFO. More details are available in the extended version of this paper.

6 EXPERIMENTS

We present experimental evaluation answering following questions:

(1) How do first-order rewriting techniques perform 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 experiments using synthetic benchmarks used in prior works and a large real-world dataset of 400GB. We compare LinCQA against several state-of-the-art CQA systems with improvements. To the best of our knowledge, this is the most comprehensive performance evaluation of existing CQA techniques and we are the first ones to evaluate different CQA techniques on a real-world dataset of this large scale.

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 [20], 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 a SATA SSD with 440GB space being available, 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 [21] 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} [28].
FastFO: our own implementation of the general method that can handle any query for which CQA is **FO**-rewritable.

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 [25].

For LinCQA, ConQuer and FastFO, we only report execution time of **FO**-rewritings, since the rewritings can be produced within 1ms for all our queries. We report the performance of each **FO**-rewriting using the best query plan. The preprocessing time required by CAvSAT *prior* to computing the consistent answers is not reported. For each rewriting and database shown in the experimental results, we run the evaluation five times (unless timed out), discard the first run and report the average time of the last four runs.

6.2 Databases and Queries

6.2.1 Synthetic workload. We consider the synthetic workload used in previous works [24, 25, 42]. Specifically, we take the seven queries that are consistent first-order rewritable in [24, 25, 42]. These queries feature joins between primary-key attributes to foreign-key attributes and projection on non-key attributes:

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

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 constraints (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 $[1, \mathrm{rSize}/10]$.

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* (bSize−1) into each block. The parameter inBlockNum is calculated by the given rSize, inRatio and bSize: inBlockNum = (inRatio · rSize)/bSize.

Table 3: A summary of the Stackoverflow Dataset

Table	# of rows (rSize)	inRatio	max. bSize	# of Attributes
Users	14,839,627	0%	1	14
Posts	53,086,328	0%	1	20
PostHistory	141,277,451	0.001%	4	9
Badges	40,338,942	0.58%	941	4
Votes	213,555,899	30.9%	1441	6

Table 4: StackOverflow queries

Q_1	`SELECT DISTINCT P.id, P.title FROM Posts P, Votes V WHERE P.Id = V.PostId AND P.OwnerUserId = V.UserId AND BountyAmount > 100`
Q_2	`SELECT DISTINCT U.Id, U.DisplayName FROM Users U, Badges B WHERE U.Id = B.UserId AND B.name = "Illuminator"`
Q_3	`SELECT DISTINCT U.DisplayName FROM Users U, Posts P WHERE U.Id = P.OwnerUserId AND P.Tags LIKE "<c++>"`
Q_4	`SELECT DISTINCT U.Id, U.DisplayName FROM Users U, Posts P, Comments C WHERE C.UserId = U.Id AND C.PostId = P.Id AND P.Tags LIKE "%SQL%" AND C.Score > 5`
Q_5	`SELECT DISTINCT P.Id, P.Title FROM Posts P, PostHistory PH, Votes V, Comments C WHERE P.id = V.PostId AND P.id = PH.PostId AND P.id = C.PostId AND P.Tags LIKE "%SQL%" AND V.BountyAmount > 100 AND PH.PostHistoryTypeId = 2 AND C.score = 0`

6.2.2 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, meaning 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.

6.2.3 Stackoverflow Dataset. We obtained the stackoverflow.com metadata as of 02/2021 from the Stack Exchange Data Dump, with 551,271,294 rows taking up 400GB. [3,4] The database tables used are summarized in Table 3. We remark that the attribute *Id* in PostHistory, Comments, Badges, and Votes are surrogate keys and therefore not imposed as natural primary keys; instead, we properly choose composite keys as primary keys. Table 4 shows the five queries used in our CQA evaluation, where the number of tables joined together increases from 2 in Q_1 to 4 in Q_5.

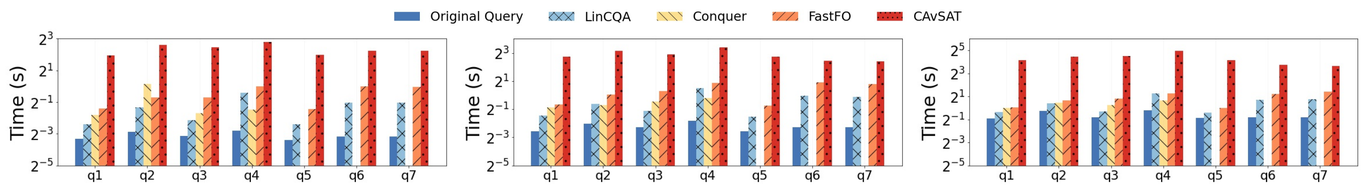
6.3 Experimental Results

In this section, we report the evaluation of LinCQA and the other CQA systems on synthetic workloads and the StackOverflow dataset.

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 [24, 25, 42]. 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 returns the possible answers).

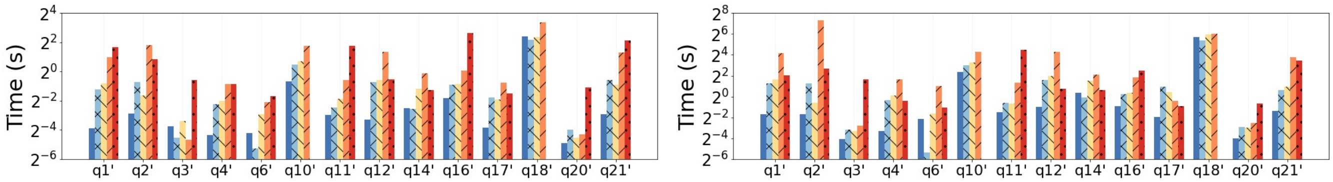
[3] https://archive.org/details/stackexchange

[4] https://sedeschema.github.io/



(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).

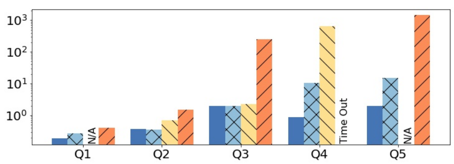


Figure 6: Runtime Comparison on StackOverflow

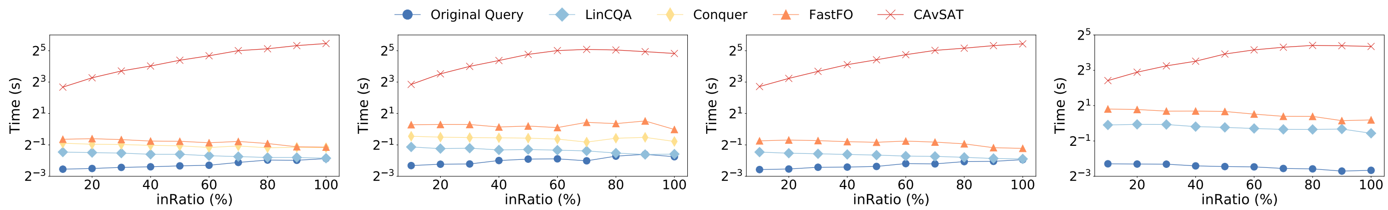
In the synthetic dataset, all three systems based on **FO**-rewriting techniques outperform CAvSAT, often by an order of magnitude. This observation shows that if CERTAINTY(q) is **FO**-rewritable, a properly implemented rewriting is more efficient than the generic algorithm in practice, refuting some observations in [25, 42]. Compared to ConQuer, LinCQA performs better or comparably on q_1 through q_4. 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. To compute the consistent answers for a certain query, we note that the actual runtime performance heavily depends on the query plan chosen by the query optimizer besides the SQL rewriting given, thus we focus on the overall performance of different CQA systems rather than a few cases in which the performance difference between different systems is relatively small.

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 7. 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.

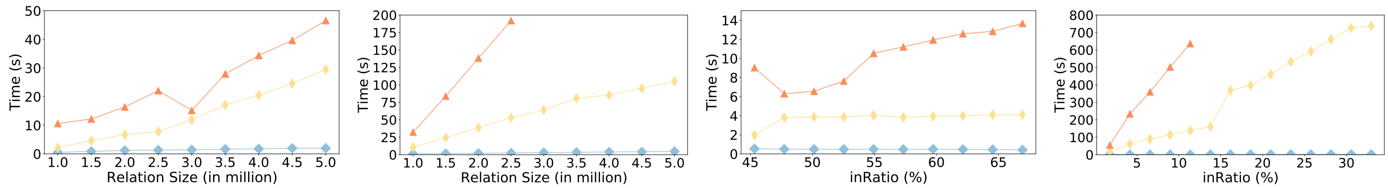
In our next experiment, we fix the the database instance size with rSize = 1M and inconsistent ratio with inRatio = 10%, running all CQA systems on databases of varying inconsistent block size bSize from 2 to 10. We observe that the performance of all CQA systems is not very sensitive to the change of inconsistent block sizes and thus we omit the results here due to the space limit.

StackOverflow Dataset We use a 400GB StackOverflow dataset to evaluate the performance of different systems on large-scale real-world datasets. CAvSAT is excluded since it requires extra disk space for preprocessing which is beyond the limit of the available disk space. Since Q_1 and Q_5 are not in C_{forest}, ConQuer cannot handle them and their execution times are marked as "N/A". Query executions that do not finish in an hour are marked as "Time Out". We observe that on all five queries, LinCQA significantly outperforms other competitors. For queries that ConQuer (Q_4) and FastFO (Q_3, Q_5) take long to compute, LinCQA manages to finish execution quickly thanks to its efficient self-pruning and pair-pruning steps, which enhance main memory computation by avoiding the generation of huge intermediate results that have to be spilled to disk.



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

Figure 7: 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 8: Performance comparison between different systems on varying relation sizes/inconsistent ratios

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 $Q_{3-\text{path}}$:

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

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

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.

We define a generic binary relation $\mathcal{D}(x, y, N)$ as

$$\mathcal{D}(x, y, N) = ([x] \times [y]) \cup \{(u, u) \mid xy + 1 \leq u \leq N, u \in \mathbb{Z}^+\},$$

where $x, y, N \in \mathbb{Z}^+$, $[n] = \{1, 2, \ldots, n\}$ and $[a] \times [b]$ denotes the cartesian product between $[a]$ and $[b]$. To generate the input instances for $Q_{2-\text{path}}$, we generate relations $\mathsf{R} = \mathcal{D}(a, b, N)$ and $\mathsf{S} = \mathcal{D}(b, c, N)$ with integer parameters a, b, c and N. For $Q_{3-\text{path}}$, we additionally generate the relation $\mathsf{T} = \mathcal{D}(c, d, N)$. 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 8a and 8b, 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 8b 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 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 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 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 8c and 8d 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 8d 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 efficient rewritings, is scalable, and robust on worst case instances.

REFERENCES

[1] Miklós Ajtai and Yuri Gurevich. 1994. Datalog vs First-Order Logic. *J. Comput. Syst. Sci.* 49, 3 (1994), 562–588.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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

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

[9] 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.

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

[11] 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.

[12] 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.

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

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

[15] 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

[16] 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.

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

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

[19] 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.

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

[21] 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.

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

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

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

[40] 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.

[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] 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.

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

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

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

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

[51] 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.

[52] 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.

[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 ATTACK GRAPH

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

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

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

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

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 $i \in \{0, \ldots, n-1\}$, $\text{vars}(F_i) \cap \text{vars}(F_{i+1}) \nsubseteq F^{+,q}$.

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 CERTAINTY(q).

Theorem A.1 ([46]). *Let q be a self-join-free BCQ. Then the attack graph of q is acyclic if and only if* CERTAINTY(q) *is in* **FO**.

B EFFICIENT CONSTRUCTION OF PPJT

Proposition B.1. *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* key(F) *or* 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.*

Proof. 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 key(G) and 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:

$$\text{for every } G \in q_{\tau_i}, \mathbf{vars}(G) \cap \mathbf{vars}(F) \subseteq \text{key}(G). \tag{2}$$

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) $\text{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \text{key}(F'_i)$.

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 $\mathbf{vars}(F_i) \cap \mathbf{vars}(F) \subseteq \text{key}(G)$. Then from $\text{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \mathbf{vars}(F_i)$, it follows $\text{vars}(q_{\tau_i}) \cap \mathbf{vars}(F) \subseteq \text{key}(G)$. 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.

□

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 [22, 23].

Proposition B.2. *Let $q() \text{ :- } R(\underline{c}, x), S(\underline{c}, y), T(\underline{x, y})$. Then there exists a linear time algorithm for* CERTAINTY(q).

C MISSING PROOFS

Proof of Proposition 4.3. 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 [46]. 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.

The proof is now complete by the classification theorem of [43]. □

Proof of Proposition 4.5. Let q be a query in C_{forest} and let G be the join graph of q as in Definition 6 of [29]. 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 $w \in \text{vars}(S)$ such that $w \in \text{vars}(R) \setminus \text{key}(R)$. By the definition of C_{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 $w \in \text{vars}(R)$, $w \in \text{vars}(S)$, and for any atom T_i in the unique path $R - T_1 - \cdots - T_k - S$, we have $w \notin \text{vars}(T_i)$. We must have $w \in \text{key}(R)$ and $w \in \text{key}(S)$, or otherwise there would be an arc between R and S, a contradiction. From the property C_{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, $w \in \text{vars}(T_1)$, 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 [29], 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. ◻

Lemma C.1. *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 () :- $R(\underline{\vec{x}}, \vec{y})$; 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 () :- $R(\underline{\vec{x}}, \vec{y}), S(\underline{\vec{x}'}, \vec{y}')$, where $S(\underline{\vec{x}'}, \vec{y}')$ is the S-atom of q.

Proof. 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 1, $\mathbf{db}$ 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 4.11. 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 **db**. Then there is a valuation μ with $\mu(q_{[\vec{u}\rightarrow\vec{c}]}) \subseteq r$. 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 $\mu^{+}(q) = \mu(q_{[\vec{u}\rightarrow\vec{c}]}) \subseteq r$ with $\mu^{+}(\vec{u}) = \theta(\vec{u}) = \vec{c}$, as desired. □

D IMPROVEMENT UPON EXISTING SYSTEMS

D.1 Conquer

Fuxman and Miller [29] identified C_{forest}, a class of CQs whose consistent answers can be computed via an **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 is not connected [28]. 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 [29], enabling us to produce the consistent SQL rewriting for every query in C_{forest}.

D.2 Conquesto

Conquesto [37] produces a non-recursive Datalog program that implements the algorithm in [46], targetting all **FO**-rewritable self-join-free CQs. However, it suffers from repeated computation and unnecessary cartesian products. For example, the Conquesto rewriting for the CQ $q(z)$:- $R_1(\underline{x}, y, z), R_2(\underline{y}, v, w)$ is shown as follows, where Rule (3) and (5) share the common predicate $\mathsf{R}_2(y, v, w)$ in their bodies, resulting in re-computation, and Rule (7) involves a Cartesian product.

$$\mathsf{Sr}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{R}_2(y, v, w). \tag{3}$$
$$\mathsf{Yes}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{Sr}_{\mathsf{R}_2}(y), \mathsf{R}_2(y, v, w). \tag{4}$$
$$\mathsf{Sr}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{R}_1(x, y, z), \mathsf{R}_2(y, v, w). \tag{5}$$
$$\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, v_2), \mathsf{Yes}_{\mathsf{R}_2}(y), v_2 = z. \tag{6}$$
$$\mathsf{Bb}_{\mathsf{R}_1}(x, z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, v_2), \neg\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z). \tag{7}$$
$$\mathsf{Yes}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, z), \neg\mathsf{Bb}_{\mathsf{R}_1}(x, z). \tag{8}$$

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.

$$\mathsf{Sr}(x, y, z) \text{ :- } \mathsf{R}_1(x, y, z), \mathsf{R}_2(y, v, w). \tag{9}$$
$$\mathsf{Yes}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{Sr}(x, y, z), \mathsf{R}_2(y, v, w). \tag{10}$$
$$\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z) \text{ :- } \mathsf{Sr}(x, \_, z), \mathsf{R}_1(x, y, v_2), \mathsf{Yes}_{\mathsf{R}_2}(y), v_2 = z. \tag{11}$$
$$\mathsf{Bb}_{\mathsf{R}_1}(x, z) \text{ :- } \mathsf{Sr}(x, \_, z), \mathsf{R}_1(x, y, v_2), \neg\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z). \tag{12}$$
$$\mathsf{Yes}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{Sr}(x, y, z), \mathsf{R}_1(x, y, z), \neg\mathsf{Bb}_{\mathsf{R}_1}(x, z). \tag{13}$$

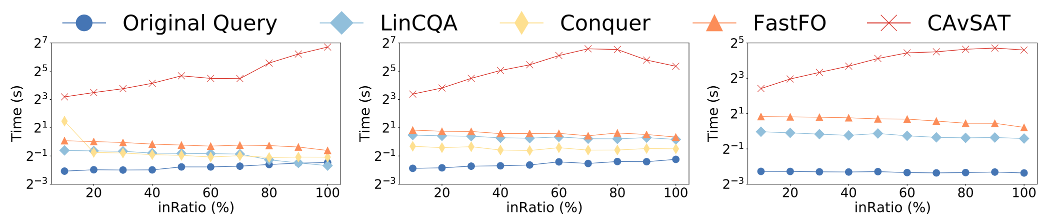
For evaluation, the rules computing each intermediate relation (i.e. all rules except for the one computing $\mathsf{Yes}_{\mathsf{R}_1}(z)$) are then translated to a SQL subquery via a `WITH` clause.

E ADDITIONAL TABLES AND FIGURES

Figure 9 presents the performance of queries q_2, q_4 and q_6 on varying inconsistency ratios, supplementing Figure 7. Figure 10 summarizes the performance of all seven synthetic queries on varying block sizes.

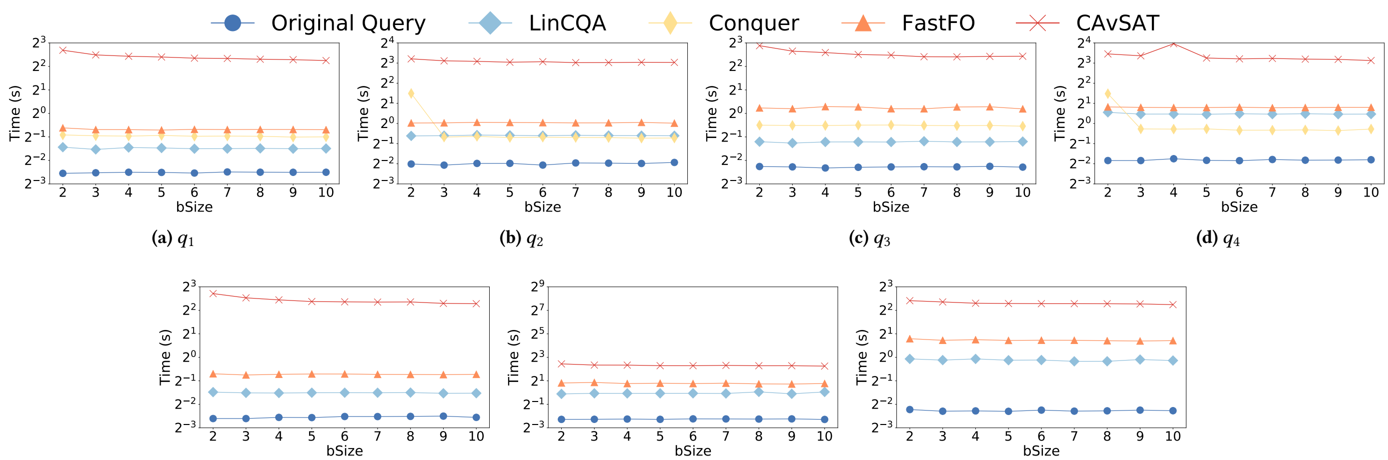
Table 5: A summary of the Stackoverflow Dataset.

Table	# of rows	inRatio	max. bSize	Attributes
Users	14,839,627	0%	1	Id, AboutMe, Age, CreationDate, DisplayName, DownVotes, EmailHash, LastAccessDate, Location, Reputation, UpVotes, Views, WebsiteUrl, AccountId
Posts	53,086,328	0%	1	Id, AcceptedAnswerId, AnswerCount, Body, ClosedDate, CommentCount, CommunityOwnedDate, CreationDate, FavoriteCount, LastActivityDate, LastEditDate, LastEditorDisplayName, LastEditorUserId, OwnerUserId, ParentId, PostTypeId, Score, Tags, Title, ViewCount
PostLinks	7,499,403	0%	1	Id, CreationDate, PostId, RelatedPostId, LinkTypeId
PostHistory	141,277,451	0.001%	4	Id, PostHistoryTypeId, PostId, RevisionGUID, CreationDate, UserId, UserDisplayName, Comment, Text
Comments	80,673,644	0.0012%	7	Id, CreationDate, PostId, Score, Text, UserId
Badges	40,338,942	0.58%	941	Id, Name, UserId, Date
Votes	213,555,899	30.9%	1441	Id, PostId, UserId, BountyAmount, VoteTypeId, CreationDate



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

Figure 9: 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 10: Performance of different systems on inconsistent database of varying block size