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 rewritings 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, 21 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 [61] 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 so as 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 [3, 32–34, 63], using techniques such as knowledge bases and machine learning [10, 13, 19, 20, 29, 35, 53, 54, 62, 65]. Data cleaning has also been studied under different contexts [11, 14, 16, 40, 41, 60]. However, the process of data cleaning is often ad hoc and arbitrary choices are frequently made regarding which data to keep in order 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 [20, 21, 62, 63], but in most cases, 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 [4] that has recently received more attention [12, 68]. 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 [37].

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 potentially requires inspecting an exponential number of 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 [42, 46, 47, 50, 52], 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 [48].

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 [43] relies on Integer Programming solvers, and CAvSAT [27,

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

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 [43]	all SPJ Queries	Big Integer Program (BIP)	DBMS & BIP solver
CAvSAT [27, 28]	all SPJ Queries	SAT formula	DBMS & SAT solver
Conquer [30]	C_{forest}	SQL rewriting	DBMS
Conquesto [39]	self-join-free SPJ Queries in **FO**	Datalog rewriting	Datalog engine
LinCQA (this paper)	PPJT	SQL rewriting / Datalog rewriting	DBMS or Datalog engine

28] 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 [30] identified a class of **FO**-rewritable queries called C_{forest} and implemented their rewriting in ConQuer, which outputs a single SQL query. Conquesto [39] 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)* joining two tables on 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 has 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 Boolean 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 Boolean queries in C_{forest}. For consistent databases, Yannakakis's algorithm [9] evaluates acyclic Boolean 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 Boolean 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 has been studied in different contexts [7, 8, 15, 17, 36, 38, 55, 64]. The notion of Consistent Query Answering (CQA) was introduced in the seminal work by Arenas, Bertossi, and Chomicki [4]. After twenty years, their contribution was acknowledged in a *Gems of PODS session* [12]. An overview of complexity classification results in CQA appeared recently in the *Database Principles* column of SIGMOD Record [68].

The term CERTAINTY(q) was coined in [66] 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 [31], and was further pursued in [42, 46–48, 50, 52], 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 [45]. The complexity of CERTAINTY(q) for self-join-free Boolean conjunctive queries with negated atoms was studied in [49]. For self-join-free Boolean conjunctive queries w.r.t. multiple keys, it remains decidable whether or not CERTAINTY(q) is in **FO** [51].

Several systems for CQA that are used for comparison in our study have already been described in the introduction: ConQuer [30], Conquesto [39], CAvSAT [27, 28], and EQUIP [43]. Most early systems for CQA used efficient solvers for Disjunctive Logic Programming and Answer Set Programming (ASP) [5, 18, 34, 55–57].

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 [37].

3 BACKGROUND

In this section, we define some notations used in our paper. We use the example **Company** database shown in Figure 1 to illustrate our constructs, where the primary key attribute of each table is highlighted in bold.

Employee

employee_id	office_city	wfh_city
0011	Boston	Boston
0011	Chicago	New York
0011	Chicago	Chicago
0022	New York	New York
0022	Chicago	Chicago
0034	Boston	New York

Manager

office_city	manager_id	start_year
Boston	0011	2020
Boston	0011	2021
Chicago	0022	2020
LA	0034	2020
LA	0037	2020
New York	0022	2020

Contact

office_city	contact_id
Boston	0011
Boston	0022
Chicago	0022
LA	0034
LA	0037
New York	0022

Figure 1: An inconsistent database (Company). A repair of this database is highlighted with the gray color.

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*, and the length of that sequence is called the *arity* of that table. Some of these attributes are declared as *primary-key attributes*, forming together the *primary key*. A *database instance* **db** associates to each table name a finite set of *tuples* that agree on the arity of the table, 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 in Figure 1 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 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 $\text{vars}(\vec{x})$ for the set of variables that appear in $\vec{x}$. An *atom* F with relation name R takes the form $R(\underline{\vec{x}}, \vec{y})$, where the primary key is underlined; we denote $\text{key}(F) = \text{vars}(\vec{x})$. 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 $\text{Employee}(\underline{x}, y, y)$, $\text{Manager}(\underline{u}, v, \texttt{2020})$, and $\text{Contact}(\underline{\text{LA}}, \texttt{2020})$. The block $\text{Manager}(\underline{\text{Boston}}, *)$ contains two facts: Manager($\underline{\text{Boston}}$, 0011, 2020) and Manager($\underline{\text{Boston}}$, 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 $\text{vars}(q)$ 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, $\text{body}(q)$.

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 a self-join-free query q is understood, an atom $R(\underline{\vec{x}}, \vec{y})$ in q can be denoted by R. 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:

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

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

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

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

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, meaning that every variable occurring in the rule must also occur in a non-negated atom of the rule body. A relation belongs to the intensional database (IDB) if it is defined by rules, i.e., if it appears as the head of some rule; otherwise it belongs to the extensional database (EBD), i.e., it is a stored table. 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 $\text{CERTAINTY}(q)$ asks for the intersection of query outputs over all repairs of **db**. If q is Boolean, the problem CERTAINTY(q) then asks whether q is satisfied by every repair of the input database instance **db**. In this work, we study the *data complexity* of CERTAINTY(q), i.e., the size of the query q is assumed to be a fixed constant.

The problem 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 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 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 the Manager table and both managers in Chicago and New York offices started in 2020.*

Based on the observation, it is sufficient to solve CERTAINTY(q) by running the following single SQL query, called an **FO**-rewriting of 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)
```
□

Acyclic queries and join trees Let q be a CQ. A *join tree* of q is an undirected tree whose nodes are the atoms of q such that for every two distinct atoms R and S, their common variables occur in all atoms on the unique path from R to S in the tree. A CQ 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.

Example 3.5. The join tree of the query q in Example 3.3 has a single edge between Employee($\underline{x}, y, z$) and Manager($\underline{y}, w, \texttt{2020}$).

Attack graphs. Let q be an acyclic, self-join-free BCQ with join tree τ. For every atom $F \in q$, we define $F^{+,q}$ as the set of all variables in q that are functionally determined by key(F) with respect to all functional dependencies of the form key(G) $\rightarrow$ vars(G) with $G \in q \setminus \{F\}$. Following [67], the *attack graph* of q is a directed graph whose vertices are the atoms of q. There is a directed edge, called *attack*, from F to G ($F \neq G$), if on the unique path between F and G in τ, every two adjacent atoms share a variable not in $F^{+,q}$. An atom without incoming edges in the attack graph is called *unattacked*. The attack graph of q is used to determine the data complexity of CERTAINTY(q): the attack graph of q is acyclic if and only if CERTAINTY(q) is in **FO** [48].

[1] Throughout this paper, whenever we say that a CQ is acyclic, we mean acyclicity as defined in [9], a notion that today is also known as α-acyclicity, to distinguish it from other notions of acyclicity.

Example 3.6. For the query q in Example 3.3, Employee$^{+,q} = \{x\}$ and Manager$^{+,q} = \{y\}$. It follows that Employee attacks Manager because the variable y is shared between atoms Employee and Manager and $y \notin$ Employee$^{+,q}$. However, Manager does not attack Employee since the only shared variable y is in Manager$^{+,q}$.

The attack graph of q is acyclic since it only contains one attack from Employee to Manager. It follows that CERTAINTY(q) is in **FO**, as witnessed by the **FO**-rewriting in Example 3.4.

4 A LINEAR-TIME REWRITING

Before presenting our linear-time rewriting for CERTAINTY(q), we first provide a motivating example. Consider the following query on the **Company** database shown in Figure 1:

Is there an office whose contact person works for the office and, moreover, manages the office since 2020?

This query can be expressed by the following CQ:

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

To the best of our knowledge, the most efficient running time for CERTAINTY(q^{ex}) guaranteed by existing systems is quadratic in the input database size, denoted N. The problem CERTAINTY(q^{ex}) admits an **FO**-rewriting by the classification theorem in [45]. However, the non-recursive Datalog rewriting of CERTAINTY(q^{ex}) produced by Conquesto contains cartesian products between two tables, which means that it runs in $\Omega(N^2)$ time in the worst case. 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 CERTAINTY(q^{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 CERTAINTY(q) (Theorem 4.5). Finally, we extend our result to all acyclic self-join-free CQs in Section 4.3 (Theorem 4.10) .

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.

Recall that an atom in a self-join-free query can be uniquely denoted by its relation name. For example, we may use Employee as a shorthand for the atom Employee($\underline{x}, y, z$) in q^{ex}.

Definition 1 (PPJT). *Let q be an acyclic self-join-free BCQ. Let τ be a join tree of q and R a node in τ. The tree (τ, R) is a pair-pruning join tree (PPJT) of q if for any rooted subtree (τ', R') of (τ, R), the atom R' is unattacked in $q_{\tau'}$.*

Example 4.1. For the join tree τ in Figure 2, the rooted tree $(\tau, \text{Employee})$ is a PPJT for q^{ex}. The atom Employee($\underline{x}, y, z$) is unattacked in q. For the child subtree $(\tau_M, \text{Manager})$ of $(\tau, \text{Employee})$, the atom

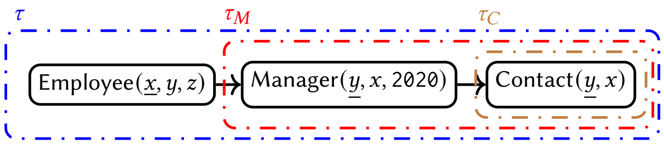


Figure 2: A pair-pruning join tree (PPJT) of the query q^{ex}.

Manager($\underline{y}$, x, 2020) is also unattacked in the following subquery

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

Finally, for the subtree $(\tau_C, \text{Contact})$, the atom Contact($\underline{y}, x$) is also unattacked in the corresponding subquery $q^{\text{ex}}_{\tau_C}() \text{ :- } \text{Contact}(\underline{y}, x)$. Hence $(\tau, \text{Employee})$ 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 4.2. *Let q be an acyclic self-join-free BCQ. If q has a PPJT, then* CERTAINTY(q) *admits an* **FO***-rewriting.*

Proposition 4.2 is proved in Appendix B.1 of the extended version of this paper, in which we show that the if q has a PPJT, then the attack graph of q must be acyclic. We note that not all acyclic self-join-free BCQs with an acyclic attack graph have a PPJT, as demonstrated in the next example.

Example 4.3. Let $q() \text{ :- } R(\underline{x, w}, y), S(\underline{y, w}, z), T(\underline{w}, z)$. The attack graph of q is acyclic. 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 [31] identified a large class of self-join-free CQs, called C_{forest}, that includes most queries with primary-key-foreign-key joins, path queries, and queries on a star schema, such as found in SSB and TPC-H [58, 59]. This class covers most of the SPJ queries seen in practical settings. In view of this, the following proposition is of practical significance.

Proposition 4.4. *Every acyclic BCQ in C_{forest} has a PPJT.*

Furthermore, it is easy to verify that, unlike C_{forest}, PPJT captures *all* **FO**-rewritable self-join-free SPJ queries on two tables, a.k.a. binary joins. For example, the binary join q_5 in Section 6.2 admits a PPJT but is not in C_{forest}.

Proposition 4.4 is proved in Appendix B.2 of the extended version of this paper.

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). For the data complexity of CERTAINTY(q), this exhaustive search runs in constant time since we assume n is a constant. In practice, the search cost is acceptable for most join queries that do not involve too many tables.

Appendix A of 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 a 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 4.5. *Let q be an acyclic self-join-free BCQ 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 BCQ also in linear time $O(N)$ [69]. Hence, the existence of a PPJT implies that computing CERTAINTY(q) will have the same asymptotic complexity.

Disconnected CQs. Every disconnected BCQ q can be written as $q = q_1, q_2, \ldots, q_n$ 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 [45].

4.2 The Rewriting Rules

We now show how to produce an efficient rewriting in Datalog and prove Theorem 4.5. In Section 5, we will discuss how to translate the Datalog program to SQL. Let q be an acyclic self-join-free BCQ with a PPJT (τ, R) and **db** an instance for the problem CERTAINTY(q): does the query q evaluate to true on every repair of **db**?

Let us first revisit Yannakakis' algorithm for evaluating q on a database **db** in linear time. Given a rooted join tree (τ, R) of q, Yannakakis' algorithm visits all nodes in a bottom-up fashion. For every internal node S of (τ, R), it keeps the tuples in table S that join with every child of S in (τ, R), where each such child has been visited recursively. In the end, the algorithm returns whether the root table R is empty or not. Equivalently, Yannakakis' algorithm evaluates q on **db** by removing tuples from each table that cannot contribute to an answer in **db** at each recursive step.

Our algorithm for CQA proceeds like Yannakakis' algorithm in a bottom-up fashion. At each step, we remove tuples from each table that cannot contribute to an answer to q in at least one repair of **db**. Informally, if a tuple cannot contribute to an answer in at least one repair of **db** containing it, then it cannot contribute to a consistent answer to q on **db**. Specifically, given a PPJT (τ, R) of q, to compute all tuples of each internal node S of (τ, R) that may contribute to a consistent answer, we need to "prune" the blocks of S in which there is some tuple that violates either the local selection condition on table S, or the joining condition with some child table of S in (τ, R). The term "pair-pruning" is motivated by the latter process, where we consider only one pair of tables at a time. This idea is formalized in Algorithm 1, where the procedures Self-Pruning and Pair-Pruning prune, respectively, the blocks that violate the local selection condition and the joining condition.

To ease the exposition of the rewriting, we now present both procedures in Datalog syntax. We will use two predicates for every atom S in the tree (let T be the unique parent of S in τ):

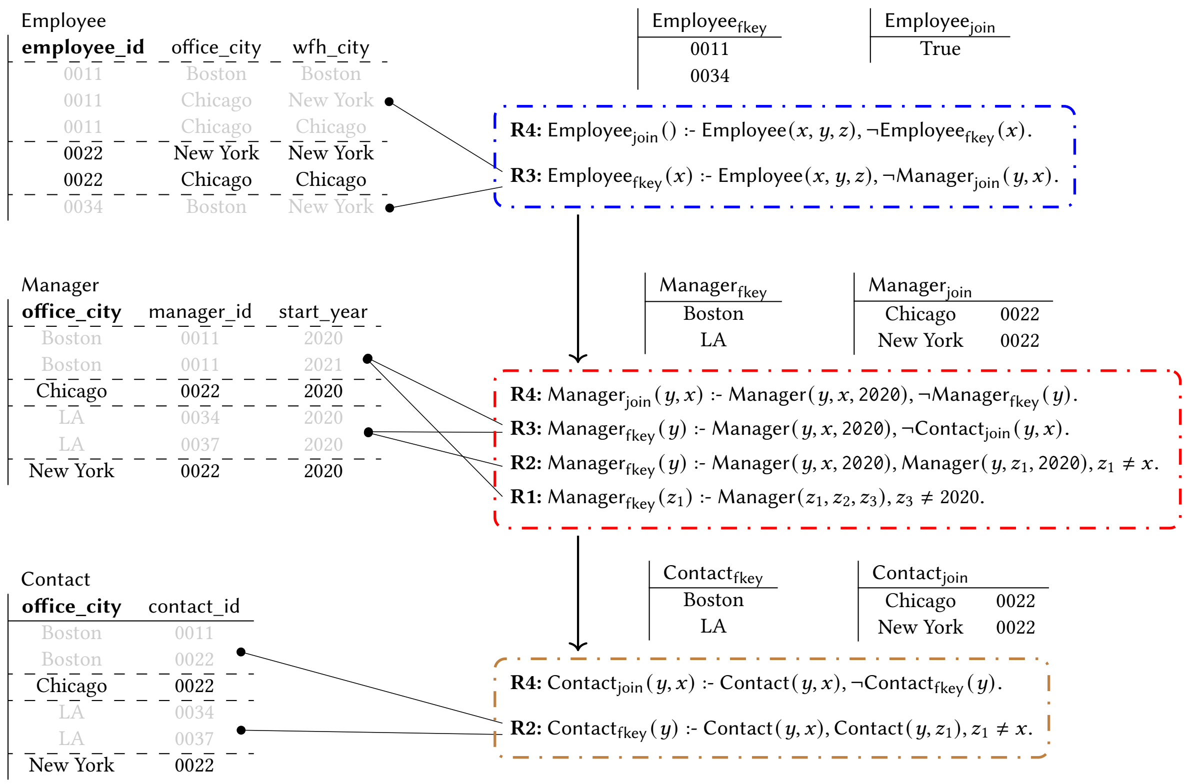


Figure 3: The non-recursive Datalog program for evaluating $\mathsf{CERTAINTY}(q^{\mathsf{ex}})$ together with an example execution on the Company database in Figure 1. The faded-out rows denote blocks that are removed since they do not contribute to any consistent answer. The arrows denote which rules remove which blocks (some blocks can be removed by multiple rules).

- the predicate S_{fkey} has arity equal to $|\mathsf{key}(S)|$ and collects the primary-key values of the S-table that cannot contribute to a consistent answer for q [2]; and
- the predicate S_{join} has arity equal to $|\mathsf{vars}(S) \cap \mathsf{vars}(T)|$ and collects the values for these variables in the S-table that may contribute to a consistent answer.

Algorithm 1: PPJT-REWRITING(τ, R)

Input: PPJT (τ, R) of q
Output: a Datalog program P deciding $\mathsf{CERTAINTY}(q)$
$P := \emptyset$
$P := P \cup$ SELF-PRUNING(R)
foreach *child node S of R in τ* **do**
  $P := P \cup$ PPJT-REWRITING(τ, S)
  $P := P \cup$ PAIR-PRUNING(R, S)
$P := P \cup$ EXIT-RULE(R)
return P

Figure 3 depicts how each step generates the rewriting rules for q^{ex}. We now describe how each step is implemented in detail.

[2] The f in fkey is for "false key".

SELF-PRUNING(R): Let $R(x_1, \dots x_k, x_{k+1}, \dots, x_n)$, 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 some tuple with that primary-key violates 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. Therefore, that R-block cannot contribute to a consistent answer.

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*

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

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

Example 4.6. The self-pruning phase on$(\tau_M$, Manager) produces one rule using Rule 1. When executed on the **Company** database, the key Boston is added to $\text{Manager}_{\text{fkey}}$, since the tuple (Boston, 0011, 2021) has start_year ≠ 2020.

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

PAIR-PRUNING(R, S): Suppose that q contains the atoms $R(\underline{\vec{x}}, \vec{y})$ and $S(\underline{\vec{u}}, \vec{v})$, 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 $\text{vars}(R) \cap \text{vars}(S)$. The third rule prunes all R-blocks containing some tuple that cannot join with any S-tuple to contribute to a consistent answer.

Rule 3. *Add the rule*

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

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 4.7. Figure 3 shows the two pair-pruning rules generated (in general, there will be one pair-pruning rule for each parent-child edge in the PPJT. In both cases, the join variables are $\{y, x\}$. For the table Employee, the rule prunes the two blocks with keys 0011, 0034 and adds them to $\text{Employee}_{\text{fkey}}$. □

EXIT-RULE(R): Suppose that q contains $R(\underline{\vec{x}}, \vec{y})$. If R is an internal node, let $\vec{w}$ be a sequence of distinct variables containing all (and only) the join variables of R and its parent node in τ. If R is the root node, let $\vec{w}$ be the empty vector. The exit rule removes the pruned blocks of R and projects on the variables in $\vec{w}$. If R is an internal node, the resulting tuples in the projection could contribute to a consistent answer, and will be later used for pair pruning; if R is the root, the projection returns the final result.

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

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

Otherwise, we produce the rule

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

Example 4.8. Figure 3 shows the three exit rules for q^{ex}—one rule for each node in the PPJT. The boolean predicate $\text{Employee}_{\text{join}}$ determines whether `True` is the consistent answer to the query. □

Runtime Analysis It is easy to see that **Rule 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 hash 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.

Sketch of Correctness Let q be an acyclic self-join-free BCQ with a PPJT (τ, R) and **db** an instance for $\text{CERTAINTY}(q)$. The easier property to show is the *soundness* of our rewriting **Rules 1, 2, 3, 4**: if the predicate R_{join} is nonempty when our rewriting is executed on **db**, then every repair of **db** must necessarily satisfy q. The argumentation uses a straightforward bottom-up induction on the PPJT: for every rooted subtree (τ', S) of (τ, R), the tuples in S_{join} are consistent answers to the corresponding subquery $q_{\tau'}$ projected on the join variables with the parent of S (i.e., on the variables $\vec{w}$ in **Rule 4**).

The more difficult property to show is the *completeness* of our rewriting rules: if every repair of **db** satisfies q, then the predicate R_{join} must be nonempty after executing the rules on **db**. The crux here is a known result (see, for example, Lemma 4.4 in [48]) which states that for every unattacked atom R in a self-join-free BCQ q, the following holds true:

> *if every repair of* **db** *satisfies* q*, then there is a nonempty block* **b** *of* R *such that in each repair of* **db***, the query* q *can be made true by using the (unique) tuple of* **b** *in that repair.*

Our recursive construction of a PPJT (τ, R) ensures that for each rooted subtree (τ', S) of (τ, R), S is unattacked in $q_{\tau'}$. Therefore, it suffices to compute the blocks in S that could contribute to a consistent answer to $q_{\tau'}$ at each recursive step in a bottom-up fashion, eventually returning the consistent answer to q in **db**.

The soundness and completeness arguments taken together imply that our rewriting rules return only and all consistent answers. The formal correctness proof is in Appendix B.3 of the extended version of this paper.

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

LEMMA 4.9. *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* $\text{CERTAINTY}(q_{[\vec{u} \to \vec{c}]})$.

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

THEOREM 4.10. *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 $O(N \cdot |\text{OUT}_p|)$; and
(2) moreover, if q is full, the set of consistent answers can be computed in time $O(N + |\text{OUT}_c|)$.

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

Proof Sketch. Our algorithm first evaluates q on **db** to yield a set S of size $|\mathsf{OUT}_p|$ in time $O(N \cdot |\mathsf{OUT}_p|)$. We then return all answers $\vec{c} \in S$ such that **db** is a "yes"-instance for $\mathsf{CERTAINTY}(q_{[\vec{u}\to\vec{c}]})$, which runs in $O(N)$ by Theorem 4.5. This approach gives an algorithm with running time $O(N \cdot |\mathsf{OUT}_p|)$.

If q is full, we proceed by (i) removing all blocks with at least two tuples from **db** to yield $\mathbf{db}^c$ and (ii) evaluating q on $\mathbf{db}^c$. In our algorithm, step (i) runs in $O(N)$ and since q is full, step (ii) runs in time $O(N + |\mathsf{OUT}_c|)$. The correctness proof of these algorithms are in Appendix B.5 of the extended version of this papar. □

Rewriting for non-Boolean Queries Let $\vec{c} = (c_1, c_2, \ldots, c_n)$ be a sequence of fresh, distinct constants. 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 4.11. Consider the non-Boolean query

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

Note that the constant `2020` in q^{ex} is replaced by the free variable w in q^{nex}. Hence, the program P for $\mathsf{CERTAINTY}(q^{\mathsf{nex}}_{[w\to c]})$ is the same as Figure 3, with the only difference that `2020` is replaced by the constant c. The ground rule produced is:

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

and Figure 4a shows how Yannakakis' algorithm evaluates q^{nex}.

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:

$$\mathsf{Manager}_{\mathsf{fkey}}(y_1) \text{ :- } \mathsf{Manager}(y_1, y_2, y_3), y_3 \neq w.$$

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 $\mathsf{Manager}_{\mathsf{fkey}}(y_1)$ becomes $\mathsf{Manager}_{\mathsf{fkey}}(y_1, w)$. The transformed rule will be:

$$\mathsf{Manager}_{\mathsf{fkey}}(y_1, w) \text{ :- } \mathsf{Manager}(y_1, y_2, y_3), y_3 \neq w, \mathsf{ground}(w).$$

The full rewriting for q^{nex} can be seen in Figure 4b. □

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

5 IMPLEMENTATION

In this section, we first present LinCQA, a system that produces the consistent **FO**-rewriting of a query q in both Datalog and SQL formats if q has a PPJT. Having a rewriting in both formats allows us to use both Datalog and SQL engines as a backend. We then 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, or it terminates otherwise.

5.1.1 *Datalog rewriting.*

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

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

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

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

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

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

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

The rationale is that appending $\mathsf{ground}(\vec{u})$ to all unsafe rules could potentially introduce a Cartesian product between $\mathsf{ground}(\vec{u})$ and some existing atom $R(\vec{v}, \vec{w})$ in the rule. The Cartesian product has size $O(N \cdot |\mathsf{OUT}_p|)$ and would take $\Omega(N \cdot |\mathsf{OUT}_p|)$ 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 $O(N + |\mathsf{ground}^*|)$ time to compute. Note that the size of ground^* can be as large as $N^k \cdot |\mathsf{OUT}_p|$ 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
```

$\mathsf{Employee}_{\mathsf{join}}(w)$:- $\mathsf{Employee}(x, y, z), \mathsf{Manager}_{\mathsf{join}}(y, x, w)$.

$\mathsf{Manager}_{\mathsf{join}}(y, x, w)$:- $\mathsf{Manager}(y, x, w), \mathsf{Contact}_{\mathsf{join}}(y, x)$.

$\mathsf{Contact}_{\mathsf{join}}(y, x)$:- $\mathsf{Contact}(y, x)$.

(a) Yannakakis' Algorithm

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

R4: $\mathsf{Employee}_{\mathsf{join}}(w)$:- $\mathsf{Employee}(x, y, z), \neg\mathsf{Employee}_{\mathsf{fkey}}(x, w), \mathsf{ground}(w)$.

R3: $\mathsf{Employee}_{\mathsf{fkey}}(x, w)$:- $\mathsf{Employee}(x, y, z), \neg\mathsf{Manager}_{\mathsf{join}}(y, x, w), \mathsf{ground}(w)$.

R4: $\mathsf{Manager}_{\mathsf{join}}(y, x, w)$:- $\mathsf{Manager}(y, x, w), \neg\mathsf{Manager}_{\mathsf{fkey}}(y), \mathsf{ground}(w)$.

R3: $\mathsf{Manager}_{\mathsf{fkey}}(y, w)$:- $\mathsf{Manager}(y, x, w), \neg\mathsf{Contact}_{\mathsf{join}}(y, x), \mathsf{ground}(w)$.

R2: $\mathsf{Manager}_{\mathsf{fkey}}(y, w)$:- $\mathsf{Manager}(y, x, w), \mathsf{Manager}(y, z_1, w), z_1 \neq x$.

R1: $\mathsf{Manager}_{\mathsf{fkey}}(z_1, w)$:- $\mathsf{Manager}(z_1, z_2, z_3), z_3 \neq w, \mathsf{ground}(w)$.

R4: $\mathsf{Contact}_{\mathsf{join}}(y, x)$:- $\mathsf{Contact}(y, x), \neg\mathsf{Contact}_{\mathsf{fkey}}(y)$.

R2: $\mathsf{Contact}_{\mathsf{fkey}}(y)$:- $\mathsf{Contact}(y, x), \mathsf{Contact}(y, z_1), z_1 \neq x$.

(b) PPJT extended to non-Boolean queries

Figure 4: The non-recursive Datalog program for q^{nex} and CERTAINTY(q^{nex}).

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 $\mathsf{S}^1_{\mathsf{join}}, \mathsf{S}^2_{\mathsf{join}}, \ldots, \mathsf{S}^k_{\mathsf{join}}$, which are readily computed in the recursive steps. For each $1 \leq i \leq k$, let the attributes of S^i be $B^i_1, B^i_2, \ldots, B^i_{m_i}$, joining with attributes $A_{\alpha^i_1}, A_{\alpha^i_2}, \ldots, A_{\alpha^i_{m_i}}$ in R respectively. We produce the following rule:

$$
\begin{aligned}
&\texttt{SELECT}\ \mathbf{KeyAttri}(\mathsf{R})\ \texttt{FROM}\ \mathsf{R}\\
&\texttt{LEFT OUTER JOIN}\ \mathsf{S}^1_{\mathsf{join}}\ \texttt{ON}\\
&\quad \mathsf{R}.A_{\alpha^1_1} = \mathsf{S}^1_{\mathsf{join}}.B^1_1\ \texttt{AND}\ \ldots\ \texttt{AND}\ \mathsf{R}.A_{\alpha^1_{m_1}} = \mathsf{S}^1_{\mathsf{join}}.B^1_{m_1}\\
&\ldots\\
&\texttt{LEFT OUTER JOIN}\ \mathsf{S}^k_{\mathsf{join}}\ \texttt{ON}\\
&\quad \mathsf{R}.A_{\alpha^k_1} = \mathsf{S}^k_{\mathsf{join}}.B^k_1\ \texttt{AND}\ \ldots\ \texttt{AND}\ \mathsf{R}.A_{\alpha^k_{m_k}} = \mathsf{S}^k_{\mathsf{join}}.B^k_{m_k}\\
&\texttt{WHERE}\ \mathsf{S}^1_{\mathsf{join}}.B^1_1\ \texttt{IS NULL OR}\ \ldots\ \mathsf{S}^1_{\mathsf{join}}.B^1_{m_1}\ \texttt{IS NULL OR}\\
&\qquad\quad \mathsf{S}^2_{\mathsf{join}}.B^2_1\ \texttt{IS NULL OR}\ \ldots\ \mathsf{S}^2_{\mathsf{join}}.B^2_{m_2}\ \texttt{IS NULL OR}\\
&\qquad\quad \ldots\\
&\qquad\quad \mathsf{S}^k_{\mathsf{join}}.B^k_1\ \texttt{IS NULL OR}\ \ldots\ \mathsf{S}^k_{\mathsf{join}}.B^k_{m_k}\ \texttt{IS NULL}
\end{aligned}
$$

The *inconsistent blocks* represented by the keys found by the above three queries are *combined* using `UNION ALL` (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.

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

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 as $\mathsf{R}_{\mathsf{join}}$ 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 attributes of q to be projected and let **CompPredicate**(q) be the comparison expression in the `WHERE` clause of q. We first produce the SQL query that computes the facts of ground*.

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

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 attributes in q_{τ_T} to be projected (i.e., corresponding to the variables in $\vec{u}_T$). Recall that $T_{\mathsf{join}}(\vec{v})$ and $T_{\mathsf{fkey}}(\vec{v})$ would be replaced with $T_{\mathsf{join}}(\vec{v}, \vec{u}_T)$ and $T_{\mathsf{fkey}}(\vec{v}, \vec{u}_T)$ 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 a rule that has some negated IDB containing a free variable corresponding to some attribute in ground (i.e., ground.A),

- if the rule is produced by **Rule 3,** in each `LEFT OUTER JOIN` with S^i_{join} we add the expression `ground.A =` $\mathsf{S}^i_{\mathsf{join}}.B$ connected by the `AND` operator, where B is an attribute to be projected in $\mathsf{S}^i_{\mathsf{join}}$. 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 =` $\mathsf{R}_{\mathsf{fkey}}$`.A`.

5.2 Improvements upon existing CQA systems

ConQuer [30] and Conquesto [39] are two other CQA systems targeting their own subclasses of **FO**-rewritable queries, both with noticeable performance issues. For a fair comparison with LinCQA, we implemented our own optimized version of both systems. Specifically, we complement Conquer presented in [30] 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[39] 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.

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 [22], 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 [23] 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} [30].

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 [27].

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 [26, 27, 44]. Specifically, we take the seven queries that are consistent first-order rewritable in [26, 27, 44]. These queries feature joins between primary-key attributes to foreign-key attributes and projection on non-key attributes:

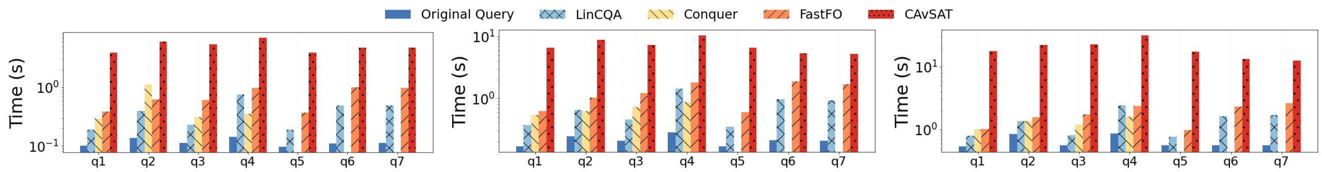
$$
\begin{aligned}
q_1(z) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_3(\underline{y}, v, w).\\
q_2(z, w) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_2(\underline{y}, v, w).\\
q_3(z) &\text{ :- } \mathrm{R}_1(\underline{x}, y, z), \mathrm{R}_2(\underline{y}, v, w), \mathrm{R}_7(\underline{v},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

The synthetic instances are generated in two phases. In the first phase, we generate the consistent instance, while in the second phase we inject inconsistency. We use the following parameters for data generation: (i) rSize: the number of tuples per relation, (ii) inRatio: the ratio of the number of tuples that violate primary key 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 $[1, \text{rSize}/10]$.

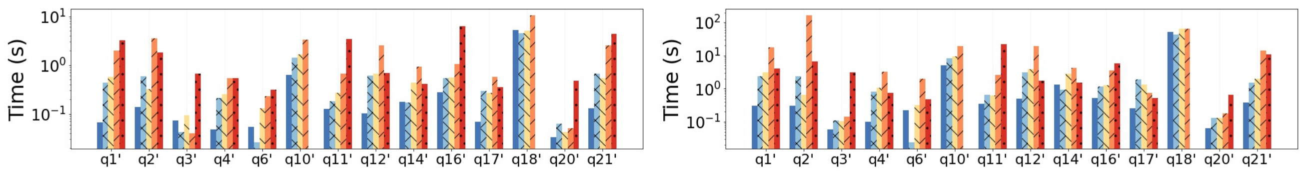
Inconsistency injection. In each relation, we first select a number of primary keys (or number of inconsistent blocks inBlockNum) from the generated consistent instance. Then, for each selected primary key, the inconsistency is injected by inserting the *same number of additional tuples* (bSize−1) into each block. The parameter inBlockNum is calculated by the given rSize, inRatio and bSize: inBlockNum = (inRatio · rSize)/bSize. We remark that there are alternative inconsistency injection methods available [2, 6].

6.2.2 TPC-H benchmark. We also altered the 22 queries from the original TPC-H benchmark [59] 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.



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

Figure 5: Performance comparison of different CQA systems on a synthetic workload with varying relation sizes.



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

Figure 6: Performance comparison of different CQA systems on the TPC-H benchmark with varying scale factor (SF).

Table 2: 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

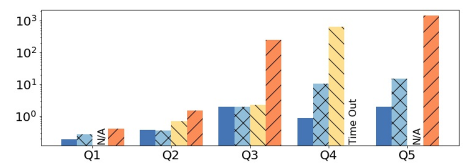


Figure 7: Runtime Comparison on StackOverflow

Table 3: 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`

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 2. 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 (or quasi-keys). Table 3 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.

[3]https://archive.org/details/stackexchange
[4]https://sedeschema.github.io/

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. Table 4 summarizes the sizes of consistent and inconsistent answers for each query in selected datasets.

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 [26, 27, 44]. We vary the size of each relation (rSize $\in \{500K, 1M, 5M\}$) in the synthetic data (Figure 5) and we evaluate on TPC-H database instances of scale factors 1 and 10 (Figure 6). Both figures include the time for running the original query on the inconsistent database (which returns the possible answers).

In the synthetic dataset, all three systems based on **FO**-rewriting techniques outperform CAvSAT, often by an order of magnitude. This observation 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 [27, 44]. Compared to ConQuer, LinCQA performs better or comparably on q_1 through q_4. LinCQA is also more efficient than ConQuer for q_1, q_2 and q_3. As the database size increases, the relative performance gap between LinCQA and ConQuer reduces for q_4. ConQuer cannot produce the SQL rewritings for queries q_5, q_6 and q_7 since they are not in C_{forest}. In summary, LinCQA is more efficient and at worst competitive to ConQuer on relatively small databases with less than $5M$ tuples, and is applicable to a wider class of acyclic 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

Table 4: The sizes of consistent and inconsistent answers for each query in selected datasets.

	Synthetic (rSize = 5M, inRatio = 10%, bSize = 2)							StackOverflow						
	q_1	q_2	q_3	q_4	q_5	q_6	q_7	Q_1	Q_2	Q_3	Q_4	Q_5		
cons. size	311573	463459	290012	408230	311434	277287	277135	27578	145	38320	3925	1245		
incons. size	571047	572244	534011	534953	574615	504907	474203	27578	145	38320	3925	1250		
	TPC-H (SF = 10)													
	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}
cons. size	4	28591	0	5	1	901514	289361	7	1	187489	1	13465732	3844	3776
incons. size	4	35206	0	5	1	1089754	318015	7	1	187495	1	16617583	4054	4010

is not, its performance is comparable to the other baselines. It is also worth noting that for most queries in the TPC-H benchmark, the overhead over running the SQL query directly is much smaller when 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. We also note that ConQuer and LinCQA exhibit comparable performance on most queries in TPC-H. 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 8. 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 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. Figure 11 and 12 in the extended version of this paper present the full results.

StackOverflow Dataset We use a 400GB StackOverflow dataset to evaluate the performance of different systems on large-scale real-world datasets. Another motivation to use such a large dataset is that LinCQA and ConQuer exhibit comparable performance on the medium-sized synthetic and TPC-H datasets. CAvSAT is excluded since it requires extra storage 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. In particular, when the database size is very large, LinCQA is much more scalable than ConQuer due to its more efficient strategy. We intentionally select queries with small inconsistent answer sizes for ease of experiments and presentation. Some queries with inconsistent answer size up to 1M would require hours to be executed and it is prohibitive to measure the performances of our baseline systems. 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.

To see the performance change of different systems when executing in small available memory, we run the experiments on a SQL server with maximum allowed memory of 120GB, 90GB, 60GB, 30GB, and 10GB respectively. Figure 9 shows that, despite the memory reduction, LinCQA is still the best performer on all five queries given different amounts of available memory. No obvious performance regression is observed on Q_1 and Q_2 when reducing memory since both queries access only two tables.

Summary Our experiments show that both LinCQA and ConQuer outperform FastFO and CAvSAT, systems that produce generic **FO**-rewritings and reduce to SAT respectively. Despite LinCQA and ConQuer showing a similar performance on most queries in our experiments, we observe that LinCQA is (1) applicable to a wider class of acyclic queries than ConQuer and (2) more scalable than ConQuer when the database size increases significantly.

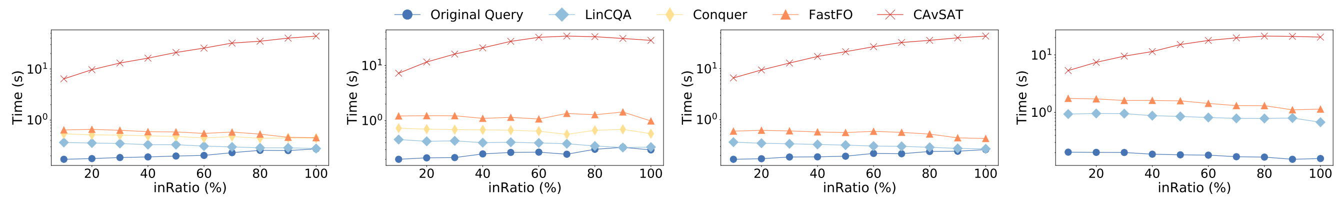
6.4 Worst-Case Study

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

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

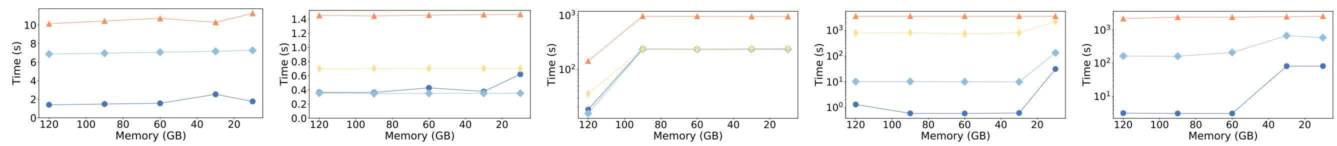
$$Q_{3-\text{path}}(x) \text{ :- } \mathsf{R}(\underline{x}, y), \mathsf{S}(\underline{y}, z), \mathsf{T}(\underline{z}, w).$$

We compare the performance of LinCQA with ConQuer and FastFO on both queries. CAvSAT does not finish its execution on



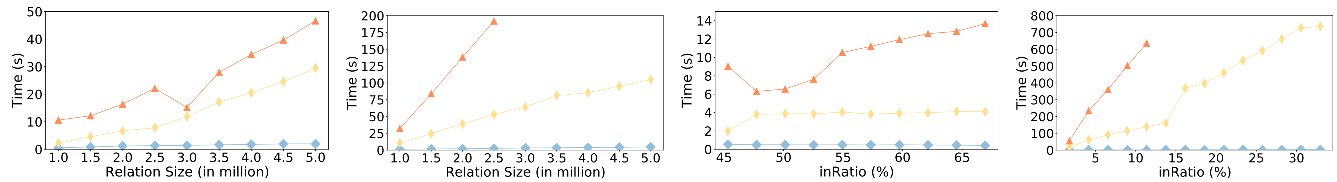
(a) q_1 (b) q_3 (c) q_5 (d) q_7

Figure 8: Performance of different systems on inconsistent database of varying inconsistent ratio



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4 (e) Q_5

Figure 9: Performance of StackOverflow queries on varying available memory



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

Figure 10: Performance comparison between different systems on varying relation sizes/inconsistent ratios

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 10a and 10b, 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 10b 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 10c and 10d 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 10d 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.

An interesting open question is whether CQA is in linear time for *all* acyclic self-join-free SPJ queries with an acyclic attack graph, including those that do not admit a PPJT. It would also be interesting to study the notion of PPJT for non-acyclic SPJ queries.

REFERENCES

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

[2] Lyublena Antova, Thomas Jansen, Christoph Koch, and Dan Olteanu. 2008. Fast and Simple Relational Processing of Uncertain Data. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). IEEE Computer Society, 983–992. https://doi.org/10.1109/ICDE.2008.4497507

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

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

[5] 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.

[6] Patricia C. Arocena, Boris Glavic, Giansalvatore Mecca, Renée J. Miller, Paolo Papotti, and Donatello Santoro. 2015. Messing Up with BART: Error Generation for Evaluating Data-Cleaning Algorithms. *Proc. VLDB Endow.* 9, 2 (2015), 36–47. https://doi.org/10.14778/2850578.2850579

[7] 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.

[8] 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.

[9] 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

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

[11] 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.

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

[13] 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.

[14] 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.

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

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

[17] 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

[18] 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.

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

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

[21] 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.

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

[23] 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.

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

[25] 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.

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

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

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

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5895)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.). Springer, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

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

[60] 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.

[61] Erhard Rahm and Hong Hai Do. 2000. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.* 23, 4 (2000), 3–13.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] Jef Wijsen. 2010. On the first-order expressibility of computing certain answers to conjunctive queries over uncertain databases. In *PODS*. ACM, 179–190.

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

[68] Jef Wijsen. 2019. Foundations of Query Answering on Inconsistent Databases. *SIGMOD Rec.* 48, 3 (2019), 6–16.

[69] 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 EFFICIENT CONSTRUCTION OF PPJT

Proposition A.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* $\mathsf{key}(F)$ *or* $\mathsf{key}(G)$ *is included in the other, then q has a PPJT that can be constructed in quadratic time in the number of atoms in q.*

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 $\mathsf{key}(G)$ and $\mathsf{key}(H)$ are not comparable by set inclusion.

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

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

We claim the following:

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

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

We distinguish two cases:

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

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

□

We remark that $\mathsf{CERTAINTY}(q)$ 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 [24, 25].

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

Proof. Let $\mathbf{db}$ be an instance for $\mathsf{CERTAINTY}(q)$. We define $X = \{a \mid R(\underline{c}, a) \in \mathbf{db}\}$ and $Y = \{b \mid S(\underline{c}, b) \in \mathbf{db}\}$. It is easy to see that $\mathbf{db}$ is a "yes"-instance for $\mathsf{CERTAINTY}(q)$ if and only if $X \times Y \subseteq T$, where $\times$ denotes the Cartesian product.

Next, consider the following algorithm that computes $X \times Y$ in linear time, exploiting that T is also part of the input to $\mathsf{CERTAINTY}(q)$.

1 Compute $X = \{a \mid R(\underline{c}, a) \in \mathbf{db}\}$ and $Y = \{b \mid S(\underline{c}, b) \in \mathbf{db}\}$
2 **if** $|X| \cdot |Y| > |T|$ **then**
3   **return** false
4 **return whether** $X \times Y \subseteq T$

Line 1 and 2 run in time $O(|R| + |S| + |T|)$. If the algorithm terminates at line 3, then the algorithm runs in linear time, or otherwise we must have $|R| \cdot |S| \leq |T|$, and the algorithm thus runs in time $O(|R| + |S| + |T| + |R| \cdot |S|) = O(|R| + |S| + |T|)$.

Note that q does not have a PPJT: in $q_1() \text{ :- } R(\underline{c}, x), T(\underline{x, y})$, R attacks T, and in $q_2() \text{ :- } S(\underline{c}, y), T(\underline{x, y})$, S attacks T. □

B MISSING PROOFS

In this section, we provide the missing proofs.

B.1 Proof of Proposition 4.2

Proof. 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 [48]. 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 [67], it must also attack U, a contradiction as well.

The proof is now complete by the classification theorem of [45]. □

B.2 Proof of Proposition 4.4

Proof. Let q be a query in C_{forest} and let G be the join graph of q as in Definition 6 of [31]. In particular, (i) the vertices of G are the atoms of q, and (ii) there is an arc from R to S if $R \neq S$ and there is some variable $w \in \mathsf{vars}(S)$ such that $w \in \mathsf{vars}(R) \setminus \mathsf{key}(R)$. By the definition of 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 $w \in \mathsf{vars}(R)$, $w \in \mathsf{vars}(S)$, and for any atom T_i in the unique path $R - T_1 - \cdots - T_k - S$, we have $w \notin \mathsf{vars}(T_i)$. We must have $w \in \mathsf{key}(R)$ and $w \in \mathsf{key}(S)$, or otherwise there would be an arc between R and S, a contradiction. From the property 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, $w \in \mathsf{vars}(T_1)$, a contradiction to our assumption.

Claim 2: the forest $\tau_1 \cup \cdots \cup \tau_n$ can be extended to a join tree τ of q. To show this, we will show that $\tau_1 \cup \cdots \cup \tau_n$ corresponds to a partial join tree as constructed by the GYO ear-removal algorithm. Indeed, suppose that atom T is a child of atom T' in τ_i. Then, T was an ear while constructing τ_i for q_{τ_i}, with T' as its witness. Recall that this means that if a variable x is not exclusive in T, then $x \in T'$. We will show that this is a valid ear removal step for q as well. Indeed, consider an exclusive variable x in T for q_{τ_i} that does not remain exclusive in q. Then, x occurs in some other tree τ_j. We will now use the fact that, by Lemma 2 of [31], 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. □

B.3 Proof of Theorem 4.5

In this section, we prove Theorem 4.5.

Definition 2. *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* $R_{\mathsf{gkey}}(q, \mathbf{db})$, *as follows:*

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

Let q be a self-join-free acyclic BCQ with a PPJT (τ, R). Lemma B.1 implies that in order to solve CERTAINTY(q), it suffices to check whether $R_{\mathsf{gkey}}(q, \mathbf{db}) \neq \emptyset$.

Lemma B.1. *Let q be a self-join-free acyclic BCQ with a PPJT (τ, R). Let* **db** *be a database instance for* CERTAINTY(q). *Then the following statements are equivalent:*

(1) **db** *is a "yes"-instance for* CERTAINTY(q); *and*
(2) $R_{\mathsf{gkey}}(q, \mathbf{db}) \neq \emptyset$.

Proof. By Lemma 4.4 in [48], we have that **db** is a "yes"-instance for CERTAINTY(q) if and only if there exists a sequence of constant $\vec{c}$ such that **db** is a "yes"-instance for CERTAINTY$(q_{[\vec{x}\to\vec{c}]})$, and the latter is equivelant to $R_{\mathsf{gkey}}(q, \mathbf{db}) \neq \emptyset$ by Definition 2. □

Example B.2. The atom Employee$(\underline{x}, y, z)$ is unattacked in q^{ex}. Observe that for employee_id = 0022, no matter whether we choose the tuple Employee(0022, New York, New York) or the tuple Employee(0022, Chicago, Chicago) in a repair, the chosen tuple will join with some corresponding tuple in the Manager and Contact table. The query $q^{\mathsf{ex}}_{[x\to 0022]}$ will then return `True` for all repairs of database **Company**, and $0022 \in \mathsf{Employee}_{\mathsf{gkey}}(q^{\mathsf{ex}}, \mathbf{Company}) \neq \emptyset$. The **Company** database is then concluded to be a "yes"-instance for CERTAINTY(q^{ex}) by Lemma B.1.

We remark that converse also holds: if **Company** is known to be a "yes"-instance for CERTAINTY(q^{ex}), then by Lemma B.1, the set $\mathsf{Employee}_{\mathsf{gkey}}(q^{\mathsf{ex}}, \mathbf{Company})$ must also be nonempty. □

Lemma B.3. *Let $R(\underline{\vec{x}}, \vec{y})$ be an atom in an acyclic self-join-free BCQ q with a PPJT (τ, R). Let* **db** *be an instance for* 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 **db** *is non-empty and for every fact $R(\underline{\vec{c}}, \vec{d})$ in* **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 (i) all facts $S(\underline{\vec{s}}, \vec{t})$ agree on the joining variables in $\mathsf{vars}(R) \cap \mathsf{vars}(S)$ and (ii) for every fact $S(\underline{\vec{s}}, \vec{t})$ in **db**, *the pair $\{R(\underline{\vec{c}}, \vec{d}), S(\underline{\vec{s}}, \vec{t})\}$ satisfies the BCQ () :- $R(\underline{\vec{x}}, \vec{y}), S(\underline{\vec{u}}, \vec{v})$, where $S(\underline{\vec{u}}, \vec{v})$ 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 **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 **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 **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 B.1, **db** is a "no"-instance for CERTAINTY(q_S), CERTAINTY(q) and thus 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 CERTAINTY(q_S). Indeed, suppose otherwise that $\mathbf{db}'$ is a "yes"-instance for CERTAINTY(q_S), then there exists some $\vec{s}$ such that $\mathbf{db}'$ is a "yes"-instance for 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 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 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 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 Theorem 4.5. It suffices to present rewriting rules to compute each $R_{\mathsf{gkey}}(q, \mathbf{db})$ for each atom R in q by Lemma B.3 , and show that these rewriting rules are equivalent to those presented in Section 4.2, which are shown to run in linear time. We denote R_{gk} as the Datalog predicate for $R_{\mathsf{gkey}}(q, \mathbf{db})$. It is easy to see that **Rule 1** computes all blocks of R violating item 2a of Lemma B.3.

To compute the blocks of R violating item 2b, we denote R_{gki} as the predicate for the subset of R_{gk} that satisfies condition (i) of item 2b. For each child $S(\underline{\vec{u}}, \vec{v})$ of R in a PPJT, let $\vec{w}$ be a sequence of all variables in $\mathsf{vars}(R) \cap \mathsf{vars}(S)$. The following rules find all blocks in R that violate condition (ii) of item 2b.

$$S_{\mathsf{join}}(\vec{w}) \text{ :- } S(\underline{\vec{u}}, \vec{v}), S_{\mathsf{gki}}(\vec{u}). \tag{3}$$

$$R_{\mathsf{fkey}}(\vec{x}) \text{ :- } R(\vec{x}, \vec{y}) \neg S_{\mathsf{join}}(\vec{w}). \tag{4}$$

We then compute the predicate R_{gk} denoting $R_{\mathsf{gkey}}(q, \mathbf{db})$ with

$$R_{\mathsf{gk}}(\vec{x}) \text{ :- } R(\underline{x}, \vec{y}), \neg R_{\mathsf{fkey}}(\vec{x}). \tag{5}$$

If R has a parent, then we may compute all blocks in R violating condition (i) of item 2b using the following rules for every variable at the i-th position of $\vec{y}$,

$$R_{\mathsf{gk}^\neg}(\vec{x}) \text{ :- } R(\vec{x}, \vec{y}), R(\vec{x}, \vec{y}'), y_i \neq y_i'. \tag{6}$$

$$R_{\mathsf{gki}}(\vec{x}) \text{ :- } R_{\mathsf{gk}}(\vec{x}), \neg R_{\mathsf{gk}^\neg}(\vec{x}). \tag{7}$$

Now we explain why Rules (3) through (7) are equivalent to **Rule 2, 3, 4**. First, the head $R_{\mathsf{gk}^\neg}$ of Rule (6) can be safely renamed to R_{fkey} and yield **Rule 2**. Rule (4) is equivalent to **Rule 3**. Finally, Rules (3), (5) and (7) can be merged to **Rule 4** since to compute each S_{join}, we only need $S(\vec{u}, \vec{v})$ and S_{fkey}. Note that the Rule (3) for the atom R will have 0 arity if R is the root of the PPJT. □

B.4 Proof of Lemma 4.9

Proof. 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 CERTAINTY($q_{[\vec{u}\to\vec{c}]}$). For the other direction, we assume that $\mathbf{db}$ is "yes"-instance for CERTAINTY($q_{[\vec{u}\to\vec{c}]}$). Let r be any repair of $\mathbf{db}$. Then there is a valuation μ with $\mu(q_{[\vec{u}\to\vec{c}]}) \subseteq r$. Let θ be the valuation with $\theta(\vec{u}) = \vec{c}$. Consider the valuation

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

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

B.5 Proof of Theorem 4.10

Proof. Our algorithm first evaluates q on $\mathbf{db}$ to yield a set S of size $|\mathsf{OUT}_p|$ in time $O(N \cdot |\mathsf{OUT}_p|)$. Here the set S must contain all the consistent answers of q on $\mathbf{db}$. By Lemma 4.9, we then return all answers $\vec{c} \in S$ such that $\mathbf{db}$ is a "yes"-instance for CERTAINTY($q_{[\vec{u}\to\vec{c}]}$), which runs in $O(N)$ by Theorem 4.5. 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 (i) removing all blocks with at least two tuples from $\mathbf{db}$ to yield $\mathbf{db}^c$ and (ii) evaluating q on $\mathbf{db}^c$. It suffices to show that every consistent answer to q on $\mathbf{db}$ is an answer to q on $\mathbf{db}^c$. Assume that $\vec{c}$ is a consistent answer to q on $\mathbf{db}$. Consider $q_{[\vec{u}\to\vec{c}]}$, a disconnected CQ where $\vec{u}$ is a sequence of all variables in q. Its **FO**-rewriting effectively contains **Rule 1** for each atom in $q_{[\vec{u}\to\vec{c}]}$, which is equivalent to step (i), and then checks whether $\mathbf{db}^c$ satisfies $q_{[\vec{u}\to\vec{c}]}$ by Lemma B.1 of [45]. By Lemma 4.9, $\mathbf{db}$ is a "yes"-instance for CERTAINTY($q_{[\vec{u}\to\vec{c}]}$), and thus the **FO**-rewriting concludes that $\mathbf{db}^c$ satisfies $q_{[\vec{u}\to\vec{c}]}$, i.e. $\vec{c}$ is an answer to q on $\mathbf{db}^c$. In our algorithm, step (i) runs in $O(N)$ and since q is full, step (ii) runs in time $O(N + |\mathsf{OUT}_c|)$. □

C IMPROVEMENT UPON EXISTING SYSTEMS

C.1 Conquer

Fuxman and Miller [31] 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 [30]. 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 [31], enabling us to produce the consistent SQL rewriting for every query in C_{forest}.

C.2 Conquesto

Conquesto [39] produces a non-recursive Datalog program that implements the algorithm in [48], 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 (8) and (10) share the common predicate $\mathsf{R}_2(y, v, w)$ in their bodies, resulting in re-computation, and Rule (12) involves a Cartesian product.

$$\mathsf{Sr}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{R}_2(y, v, w). \tag{8}$$

$$\mathsf{Yes}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{Sr}_{\mathsf{R}_2}(y), \mathsf{R}_2(y, v, w). \tag{9}$$

$$\mathsf{Sr}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{R}_1(x, y, z), \mathsf{R}_2(y, v, w). \tag{10}$$

$$\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, v_2), \mathsf{Yes}_{\mathsf{R}_2}(y), v_2 = z. \tag{11}$$

$$\mathsf{Bb}_{\mathsf{R}_1}(x, z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, v_2), \neg\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z). \tag{12}$$

$$\mathsf{Yes}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{Sr}_{\mathsf{R}_1}(z), \mathsf{R}_1(x, y, z), \neg\mathsf{Bb}_{\mathsf{R}_1}(x, z). \tag{13}$$

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.

$$\mathsf{Sr}(x, y, z) \text{ :- } \mathsf{R}_1(x, y, z), \mathsf{R}_2(y, v, w). \tag{14}$$

$$\mathsf{Yes}_{\mathsf{R}_2}(y) \text{ :- } \mathsf{Sr}(x, y, z), \mathsf{R}_2(y, v, w). \tag{15}$$

$$\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z) \text{ :- } \mathsf{Sr}(x, \_, z), \mathsf{R}_1(x, y, v_2), \mathsf{Yes}_{\mathsf{R}_2}(y), v_2 = z. \tag{16}$$

$$\mathsf{Bb}_{\mathsf{R}_1}(x, z) \text{ :- } \mathsf{Sr}(x, \_, z), \mathsf{R}_1(x, y, v_2), \neg\mathsf{Gf}_{\mathsf{R}_1}(v_2, x, y, z). \tag{17}$$

$$\mathsf{Yes}_{\mathsf{R}_1}(z) \text{ :- } \mathsf{Sr}(x, y, z), \mathsf{R}_1(x, y, z), \neg\mathsf{Bb}_{\mathsf{R}_1}(x, z). \tag{18}$$

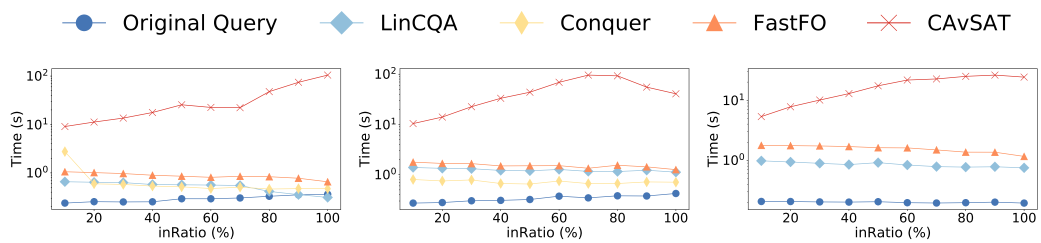
For evaluation, the rules computing each intermediate relation (i.e. all rules except for the one computing $\mathsf{Yes}_{\mathsf{R}_1}(z)$) are then translated to a SQL subquery via a `WITH` clause.

D ADDITIONAL TABLES AND FIGURES

Figure 11 presents the performance of queries q_2, q_4 and q_6 on varying inconsistency ratios, supplementing Figure 8. Figure 12 summarizes the performance of all seven synthetic queries on varying block sizes.

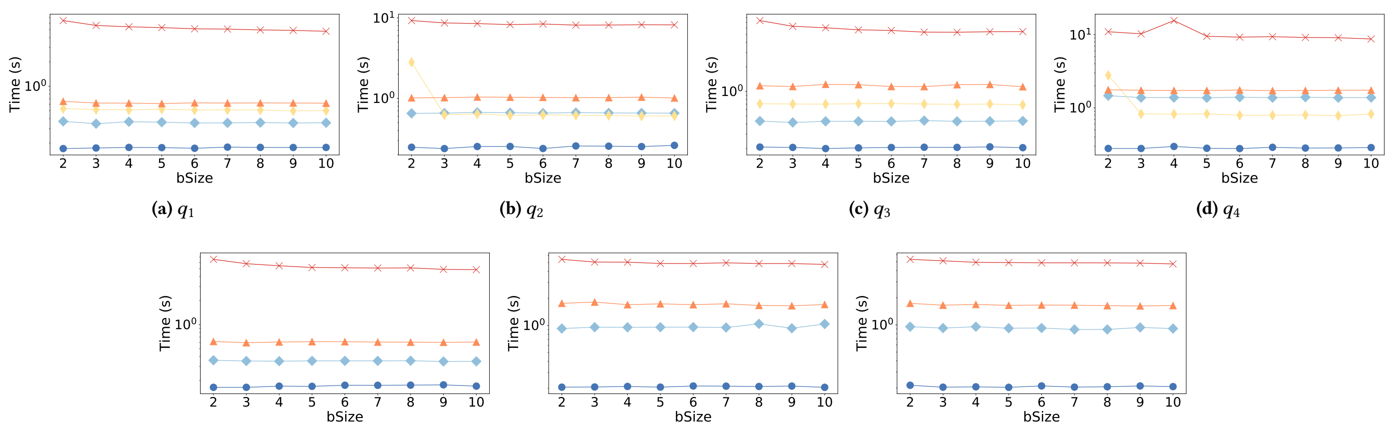
Table 5: A summary of the Stackoverflow Dataset.

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 11: 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 12: Performance of different systems on inconsistent database of varying block size