Computing the Difference of Conjunctive Queries Efficiently

XIAO HU, University of Waterloo, Canada
QICHEN WANG, Hong Kong Baptist University, Hong Kong

We investigate how to efficiently compute the difference result of two (or multiple) conjunctive queries, which is the last operator in relational algebra to be unraveled. The standard approach in practical database systems is to materialize the results for every input query as a separate set, and then compute the difference of two (or multiple) sets. This approach is bottlenecked by the complexity of evaluating every input query individually, which could be very expensive, particularly when there are only a few results in the difference. In this paper, we introduce a new approach by exploiting the structural property of input queries and rewriting the original query by pushing the difference operator down as much as possible. We show that for a large class of difference queries, this approach can lead to a linear-time algorithm, in terms of the input size and (final) output size, i.e., the number of query results that survive from the difference operator. We complete this result by showing the hardness of computing the remaining difference queries in linear time. Although a linear-time algorithm is hard to achieve in general, we also provide some heuristics that can provably improve the standard approach. At last, we compare our approach with standard SQL engines over graph and benchmark datasets. The experiment results demonstrate order-of-magnitude speedups achieved by our approach over the vanilla SQL engine.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: conjunctive query, query optimization, difference operator

ACM Reference Format:
Xiao Hu and Qichen Wang. 2023. Computing the Difference of Conjunctive Queries Efficiently. *Proc. ACM Manag. Data* 1, 2, Article 153 (June 2023), 26 pages. https://doi.org/10.1145/3589298

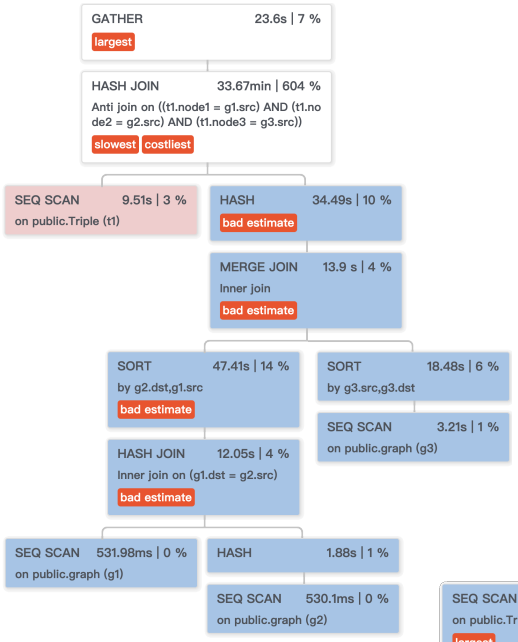
1 INTRODUCTION

Conjunctive queries with aggregation, union, and difference (also known as negation) operators form the full relational algebra [12]. While conjunctive queries [15, 16, 24, 29, 33, 38], with aggregation [30] and unions [21, 23], have been extensively studied in the literature, the difference operator received much less attention. In modern database systems, there are several different equivalent expressions for computing the difference between two queries, such as NOT IN, NOT EXIST, EXCEPT, MINUS, DIFFERENCE, and LEFT-OUTER JOIN followed by a non-NULL filter. In contrast to its powerful expressibility, the execution plan of difference operator in existing database systems or data analytic engines (e.g., MySQL [2], Oracle [3], Postgre SQL [4], Spark SQL [6]) is quite brute-force. Given two (or multiple) conjunctive queries, their difference is simply done by materializing the answers for each participated conjunctive query separately, and then computing the difference of two (or multiple) sets. Hashing or other indexes may be built on top of the query answers to speed up the computation of the set difference at last. However, this approach is severely

Authors' addresses: Xiao Hu, xiaohu@uwaterloo.ca, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada, N2L 3G1; Qichen Wang, qcwang@hkbu.edu.hk, Hong Kong Baptist University, 224 Waterloo Road, Kowloon Tong, Hong Kong, Hong Kong.

2836-6573/2023/6-ART153 $15.00
https://doi.org/10.1145/3589298

(a) Execution Plan for Original Query



(b) Execution Plan for Rewritten Query

Fig. 1. Execution plans for SQL queries in Example 1.1 generated by PEV(https://tatiyants.com/pev/). Blocks in red indicate Q_1 in Q and Q'. Blocks in blue of (a) indicate Q_2 in Q, and that of (b) indicate Q_3 in Q'. Blocks in white indicate the difference operator for both Q and Q'.

bottlenecked by evaluating every input query individually and materializing a large number of intermediate query results that do not contribute to the final results due to the difference operator.

Let's consider an example of friend recommendation in social networks (such as Twitter, Facebook, Sina Weibo). A friend recommendation is represented as a triple (a, b, c) extracted from the network semantics, such that user c is recommended to user a since user b is a friend of user a and user c is a friend of user b, together with other customized constraints. We also avoid the recommendation when a and c are already friends. The task of finding all valid recommendations can be captured by a SQL query in Example 1.1, as the difference of two sub-queries.

Example 1.1. Let Graph(src, dst) be a table storing all edges in the social network, and Triple(node1, node2, node3) be a table storing all candidate recommendations. The following SQL query (Q) finds all triples from Triple that do not form a triangle in the graph:

```
Q : SELECT node1, node2, node3 FROM Triple t1
    WHERE NOT EXISTS
    (SELECT * FROM Graph g1, Graph g2, Graph g3
     WHERE g1.dst = g2.src and g2.dst = g3.src and g3.dst = g1.src and g1.src = t1.node1
           and g2.src = t1.node2 and g3.src = t1.node3);
```

such that Q is the difference of two sub-queries Q_1 and Q_2, where Q_1 returns all candidate recommendations from Triple and Q_2 returns all triangle friendship in the social network.

```
Q1 : SELECT node1, node2, node3 FROM Triple t1
Q2 : SELECT * FROM Graph g1, Graph g2, Graph g3
     WHERE g1.dst = g2.src and g2.dst = g3.src and g3.dst = g1.src and g1.src = t1.node1
           and g2.src = t1.node2 and g3.src = t1.node3;
```

We note that Q can be rewritten as the following SQL query Q':

```
Q' : SELECT node1, node2, node3 FROM Triple t1
     WHERE NOT EXISTS
    (SELECT * FROM Triple t2
     WHERE EXISTS (SELECT * FROM graph g1 WHERE t2.node1 = g1.src and t2.node2 = g1.dst)
     AND EXISTS (SELECT * FROM graph g2 WHERE t2.node2 = g2.src and t2.node3 = g2.dst)
     AND EXISTS (SELECT * FROM graph g3 WHERE t2.node3 = g3.src and t2.node1 = g3.dst)
     AND t2.node1 = t1.node1 and t2.node2 = t1.node2 and t2.node3 = t1.node3)
```

such that Q' is the difference of Q_1 and another sub-query Q_3, where Q_3 finds all candidate recommendations in Triple that also form a triangle in the social network:

```
Q3 : SELECT * FROM Triple t2
     WHERE EXISTS (SELECT * FROM graph g1 WHERE t2.node1 = g1.src and t2.node2 = g1.dst)
     And EXISTS (SELECT * FROM graph g2 WHERE t2.node2 = g2.src and t2.node3 = g2.dst)
     And EXISTS (SELECT * FROM graph g3 WHERE t2.node3 = g3.src and t2.node1 = g3.dst)
     AND t2.node1 = t1.node1 and t2.node2 = t1.node2 and t2.node3 = t1.node3
```

Figure 1(a) illustrates the execution plan for Q generated by PostgreSQL optimizer. It first materializes all triangles in the graph as Q_2, and then computes the difference of Q_1 and Q_2 by anti-join. Moreover, hashing index is built on top of all triangles of Q_2 so that the anti-join can be executed by checking whether every candidate recommendation in Q_1 appears as a triangle in Q_2. At last, all "survived" recommendations are outputted as final answers. In plan (a), computing the set difference at last is the most time-consuming step, which is predicted to take 33.67 minutes by PostgreSQL optimizer. Although computing the subquery Q_2 is not that expensive, which only takes about 132 seconds, the number of intermediate results materialized for Q_2 is quite large as expected, which finally leads to the inefficiency of the subsequent computation on $Q_1 - Q_2$. In Section 6, plan (a) actually runs in 308.175 seconds in practice.

To tackle the challenges brought by the difference operator, we take two input sub-queries as a whole into account for algorithm design. We are interested in efficient algorithms with running times linear in the final result size. This requirement rules out the standard approach of materializing the results for each input sub-query separately and then computing their set difference. Indeed, the final output size can be many magnitudes smaller than the number of intermediate results that materialized. To overcome the curse of large intermediate results, we introduce a rewriting-based approach by *exploiting the joint structural properties of two input sub-queries* and *pushing the difference operator down as far as possible.*

In Example 1.1, we can rewrite the original SQL query Q into a new one Q'. Instead of computing Q_2, it finds all candidate recommendations that also form a triangle in the social network as Q_3, which is exactly the intersection of Q_1 and Q_2, and then computes the difference of Q_1 and Q_3. Figure 1(b) illustrates the execution plan of this new query. We observe that computing the set difference at last is predicated to only take 21.67 seconds, which is much faster than (a). This is as expected, since the number of intermediate results generated by Q_3 is much smaller than Q_2, after

taking Q_1 into consideration, which is the key to the overall improvement. As the price, computing Q_3 is predicated to take a few more minutes than Q_2, but this is totally tolerable. In Section 6, plan (b) actually runs in 78.918 seconds, which already achieves 4x speedup over (a). This significant improvement from (a) to (b) motivates us to further investigate this interesting problem for general queries.

Our contributions. In this paper, we formulate the *difference of conjunctive queries* (DCQ) problem and study the data complexity of this problem. Our contributions can be summarized as:

- **Complexity Dichotomy:** We give a dichotomy for computing DCQs in linear time in terms of input and output size. We characterize a class of "easy" DCQs exploiting the joint properties of two input CQs, and present a linear-time algorithm. On the other hand, we prove the hardness of obtaining a linear-time algorithm for the remaining "hard" DCQs via several well-known conjectures. **(Section 3 and 4.1)**
- **Efficient Heuristic:** We propose an efficient heuristic for computing "hard" DCQs, which does not lead to a linear-time algorithm but still improves the baseline approach greatly. The heuristic investigates the *intersection* of two input CQs and incorporates the state-of-the-art algorithms for CQ evaluation. **(Section 4.2)**
- **Extension:** We explore several interesting extensions. First, we design a recursive algorithm for computing the difference of *multiple* conjunctive queries. We also extend our algorithm to support other relational operators, such as selection, projection, join, and aggregation. At last, we investigate the DCQ problem under the bag semantics. **(Section 5)**
- **Experimental Evaluation:** We provide an experimental evaluation of our approach and standard approach on real-world datasets in both centralized and parallel database systems. The experimental results show that our approach out-performs the baseline on different classes of queries and datasets. **(Section 6)**

Roadmap. In Section 2, we formally define the DCQ problem and review the literature on evaluating a single CQ. In Section 3, we provide a linear-time algorithm for "easy" DCQs. In Section 4, we prove the hardness for the remaining DCQs and show efficient heuristics. In Section 5, we study several extensions of DCQs with other relational operators and bag semantics. In Section 6, we present the experimental evaluation. At last, we review related work in Section 7 and Section 8.

2 PRELIMINARIES

2.1 Problem Definition

Conjunctive Query (CQ). We consider the standard setting of multi-relational databases. Let $\mathbb{R}$ be a database schema that contains n relations $R_1, R_2, \cdots, R_n$. Let $\mathcal{V}$ be the set of attributes in the database $\mathbb{R}$. Each relation R_i is defined on a subset of attributes $e_i \subseteq \mathcal{V}$. Let $\mathcal{E} = \{e_1, e_2, \cdots, e_n\}$ be the set of the attributes for all relations. Let $\mathrm{dom}(x)$ be the domain of attribute $x \in \mathcal{V}$, and let $\mathrm{dom}(U) = \prod_{x \in U} \mathrm{dom}(x)$ be the domain of attributes $U \subseteq \mathcal{V}$.

Given the database schema $\mathbb{R}$, let an input instance be D, and the corresponding instances of $R_1, \cdots, R_n$ be $R_1^D, \cdots, R_n^D$. Where D is clear from the context, we will drop the superscript and use $R_1, \cdots, R_n$ for both the schema and instances. Any tuple $t \in R_i$ is defined on e_i. For any attribute $x \in e_i$, $\pi_x t \in \mathrm{dom}(x)$ denotes the value of attribute x in tuple t. Similarly, for a set of attributes $U \subseteq e_i$, $\pi_U t$ denotes the values of attributes in U for t with an implicit ordering on the attributes.

We consider the class of *conjunctive queries without self-joins* formally defined as

$$Q := \pi_{\mathbf{y}} \left(\sigma_{\phi_1} R_1(e_1) \bowtie \cdots \sigma_{\phi_2} R_2(e_2) \bowtie \cdots \bowtie \sigma_{\phi_n} R_n(e_n) \right), \tag{1}$$

where σ is the selection operator, ϕ_i is the predicate defined over relation R_i, $\sigma_{\phi_i} R_i(e_i)$ selects out tuples from R_i passing the predicate ϕ_i and $\mathbf{y} \subseteq \mathcal{V}$ denotes the *output attributes*. If $\mathbf{y} = \mathcal{V}$, such

a CQ query is known as *full join*, which represents the natural join of the underlying relations. We usually use a triple $(\mathbf{y}, \mathcal{V}, \mathcal{E})$ to represent a CQ Q, and simply use a pair $(\mathcal{V}, \mathcal{E})$ to represent a full join. Each relation R_i in Q is distinct, i.e., the CQ does not have a self-join. As a simplification, we ignore all the selection operators since it just takes $O(1)$ time to decide if a tuple passes the predicate ϕ_i. Moreover, we assume every R_i is defined on different subset of attributes; otherwise, we can simply keep the intersection of all relations defining on the same subset of attributes. Hence, we also use R_e to denote the relation defined on $e \in \mathcal{E}$.

The result of Q over instance D noted as $Q(D)$, is defined as:

$$Q(D) = \left\{ t \in \text{dom}(\mathbf{y}) : \exists t' \in \text{dom}(\mathcal{V}), s.t. \pi_{\mathbf{y}} t' = t, \pi_{e_i} t \in R_i, \forall i \in [n] \right\},$$

i.e., the projection of all combinations of tuples from every relation onto $\mathbf{y}$, such that tuples in each combination have the same value(s) on the common attribute(s). Let $N = |D|$ be the input size, i.e., the total number of tuples in the input instance. Let $\text{OUT} = |Q(D)|$ be the output size, i.e., the number of query results of Q over D.

Difference of Conjunctive Queries (DCQ). A DCQ $Q_1 - Q_2$ consists of two CQs without self-joins Q_1, Q_2 with the same output attributes. We also assume that the DCQ does not have a self-join, i.e., there exists no pair of relations R_i from Q_1 and R_j from Q_2 such that R_i, R_j are the same. Note that our algorithms presented in this work also applied to the case when self-join exists in DCQ, but our lower bound assumes that no self-join exists. The input to a DCQ $Q_1 - Q_2$ is a pair of database instances D_1, D_2 defined for Q_1, Q_2 respectively[1]. The result of $Q_1 - Q_2$ over D_1, D_2 is $Q_1(D_1) - Q_2(D_2)$, i.e., tuples that appear in the result of Q_1 over instance D_1, but not in the result of Q_2 over instance D_2. Let $N = |D_1| + |D_2|$ be the input size, i.e., the total number of tuples in both input instances. Let $\text{OUT} = |Q_1(D_1) - Q_2(D_2)|$ be the output size.

In this paper, we adopt standard data complexity [36]; that is, we measure the complexity of algorithms with input size N and output size OUT, and assume the query size as a constant.

2.2 Literature Review of CQ Evaluation

Before diving into the massive literature, we mention two classes of CQs that play an important role in query evaluation.

- **(α-acyclic).** A CQ $Q = (\mathbf{y}, \mathcal{V}, \mathcal{E})$ is α-acyclic [17, 25] if there exists a tree $\mathcal{T}$ (called a *join tree*) such that each node in $\mathcal{T}$ corresponds to a relation in $\mathcal{E}$, and for each attribute $x \in \mathcal{V}$, the set of nodes containing x form a connected subtree of $\mathcal{T}$. Moreover, we define $\text{top}(x)$ as the highest node of $\mathcal{T}$ that attribute x appears.
- **(free-connex).** A CQ $Q = (\mathbf{y}, \mathcal{V}, \mathcal{E})$ is *free-connex* [16] if there exists a tree $\mathcal{T}$ (called a *free-connex join tree*) such that $\mathcal{T}$ is a join tree for Q, and for any pair of attributes $x_1 \in \mathbf{y}, x_2 \in \mathcal{V} - \mathbf{y}$, $\text{top}(x_2)$ is not an ancestor of $\text{top}(x_1)$. It has been proved equivalently that a CQ $Q = (\mathbf{y}, \mathcal{V}, \mathcal{E})$ is *free-connex* if Q is α-acyclic and $(\mathbf{y}, \mathcal{V}, \mathcal{E} \cup \{\mathbf{y}\})$ is also α-acyclic.

Their relationships are illustrated in Figure 2. A free-connex CQ must be α-acyclic. An α-acyclic full join must be free-connex. Below, when the context is clear, we always refer "acyclic" to "α-acyclic".

There has been a long line of research on CQ evaluation [17, 22, 31, 34, 36]. Yannakakis's seminal algorithm [17] was proposed for acyclic CQs, whose running time differs over different sub-classes of acyclic CQs. A free-connex CQ can be evaluated in $O(N + \text{OUT})$ time, which is already optimal since any algorithm needs to read input data and output all query results. On the other hand, an acyclic but non-free-connex CQ can be evaluated in $O(N \cdot \text{OUT})$ time. Subsequent works have progressively defined different notions of "width" [26, 27], measuring how far a query is from being

[1]We distinguish the input instances D_1, D_2 of Q_1, Q_2 for simplifying algorithmic description later, which is different from conventional definition.

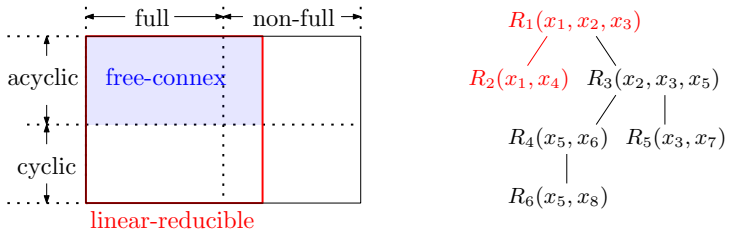


Fig. 2. Left is the classification of CQs via α-cyclic/cyclic and full/non-full metrics. Dashed lines indicate the boundaries of different classes. Right is a join tree of an α-acyclic full CQ $Q = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \{x_1, x_2, \cdots, x_8\}$ and $\mathcal{E} = \{e_1 = \{x_1, x_2, x_3\}, e_2 = \{x_1, x_4\}, e_3 = \{x_2, x_3, x_5\}, e_4 = \{x_5, x_6\}, e_5 = \{x_3, x_7\}, e_6 = \{x_5, x_8\}\}$. It is also a free-connex join tree for non-full CQ $Q' = (\mathbf{y}, \mathcal{V}, \mathcal{E})$ with $\mathbf{y} = \{x_1, x_2, x_3, x_4\}$, but is not a free-connex join tree with $\mathbf{y} = \{x_1, x_2, x_5\}$, since $\mathsf{top}(x_3) = R_1$ is an ancestor of $\mathsf{top}(x_5) = R_3$. The subtree in red is the reduced query of Q' after running Algorithm 1.

acyclic and tackle cyclic queries with decomposition. This line of algorithms run in $O(N^w + \text{OUT})$ time, where w can be the fractional hypertree width [26, 33], submodular width [14], or FAQ-width [14]. In addition, some specific classes of CQs can be speedup by fast matrix multiplication techniques [15, 18, 24], but we won't go into that direction further. CQ evaluation is still an actively researched problem; any improvement here will also improve DCQ evaluation when plugged into the baseline as well as our approach.

In the remaining, we often use $\mathtt{cost}(Q)$ to denote the time complexity of evaluating a CQ Q.

Implications to the Baseline Approach of DCQ Evaluation. Given a DCQ $Q_1 - Q_2$, the baseline approach of computing Q_1, Q_2 separately and then set difference incurs the following cost:

Corollary 2.1. *Given two CQs Q_1 and Q_2, the DCQ $Q_1 - Q_2$ can be computed in $O(\mathtt{cost}(Q_1) + \mathtt{cost}(Q_2))$ time.*

For example, when both Q_1 and Q_2 are free-connex, the baseline approach runs in $O(N + \text{OUT}_1 + \text{OUT}_2)$ time, where $\text{OUT}_1, \text{OUT}_2$ are the output sizes of Q_1, Q_2 respectively.

2.3 New Results of DCQ Evaluation

Our new complexity results for DCQ evaluation are summarized in Table 1. To help understand these results, we first introduce the class of *linear-reducible* CQs, and the *reduce* procedure.

Definition 2.2 (Linear-reducible). A CQ $Q = (\mathbf{y}, \mathcal{V}, \mathcal{E})$ is linear-reducible if $(\mathbf{y}, \mathcal{V}, \mathcal{E} \cup \{\mathbf{y}\})$ is free-connex.

The relationship between linear-reducible CQs and existing classifications of CQs is illustrated in Figure 2. Any full or free-connex CQ must be linear-reducible. In addition, some cyclic but non-full CQs are also linear-reducible, for example, $Q = \pi_{x_1,x_2,x_3}(R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_1, x_3) \bowtie R_4(x_3, x_4))$, since adding $R_5(x_1, x_2, x_3)$ will result in a free-connex CQ (see Figure 2). It is also noted that any non-free-connex but acyclic CQ is non-linear-reducible.

Moreover, we introduce a *reduce* procedure in Algorithm 1, that can transform any linear-reducible CQ into a full join query in $O(N)$ time, while preserving the query results. This algorithm is similar to the semi-join phase of Yannakakis algorithm [38]. Intuitively, we remove attributes or relations in a bottom-up ordering of nodes in a free-connex join tree. Recall that each node in the join tree corresponds to a relation in $\mathcal{E}$. When a node e is visited, we distinguish two more cases: (line 4-5) if its output attributes are fully contained in its parent, we remove e and update its parent relation via semi-joins; (line 6-7) and otherwise, we remove all non-output attributes (if

$Q_1 - Q_2$	Baseline	Our Approach
$Q_1 - Q_2$ is difference-linear	$\texttt{cost}(Q_1)$ + $\texttt{cost}(Q_2)$	$N + \text{OUT}$ [Theorem 3.1]
Q_2 is linear-reducible		$\texttt{cost}(Q_1)$ [Corollary 2.5]
Q_2 is non-linear-reducible		$\texttt{cost}(Q_1) + \min \begin{cases} \text{OUT}_1 \cdot \texttt{cost}(Q_2^{\emptyset}) & \text{[Theorem 4.8]} \\ \texttt{cost}(Q_2^{\oplus}) & \text{[Theorem 4.10]} \end{cases}$

Table 1. Summary of complexity results by baseline and our approach. $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$ are two input CQs. $(\mathbf{y}, \mathcal{E}_1')$ and $(\mathbf{y}, \mathcal{E}_2')$ are reduced queries of Q_1, Q_2 respectively. $Q_2^{\emptyset} = (\emptyset, \mathcal{V} - \mathbf{y}, \{e - \mathbf{y} : e \in \mathcal{E}_2\})$ and $Q_2^{\oplus} = (\mathbf{y}, \mathcal{V}_2, \{\mathbf{y}\} \cup \mathcal{E}_2)$ are formally defined in Section 4.2. N is the input size. OUT_1, OUT are the output sizes of $Q_1, Q_1 - Q_2$ respectively. $\texttt{cost}(\cdot)$ is the time complexity of evaluating a single CQ.

there exists any) in e via projections. The output of Algorithm 1 is a full join query $Q' = (\mathbf{y}, \mathcal{E})$ (called the *reduced query*) and an instance D' (called the *reduced instance*) such that $Q(D) = Q'(D')$. An example of reduced query is illustrated in Figure 2.

We are now ready to present the new results for DCQ evaluation.

Dichotomy for Linear-time Algorithm. Our main complexity result is a complete characterization of Q_1, Q_2 for which a linear algorithm can be achieved for computing $Q_1 - Q_2$:

Definition 2.3 (Difference-Linear). Given two CQs $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$, the DCQ $Q_1 - Q_2$ is difference-linear if Q_1 is free-connex, Q_2 is linear-reducible, and $(\mathbf{y}, \mathcal{E}_1' \cup \{e\})$ is α-acyclic for every $e \in \mathcal{E}_2'$, where $(\mathbf{y}, \mathcal{E}_1')$ and $(\mathbf{y}, \mathcal{E}_2')$ are the reduced queries of Q_1, Q_2 respectively.

Theorem 2.4 (Dichotomy). *Given two CQs $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$, the DCQ $Q_1 - Q_2$ can be computed in $O(N + \text{OUT})$ time if and only if it is difference-linear.*

Our proof of Theorem 2.4 consists of two steps. In Section 3, we prove the "if"-direction by designing a linear algorithm for the class of "easy" queries as characterized. In Section 4.1, we prove the "only-if" direction by showing the lower bound for the remaining class of "hard" queries, based on some well-established conjectures.

Improvement Achieved by Heuristics. For the class of "hard" DCQs on which obtaining a linear-time algorithm is hopeless, we further show some efficient heuristics that strictly improve the baseline. The complete results are presented in Section 4.2 and we mention an interesting case:

Corollary 2.5. *Given two CQs Q_1 and Q_2, if Q_2 is linear-reducible, then DCQ $Q_1 - Q_2$ can be computed in $O(\texttt{cost}(Q_1))$ time.*

We summarize all these results above in Table 1: (1) our approach strictly improves the baseline as long as Q_2 is linear-reducible; (2) furthermore, our approach leads to a linear-time algorithm if Q_1 also satisfies some specific conditions; (3) in the remaining case when Q_2 is non-linear-reducible, the comparison of our approach and baseline depends on specific queries or even input instances.

3 EASY DCQS

In this section, we show a linear-time algorithm for computing the class of "easy" DCQ characterized in Theorem 2.4. The main technique we used is simply query rewriting, but by exploiting the structures of two input queries in a non-trivial way.

Theorem 3.1. *Given two CQs $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$, if $Q_1 - Q_2$ is difference-linear, then DCQ $Q_1 - Q_2$ can be computed in $O(N + \text{OUT})$ time.*

Algorithm 1: REDUCE($Q = (\mathbf{y}, \mathcal{V}, \mathcal{E}), D$)

$\mathcal{T} \leftarrow$ the free-connex join tree of $(\mathbf{y}, \mathcal{V}, \mathcal{E} \cup \{\mathbf{y}\})$;
foreach $e \in \mathcal{E}$ *in a bottom-up way (excluding root) of* $\mathcal{T}$ **do**
  Let $e' \leftarrow$ the (unique) parent node of e;
  if $e \cap \mathbf{y} \subseteq e' \cap \mathbf{y}$ **then**
    $R_{e'} \leftarrow R_{e'} \ltimes R_e$ **and** $\mathcal{E} \leftarrow \mathcal{E} - \{e\}$;
  else
    if $e \not\subset \mathbf{y}$ **then** $R_e \leftarrow \pi_{e \cap \mathbf{y}} R_e$ **and** $e \leftarrow e \cap \mathbf{y}$;
return $((\mathbf{y}, \mathcal{E}), D)$;

We start with a special class of DCQs that two input CQs share the same schema. In Section 3.1, we introduce an algorithm based on query rewriting, which always pushes the difference operator down to the input relations and avoids materializing a large number of intermediate results that do not participate in the final query result. In Section 3.2, we move to general case that Q_1 and Q_2 can have different schemas.

3.1 Q_1 and Q_2 share the same schema

We first note that if Q_1, Q_2 share the same schema, i.e., there is a one-to-one correspondence between the relations/attributes in Q_1 and Q_2, Theorem 3.1 degenerates to the following lemma:

LEMMA 3.2. *Given two CQs $Q_1 = Q_2 = (\mathbf{y}, \mathcal{V}, \mathcal{E})$, if $(\mathbf{y}, \mathcal{V}, \mathcal{E})$ is free-connex, then the DCQ $Q_1 - Q_2$ can be computed in $O(N + \text{OUT})$ time.*

Let's start with an example falling into this special case.

Example 3.3. Consider a DCQ $Q_1 - Q_2$ with $Q_1 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$ and $Q_2 = R'_1(x_1, x_2) \bowtie R'_2(x_2, x_3)$. We can rewrite it as the union of two join queries: $Q_1 - Q_2 = (R_1 - R'_1) \bowtie R_2 + R_1 \bowtie (R_2 - R'_2)$, where the difference operator is only applied for computing $R_1 - R'_1$ and $R_2 - R'_2$. Intuitively, for every join result $(a, b, c) \in Q_1 - Q_2$, it must be $(a, b) \notin R'_1$ or $(b, c) \notin R'_2$; otherwise, $(a, b, c) \in Q_2$, coming to a contradiction. The correctness of this rewriting will be formally presented in the proof of Lemma 3.4. In addition, the difference operators can be evaluated in $O(N)$ time, and the join operators can be evaluated in $O(N + \text{OUT})$ time.

Rewrite Rule. We now generalize the rewriting rule in Example 3.9 to general DCQ $Q_1 - Q_2$ for $Q_1 = Q_2 = (\mathbf{y}, \mathcal{V}, \mathcal{E})$, where $\mathbf{y} = \mathcal{V}$ and $(\mathcal{V}, \mathcal{E})$ is α-acyclic. In other words, both Q_1, Q_2 correspond to the same acyclic full join query. For any $e \in \mathcal{E}$, let R_e, R'_e be the corresponding relations in Q_1, Q_2 respectively. Our rule is built on the observation that for any query result $t \in Q_1 - Q_2$, $\pi_e t \in R_e$ must hold for every $e \in \mathcal{E}$, but $\pi_e t \notin R'_e$ happens for some $e \in \mathcal{E}$. Applying this observation, we can rewrite such a DCQ as the (disjoint) union of a constant number of join queries as follows:

LEMMA 3.4. *Given two CQs $Q_1 = Q_2 = (\mathbf{y}, \mathcal{V}, \mathcal{E})$, if $(\mathcal{V}, \mathcal{E})$ is α-acyclic and $\mathbf{y} = \mathcal{V}$, $Q_1 - Q_2 = \bigcup_{e \in \mathcal{E}} \left((R_e - R'_e) \bowtie (\bowtie_{e' \in \mathcal{E} - \{e\}} R_{e'})\right)$.*

Algorithm and Complexity. An algorithm directly follows the rewriting rule above. It first computes the difference of every pair of input relations, i.e., $R_e - R'_e$ for each $e \in \mathcal{E}$, and then computes a full join query $(R_e - R'_e) \bowtie Q_1$ derived for each $e \in \mathcal{E}$. Actually, we can handle a slightly larger class of DCQ. For $Q_1 = Q_2 = (\mathbf{y}, \mathcal{V}, \mathcal{E})$, if $(\mathbf{y}, \mathcal{V}, \mathcal{E})$ is free-connex, we simply remove all non-output attributes for Q_1, Q_2 separately in the preprocessing step, and then tackle two acyclic full joins, that share the same structure.

Algorithm 2: EasyDCQ(Q_1, Q_2, D_1, D_2)

```
if y ≠ V_1 then ((y, E_1), D_1) ← Reduce(Q_1, D_1);
if y ≠ V_2 then ((y, E_2), D_2) ← Reduce(Q_2, D_2);
S ← ∅;
foreach e ∈ E_2 do
    S_e ← Yannakakis((e, y, E_1), D_1);
    S ← S ∪ Yannakakis((y, E_1 ∪ {e}), D_1 ∪ {S_e − R'_e});
return S;
```

As the pre-processing step and difference operators can be evaluated in $O(N)$ time, this algorithm is bottlenecked by evaluating the join query $(\mathcal{V}, \mathcal{E})$, which takes $O(N + \text{OUT})$ time. Putting everything together, we come to Lemma 3.2.

3.2 Q_1 and Q_2 have different schemas

We next move to the general case when these two input CQs have different schemas. We focus on the case when both Q_1, Q_2 are full and then extend to non-full case.

DCQ with full CQs. Now, we assume that $\mathbf{y} = \mathcal{V}_1 = \mathcal{V}_2 = \mathcal{V}$. Theorem 3.1 simply degenerates to the Lemma 3.5.

Lemma 3.5. *Given two full joins $Q_1 = (\mathcal{V}, \mathcal{E}_1)$ and $Q_2 = (\mathcal{V}, \mathcal{E}_2)$, if Q_1, Q_2 are α-acyclic, and $(\mathcal{V}, \mathcal{E}_1 \cup \{e\})$ is α-acyclic for every $e \in \mathcal{E}_2$, then $Q_1 - Q_2$ can be computed in $O(N + \text{OUT})$ time.*

A straightforward solution is to transform both $Q_1 = (\mathcal{V}, \mathcal{E}_1)$ and $Q_2 = (\mathcal{V}, \mathcal{E}_2)$ into one auxiliary query $(\mathcal{V}, \mathcal{E}_1 \cup \mathcal{E}_2)$, and then invoke the algorithm in Section 3.1 to handle the degenerated case. However, this solution does not necessarily lead to a linear-time algorithm. Let's gain some intuition from the example below.

Example 3.6. Consider a DCQ $Q_1 - Q_2$ with $Q_1 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3, x_4)$ and $Q_2 = R_3(x_1, x_2, x_3) \bowtie R_4(x_3, x_4)$. For an auxiliary query, we introduce the following intermediate relations $R_5 = R_1 \bowtie \pi_{x_2,x_3} R_2$, $R_6 = \pi_{x_3,x_4} R_2$, $R_7 = \pi_{x_1,x_2} R_3$ and $R_8 = \pi_{x_2,x_3} R_3 \bowtie R_4$. Then, we can rewrite Q_1, Q_2 as follows:

$$Q_1 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3, x_4) \bowtie R_5(x_1, x_2, x_3) \bowtie R_6(x_3, x_4)$$
$$Q_2 = R_7(x_1, x_2) \bowtie R_8(x_2, x_3, x_4) \bowtie R_3(x_1, x_2, x_3) \bowtie R_4(x_3, x_4)$$

Then, we are left with two queries that share the same schema. However, this strategy does not necessarily lead to a linear-time algorithm, since materializing the intermediate relation R_8 requires super-linear time, which could be much larger than the final output size OUT.

Careful inspection reveals that a simpler rewriting rule can avoid materializing R_8. More specifically, we keep Q_2 unchanged and rewrite Q_1 as above. Then, $Q_1 - Q_2$ can be rewritten as $(R_5 - R_3) \bowtie R_1 \bowtie R_2 + R_1 \bowtie R_2 \bowtie (R_6 - R_4)$. Intuitively, for every join result $(a, b, c, d) \in Q_1 - Q_2$, it must be $(a, b, c) \notin R_3$ or $(c, d) \notin R_4$; otherwise, $(a, b, c, d) \in Q_2$, coming to a contradiction. The correctness of this rewriting will be formally presented in the proof of Lemma 3.7. In this case, materializing R_6 only takes $O(N)$ time, but materializing R_5 might take super-linear time. Fortunately, we can bound the size of R_5 by $O(N + \text{OUT})$. The rationale is that every tuple in $R_5 - R_3$ will participate in at least on one join result of $(R_5 - R_3) \bowtie R_1 \bowtie R_2$, i.e., the final result of the difference query $Q_1 - Q_2$, thus $|R_5 - R_3| \leq$ OUT. For the difference operator, $R_5 - R_3$ takes $O(N + \text{OUT})$ time, and $R_6 - R_4$ takes $O(N)$ time. For the join operator, both simple join queries take linear time in terms of their input size and output size. Overall, this rewriting rule can compute the example query in $O(N + \text{OUT})$ time.

Algorithm 3: YANNAKAKIS(Q, D) [38]

```
(Q, D) ← REDUCE(Q, D);
T ← the free-connex join tree of Q rooted at r;
foreach v ∈ T in a bottom-up way (excluding root) do
    R_u ← R_u ⋉ R_v for the parent node p(u) of u;
foreach u ∈ T in a top-down way (excluding leaves) do
    foreach v is a child of u do R_u ← R_u ⋉ R_v;
foreach v ∈ T in a bottom-up way (excluding root) do
    R_u ← R_u ⋈ R_v for the parent node p(u) of u;
return R_r;
```

Rewrite Rule. Generalizing this observation, we develop the following rewriting rule for arbitrary full joins Q_1, Q_2. The high-level idea is to introduce an intermediate relation $S_e = \pi_e Q_1$ for every $e \in \mathcal{E}_2$, i.e., the projection of join results of Q_1 onto attributes e. Now we can rewrite $Q_1 - Q_2$ using input relations in Q_1 and intermediate relations corresponding to $\mathcal{E}_2$, as well as input relations in Q_2, which results in the disjoint union of multiple full joins.

LEMMA 3.7. *Given two CQs $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$, if $\mathbf{y} = \mathcal{V}_1 = \mathcal{V}_2$, $Q_1 - Q_2 = \bigcup_{e \in \mathcal{E}_2} \left((S_e - R'_e) \bowtie Q_1\right)$, for $S_e = \pi_e Q_1$.*

PROOF. Direction $\subseteq$. Consider an arbitrary result $t \in Q_1 - Q_2$. By definition, $\pi_{e_1} t \in R_{e_1}$ for every $e_1 \in \mathcal{E}_1$, and $\pi_{e_2} t \notin R'_{e_2}$ for some $e_2 \in \mathcal{E}_2$. Moreover, $\pi_{e_2} t \in S_{e_2} = \pi_{e_2} Q_1$. So, $t \in (S_{e_2} - R'_{e_2}) \bowtie Q_1$. Direction $\supseteq$. Consider an arbitrary $e_2 \in \mathcal{E}_2$ and a query result $t \in (S_{e_2} - R'_{e_2}) \bowtie Q_1$. By definition, $t \in Q_1$ and $t \notin R'_{e_2}$, which further implies $t \notin Q_2$. This way, $t \in Q_1 - Q_2$. □

Algorithm and Complexity. An algorithm for computing the difference of two full join queries follows the rewriting rule above. For each $e \in \mathcal{E}_2$, it first materializes the query results of $\pi_e Q_1$, then computes the difference operator $\pi_e Q_1 - R'_e$, and finally the full join $(\pi_e Q_1 - R'_e) \bowtie Q_1$ by invoking the classical Yannakakis algorithm. We next analyze the complexity of the algorithm above. To establish the complexity, we first show an upper bound on the size of any intermediate relation constructed:

LEMMA 3.8. *$|S_e| = O(N + \mathrm{OUT})$ for any $e \in \mathcal{E}_2$, where $S_e = \pi_e Q_1$.*

PROOF. Consider an arbitrary tuple $t \in S_e - R'_e$. First, t participates in at least one query result of Q_1. As $t \in S_e$, $t \in \pi_e Q_1$ by definition. There must exist some tuple $t' \in Q_1$ such that $\pi_e t' = t$. Thus, t participates in some query results of Q_1. Meanwhile, t does not participate in any query result of Q_2, since $t \notin R'_e$. In this way, t participates in at least one result in $Q_1 - Q_2$, thus $|S_e - R'_e| \le \mathrm{OUT}$. Moreover, $|R'_e| \le N$. Together, we obtain $|S_e| = O(N + \mathrm{OUT})$. □

Let $\mathcal{V} = \mathcal{V}_1 = \mathcal{V}_2$. If $(\mathcal{V}, \mathcal{E}_1)$ is α-acyclic, and $(\mathcal{V}, \mathcal{E}_1 \cup \{e\})$ is also α-acyclic for every $e \in \mathcal{E}_2$, then the constructed CQ $\pi_e Q$ is free-connex. Implied by the existing result on CQ evaluation, S_e can be computed in $O(N + |S_e|) = O(N + \mathrm{OUT})$ time by the classic Yannakakis algorithm, where OUT is the output size of the difference query! The invocation of Yannakakis algorithm here is crucial for achieving linear complexity. For example, if S_e is computed by first materializing the query results of Q_1 and then computing their projection onto e, the time complexity would be as large as $O(N + \mathrm{OUT}_1)$, where OUT_1 is the output size of Q_1. Now, each full join $(S_e - R'_e) \bowtie Q_1$ is α-acyclic with input size $O(N + \mathrm{OUT})$ and output size OUT, thus can be computed in $O(N + \mathrm{OUT})$ time.

Therefore, the total time complexity is bounded by $O(N + \text{OUT})$, since there are $O(1)$ sub-queries in $Q_1 - Q_2$. Putting everything together, we come to Lemma 3.5.

DCQ with general CQs. Now, we are ready to present an linear-time algorithm for computing $Q_1 - Q_2$, such that Q_1 is free-connex, Q_2 is linear-reducible, and $(\mathbf{y}, \mathcal{E}_1' \cup \{e\})$ is α-acyclic for every $e \in \mathcal{E}_2'$, where $(\mathbf{y}, \mathcal{E}_1')$ and $(\mathbf{y}, \mathcal{E}_2')$ are the reduced queries of Q_1, Q_2 respectively. As described in Algorithm 2, we first apply a preprocessing step to Q_1 and Q_2 (line 1-4), which removes non-output attributes in Q_1 and Q_2 if they are non-full.

As shown in Algorithm 1, this reduce step is quite standard by first building a free-connex join tree for the derived query $(\mathbf{y}, \mathcal{V}, \mathcal{E} \cup \{\mathbf{y}\})$, and then traversing the tree in a bottom-up way. In the traversal, when a relation is visited and contains some non-output attributes, we just update its parent relation by applying a semi-join and removing it. Note that if a relation does not contain any non-output attribute, then its ancestor also does not contain any, implied by the property of the free-connex join tree. Thus, the residual tree is a connected subtree that contains the root. Note that no physical relation is defined to $\mathbf{y}$, but this is not an issue since when such a relation is visited, Algorithm 1 simply skips it (line 4) as well as its ancestors. This algorithm only takes $O(N)$ time.

Then, we are left with two full joins, and invoke our rewriting rule proposed in Section 3.2 (line 6-8). As the reduce procedure takes $O(N)$ time, and the join phase takes $O(N + \text{OUT})$ time implied by Lemma 3.5, we can obtain the complexity result in Theorem 3.1.

Improvement over Baseline. When Q_1, Q_2 fall into the class of "easy" DCQs as characterized by Theorem 3.1, our algorithm only takes $O(N + \text{OUT})$ time for computing $Q_1 - Q_2$, while the baseline takes $O\left(N + \text{OUT}_1 + \texttt{cost}(\text{OUT}_2)\right)$ time, since $\texttt{cost}(Q_1) = O(N + \text{OUT})$ for free-connex Q_1. We next use a few examples of "easy" DCQs to illustrate the improvement achieved by our approach.

Example 3.9. Consider a DCQ with $Q_1 = R_1(x_1, x_2, x_3)$ and $Q_2 = R_2(x_1, x_2) \bowtie R_3(x_2, x_3) \bowtie R_4(x_1, x_3)$. The baseline takes $O\left(N^{\frac{2\cdot\omega}{\omega+1}} + N^{\frac{3(\omega-1)}{\omega+1}} \cdot \text{OUT}_2^{\frac{3-\omega}{\omega+1}}\right)$ time to compute the triangle join $R_2 \bowtie R_3 \bowtie R_4$ in Q_2, where ω is the exponent of fast matrix multiplication. In contrast, our approach only takes $O(N)$ time since $\text{OUT} \leq N$, improving the baseline by a factor of $O\left(N^{\frac{\omega-1}{\omega+1}} + N^{\frac{2\omega-4}{\omega+1}} \cdot \text{OUT}_2^{\frac{3-\omega}{\omega+1}}\right)$.

Example 3.10. Consider a DCQ with $Q_1 = R_1(x_1, x_2) \bowtie R_2(x_3, x_4)$ and $Q_2 = R_3(x_1, x_2) \bowtie R_4(x_2, x_3) \bowtie R_5(x_1, x_3)$. The baseline takes $O(N^2)$ time to materialize Q_1, which degenerates to the Cartesian product of R_1 and R_2. In contrast, our approach only requires $O(N + \text{OUT})$ time, improving the baseline by a factor of $O\left(\frac{N^2}{\text{OUT}}\right)$, since OUT can be much smaller than N^2.

Example 3.11. Consider a DCQ with $Q_1 = \bowtie_{e \subseteq U : |e| = 1} R_e(\{x_1\} \cup e)$ and $Q_2 = \bowtie_{e' \subseteq U : |e'| = 2} R_{e'}(\{x_1\} \cup e')$ for $U = \{x_2, \cdots, x_{k+1}\}$. The baseline takes $O(N)$ time to materialize Q_1, and $O(N^{\frac{k}{2}})$ time to materialize Q_2. In contrast, our approach can compute it in $O(N + \text{OUT})$ time, improving the baseline by a factor of $O\left(\frac{N^{k/2}}{\text{OUT}}\right)$, since OUT can be much smaller than $N^{\frac{k}{2}}$.

4 HARD DCQS

In this section, we turn to the class of "hard" DCQs characterized by Theorem 2.4. We first prove the hardness of computing DCQs in linear time via some well-known conjectures, and then show an efficient heuristic for hard DCQs by further exploiting the query structures.

4.1 Hardness

We will prove the hardness of computing a hard DCQ $Q_1 - Q_2$, in particular: (1) Q_1 is non-free-connex; or (2) Q_1 is free-connex but Q_2 is non-linear-reducible; or (3) Q_1 is free-connex, Q_2 is linear-reducible, but there exists some $e \in \mathcal{E}_2'$ such that $(\mathbf{y}, \mathcal{E}_1' \cup \{e\})$ is cyclic, where $(\mathbf{y}, \mathcal{E}_1')$ and

$(\mathbf{y}, \mathcal{E}_2')$ are the reduced queries of $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1), Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$ respectively. We will prove the hardness for each class of hard DCQs separately.

Hardness-(1). The hardness of computing DCQs in case (1) comes from computing a non-free-connex CQ [16]. By setting the result of Q_2 as $\emptyset$, $Q_1 - Q_2$ simply degenerates to Q_1, hence we obtain:

Lemma 4.1. *For any DCQ $Q_1 - Q_2$, if Q_1 is non-free-connex, any algorithm computing $Q_1 - Q_2$ requires at least $\Omega(N + \text{OUT})$ time.*

The hardness of case (2) and (3) is built on the *strong triangle conjecture* in the literature:

Conjecture 4.2 (Strong Triangle conjecture [11]). *Detecting whether an n-node m-edge graph contains a triangle requires $\Omega\left(\min\left\{n^{\omega-o(1)}, m^{2\omega/(\omega+1)-o(1)}\right\}\right)$ time in expectation, where $\omega = 2+o(1)$ is assumed as the exponent of fast matrix multiplication.*

Hardness-(2). We start with two hardcore DCQs in Lemma 4.3 and Lemma 4.4. The proof of Lemma 4.5 for general DCQs in case (2) is given in the full version [10].

Lemma 4.3. *Any algorithm for computing the following DCQ:*

$$Q_1 - Q_2 = R_1(x_1, x_3) - \pi_{x_1,x_3}\left(R_2(x_1, x_2) \bowtie R_3(x_2, x_3)\right)$$

requires $\Omega(N)$ time, assuming the strong triangle conjecture.

Proof. For a graph $G = (V, E)$, we denote $m = |E|$ and $n = |V|$. Note that $n < m < n^2$; otherwise, we simply remove vertices that do not incident to any edges in G. We then construct an instance D_1, D_2 for Q_1, Q_2 by setting $R_1 = R_2 = R_3 = E$. Hence, $N = m$. Note that there exists some triangle in G if and only if $Q_1 \cap Q_2$ is non-empty. Together with $Q_1 \cap Q_2 = Q_1 - (Q_1 - Q_2)$, we output "a triangle is detected in G" if and only if $|Q_1 - Q_2| < N$. If $Q_1 - Q_2$ can be computed in $O(N)$ time, whether there exists a triangle in G can be detected in $O(\min\{n^2, m^{4/3}\})$ time, coming to a contradiction of strong triangle conjecture. □

Lemma 4.4. *Any algorithm for computing the following DCQ:*

$$Q_1 - Q_2 = R_1(x_1) - \pi_{x_1}\left(R_2(x_1, x_3) \bowtie R_3(x_2, x_3) \bowtie R_4(x_1, x_3)\right),$$

requires $\Omega(N)$ time, assuming the strong triangle conjecture.

Proof. This is similar to the proof of Lemma 4.3. For a graph $G = (V, E)$, we construct $R_2 = R_3 = R_4 = E$ and $R_1 = V$, with $m = |E| = N$ and $n = |V|$. Note that there exists some triangle in G if and only if $Q_1 \cap Q_2$ is non-empty. Together with $Q_1 \cap Q_2 = Q_1 - (Q_1 - Q_2)$, we output "a triangle is detected in G" if and only if $|Q_1 - Q_2| < |R_1|$. If $Q_1 - Q_2$ can be computed in $O(N)$, whether there exists a triangle in G can be detected in $O(\min\{n^2, m^{4/3}\})$ time, coming to a contradiction of strong triangle conjecture. □

Lemma 4.5. *Given two CQs Q_1, Q_2, if Q_1 is free-connex and Q_2 is non-linear-reducible, any algorithm computing $Q_1 - Q_2$ requires $\Omega(N + \text{OUT})$ time, assuming the strong triangle conjecture.*

Hardness-(3). The hardness of evaluating a DCQ in case (3) inherits the hardness of *deciding* a DCQ: given a DCQ $Q_1 - Q_1$ and input databases D_1, D_2, the *decidability* problem asks to decide whether there exists a query result in $Q_1 - Q_2$. We identify a few hardcore DCQs in Lemma 4.6. The proof of Lemma 4.7 for general DCQs in case (3) is given in the full version [10].

Lemma 4.6. *Any algorithm for deciding the following DCQ*

$$Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_3) \bowtie R_4(x_2)$$
$$Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_3) \bowtie R_4(x_2, x_3)$$
$$Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_3) \bowtie R_5(x_1, x_2)$$
$$Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_3) \bowtie R_4(x_2, x_3) \bowtie R_5(x_1, x_2)$$

requires $\Omega(N)$ *time, assuming the strong triangle conjecture.*

Proof. We first focus on the first DCQ and the remaining ones can be proved similarly. Given an arbitrary graph $G = (V, E)$ with $m = |E|$ and $n = |V|$, we develop an algorithm to detect whether there exists a triangle in G. Note that $n < m < n^2$; otherwise, we simply remove vertices that do not incident to any edges in G. The degree $\deg(u)$ of a vertex $u \in V$ is defined as the size of neighbors of u, i.e., those incident to u with an edge in E. We partition vertices in V into two subsets: $V^H = \{v \in V : \deg(v) > m^{1/3}\}$ and $V^L = V - V^H$. From G, we construct following relations: $R = E$, $R_0 = \{(u, v) \in E : u \in V^L \text{ or } v \in V^L\}$, $R_1 = \{(u, v) \in E : u \in V^H\}$, $R_2 = \{(u, v) \in E : v \in V^H\}$ and $R_3 = V^H \times V^H - E$. Set $N = m^{4/3}$. It can be easily checked that each relation contains at most $m^{4/3}$ tuples, hence $N = m^{4/3}$. We further define a CQ Q as follows:

$$Q = R(x_1, x_2) \bowtie R(x_2, x_3) \bowtie R_0(x_1, x_3)$$

For $Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_3) \bowtie R_4(x_2)$, we set $R_4 = V$ and output "a triangle is detected" if and only if Q or $Q_1 - Q_2$ is not empty. We first prove the correctness of this algorithm, i.e., a triangle exists in G if and only if Q or $Q_1 - Q_2$ is not empty. *Direction Only-If*. Consider an arbitrary triangle (u, v, w) in G. We distinguish two cases: (i) at least one of u, w is light; (ii) both u and w are heavy. In (i), assume u is light. Then, $(u, v), (u, w) \in R_0$. We come to $(u, v, w) \in Q$. In (ii), $(u, v) \in R_1$,$(v, w) \in R_2$, $(u, w) \notin R_3$, so we come to $(u, v, w) \in Q_1 - Q_2$. *Direction If*. If $Q \neq \emptyset$, say $(u, v, w) \in Q$, then $(u, v), (v, w), (u, w) \in E$ and therefore (u, v, w) is a triangle in G. If $Q_1 - Q_2 \neq \emptyset$, say $(u, v, w) \in Q_1 - Q_2$, then $(u, v) \in R_1, (v, w) \in R_2, (u, w) \notin R_3$, i.e., $(u, v), (v, w), (u, w) \in E$, and therefore (u, v, w) is a triangle in G.

We next turn to the time complexity. All statistics and relations R_1, R_2, R_4 can be computed in $O(m)$ time. Moreover, relation R_3 can be constructed in $O(m^{4/3})$ time since $|V^H| = O(m^{2/3})$. Q can be evaluated in $O(m^{4/3})$ time, since each of $R_0(x_1, x_2) \bowtie R(x_2, x_3)$ generates at most $O(m^{4/3})$ intermediate join results if $x_2 \in V^L$, and each of $R_0(x_1, x_2) \bowtie R(x_1, x_3)$ generates at most $O(m^{4/3})$ intermediate join results if $x_1 \in V^L$. If $Q_1 - Q_2$ can be decided in $O(N)$ time, whether there exists a triangle in G can be decided in $O(m^{4/3})$ time, coming to a contradiction to strong triangle conjecture.

For $Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_3) \bowtie R_4(x_2, x_3)$, we set $R_4 = E$ and output "a triangle is detected" if and only if Q or $Q_1 - Q_2$ is not empty. For $Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_3) \bowtie R_5(x_1, x_2)$, we set $R_5 = R$ and output "a triangle is detected" if and only if Q or $Q_1 - Q_2$ is not empty. For $Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_3) \bowtie R_4(x_2, x_3) \bowtie R_5(x_1, x_2)$, we set $R_4 = R_5 = E$ and output "a triangle is detected" if and only if Q or $Q_1 - Q_2$ is not empty. Similarly, Q can be computed in $O(m^{4/3})$ time. This way, if $Q_1 - Q_2$ can be decided in $O(N)$ time, whether there exists a triangle in G can be decided in $O(m^{4/3})$ time, coming to a contradiction to strong triangle conjecture. Together, we have completed the proof. □

Lemma 4.7. *Given two CQs* $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1), Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$*, if* Q_1 *is free-connex,* Q_2 *is linear-reducible, and there exists some* $e \in \mathcal{E}_2'$ *such that* $(\mathbf{y}, \mathcal{E}_1' \cup \{e\})$ *is cyclic where* $(\mathbf{y}, \mathcal{E}_1')$ *and* $(\mathbf{y}, \mathcal{E}_2')$ *are the reduced queries of* Q_1, Q_2 *respectively, any algorithm computing* $Q_1 - Q_2$ *requires* $\Omega(N + \text{OUT})$ *time, assuming the strong triangle conjecture.*

4.2 Efficient Heuristics

Although the hardness results in Section 4 have ruled out a linear-time algorithm for the "hard" DCQs, we find that it is still possible to explore efficient heuristics that can outperform the baseline approach. Our heuristic is based on a simple fact that $Q_1 - Q_2 = Q_1 - Q_1 \cap Q_2$. After computing the query results for Q_1, a straightforward way of deciding $Q_1 \cap Q_2$ is to decide for each result $t \in Q_1$, whether $t \in Q_2$ nor not. This decidability query can be viewed as a special Boolean query by replacing every output attribute $x \in \mathbf{y}_2$ with a constant $\pi_x t$. More specifically, for $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$, the derived a Boolean query can be represented as $(\emptyset, \mathcal{V}_2 - \mathbf{y}, \{e - \mathbf{y} : e \in \mathcal{E}_2\})$. Putting everything together, we come to Theorem 4.8.

Theorem 4.8. *Given two CQs $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$, $Q_1 - Q_2$ can be computed in $O(\texttt{cost}(Q_1) + \mathrm{OUT}_1 \cdot \texttt{cost}(Q_2^{\emptyset}))$ time, where $Q_2^{\emptyset} = (\emptyset, \mathcal{V}_2 - \mathbf{y}, \{e - \mathbf{y} : e \in \mathcal{E}_2\})$.*

Remark. If Q_2 is linear-reducible, Q_2 can be reduced to a full join in $O(N)$ time by Algorithm 1. Then, $Q_2^{\emptyset}$ becomes empty. A faster solution is to build hashing indexes on every relation in the reduced Q_2. For each tuple $t \in Q_1$, it suffices to check for every $e \in \mathcal{E}_2$ whether $\pi_e t \in R'_e$, which only takes $O(1)$ time. We note that the rewriting rule in Lemma 3.7 can also apply to this case and lead to the same complexity. Suppose Q_2 is reduced. Each $e \in \mathcal{E}_2$ induces a CQ $(\pi_e Q_1 - R'_e) \bowtie Q_1$. After materializing the results of Q_1, it suffices to check for each tuple $t \in Q_1$, whether $\pi_e t \in R'_e$ or not. This is exactly how our heuristic proceeds. Hence, Corollary 2.5 follows.

Example 4.9. Consider a DCQ with $Q_1 = \pi_{x_1,x_2,x_3} R_1(x_1, x_4) \bowtie R_2(x_4, x_2, x_3)$ and $Q_2 = \pi_{x_1,x_2,x_3} R_3(x_1, x_2) \bowtie R_4(x_2, x_3) \bowtie R_5(x_1, x_3) \bowtie R_6(x_3, x_4)$. The baseline spends $O(N^{\frac{2\cdot\omega}{\omega+1}} + N^{\frac{\omega-1}{\omega+1}} \cdot \mathrm{OUT}_1)$ time computing Q_1 and $O(N^{\frac{2\cdot\omega}{\omega+1}} + N^{\frac{3(\omega-1)}{\omega+1}} \cdot \mathrm{OUT}_2^{\frac{3-\omega}{\omega+1}})$ time computing the hidden triangle join $R_3 \bowtie R_4 \bowtie R_5$ in Q_2, where ω is the exponent of fast matrix multiplication. In contrast, our algorithm only spends $O(N^{\frac{2\cdot\omega}{\omega+1}} + N^{\frac{\omega-1}{\omega+1}} \cdot \mathrm{OUT}_1)$ time for computing Q_1, without computing the expensive Q_2, hence can improve the baseline by a factor of $O\left(N^{\frac{2(\omega-1)}{\omega+1}} \cdot \mathrm{OUT}_2^{\frac{3-\omega}{\omega+1}} / \mathrm{OUT}_1\right)$ when $N^{\frac{2(\omega-1)}{\omega+1}} \cdot \mathrm{OUT}_2^{\frac{3-\omega}{\omega+1}} \geq \mathrm{OUT}_1$.

We can show some further improvement when Q_2 is non-linear-reducible. Instead of issuing an individual Boolean query for every query result $t \in Q_1$, we take all the Boolean queries into account as whole. To do so, we further explore the structural property of the intersection query $Q_2^{\oplus} = (\mathbf{y}, \mathcal{V}_2, \{\mathbf{y}\} \cup \mathcal{E}_2)$, by treating the query results of Q_1 as a single relation over attributes $\mathbf{y}$. We note that it is always cheaper (or at least not more expensive) to compute $Q_2^{\oplus}$ than Q_2, as one can always compute $Q_2^{\oplus}$ by materializing Q_2 with an additional semi-join with Q_1. The cost of this naive method evaluating $Q_2^{\oplus}$ is bounded by $O(\texttt{cost}(Q_1) + \texttt{cost}(Q_2))$. It is possible to compute $Q_2^{\oplus}$ in a more efficient way.

Theorem 4.10. *Given two CQs $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$, $Q_1 - Q_2$ can be computed in $O(\texttt{cost}(Q_1) + \texttt{cost}(Q_2^{\oplus}))$ time, where $Q_2^{\oplus} = (\mathbf{y}, \mathcal{V}_2, \{\mathbf{y}\} \cup \mathcal{E}_2)$.*

Example 4.11. Consider a DCQ with $Q_1 = R_1(x_1, x_3)$ and $Q_2 = \pi_{x_1,x_3}(R_3(x_1, x_2) \bowtie R_4(x_2, x_3))$. The baseline takes $O(N + N \cdot \sqrt{\mathrm{OUT}_2})$ time to materialize Q_2. The first heuristics of issuing $Q_2^{\emptyset}$ for each tuple $t \in R_1$ takes $O(N^{3/2})$ time. We note that $Q_{12} = \pi_{x_1,x_3}(R_1(x_1, x_3) \bowtie R_2(x_1, x_2) \bowtie R_3(x_2, x_3))$ lists edges that participate in at least one triangle. The existing best algorithm takes $O(N^{\frac{2\omega}{\omega+1}})$ time to compute Q_{12}, where ω is the exponent of fast matrix multiplication, dominating the overall complexity. Our approach will improve the baseline if $\mathrm{OUT}_2 > N^{\frac{2(\omega-1)}{\omega+1}}$, and strictly outperforms the naive heuristic.

Example 4.12. Consider a DCQ with $Q_1 = \pi_{x_1,x_3} R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$ and $Q_2 = \pi_{x_1,x_3} R_3(x_1, x_2) \bowtie R_4(x_2, x_3)$. Let $R_5(x_1, x_3) = Q_1$. Here, $Q_{12} = \pi_{x_1,x_3} R_3(x_1, x_2) \bowtie R_4(x_2, x_3) \bowtie R_5(x_1, x_3)$ with $|R_5| = \mathrm{OUT}_1$. Similarly, the existing best algorithm takes $O(\mathrm{OUT}_1^{\frac{\omega}{\omega+1}} \cdot N^{\frac{\omega}{\omega+1}})$ time to compute Q_{12}. It is

worth mentioning that $Q_{12} \neq \pi_{x_1,x_3} R_1(x_1,x_2) \bowtie R_2(x_2,x_3) \bowtie R_3(x_1,x_2) \bowtie R_4(x_2,x_3)$. *Suppose* $(a,c) \in Q_{12}$, *is witnessed by* (a,b_1,c) $\inf R_1 \bowtie R_2$ *and* $(a,b_2,c) \in R_3 \bowtie R_4$, *but* $(a,b_1,c) \notin R_3 \bowtie R_4$ *and* $(a,b_2,c) \notin R_1 \bowtie R_2$. *Then,* $(a,b_1,c),(a,b_2,c) \notin R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$, *hence the result* (a,c) *will be missed in this rewriting.*

5 EXTENSIONS

Based on the basic DCQ over two CQs discussed so far, we next consider several interesting extensions of DCQ with rich interaction between difference and other relational algebra operators.

5.1 Difference of Multiple CQs

The first extension is adapting our result for computing DCQ involving two CQs to multiple CQs, say $Q = Q_1 - Q_2 - \cdots - Q_k$. Given k CQs $Q_1, Q_2, \cdots, Q_k$ with $Q_i = (\mathbf{y}, \mathcal{V}_i, \mathcal{E}_i)$ for $i \in \{2,3,\cdots,k\}$, we suppose Q_1 is free-connex, Q_i for $i \geq 2$ is linear-reducible for every $i \in \{2,3,\cdots,k\}$, and the subquery induced by $\mathcal{E}'_1 \cup \{e\}$ is α-acyclic for every $e \subseteq \mathcal{E}'_2 \cup \mathcal{E}'_3 \cup \cdots \cup \mathcal{E}'_k$, where $(\mathbf{y}, \mathcal{E}'_i)$ is the reduced query of $(\mathbf{y}, \mathcal{V}_i, \mathcal{E}_i)$. We can invoke our algorithm in Section 3 in a round-robin fashion, by computing $Q_1 - Q_i$ for each $i \in \{2,3,\cdot,4\}$ and stop once some $Q_1 - Q_i$ finishes the computation. Suppose $Q_1 - Q_i$ finishes the first. We then check for each result $t \in Q_1 - Q_i$ whether $t \in Q_j$ for every $j \in \{2,3,\cdots,k\} - \{i\}$, and output $t \in Q_1 - Q_i$ if $t \notin Q_j$ for every $j \in \{2,3,\cdots,k\} - \{i\}$.

Theorem 5.1. *Given* k *CQs* $Q_1, Q_2, \cdots, Q_k$ *with* $Q_i = (\mathbf{y}, \mathcal{V}_i, \mathcal{E}_i)$ *for* $i \in \{2,3,\cdots,k\}$, *the DCQ* $Q = Q_1 - Q_2 - \cdots - Q_k$ *can be evaluated in* $O\left(N + \min_{i \in \{2,3,\cdots,k\}} \mathrm{OUT}_i\right)$ *time where* $\mathrm{OUT}_i = |Q_1 - Q_i|$, *if* Q_1 *is free-connex,* Q_i *for* $i \geq 2$ *is linear-reducible for every* $i \in \{2,3,\cdots,k\}$, *and the subquery induced by* $\mathcal{E}'_1 \cup \{e\}$ *is* α*-acyclic for every* $e \subseteq \mathcal{E}'_2 \cup \mathcal{E}'_3 \cup \cdots \cup \mathcal{E}'_k$, *where* $(\mathbf{y}, \mathcal{E}'_i)$ *is the reduced query of* $(\mathbf{y}, \mathcal{V}_i, \mathcal{E}_i)$.

We also mention an important property that can decompose a DCQ into the union of multiple DCQs in a special form:

Lemma 5.2. *Given a DCQ* $Q = Q_1 - Q_2 - \cdots - Q_k$ *with full CQ* $Q_i = (\mathbf{y}, \mathcal{E}_i)$ *for every* $i \in \{1,2,\cdots,k\}$, *then* $Q = \bigcup_{(e_2,e_3,\cdots,e_k) \in \mathcal{E}_2 \times \mathcal{E}_3 \times \cdots \times \mathcal{E}_k} \left(Q_1 - R_{e_2} \bowtie Q_1 - R_{e_3} \bowtie Q_1 - \cdots - R_{e_k} \bowtie Q_1\right)$.

Proof. Direction $\subseteq$. Consider an arbitrary tuple $t \in Q_1 - Q_2 - \cdots - Q_k$. We know $t \in Q_1$ and $t \notin Q_i$ for every $i \notin \{2,3,\cdots,k\}$. If $t \notin Q_i$, there must exist some $e_i \in \mathcal{E}_i$, such that $\pi_{e_i} t \notin R_{e_i}$, i.e., $t \notin R_{e_i} \bowtie Q_1$. Together, there must exist a combination $(e_2,e_3,\cdots,e_k) \in \mathcal{E}_2 \times \mathcal{E}_3 \times \cdots \times \mathcal{E}_k$ such that $Q_1 - R_{e_2} \bowtie Q_1 - R_{e_3} \bowtie Q_1 - \cdots - R_{e_k} \bowtie Q_1$. Direction $\supseteq$. Consider an arbitrary combination $(e_2,e_3,\cdots,e_k) \in \mathcal{E}_2 \times \mathcal{E}_3 \times \cdots \times \mathcal{E}_k$ and an arbitrary tuple $t \in Q_1 - R_{e_2} \bowtie Q_1 - R_{e_3} \bowtie Q_1 - \cdots - R_{e_k} \bowtie Q_1$. By definition, $t \in Q_1$ and $\pi_{e_i} t \notin R_{e_i}$ for every $i \in \{2,3,\cdots,k\}$, i.e., $t \notin Q_i$. Hence, $t \in Q_1 - Q_2 - \cdots - Q_k$. □

Then, it suffices to compute $Q_1 - R_{e_2} \bowtie Q_1 - R_{e_3} \bowtie Q_1 - \cdots - R_{e_k} \bowtie Q_1$ for each combination $(e_2,e_3,\cdots,e_k) \in \mathcal{E}_2 \times \mathcal{E}_3 \times \cdots \times \mathcal{E}_k$. We notice that deciding whether such a query has any result can be done in $O(N)$ time, if any subquery induced by $\mathcal{E}_1 \cup S$ is α-acyclic for every $S \subseteq \{e_2,e_3,\cdots,e_k\}$ [20] (see Theorem 7.5). We believe that it is promising to adapt their algorithm to further speed the evaluation of DCQ with multiple CQs, and we won't pursue this direction here.

5.2 Select, Project and Join

- If there is a selection operator σ_ϕ over $Q = Q_1 - Q_2$, we can push it down such that $Q = \sigma_\phi Q_1 - \sigma_\phi Q_2$. If ϕ is a predicate on a base relation R_e of Q_1 (resp. Q_2), we can we simply check if $\phi(t)$ is true for each tuple $t \in \sigma_\phi R_e$, and discard it if not. This only takes $O(N)$ time. It is challenging that ϕ is a predicate not on any base relation, even for a single CQ evaluation.

- If there is a projection operator π_θ over $Q = Q_1 - Q_2$, we can push it down such that $Q = \pi_\theta Q_1 - \pi_\theta Q_2$, and handle a new DCQ $Q_1' - Q_2'$ with $Q_1' = \pi_\theta Q_1$ and $Q_2' = \pi_\theta Q_2$.
- If there is a join operator over multiple DCQs, we first rewrite the join into a DCQover multiple CQs and invoke our previous algorithm in Section 5.1. More specifically, given k DCQs $Q^1, Q^2, \cdots, Q^k$ with $Q^i = Q_1^i - Q_2^i$ for any $i \in [k]$, we can rewrite $Q^1 \bowtie Q^2 \bowtie \cdots \bowtie Q^k$ as a DCQ with multiple CQs: $(\bowtie_{i\in[k]} Q_1^i) - \{(\bowtie_{i\in I} Q_1^i) \bowtie (\bowtie_{j\in J} Q_2^j) : I \subsetneq [k], J = [k] - I\}$.

5.3 Aggregation

Our algorithm for DCQ can also be extended to support aggregations over annotated relations [14, 30]. Let $(S, \oplus, \otimes)$ be a commutative ring. For a CQ Q over an annotated instance D, every tuple $t \in R_e$ has an annotation $w(t) \in S$. For a full query $Q = (\mathcal{V}, \mathcal{E})$, the annotation for any join result $t \in Q(D)$ is defined as $w(t) := \bigotimes_{e\in\mathcal{E}} w(\pi_e t)$. For a non-full query $Q = (\mathbf{y}, \mathcal{V}, \mathcal{E})$, the aggregation becomes `GROUP BY` $\mathbf{y}$, and the annotation for each result $t \in Q$ (i.e., the aggregate of each group) is $w(t) := \bigoplus_{t'\in\bowtie_{e\in\mathcal{E}} R_e : \pi_{\mathbf{y}} t' = t} w(t')$. Below, we introduce two commonly-used formulations. Given $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$, $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$ and instances D_1, D_2, let w_1, w_2 be the annotations of tuples in Q_1, Q_2 respectively. For completeness, we set $w_1(t) = 0$ if $t \notin Q_1(D_1)$ and $w_2(t) = 0$ if $t \notin Q_2(D_2)$.

Relational difference. For DCQs defined on relational difference, a tuple t appears in the query results of $Q_1 - Q_2$ if and only if $t \in Q_1(D_1)$ and $t \notin Q_2(D_2)$. For $t \in \pi_{\mathbf{y}'}(Q_1(D_1) - Q_2(D_2))$, the annotation of t is defined as $w(t) = \oplus_{t'\in Q_1(D_1)-Q_2(D_2):\pi_{\mathbf{y}'} t' = t} w(t')$. The input size is defined as $N = |D_1| + |D_2|$, and the output size is $\text{OUT} = |\pi_{\mathbf{y}'}(Q_1(D_1) - Q_2(D_2))|$. Again, our target is to find a linear-time algorithm in terms of N and OUT. Our algorithms can be applied directly, followed by aggregation, and its complexity is bottlenecked by the output size of the difference query, i.e., $|Q_1(D_1) - Q_2(D_2)|$, which could be much larger than OUT.

Numerical difference. For DCQs defined on numerical difference, a tuple t appears in the query results of $Q_1 - Q_2$ if and only if $t \in Q_1(D_1)$ or $t \in Q_2(D_2)$, with annotation $w(t) = w_1(t) - w_2(t)$. Then, the aggregation operator defined over attributes $\mathbf{y}'$ on top of $Q_1 - Q_2$ can be rewritten as the numerical difference of two new annotated queries, i.e., $\pi_{\mathbf{y}'}Q_1 - \pi_{\mathbf{y}'}Q_2$. The input size is defined as $N = |D_1| + |D_2|$, and the output size is $\text{OUT} = |\pi_{\mathbf{y}'}Q_1(D_1) \cup \pi_{\mathbf{y}'}Q_2(D_2)|$. Again, our target is to find an linear-time algorithm in terms of N and OUT. Here, any algorithm with time complexity $O(N + |\pi_{\mathbf{y}'}Q_1(D_1)| + |\pi_{\mathbf{y}'}Q_2(D_2)|)$ is already optimal, since $|\pi_{\mathbf{y}'}Q_1(D_1) \cup \pi_{\mathbf{y}'}Q_2(D_2)| \geq \frac{1}{2}\left(|\pi_{\mathbf{y}'}Q_1(D_1)| + |\pi_{\mathbf{y}'}Q_2(D_2)|\right)$. Hence, if $\pi_{\mathbf{y}'}Q_1$ and $\pi_{\mathbf{y}'}Q_2$ are free-connex, both our algorithm and baseline are optimal.

Theorem 5.3. *Given two CQs $Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$, and a subset of aggregation attributes $\mathbf{y}' \subseteq \mathbf{y}$, if $(\mathbf{y}', \mathcal{V}_1, \mathcal{E}_1)$ and $(\mathbf{y}', \mathcal{V}_2, \mathcal{E}_2)$ are free-connex, $\pi_{\mathbf{y}'}(Q_1 - Q_2)$ with numerical difference can be computed in $O(N + \text{OUT})$ time.*

Example 5.4. Consider an example DCQ $Q = \pi_{x_1}(R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_2) \bowtie R_4(x_2, x_3))$ over an instance in Figure 3. This DCQ can capture Q16 in the TPC-H benchmark [9] as a special case. For relational difference, the query result of Q includes 2 tuples as $\{(a_1, 1), (a_2, 1)\}$. For numerical difference, the query result of Q includes 3 tuples as $\{(a_1, 1), (a_2, 2), (a_3, -2)\}$.

5.4 Bag Semantics

We consider the bag semantics that the set of query result is a multi-set. For simplicity, each distinct tuple t is annotated with a positive integer $w(t)$ to indicate the number of copies. In a full CQ $Q = (\mathcal{V}, \mathcal{E})$, the annotation of $t \in Q$ is defined as $w(t) = \times_{e\in\mathcal{E}} w(\pi_e t)$. For a projection of R_e onto attributes e', the annotation of $t \in \pi_{e'} R_e$ is defined as $w(t) = \sum_{t'\in R_e:\pi_{e'} t' = t} w(t')$. Given two CQs

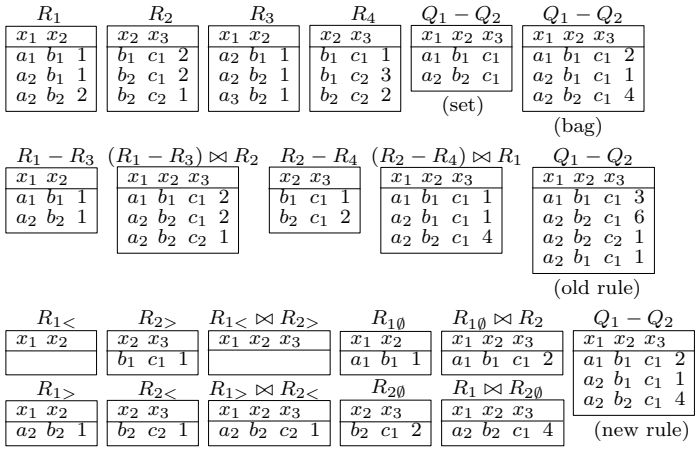


Fig. 3. An example of DCQ $Q_1 - Q_2$ with $Q_1 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$ and $Q_2 = R_3(x_1, x_2) \bowtie R_4(x_2, x_3)$. In the bag semantics, the input size is 18, and the output size is 7. (a_1, b_1, c_1) is a result under bag semantics, but not a result under set semantics. The results of $Q_1 - Q_2$ computed by the old and new rewriting rules are also illustrated separately.

$Q_1 = (\mathbf{y}, \mathcal{V}_1, \mathcal{E}_1), Q_2 = (\mathbf{y}, \mathcal{V}_2, \mathcal{E}_2)$ and two input instances D_1, D_2, let w_1, w_2 be the annotations of tuples in Q_1, Q_2 respectively. For completeness, we set $w_1(t) = 0$ if $t \notin Q_1(D_1)$ and $w_2(t) = 0$ if $t \notin Q_2(D_2)$. A tuple t is a query result of $Q_1 - Q_2$ if and only if $t \in Q_1(D_1)$ and $w_1(t) > w_2(t)$. An example is given in Figure 3.

The input size is $N = |D_1| + |D_2|$, and the output size is $\text{OUT} = \sum_{t \in Q_1(D_1)} \max\{0, w_1(t) - w_2(t)\}$. Again, our target is to find an linear-time algorithm in terms of N and OUT. Unfortunately, our rewriting rule in Section 3 cannot be adapted here. Figure 3 shows several incorrect behaviors: some tuple has a much higher annotation (e.g., (a_1, b_1, c_1)); and some tuple should not appear (e.g., (a_2, b_2, c_2)), which motivates us to explore new rules here.

Example 5.5. Consider a DCQ with $Q_1 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$ and $Q_2 = R_3(x_1, x_2) \bowtie R_4(x_2, x_3)$ under the bag semantics. Any result $(a, b, c) \in Q_1 - Q_2$ falls into one of the three cases: (1) $(a, b) \notin R_2$ or $(b, c) \notin R_4$; (2) $w_1(a, b) > w_2(a, b)$ and $w_1(b, c) > w_2(b, c)$; (3) either $w_1(b, c) \le w_2(b, c)$ or $w_1(a, b) \le w_2(a, b)$, but $w_1(a, b) \cdot w_1(b, c) > w_2(a, b) \cdot w_2(b, c)$. We partition R_1 into three subsets, $R_{1\emptyset} = \{t \in R_1 : t \notin R_3\}$, $R_{1<} = \{t \in R_1 : t \in R_3, w_1(t) \le w_2(t)\}$ and $R_{1>} = \{t \in R_1 : t \in R_3 : w_1(t) > w_2(t)\}$. Similarly, we partition R_2 into $R_{2\emptyset}$, $R_{2<}$, $R_{2>}$ with respect to R_4. Results falling into (1) can be found by $(R_{1\emptyset} \bowtie R_2) \cup (R_1 \bowtie R_{2\emptyset})$. Results falling into (2) can be found by $R_{1>} \bowtie R_{2>}$. Results falling into (3) can be found by two new θ-joins $(R_{1>} \bowtie_\theta R_{2<}) \cup (R_{1<} \bowtie_\theta R_{2>})$, where a pair of tuples (t_1, t_2) can be θ-joined if and only if $w_1(t_1) \cdot w_1(t_2) > w_2(t_1) \cdot w_2(t_2)$.

All auxiliary relations as well as $(R_{1\emptyset} \bowtie R_2) \cup (R_1 \bowtie R_{2\emptyset})$ and $R_> \bowtie R_{2>}$ can be computed efficiently. We consider $R_{1>} \bowtie_\theta R_{2<}$ ($(R_{1<} \bowtie_\theta R_{2>})$ is symmetric). The solution of checking θ-condition for all combinations of tuples in $R_{1>}$ and $R_{2<}$ incurs quadratic complexity. A smarter way is to sort $R_{1>}$ and $R_{2<}$ by B first, and then by the ratio of $\frac{w_1(\cdot)}{w_2(\cdot)}$ decreasingly. Then, we start with $(a, b) \in R_{1>}$ with maximum ratio, and linearly scan tuples in $R_{2<}$ with the join value b until we meet some tuple (b, c) such that $\frac{w_1(b,c)}{w_2(b,c)} \le \frac{w_2(a,b)}{w_1(a,b)}$. We then stop and proceed with the next tuple in $R_{1>}$. If no join result is produced by (a, b), we skip the subsequent tuples with the same join value b and continue. Overall, this algorithm takes $O(N \log N + \text{OUT})$ time.

Theorem 5.6. *Given two CQs $Q_1 = Q_2 = (\mathbf{y}, \mathcal{V}, \mathcal{E})$, if $(\mathbf{y}, \mathcal{V}, \mathcal{E})$ is free-connex, then $Q_1 - Q_2$ under the bag semantics can be computed in $O(N \log N + \text{OUT})$ time.*

Our observation above can be extended to the case when both Q_1, Q_2 correspond to the same free-connex query. The proof of Theorem 5.6 is given in the full version [10]. The case when Q_1, Q_2 have different schemas is left as future work.

6 EXPERIMENTS

6.1 Experimental Setup

Prototype implementation. Our newly developed algorithms can be easily integrated into any SQL engine by rewriting the original SQL query. It can be further optimized if we directly integrate the rewrite procedure into the SQL parser and have customized index support. Our ultimate goal is to implement our algorithms into a system prototype with three components: a SQL parser, a query optimizer, and new indices. At the current stage, we choose to manually rewrite all SQL queries and demonstrate the power of our optimizations via the comparison with vanilla SQL queries .

Query processing engines compared. To compare the performance of all optimized techniques we proposed in the paper, we choose PostgreSQL [4], DuckDB[1], SQLite[7], MySQL[2] running in centralized settings, and Spark SQL [6] running in parallel/distributed settings, as the query processing engines. All of them are widely used in academia and industry. In the experiments, we observed that SQLite and MySQL show very poor performance, with most of the test points being timed out. Hence, we built full indices on these systems to expedite the execution. Moreover, DuckDB is a columnar-vectorized query execution engine, and indices are built when importing input data. During the experiments, we test the single-thread performance of our new optimization techniques over PostgreSQL, DuckDB, SQLite and MySQL, and parallel performance over Spark SQL. In order to separate the I/O cost from the total execution time, we load all data into the memory in advance by using pg-prewarm in PostgreSQL and cache in Spark SQL. For DuckDB and SQLite, the data need to be loaded into memory before execution, so we only count the query execution time.

Experimental environment. We perform all experiments in two machines. For experiments conducted on PostgreSQL and MySQL, we use a machine equipped with two Intel Xeon 2.1GHz processors, each having 12 cores/24 threads and 416 GB memory. For all experiments on Spark SQL, DuckDB and SQLite, we use a machine equipped with two Xeon 2.0GHz processors, each having 28 cores / 56 threads and 1TB of memory. All machines run Linux, with Scala 2.13.9 and JVM 1.8.0. We use Spark 3.3.0 and PostgreSQL 16.0. We assign 8 cores for Spark and 1 core for the rest platforms during the experiments. Each query is evaluated 10 times with each engine, and we report the average running time. Each query runs at most 10 hours to obtain meaningful results.

6.2 Datasets and Queries

The experiments consist of graph queries and benchmark queries.

Benchmark queries. For relational queries, we adopt two standard benchmarks (TPC-DS [8] and TPC-H [9]) in industry and select 3 queries with difference operator (TPC-H Q16, TPC-DS Q35, and TPC-DS Q69). These three benchmark queries connect DCQ with other relational operators like selection, projection, join, and aggregation. All benchmark queries can be captured by a common schema $Q = R_1(x_1, x_2) \bowtie (\pi_{x_2} R(x_1, x_2) - \pi_{x_2} R_2(x_2, x_3) \bowtie R_3(x_3, x_4))$ and the joins are all primary-key foreign-key joins.

Graph queries. For graph pattern queries, we use real-world graphs (such as BitCoin, DBLP, Eponions, Google, and Wiki) from SNAP (Stanford Network Analysis Project) [5], summarized in Table 2. We store edge information as a relation $\text{Graph}(\text{src}, \text{dst})$ and manually create a triple

Graph	#edge	#vertex	#l2 path	#triangle	$\#Q_{G1}$	$\#Q_{G2}$	$\#Q_{G3}$	$\#Q_{G4}$	$\#Q_{G5}$	$\#Q_{G6}$
Bitcoin	24,186	3,783	1,256,332	88,753	820	1.0×10^7	585,958	331,497	3.8×10^7	5.7×10^8
Epinions	508,837	75,879	3.9×10^7	3,586,405	25,947	9.3×10^8	1.8×10^7	1.0×10^7	3.5×10^9	2.5×10^{11}
DBLP	1,049,866	317,080	7,064,738	2,224,385	466,646	1.6×10^8	3,532,369	2,203,597	6.7×10^7	-
Google	5,105,039	875,713	6.0×10^7	2.8×10^7	372,042	2.1×10^8	2.4×10^7	1.5×10^7	7.8×10^8	-
Wiki	2.8×10^7	2,394,385	2.6×10^9	8.1×10^7	0	1.1×10^{10}	1.3×10^8	6.6×10^7	-	-

Table 2. Graph datasets and their statistics. #edge is the input size of graph datasets. $\#Q_{Gi}$ is the output size of Q_{Gi} over the corresponding graph datasets. '-' indicates that the output size is too huge such that all systems cannot report the output size within the time limit.

$$Q_{G1} = \text{Graph}(\text{node}_1, \text{node}_2) - \pi_{\text{node}_1,\text{node}_2}(\text{Graph}(\text{node}_1, \text{node}_2) \bowtie \text{Graph}(\text{node}_2, \text{node}_3))$$

$$Q_{G2} = \text{Graph}(\text{node}_1, \text{node}_2) \bowtie \text{Triple}(\text{node}_2, \text{node}_3, \text{node}_4)$$
$$- \text{Triple}(\text{node}_1, \text{node}_2, \text{node}_3) \bowtie \text{Graph}(\text{node}_3, \text{node}_4)$$

$$Q_{G3} = \text{Triple}(\text{node}_1, \text{node}_2, \text{node}_3) - \text{Graph}(\text{node}_1, \text{node}_2) \bowtie \text{Graph}(\text{node}_2, \text{node}_3) \bowtie \text{Graph}(\text{node}_3, \text{node}_1)$$

$$Q_{G4} = \text{Triple}(\text{node}_1, \text{node}_2, \text{node}_3)$$
$$- \pi_{\text{node}_1,\text{node}_2,\text{node}_3}\text{Graph}(\text{node}_1, \text{node}_2) \bowtie \text{Graph}(\text{node}_2, \text{node}_3) \bowtie \text{Graph}(\text{node}_3, \text{node}_4)$$

$$Q_{G5} = \text{Graph}(\text{node}_1, \text{node}_2) \bowtie \text{Graph}(\text{node}_2, \text{node}_3) \bowtie \text{Graph}(\text{node}_3, \text{node}_4)$$
$$- \text{Graph}(\text{node}_2, \text{node}_3) \bowtie \text{Graph}(\text{node}_3, \text{node}_4) \bowtie \text{Graph}(\text{node}_4, \text{node}_1)$$

$$Q_{G6} = \text{Graph}(\text{node}_1, \text{node}_2) \bowtie \text{Graph}(\text{node}_3, \text{node}_4)$$
$$- \text{Graph}(\text{node}_1, \text{node}_2) \bowtie \text{Graph}(\text{node}_2, \text{node}_3) \bowtie \text{Graph}(\text{node}_3, \text{node}_1) \bowtie \text{Graph}(\text{node}_3, \text{node}_4)$$

Fig. 4. Graph queries.

relation Triple(node1, node2, node3) from the graph. Tuples in Triple are generated by following rules: (rule 1) a random length-2 path in the graph as (node1, node2, node3); or (rule 2) a random edge in the graph as (node1, node2), together with a random vertex in the graph as node3; or (rule 3) a triple (node1, node3, node5) from a random length-4 path (node1, node2, node3, node4, node5) in the graph. Triple may involve different portions of tuples generated by three rules in different queries. For a graph with n length-2 paths, we set the size of Triple to be $0.05n$ for Wiki (since it is too large to process as shown in Table 2), and $0.5n$ for the remaining graphs. We evaluate 6 graph queries as described in Figure 5, whose original SQL queries as well as optimized SQL queries after rewriting are given in the full version [10].

More specifically, Q_{G1} finds all edges in the graph that do not participate in any length-2 path. Q_{G2} finds all length-3 paths that the third node (node_3) is not sampled together with the edge ($\text{node}_1, \text{node}_2$). Q_{G3} finds length-2 paths that do not form a triangle. Q_{G4} finds all generated triples that cannot extend to a length-4 path. Q_{G5} finds all length-4 paths that do not form a length-4 cycle. Q_{G6} finds all pairs of edges in the graph, which do not form a length-4 cycle.

6.3 Experiment Results

Running time. Figure 5 shows the running time of different engines on graph queries. The input and output size of all graphs queries are given in Table 2. All bars reaching the axis boundary indicate that the system did not finish within the 8-hour limit, or ran out of memory. As Q_{G6} contains an expensive Cartesian product as sub-query, materializing its query result exceeds the memory capacity of our machines on most datasets. PostgreSQL can only evaluate the original SQL query of Q_{G6} on Bitcoin dataset. By adding the parallelism from 8 to 80, our optimized Spark SQL can evaluate Q_{G6} on Epinions dataset within the time limit, while the vanilla Spark SQL cannot complete the evaluation. For Q_{G5}, all systems cannot finish the evaluation on Wiki dataset due

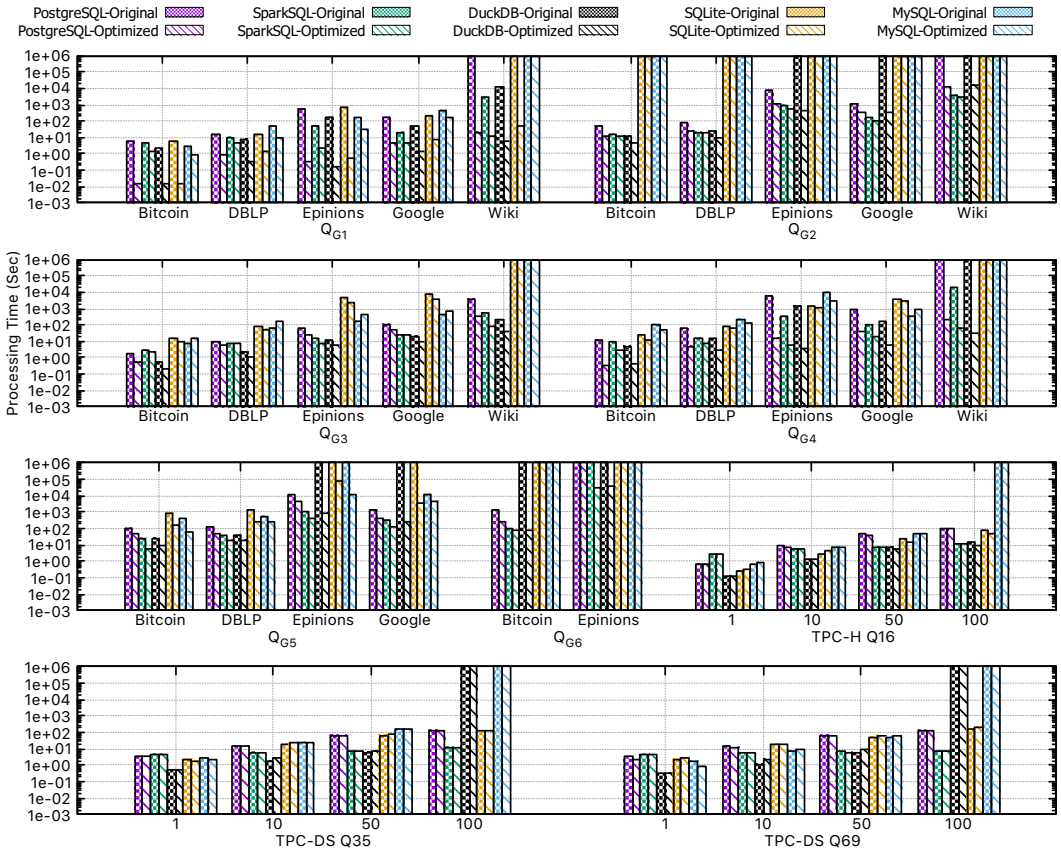


Fig. 5. Running time of graph and benchmark queries.

to the large intermediate results created. We also observe that both SQLite and MySQL cannot finish all test points for Q_{G2} and Q_{G6}, and most test points over Wiki dataset. It could be the reason that both systems are not designed for analytical queries. Our optimization techniques already achieve a speedup ranging from 2x to 1760x on PostgreSQL, from 1.2x to 270x on Spark SQL, from 2x to 1848x on DuckDB, from 1.25x to 1095x on SQLite, and from 1.8x to 5.1x on MySQL for graph queries, even without considering the queries that could not finish within the time limit. We also observe an unusual test point for Q_{G3} in MySQL, that our optimized SQL query takes more time than the vanilla SQL query, which may be due to some unknown deficiencies in MySQL internals.[2]

Figure 5 also shows the running time of all query engines on benchmark queries under different scale factors (i.e., parameters used to generate benchmark dataset, which is roughly proportional to the input data size). DuckDB and MySQL fail to finish some test points with scale factor 100. However, the improvement in benchmark queries achieved by our optimized techniques is minor, as expected. More specifically, the vanilla benchmark query consists of two free-connex sub-queries, hence can be evaluated in $O(N + \text{OUT}_1 + \text{OUT}_2)$ time, and its optimized query can be evaluated in $O(N + \text{OUT})$ time. Due to the special primary-key foreign-key joins and group-by aggregations,

[2]We review the execution plan in MySQL and find that the predicated run-time of our optimized SQL query is much smaller than the vanilla SQL query, which is also consistent with our observations in other platforms. The actual running time does not match the expected cost because of some unknown deficiencies in MySQL.

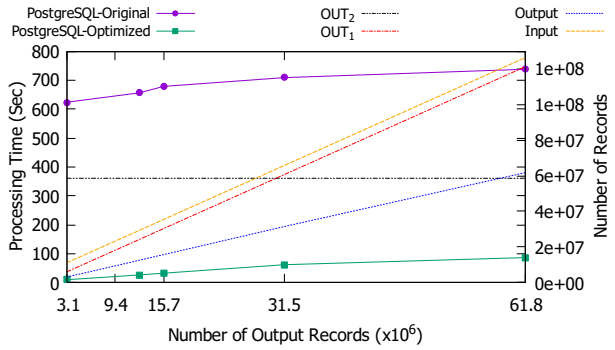


Fig. 6. Running time of Q_{G4} on Google dataset in PostgreSQL with different OUT_1.

$\text{OUT}_1 \approx \text{OUT}_2 \approx \text{OUT} << N$, such that the input contains a few hundred of million records while the query result only involves thousands of records. The improvement of our optimized techniques in SQLite, DuckDB, and MySQL is also limited. On some test points, our optimized SQL queries are even more time-consuming than vanilla SQL queries. We find that the vanilla SQL queries can greatly benefit from the indices built for primary-key foreign-key join and outperform our optimized SQL queries, which do not enjoy efficient indices for set difference or anti-join operators. How to build indices to accelerate relational operators in these systems could be interesting future work. Meanwhile, we notice that loading input data and building indices are much more time-consuming than evaluating the query; for example, it takes DuckDB 16 minutes to load a 50G-sized TPC-DS dataset, while only 8 seconds to execute the whole query.

Impact of OUT**,** OUT_1 **and** OUT_2**.** Implied by the theoretical results, the sizes $\text{OUT}_1, \text{OUT}_2$ of sub-queries Q_1, Q_2 impact the performance of vanilla SQL queries, while only the actual output size OUT affect the performance of our optimized SQL queries. Below, we study the impact of OUT_1, OUT_2 and OUT on the performance of both approaches over Q_{G4}.

In Figure 6, we investigate the impact of OUT_1 for computing DCQ. We fix Q_2 (as well as OUT_2) and only vary the size of Triple (as well as N and OUT_1). Note that OUT also increases as OUT_2 decreases. The running time of our optimized SQL query grows slowly with OUT, while the vanilla SQL query incurs a fixed overhead for evaluating Q_2, even when OUT_1 (as well as OUT) decreases to as small as 1.

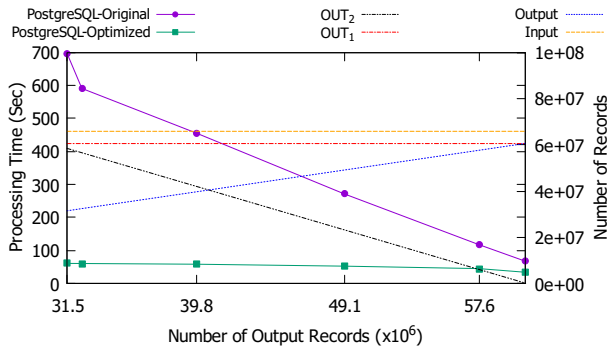


Fig. 7. Running time of Q_{G4} on Google dataset in PostgreSQL with different OUT_2.

In Figure 7, we investigate the impact of OUT_2 for computing DCQ. We fix Q_1 (as well as N and OUT_1) and vary a filter predicate applied to relation Graph in Q_2. When the predicate is more selective, OUT_2 becomes smaller, and OUT becomes larger. The running time of vanilla SQL query decreases as OUT_2 decreases, and the running time of our optimized SQL query does not change, which is only affected by N and OUT.

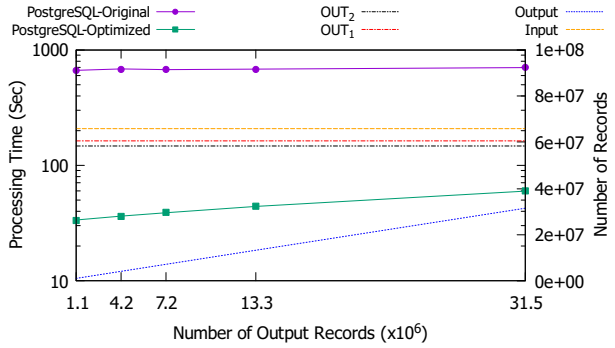


Fig. 8. Running time of Q_{G4} on Google dataset in PostgreSQL with different OUT.

In Figure 8, we investigate the impact of OUT for computing DCQ. We adjust Triple by changing the proportion of tuples generated by different rules, which will only change OUT, while OUT_1, OUT_2, and N stay the same. The running time of our optimized SQL query increases slowly as OUT increases. In contrast, the running time of vanilla SQL query remains stably high even when OUT decreases to 1, since its running time is only impacted by OUT_1 and OUT_2, both of which stay unchanged.

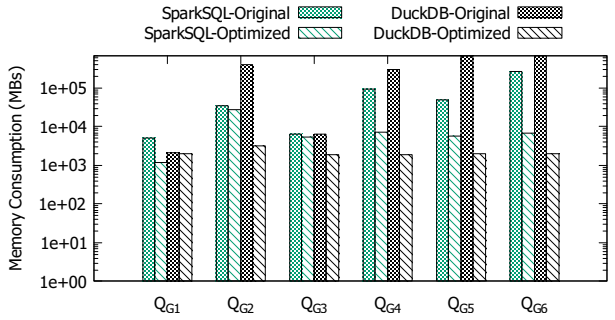


Fig. 9. Memory usage by graph queries on Epinions.

Memory Consumption. We also test the memory consumption on both graph and benchmark queries by different engines. Due to the simplicity of memory consumption measurement, we report the results for PostgreSQL and DuckDB here. For benchmark queries, the optimized and vanilla SQL queries have similar behaviors on memory consumption, since the input size dominates the overall consumption. Below, we focus on the memory consumption of graph queries. In Figure 9, our optimized SQL queries achieve overall improvements for all graph queries on Epinions dataset in terms of space consumption. For example, our optimized SQL query only requires 6.53GB on Spark SQL for evaluating Q_{G6}, while the vanilla SQL query fails to finish evaluating Q_{G6} even using

256G memory. The improvement of our optimized SQL query is more significant on DuckDB. For Q_{G4}, our optimized SQL query consumes 99.4% less memory than the vanilla SQL query. For Q_{G5} and Q_{G6}, our optimized SQL queries consume roughly 2G memory. In contrast, the vanilla SQL queries fail to execute due to out-of-memory errors even after using 738G memory.

7 CONNECTION WITH SIGNED CONJUNCTIVE QUERY

The class of *signed conjunctive queries (SCQ)* [19], or noted as *conjunctive queries with negation* [32] in the literature, is defined as

$$Q := \pi_{\mathbf{y}} \left(\eta_1 R_1(e_1) \bowtie \cdots \bowtie \eta_2 R_2(e_2) \bowtie \cdots \bowtie \eta_n R_n(e_n)\right),$$

where η_i is either empty or a negation operator $\neg$. If $\eta_i = \neg$ for all $i \in [n]$, such an SCQ is also known as a *negative conjunctive queries* (NCQ). If $\eta_i = \emptyset$ for all $i \in [n]$, such an SCQ is also known as a CQ. Recall that $\mathcal{V} = e_1 \cup e_2 \cup \cdots \cup e_n$ and $\mathcal{E} = \{e_1, e_2, \cdots, e_n\}$. The query result of Q over an instance D denoted as $Q(D)$ is defined as

$$Q(D) = \{t \in \mathrm{dom}(\mathbf{y}) : \exists t' \in \mathrm{dom}(\mathcal{V}), \pi_{e_i} t \in R_i, \forall \eta_i = \emptyset, \pi_{e_j} t \notin R_j, \forall \eta_j = \neg\}.$$

We establish the connection between SCQ and DCQ via Lemma 7.1 and Lemma 7.2.

From DCQ to SCQ. Intuitively, every DCQ can be rewritten as the union of a set of SCQs. Moreover, each resulted SCQ has exactly one negated relation, and each relation of Q_2 participates in one distinct SCQ as the negated relation. For example, $Q_1 - Q_2 = R_1(x_1, x_2) \bowtie R_2(x_2, x_3) - R_3(x_1, x_2) \bowtie R_4(x_2, x_3)$ can be rewritten as $(R_1 \bowtie R_2 \bowtie \neg R_3) \cup (R_1 \bowtie R_2 \bowtie \neg R_4)$.

Lemma 7.1. *For a DCQ $Q_1 - Q_2$, $Q_1 - Q_2 = \bigcup_{e \in \mathcal{E}_2} (Q_1 \bowtie \neg R_e)$.*

Proof. Direction $\subseteq$. For every join result $t \in Q_1 - Q_2$, there must exist a relation $e \in \mathcal{E}_2$ such that $\pi_e t \notin R_e$; otherwise, $t \in Q_2$, coming to a contradiction. Wlog, let $e \in \mathcal{E}_2$ be such a relation for t. Together with $t \in Q_1$, there must be $t \in Q_1 \bowtie \neg R_e$. Direction $\supseteq$. Consider an arbitrary relation $e \in \mathcal{E}_2$, and an arbitrary join result $t \in Q_1 \bowtie \neg R_e$. Obviously, $t \notin Q_2$ since $\pi_e t \notin R_2$. Together with $t \in Q_1$, there must be $t \in Q_1 - Q_2$. □

From SCQ to DCQ. On the other hand, SCQ can be rewritten as the intersection of a set of DCQs. For a SCQ $Q = (\mathcal{V}, \mathcal{E})$, let $\mathcal{E}^+, \mathcal{E}^- \subseteq \mathcal{E}$ denote the set of relations with positive, negative sign separately. Let $\mathcal{V}^+, \mathcal{V}^-$ be the set of attributes that appear in positive, negative relations separately. Let $Q^+ = (\bowtie_{e' \in \mathcal{E}^+} R_{e'} \times_{x \in \mathcal{V}^- - \mathcal{V}^+} \mathrm{dom}(x))$ denote the *positive subquery* defined by positive relation as well as the whole domain of attributes which do not appear in any positive relation.

Lemma 7.2. *For a SCQ Q, $Q = \cap_{e \in \mathcal{E}^-} (Q^+ - Q^+ \bowtie R_e)$.*

Proof. Direction $\subseteq$. Consider an arbitrary query result $t \in Q$. By definition, $\pi_e t \in R_e$ holds for every $e \in \mathcal{E}^+$ and $\pi_e t \notin R_e$ holds for every $e \in \mathcal{E}^-$. This way, for each $e \in \mathcal{E}^-$, we have $t \in (\bowtie_{e' \in \mathcal{E}^+} R_{e'} \times_{x \in \mathcal{V}^- - \mathcal{V}^+} \mathrm{dom}(x)) \bowtie \neg R_e$. Direction $\supseteq$. Consider an arbitrary t such that for every $e \in \mathcal{E}^-$, $t \in (\bowtie_{e' \in \mathcal{E}^+} R_{e'} \times_{x \in \mathcal{V}^- - \mathcal{V}^+} \mathrm{dom}(x)) \bowtie \neg R_e$. Then, $\pi_{e'} t \in R_{e'}$ for every $e' \in \mathcal{E}^+$ but $t \notin R_e$ for every $e \in \mathcal{E}^-$. Thus, $t \in Q$. □

For example, a SCQ $Q = R_1(x_2, x_3, x_4) \bowtie R_2(x_1, x_3, x_4) \bowtie \neg R_3(x_1, x_2, x_4) \bowtie \neg R_4(x_1, x_2, x_3)$ can be rewritten as: $(R_1 \bowtie R_2 - R_1 \bowtie R_2 \bowtie R_3) \cap (R_1 \bowtie R_2 - R_1 \bowtie R_2 \bowtie R_4)$.

Decidability of SCQ. Given a SCQ Q, the domain of attributes, and input database D, the *decidability* problem asks to decide whether there exists a query result in Q. For example, a NCQ $Q = \neg R_1(x_1, x_2) \bowtie \neg R_2(x_2, x_3)$ decides if there exists any tuple $(a, b, c) \in \mathrm{dom}(x_1) \times \mathrm{dom}(x_2) \times \mathrm{dom}(x_3)$ such that $(a, b) \notin R_1$ and $(b, c) \notin R_2$, and a CQ $Q = R_1(x_1, x_2) \bowtie R_2(x_2, x_3)$ decides if there exists

any tuple (a, b, c) such that $(a, b) \in R_1$ and $(b, c) \in R_2$. The decidability problem for CQ , NCQ and SCQ has been well studied separately:

Theorem 7.3 ([16]). *A CQ Q can be decided in linear time if and only if it is α-acyclic.*

Theorem 7.4 ([19]). *A NCQ Q can be decided in linear time if and only if it is β-acyclic.*

Theorem 7.5 ([20]). *A SCQ Q can be decided in linear time if and only if $(\mathbf{y}, \mathcal{E}^+ \cup S)$ is α-acyclic for every $S \subseteq \mathcal{E}^-$.*

Note that β-acyclicity is a more restricted notion than α-acyclicity, such that Q is β-acyclic if all sub-hypergraphs of Q are α-acyclic. Obviously, β-acyclicity strictly implies α-acyclicity. In [32], this notion of β-acyclicity has been extended to nest-set width for capturing the tractability of SCQ in terms of both query and data complexity. We won't pursue this direction further.

Decidability of DCQ. Implied by Lemma 7.1 and Theorem 7.5, we come to the following lemma:

Lemma 7.6. *Given two full joins $Q_1 = (\mathbf{y}, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{E}_2)$, the DCQ $Q_1 - Q_2$ can be decided in linear time, if $(\mathbf{y}, \mathcal{E}_1)$ is α-acyclic, and $(\mathbf{y}, \mathcal{E}_1 \cup \{e\})$ is α-acyclic for every $e \in \mathcal{E}_2$.*

Lemma 7.6 can be easily proved by a linear-time algorithm. Note that $Q_1 - Q_2 = \bigcup_{e \in \mathcal{E}_2} (Q_1 \bowtie \neg R_e)$. We can enumerate every tuple in $S_e = \pi_e Q_1$ within $O(1)$ delay, as $(\mathbf{y}, \mathcal{E}_1 \cup \{e\})$ is α-acyclic. For each tuple $t \in S_e$ enumerated, we check whether it belongs to R_e. If $t \notin R_e$, a query result of $Q \bowtie \neg R_e$ is found; otherwise, we skip it and continue to the next one. It is easy to see that at most $O(|R_e|)$ tuples are checked, so this algorithm runs in $O(N)$ time.

Theorem 7.7. *Given two full joins $Q_1 = (\mathbf{y}, \mathcal{E}_1)$ and $Q_2 = (\mathbf{y}, \mathcal{E}_2)$, the DCQ $Q_1 - Q_2$ can be decided in linear time, if and only if $(\mathbf{y}, \mathcal{E}_1)$ is α-acyclic, as well as $(\mathbf{y}, \mathcal{E}_1 \cup \{e\})$ is α-acyclic for every $e \in \mathcal{E}_2$.*

Proof. The if direction follows Lemma 7.6. We next distinguish two more cases for the only-if direction. (1) if $(\mathbf{y}, \mathcal{E}_1)$ is cyclic; and (2) if $(\mathbf{y}, \mathcal{E}_1)$ is α-acyclic, and there exists some $e \in \mathcal{E}_2$ such that $(\mathbf{y}, \mathcal{E}_1 \cup \{e\})$ is cyclic. (1) follows Theorem 7.3 by simply setting $Q_2 = \emptyset$. (2) follows Lemma 4.7. □

8 RELATED WORK

Union of CQs. [21] studied the enumeration complexity of union of conjunctive queries (UCQs), i.e., the goal is to find a data structure that after linear preprocessing time, the query answers (without duplication) can be enumerated within a small delay. Their results implied a linear algorithm in terms of input and output size for the class of union-free-connex UCQs, but whether a linear algorithm can be achieved (and, if possible, how to achieve it) is unknown for the remaining class of UCQ. [23] also investigated the enumeration complexity of UCQs but in the dynamic scenario.

Selection over CQs. Recently, multiple works have studied the complexity of selections over conjunctive queries. [37] investigated the selection in the form of comparisons between two attributes or values. The work identifies an acyclic condition under which a near-linear-time algorithm can be achieved for conjunctive queries with comparisons. [13] worked on the selections over intervals, also known as intersection queries, which are special cases for comparison queries since each intersection query can be decomposed into a union of multiple comparison queries. They show a dichotomy result that an intersection join can be computed in linear time if and only if it is ι-acyclic. [28] studied the complexity of temporal queries, where the intersection condition only exists for one global attribute. Their result suggested that a temporal query can be solved in linear time if and only if it is r-hierarchical. [35] also investigated the complexity of intersection queries in dynamic settings.

REFERENCES

[1] DuckDB. https://duckdb.org/.
[2] MySQL. https://www.mysql.com/.
[3] Oracle. https://www.oracle.com/.
[4] PostgreSQL. https://www.postgre.org/.
[5] SNAP. https://snap.stanford.edu/snap/.
[6] SparkSQL. https://spark.apache.org/sql/.
[7] SQLite. https://www.sqlite.org/.
[8] TPC-DS. https://www.tpc.org/tpcds/.
[9] TPC-H. https://www.tpc.org/tpch/.
[10] https://arxiv.org/abs/2302.13140.
[11] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular conjectures imply strong lower bounds for dynamic problems. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*. IEEE, 434–443.
[12] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.
[13] Mahmoud Abo Khamis, George Chichirim, Antonia Kormpa, and Dan Olteanu. 2022. The Complexity of Boolean Conjunctive Queries with Intersection Joins. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Philadelphia, PA, USA) *(PODS '22)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3517804.3524156
[14] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. 2016. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 13–28.
[15] Rasmus Resen Amossen and Rasmus Pagh. 2009. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*. 121–126.
[16] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*. Springer, 208–222.
[17] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. 1983. On the desirability of acyclic database schemes. *Journal of the ACM (JACM)* 30, 3 (1983), 479–513.
[18] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. 2014. Listing triangles. In *International Colloquium on Automata, Languages, and Programming*. Springer, 223–234.
[19] Johann Brault-Baron. 2012. A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In *Computer Science Logic (CSL'12)-26th International Workshop/21st Annual Conference of the EACSL*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[20] Johann Brault-Baron. 2013. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. Ph. D. Dissertation. Université de Caen.
[21] Nofar Carmeli and Markus Kröll. 2019. On the Enumeration Complexity of Unions of Conjunctive Queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 134–148.
[22] Chandra Chekuri and Anand Rajaraman. 2000. Conjunctive query containment revisited. *Theoretical Computer Science* 239, 2 (2000), 211–229.
[23] Berkholz Christoph, Keppeler Jens, and Schweikardt Nicole. 2018. Answering UCQs under Updates and in the presence of integrity constraints. In *Proceedings of the 21st International Conference on Database Theory (ICDT'18)*, Vol. 98. 1–8.
[24] Shaleen Deep, Xiao Hu, and Paraschos Koutris. 2020. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1213–1223.
[25] R. Fagin. 1983. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM* 30, 3 (1983), 514–550.
[26] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree decompositions and tractable queries. *J. Comput. System Sci.* 64, 3 (2002), 579–627.
[27] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. 2009. Generalized hypertree decompositions: NP-hardness and tractable variants. *Journal of the ACM (JACM)* 56, 6 (2009), 1–32.
[28] Xiao Hu, Stavros Sintos, Junyang Gao, Pankaj K. Agarwal, and Jun Yang. 2022. Computing Complex Temporal Join Queries Efficiently. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2076–2090. https://doi.org/10.1145/3514221.3517893
[29] Zichun Huang and Shimin Chen. 2022. Density-Optimized Intersection-Free Mapping and Matrix Multiplication for Join-Project Operations. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2244–2256. https://doi.org/10.14778/3547305.3547326
[30] Manas R. Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. AJAR: Aggregations and Joins over Annotated Relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (San Francisco, California, USA) *(PODS '16)*. Association for Computing Machinery, New York, NY, USA, 91–106. https://doi.org/10.1145/2902251.2902293

[31] Phokion G Kolaitis and Moshe Y Vardi. 1998. Conjunctive-query containment and constraint satisfaction. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 205–213.
[32] Matthias Lanzinger. 2021. Tractability Beyond ß-Acyclicity for Conjunctive Queries with Negation. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 355–369.
[33] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40.
[34] Christos H Papadimitriou and Mihalis Yannakakis. 1997. On the complexity of database queries. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 12–19.
[35] Yufei Tao and Ke Yi. 2022. Intersection joins under updates. *J. Comput. System Sci.* 124 (2022), 41–64. https://doi.org/10.1016/j.jcss.2021.09.004
[36] Moshe Y Vardi. 1982. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. 137–146.
[37] Qichen Wang and Ke Yi. 2022. Conjunctive Queries with Comparisons. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3514221.3517830
[38] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.

Received October 2022; revised January 2023; accepted February 2023