# Aggregated Deletion Propagation for Counting Conjunctive Query Answers

Xiao Hu, Shouzhuo Sun, Shweta Patwa, Debmalya Panigrahi, and Sudeepa Roy
Duke University, Durham, NC, USA
{xh102, ss1060, sjpatwa, debmalya, sudeea}@cs.duke.edu

**Abstract**

We investigate the computational complexity of minimizing the source side-effect in order to remove a given number of tuples from the output of a conjunctive query. This is a variant of the well-studied *deletion propagation* problem, the difference being that we are interested in removing the smallest subset of input tuples to remove a given number of output tuples while deletion propagation focuses on removing a specific output tuple. We call this the *Aggregated Deletion Propagation* problem. We completely characterize the poly-time solvability of this problem for arbitrary conjunctive queries without self-joins. This includes a poly-time algorithm to decide solvability, as well as an exact structural characterization of NP-hard instances. We also provide a practical algorithm for this problem (a heuristic for NP-hard instances) and evaluate its experimental performance on real and synthetic datasets.

## 1 Introduction

The problem of *view update* (e.g., [**2, 10**]) – how to change the input to achieve desired changes to the query output or *view* – is a well-studied problem in the database literature. View update problems enable users to tune the output in order to meet their prior expectation, satisfy external constraints, or examine and compare multiple options. A particularly well-studied class of view update problems is what is known as *deletion propagation* problems (see Buneman, Khanna, and Tan [**3**]; for follow up literature, see related work). In these problems, the goal is to remove a specific tuple from the output of a query by removing input tuples. In this paper, we study a natural variant of this problem where we seek to remove *at least a given number of output tuples* rather than any specific output tuple. We call this the *Aggregated Deletion Propagation* problem.

Formally, in the *Aggregated Deletion Propagation* (`ADP`), we are given a query $Q$, a database $D$, and a target integer $k$. The goal is to remove at least $k$ tuples from $Q(D)$ by removing the *minimum number of input tuples* from $D$ (this objective is called *source side-effect* in the literature). Our main motivation for the `ADP` problem comes from two generic application settings. First, `ADP` can be used to obtain a desired change in the *output size* with minimum intervention on the input. As we will describe below, in many practical situations, the goal is to create a sufficiently large impact on the output by removing a given number of output tuples rather than removing any specific tuple. Our problem applies to these situations. Second, `ADP` can be used to analyze the *robustness* of the output with respect to possible disruptions in the input. In other words, if there are inadvertent changes to the input that are not within our control, how badly can it effect the output of a query? We give examples of these two applications below.

**Example 1.** *Suppose a university wants to plan ahead in terms of managing waitlists for its classes. This can be achieved via the following query:*

$$Q_{WL}(S, C) : -Major(S, M), Req(M, C), NoSeat(C)$$

*The first query $Q_{WL}$ says that a student $S$ is on the waitlist for a class $C$ if the following happen: (1) $S$ intends to major in $M$ (we assume students can have multiple majors), (2) major $M$ requires class $C$, and (3) there are no seats available in $C$. The university may try to figure out the easiest alternative for reducing the size of the waitlist to some target, which amounts to reducing the size of the output of query $Q_{WL}$ by the* *same amount. The waitlist entries can be removed by steering students away from the major (or creating an entry condition), relaxing the requirements for the major, or by increasing the number of seats in the class; all of these options correspond to removing tuples from the input relations of $Q_{WL}$.*

**Example 2.** *We consider the same context as in the previous example, but suppose the new task is to estimate what classes can be reliably offered in a future semester. This can be done using the following query*

$$Q_{Possible}(C) : -Teaches(P, C), NotOnLeave(P).$$

*This query lists the possible courses that can be offered in a semester. A course $C$ can be offered if there is a professor $P$ who is able to teach $C$ and is not on leave. If all professors who are able to teach $C$ go to leave (removal of entries from NotOnLeave) or do not want to teach $C$ (removal of entries from Teaches), $C$ cannot be offered. While approving the leave requests and asking for teaching preferences, the university may want to study the robustness of $Q_{Possible}$ with respect to these changes: e.g., what is the minimum changes in the input that would lead to more than 10% of the courses not being able to be offered in that semester. If this size is small, i.e., many courses are critically dependent on a few professors, the university would be able to decide whether all can be on leave or change teaching preferences appropriately. Alternatively, this information might also inform the decision to hire faculty in a particular area.*

**Example 3.** *We now turn to a third example from the area of robustness of networks. Consider a query*

$$Q_{3-path}(A, B, C, D) : -R_1(A, B), R_2(B, C), R_3(C, D)$$

*that stores all possible paths between two end vertices that go through two layers of intermediate vertices in a communication or transportation network. If it were possible to disrupt (say) 80% of the paths by only removing (say) 1% of links, then the network is clearly not robust. On the other hand, if this would require removing (say) 80% of the links, that's a much more robust network. This is precisely the information the `ADP` can provide us on this query. Therefore, `ADP` can estimate the inherent robustness of a network to either malicious attacks or even just random failures.*

**Our contributions.** In this paper, we propose the `ADP` problem and study its complexity in depth for the class of *conjunctive queries without self-joins* (CQ). Here, the results can be an arbitrary projection of the *natural join* of the relations appearing on the body of the query (as illustrated in $Q_{WL}$, $Q_{Possible}$, and $Q_{3-path}$ above). Our contributions can be summarized as follows:

- **Algorithmic Dichotomy:** We give an algorithm that only takes the query $Q$ as input, and decides in time that is polynomial in the size of the query, whether `ADP` can be efficiently solved (in polynomial time data complexity [**26**]) on $Q$ for all instances $D$ and all values of $k$. The algorithm uses a few simplification steps that preserve the complexity of the problem. At the end, the query is NP-hard if the simplification steps reduce it to a small number of 'core' hard queries; otherwise, it is poly-time solvable. **(Section 4)**

- **Structural Dichotomy:** To complement our algorithmic characterization of the complexity of the `ADP` problem, we also provide a structural characterization of the complexity by identifying three simple structures – *triad-like*, *non-hierarchical head join*, and *strand* – whose presence exactly captures all queries where `ADP` in NP-hard. **(Section 5)**

- **Approximation:** We study the approximation for the `ADP` problem when it is NP-hard. We show that greedy and prime-dual achieve approximation factors of $O(\log k)$ and $p$ respectively for full CQs, where $p$ is the number of relations in the input query. Meanwhile, we present some inapproximability result when projection exists, such that obtaining even sub-polynomial approximations for the `ADP` problem on general CQs is unlikely. **(Section 6)**

- **Efficient unified algorithm:** We give a poly-time (in data complexity) algorithm for solving `ADP` for all CQs without self-joins. It returns the optimal solution for queries on which `ADP` is poly-time solvable, and provides a poly-time heuristic for queries on which `ADP` is NP-hard. We also extend the algorithm to support *selection* operations. **(Section 7)**

- **Experimental evaluations:** We provide experimental evaluation of our algorithms on synthetic and real datasets in terms of efficiency, quality, scalability, various classes of queries as well as data distribution. **(Section 8)**

## 2 Related Work

The classical view update problem, of which deletion propagation is an instantiation, has been studied extensively over the last four decades (e.g., [**2, 10**]). The deletion propagation problem has been popular more recently, starting with the seminal work by Buneman, Khanna, and Tan [**3**]. They studied the complexity of both the *source side-effect* (objective is to delete the *minimum number of input tuples*) and the *view side-effect* (objective is to delete the *minimum number of other output tuples*) versions, in order to delete a particular output tuple. For source side-effect and select-project-join-union (SPJU) operators, they showed that for PJ or JU queries, finding the optimal solution is NP-hard, while for others (e.g., SPU or SJ) it is poly-time solvable. This work was extended to multi-tuple deletion propagation by Cong, Fan, and Geerts [**8**]. They showed that for single tuple deletion propagation, a property called *key preservation* makes the problem tractable for SPJ views; however, if multiple tuples are to be deleted, the problem becomes intractable for SJ, PJ, and SPJ views. Kimelfeld, Vondrak, and Williams [**15, 14, 16**] extensively studied the complexity of deletion propagation for the view side-effect version and provided structural dichotomy and trichotomy (poly-time, APX-hard/constant approximation, and inapproximable) for single and multiple output tuple deletions.

Beyond the context of deletion propagation, several dichotomy results have been obtained for problems motivated by data management, e.g., in the context of probabilistic databases [**9**], responsibility [**21**], or database repair [**19**]. Another problem related to `ADP` is *reverse data management* and *how-to* queries [**22, 23**]. Given some desired changes in the output (e.g., modifying aggregate values, creating or removing tuples), the goal is to obtain a feasible modification of the input that satisfies a given set of constraints and optimizes on some criteria. In this line of research, the focus has been on developing an end-to-end system using provenance and mixed integer programming, and not on the complexity of the problem. `ADP` is also related to explanations by intervention [**28, 25, 24**], where the goal is to find a set of input tuples captured by a predicate whose deletion changes one or more aggregate answers to the maximum extent. `ADP` differs in that the aim is to make a desired change in the output by removing the minimum number of input tuples.

Finally, closely related to the `ADP` is the *resilience* problem, originally studied by Freire et al. for the class of CQs without self-joins and functional dependencies [**11**] (see also [**12**] for an extension to a class of queries with self-joins). The input to the resilience problem is a Boolean CQ and a database $D$ such that $Q(D)$ is true, and the goal is to remove a minimum set of tuples from $D$ to make $Q$ false on $D$. Observe that the resilience problem is identical to `ADP` with $k = |Q(D)|$. [**11**] gave a "structural dichotomy" characterizing whether a given query is poly-time solvable or NP-hard using a core hard structure called "triad". The generalization to arbitrary values of $k$ leads to interesting consequences, e.g., queries that are poly-time solvable for resilience become hard for `ADP`), whereas the presence of arbitrary projections in the output makes `ADP` even more NP-hard for `ADP`. Nevertheless, we use the characterization for resilience from [**11**] as a special case of our algorithmic and structural characterization for `ADP` and discuss the resilience problem further in subsequent sections.

## 3 Preliminaries

In this section, we start with some basic definitions in relational databases. Then, we formally define the `ADP` problem and discuss some special cases that will motivate our general technique.

### 3.1 Background

We consider the standard setting of multi-relational data-bases and conjunctive queries. Let $\mathbb{R}$ be a database schema that contains $p$ tables $R_1, \cdots, R_p$. Let $\mathbb{A}$ be the set of all attributes in the database $\mathbb{R}$. Each relation $R_i$ is defined on a subset of attributes $\texttt{attr}(R_i) \subseteq \mathbb{A}$. A relation $R_i$ is *vacuum* if $\texttt{attr}(R_i) = \emptyset$, and *non-vacuum* otherwise. We use $A, B, C, A_1, A_2, \cdots$ etc. to denote the attributes in $\mathbb{A}$ and $a, b, c, \cdots$ etc. to denote their values. For each attribute $A \in \mathbb{A}$, $\texttt{rels}(A)$ denotes the set of relations that $A$ appears, i.e., $\texttt{rels}(A) = \{R_i : A \in \texttt{attr}(R_i)\}$.

Given the database schema $\mathbb{R}$, let $D$ be a given instance of $\mathbb{R}$, and the corresponding instances of $R_1, \cdots, R_p$ be $R_1^D, \cdots, R_p^D$. Where $D$ is clear from the context, we will drop the superscript and use $R_1, \cdots, R_p$ for both the schema and instances. Any tuple $t \in R_i$ is defined on $\texttt{attr}(R_i)$. For any attribute

$R_1$

| A | B |
|---|---|
| a1 | b1 |
| a2 | b2 |
| a3 | b3 |

$R_2$

| B | C |
|---|---|
| b1 | c1 |
| b2 | c2 |
| b2 | c3 |
| b3 | c3 |

$R_3$

| C | E |
|---|---|
| c1 | e1 |
| c2 | e3 |
| c3 | e3 |

$Q_1(D)$

| A | B | C | E |
|---|---|---|---|
| a1 | b1 | c1 | e1 |
| a2 | b2 | c2 | e3 |
| a2 | b2 | c3 | e3 |
| a3 | b3 | c3 | e3 |

$Q_2(D)$

| A | E |
|---|---|
| a1 | e1 |
| a2 | e3 |
| a3 | e3 |

Figure 1: An example of database schema $\mathbb{R} = \{R_1, R_2, R_3\}$ with $\mathbb{A} = \{A, B, C, E\}$, $\texttt{attr}(R_1) = \{A, B\}$, $\texttt{attr}(R_2) = \{B, C\}$, and $\texttt{attr}(R_3) = \{C, E\}$. An instance $D$ with 10 tuples is also shown. The results for $Q_1(A, B, C, E) : -R_1(A, B), R_2(B, C), R_3(C, E)$ and $Q_2(A, E) : -R_1(A, B), R_2(B, C), R_3(C, E)$ are $Q_1(D)$ and $Q_2(D)$.

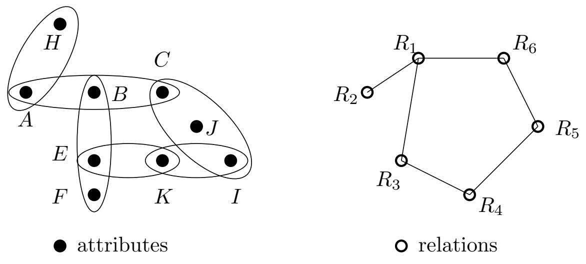


Figure 2: Hypergraph (left) and graph (right) representation for an example CQ $Q(A, C, F, K) : -R_1(A, B, C), R_2(A, H), R_3(B, E, F), R_4(E, K), R_5(K, I), R_6(C, I, J)$.

$A \in \texttt{attr}(R_i)$, $\pi_A t \in \texttt{dom}(A)$ denotes the value of attribute $A$ in tuple $t$. Similarly, for a set of attributes $\mathbb{B} \subseteq \texttt{attr}(R_i)$, $\pi_{\mathbb{B}} t$ denotes the values of attributes in $\mathbb{B}$ for $t$ with an implicit ordering on the attributes. It should be noted that for a vacuum relation $R_i$, either $R_i = \{\emptyset\}$ or $R_i = \emptyset$ (respectively interpreted as "true" and "false").

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

$$Q(\mathbf{A}) : -R_1(\mathbb{A}_1), R_2(\mathbb{A}_2), \cdots, R_p(\mathbb{A}_p)$$

where $\mathbf{A} \subseteq \mathbb{A}$ denotes the *output attributes* and $\mathbb{A} - \mathbf{A}$ the *non-output attributes* (also called the *existential variables*). Note that we do not have any projection in the body. Each $R_i$ in $Q$ is distinct, i.e., the CQ does not have a self-join. If $\mathbf{A} = \mathbb{A}$, such a CQ query is known as *full CQ* which represents the natural join among the given relations. If $\mathbf{A} = \emptyset$, such a CQ is *boolean* which indicates whether the result of natural join among the given relations is empty or not; otherwise, it is *non-boolean*.

Extending the notation, we use $\texttt{rels}(Q)$ to denote all the relations that appear in the body of $Q$, $\texttt{attr}(Q)$ to denote all the attributes that appear in the body of $Q$, and $\texttt{head}(Q) \subseteq \texttt{attr}(Q)$ to denote all the attributes that appear in the head of $Q$ (so, $\texttt{head}(Q) = \mathbb{A}$ in the previous paragraph). When a full CQ query $Q$ is evaluated on an instance $D$, if $R_i = \emptyset$ for some vacuum relation $R_i \in \texttt{rels}(Q)$, then $Q(D)$ is also empty; otherwise, the result $Q(D)$ is evaluated on non-vacuum relations. When a CQ query $Q$ is evaluated on an instance $D$, the result is exactly the projection of the full join result on attributes in $\texttt{head}(Q)$ (after removing duplicates). We give an example in Figure 1.

A classical representation of a CQ $Q$ is to model it as a hypergraph, where each attribute in $\texttt{attr}(Q)$ is a vertex and each relation in $\texttt{rels}(Q)$ is a hyperedge. In this work, we use a simpler representation for capturing the *connectivity* of queries and model it as a graph $G_Q$, where each relation is a vertex and there is an edge between $R_i, R_j \in \texttt{rels}(Q)$ if $\texttt{attr}(R_i) \cap \texttt{attr}(R_j) \neq \emptyset$. This graph is denoted $G_Q$. A CQ $Q$ is *connected* if $G_Q$ is connected, and *disconnected* otherwise. An example is illustrated in Figure 2.

## 3.2 Problem Definition

Below, we formally define the ADP problem in terms of the count of output tuples of a CQ:

**Definition 1.** *Given a CQ $Q$ on $\mathbb{R}$, an instance $D$, and a positive integer $k \geq 1$, the* aggregated deletion propagation (`ADP`) *problem aims to remove at least $k$ results from $Q(D)$ by removing the minimum number of input tuples from $D$.*

Given $Q$, $k$, and $D$, we denote the above problem by $\texttt{ADP}(Q, D, k)$. Note that an implicit constraint on the input parameter $k$ is $1 \leq k \leq |Q(D)|$. For instance, in Figure 1, $\texttt{ADP}(Q_1, D, 2)$ will return a single tuple $R_3(c3, e3)$ since removing it would remove the last two output tuples in $Q_1(D)$. In this paper, we study the data complexity [**26**] of the `ADP` problem, i.e., the size of the query and schema are fixed, and the complexity is in terms of the size of the database $D$. More precisely, we say that $\texttt{ADP}(Q, D, k)$ is *polynomial-time solvable* for a query $Q$ if, for an arbitrary instance $D$ and integer $k$, the solution of $\texttt{ADP}(Q, D, k)$ can be computed in polynomial time in the size of $D$; otherwise, it is *NP-hard*.

For simplicity, we assume that all relations have distinct set of attributes in an input CQ $Q$, i.e., $\texttt{attr}(R_i) \neq \texttt{attr}(R_j)$ for every pair of relations $R_i, R_j \in \texttt{rels}(Q)$. The rationale is that removing duplicated relations won't change the poly-time solvability of the original CQ.

## 3.3 Special Cases

Before we discuss the complexity of the `ADP` problem in general, we note the following special cases:

**`ADP` on boolean CQ.** The `ADP` problem on boolean CQ is also known as the *resilience* problem, i.e., removing the minimum number of input tuples to make the true query become false. The next theorem in [**11**] gives a decidability result of the `ADP` problem on boolean CQ.

**Theorem 1** ([**11**])**.** *On a boolean CQ $Q$, the poly-time solvability (in data complexity) of the $\texttt{ADP}(Q, D, 1)$ problem can be decided in polynomial time (in query complexity).*

**`ADP` on CQ with vacuum relations.** The `ADP` problem becomes easy when $Q$ contains a vacuum relation. Consider an arbitrary input instance $D$ for $Q$ and integer $k$. If every vacuum relation in $Q$ has instance $\{\emptyset\}$, we can remove query results in $Q(D)$ by removing the tuple $\{\emptyset\}$ in any one vacuum relation; otherwise, $Q(D) = \emptyset$ by definition, and there is no need to remove anything. Therefore:

**Lemma 1.** *For a CQ $Q$, if there exists some vacuum relation, the $\texttt{ADP}(Q, D, k)$ problem is poly-time solvable (in data complexity).*

**`ADP` with different choices of $k$:** When $k = |Q(D)|$ or $k = 1$, the `ADP` problem is equivalent to the resilience problem, which implies that $\texttt{ADP}(Q, D, k)$ is NP-hard even for a constant $k$ for general CQs. In contrast, `ADP` can be shown to be poly-time solvable (in data complexity) for any fixed $k$ if the query $Q$ is a full CQ.

For full CQs, it is indeed the case that $\texttt{ADP}(Q, D, k)$ is polynomial-time solvable for constant $k$. Enumerate all $\binom{|Q(D)|}{k}$ ways of selecting the output tuples to be removed, which is polynomial in $|D| = n$ assuming data complexity. So, the problem reduces to finding a minimum set of input tuples whose removal results in a *fixed* set of $k$ output tuples being removed. Let us fix such a set of $k$ output tuples. Now partition the $n$ input tuples into $2^k$ subsets depending on which subset of these $k$ output tuples they remove – since the CQ is full, each input tuple will remove zero or more output tuples from the $k$ chosen output tuples. All input tuples in any subset of this partition behave identically with respect to the $k$ output tuples we chose to delete; hence, we can only keep any one of these input tuples. That leaves us with $2^k$ input tuples and the input size becomes constant for fixed $k$. So, by any brute force method (e.g., trivially enumerating all $2^{2^k}$ subsets of these $2^k$ input tuples), the problem can be solved in $O(2^{2^k})$ for the fixed set of $k$ output tuples. Overall, the running time becomes $O(|Q(D)|^k \cdot 2^{2^k})$ time, which is polynomial for fixed $k$.

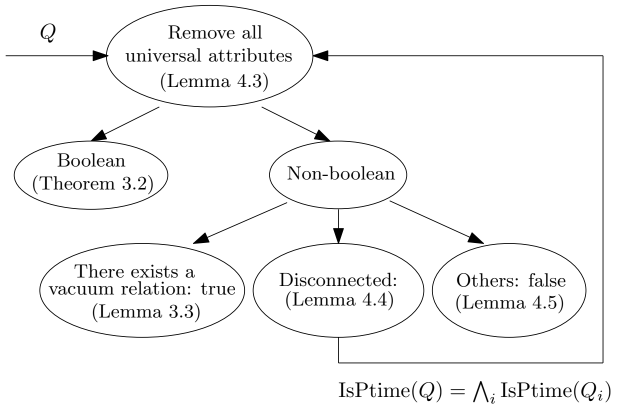
# 4 Poly-time Decidability

In this section, we give an algorithm that can decide poly-time solvability of the `ADP` problem on general CQs.

**Theorem 2.** *On a CQ $Q$,* IsPtime($Q$) *can decide poly-time solvability of the $\texttt{ADP}(Q, D, k)$ problem, which runs in polynomial time.*

**Algorithm 1:** IsPtime($Q$)

```
1  Remove all universal attributes from each relation in $Q$;
2  if $head(Q) = \emptyset$ then
3      if there is no triad structure in $Q$ then
4          return true
5  else
6      if there exists a relation $R_i$ with $attr(R_i) = \emptyset$ then
7          return true
8      else
9          if $Q$ is disconnected then
10             Let $Q_1, Q_2, \cdots, Q_s$ be its connected components;
11             return $\cap_{i=1}^{s}$ IsPtime($Q_i$);
12 return false
```



$Q$
Remove all universal attributes (Lemma 4.3)
Boolean (Theorem 3.2)
Non-boolean
There exists a vacuum relation: true (Lemma 3.3)
Disconnected: (Lemma 4.4)
Others: false (Lemma 4.5)
IsPtime($Q$) = $\bigwedge_i$ IsPtime($Q_i$)

Figure 3: Procedure IsPtime($Q$).

The procedure IsPtime($Q$) is illustrated in Figure 3. Note that when IsPtime($Q$) returns `true`, the `ADP`($Q, D, k$) problem is poly-time solvable, and NP-hard otherwise. The algorithmic description of IsPtime is given in Algorithm 1. IsPtime($Q$) runs in polynomial time in the query size.

The high-level idea is to alternately apply two simplifications steps on the input query, until a "base case" is arrived at. The first simplification step is that of removing all *universal* attributes in the input query. An attribute is *universal* if it is an output attribute appearing in all relations. After applying this step, if $Q$ becomes boolean or contains a vacuum relation (two of the base cases), it is decidable in polynomial time by Theorem 1 and Lemma 1.

Next, we check whether $Q$ is connected or not. For a disconnected query $Q$, we can *decompose* it into multiple *connected subqueries* as follows: apply breadth-first search or depth-first search algorithm on the graph $G_Q$, and find all connected components for $G_Q$. The set of relations corresponding to the set of vertices in one connected component of $G_Q$ form a connected subquery of $Q$. In this case, we perform the second simplification step of decomposing $Q$ into multiple connected subqueries, followed by calling IsPtime recursively on each connected subquery. More specifically, let $Q_1, Q_2, \cdots, Q_s$ be the connected subqueries of $Q$; then, IsPtime($Q$) will return $\bigwedge_{i=1}^{s}$ IsPtime($Q_i$). Otherwise, $Q$ ends up in "Others" (the third base case). In this case, $Q$ is connected, non-boolean, and does not contain either a vacuum relation or a universal attribute. For all queries in "Others", IsPtime returns `false`.

**Example 4.** *Consider an example CQ $Q(A, F, G, H) : -R_1(A, B)$, $R_2(F, G), R_3(B, C), R_4(C), R_5(G, H)$. Observe that $Q$ is non-boolean without any universal attribute and vacuum relations. The simplification step applied to $Q$ is to decompose it into two connected subqueries, $Q_1$ (with $R_1, R_3, R_4$) and $Q_2$ (with $R_2, R_5$). For $Q_2$, after removing the universal attribute $G$, it becomes disconnected. On applying the simplification step again to $Q_2$, it decomposes into two connected subqueries,*

*$Q_{21}$ (with $R_2$) and $Q_{22}$ (with $R_5$). After removing the universal attribute $F$ in $Q_{21}$, relation $R_2$ becomes vacuum and* IsPtime*($Q_{21}$) returns* `true`*. Similarly,* IsPtime*($Q_{22}$) returns* `true`*. However, $Q_1$ is non-boolean and contains no vacuum relation. Both simplifications fail on $Q_1$, so* IsPtime*($Q_1$) returns* `false`*. Therefore,* IsPtime*($Q$) returns* `false` *and* `ADP`*($Q, D, k$) is* NP-hard.

The essence of IsPtime is in the two simplifications steps: removing universal attributes and decomposing a disconnected query. Both these steps preserve the complexity of the problem as formally stated in Lemma 2 and Lemma 3. Intuitively, for any universal attribute, we can partition the query results by the value of the universal attribute, and interpret each class in the partition as the result of the same query over a distinct sub-instance. Moreover, the deletion of any input tuple $t$ can only affect a single sub-instance that shares the value of the universal attribute with $t$. The original `ADP` instance now degenerates to finding an optimal combination of solutions to the `ADP` problem defined over each of the sub-instances, after removing the universal attribute. Similarly, if the query is disconnected, the results of all connected subqueries will join by cross product. Then, the original `ADP` instance also degenerates to finding an optimal combination of solutions to the `ADP` problem defined for each connected subqueries. Finding the optimal combination is polynomial-time solvable since the size of the query as well as the query result is polynomial. Thus, the complexity of the original query can be deduced from that of the simplified queries.

Our proof of Theorem 2 also follows the logical diagram of IsPtime($Q$), which is divided into two parts. First, we show that these two simplification steps preserve the complexity of the problem, as described above. Then, we deal with the base cases. Note that the correctness for boolean queries and vacuum relations are implied by Theorem 1 and Lemma 1. Therefore, it suffices to show the NP-hardness of the `ADP` problem on $Q$, when $Q$ is non-boolean, connected, and contains no universal attribute or vacuum relation; we show this in Lemma 4. Putting everything together, the correctness for Theorem 2 then follows from induction over the size of the query.

## 4.1 Hardness Preservation in Simplifications

In the first part, we show that when the simplifications are applied to the input query, the complexity of the `ADP` problem is preserved.

**Lemma 2.** *Let $A$ be a universal attribute in $Q$. Then,* `ADP`*($Q, D, k$) is NP-hard if and only if* `ADP`*($Q_{-A}, D, k$) is NP-hard, where $Q_{-A}$ is the residual query after removing attribute $A$ from all relations in $Q$.*

**Lemma 3.** *Let $Q_1, Q_2, \cdots, Q_s$ be the connected subqueries of $Q$ for $s \geq 2$. The* `ADP`*($Q, D, k$) problem is NP-hard if and only if there exists some $Q_i$ for which the* `ADP`*($Q_i, D, k$) problem is NP-hard.*

The proofs of these lemmas are similar in spirit. Namely, we have two parts corresponding to the "if" and "only if" directions. To prove the "if" direction, we show that if `ADP` is NP-hard for $Q_{-A}$ (resp., there exists some $Q_i$ for which `ADP` is NP-hard), then the `ADP` problem on $Q$ is also NP-hard. To prove the "only-if" direction, we show that if `ADP` is poly-time solvable for $Q_{-A}$ (resp., `ADP` is poly-time solvable for each connected subquery $Q_i$), then `ADP` is also poly-time solvable for $Q$ as well. More specifically, given a poly-time algorithm for solving `ADP` on $Q_{-A}$ (resp., given poly-time algorithms for solving `ADP` on each $Q_i$), we design a poly-time algorithm for solving `ADP` problem on $Q$.

*Proof of Lemma 2.* **The "if" direction.** Given any instance $D'$ for $Q_{-A}$, we construct another instance $D$ for $Q$ as follows. Consider any relation $R'_i \in \texttt{rels}(Q_{-A})$. For each tuple $t' \in R'_i$, we create a new tuple $t \in R_i$ such that $\pi_A t = *$ (a fixed value for all tuples and all relations in attribute $A$), and $\pi_B t = \pi_B t'$ for every other attribute $B \in \texttt{attr}(R_i) - A$.

Hence there is a one-to-one correspondence between the output tuples in $Q(D)$ and $Q_{-A}(D')$, and also in the input $D$ and $D'$. Therefore, a solution to `ADP`$(Q, D, k)$ of size $c$ corresponds to a solution to `ADP`$(Q_{-A}, k, D')$ of size $c$, and vice versa. The proof follows.

**The "only-if" direction.** Assume there is a poly-time algorithm $\mathcal{A}$ for computing `ADP`$(Q_{-A}, D, k)$ for any instance $D$ and integer $k$. We design a poly-time algorithm $\mathcal{A}'$ for `ADP`$(Q, D, k)$ as follows:

Consider any input instance $D$ for $Q$ and integer $k$. We first partition $D$ into $D_1, D_2, \cdots, D_g$ corresponding to $a_1, a_2, \cdots, a_g$, which are all the possible values in the domain of attribute $A$. In $D_i$, each tuple $t$ has $\pi_A t = a_i$. Note that the query result $Q(D)$ is a disjoint union of the subquery results $Q(D_1), Q(D_2), \cdots, Q(D_i)$.

Now, we run a dynamic program to compute the optimal solution with cost $\textsc{Opt}$. Let $\textsc{Opt}[i][s]$ denote the minimum number of input tuples that have to be removed in order to remove at least $s$ output tuples from $Q(D)$, under the constraint that the input tuples can only be chosen from $D_1$ to $D_i$. Using this notation, we can now write the following dynamic program:

$$\textsc{Opt}[i][s] = \min_{m=0}^{s} \Big\{ \textsc{Opt}[i-1][s-m] + c_{i,m} \Big\}. \tag{1}$$

Here, $m$ denotes the number of output tuples being removed from the subproblem on $D_i$. And, $c_{i,m}$ is the cost of the solution for subproblem $\texttt{ADP}(Q, D_i, m)$, i.e., the minimum number of input tuples in $D_i$ whose removal would remove at least $m$ output tuples from $Q(D_i)$. Note that $c_{i,0} = 0$ for every $i$.

Note that each tuple in $D_i$ has the same value $a_i$ in attribute $A$. Hence, computing $\texttt{ADP}(Q, D_i, m)$ is equivalent to computing $\texttt{ADP}(Q_{-A}, D_i, m)$, which can be solved in poly-time by algorithm $\mathcal{A}$. Recall that there are $g$ distinct values in attribute $A$, thus $g \le |D|$. Moreover, $k$ is bounded by the size of query results, i.e. $k \le |Q(D)|$. The number of cells in $\textsc{Opt}$ is $g \cdot k = O(|D| \cdot |Q(D)|)$, which is polynomial in terms of $|D|$. Thus, algorithm $\mathcal{A}$ runs in polynomial time in data complexity. □

*Proof of Lemma 3.* **The "if" direction.** W.l.o.g., assume the $\texttt{ADP}$ problem on $Q_1$ is NP-hard. Given an instance $D'$ for $Q_1$, we construct another instance $D$ for $Q$ as follows. All relations in $Q_1$ have the same tuples as in $D'$. Set $L = |Q_1(D')| \cdot |D'|$. Recall that $|Q_1(D')|$ denotes the number of results in query $Q_1$ over instance $D'$. Each relation $R_j \in \texttt{rels}(Q_\ell)$ for $\ell \ge 2$ contains $L$ tuples, where each tuple is given a unique label that appears as the value of every attribute in that tuple. (Note that the size of $D$ is polynomial in the size of $D'$.) This ensures that for any connected subquery $Q_\ell$, there are exactly $L$ output tuples in $Q_\ell(D)$ corresponding to the $L$ unique labels given to the tuples in every relation. Then, the number of output tuples in $Q(D)$ is $|Q_1(D')| \cdot L^{s-1}$, since the join across the disconnected components results in a cross product.

We argue that $\texttt{ADP}(Q_1, D', k')$ has a solution of size $\le c$ if and only if $\texttt{ADP}(Q, D, k' \cdot L^{s-1})$ has a solution of size $\le c$.

In one direction, if we can remove $k'$ results from $Q_1(D')$ by removing at most $c$ tuples from $D'$, removing these tuples from $D$ removes $k' \cdot L^{s-1}$ results from $Q(D)$, which is also a solution for $\texttt{ADP}(Q, D, k' \cdot L^{s-1})$.

In the other direction, suppose we are given a solution for $\texttt{ADP}(Q, D, k' \cdot L^{s-1})$ of size at most $c$. Observe that $c \le |D'|$; otherwise, there is always a better solution for $\texttt{ADP}(Q, D, k)$ by removing all input tuples from relations in $Q_1$. Let $x_i$ be the number of input tuples removed from relations in $Q_i$, and $y_i$ be the number of output tuples removed from $Q_i(D)$. A key observation is that there exists a solution for $\texttt{ADP}(Q, D, k' \cdot L^{s-1})$ of size $\le c$ such that (i) $y_i = x_i$ for any $i \ge 2$; (ii) $x_i \ne 0$ for at most one $i \ge 2$; and (iii) $y_1 \ge k'$. We will prove these one by one.

For (i), we can always remove $x_i$ output tuples from $Q_i(D)$ by removing $x_i$ tuples from one specific relation in $Q_i$. Thus, the total number of results removed can be written as:

$$f(x_1, x_2, \cdots, x_s) = |Q_1(D')| \cdot L^{s-1} - (|Q_1(D')| - y_1) \cdot \prod_{i \ge 2}^{s} (L - x_i) \ge k.$$

For (ii), suppose $s \ge 3$ and $x_2, x_3 \ne 0$ without loss of generality. We can construct another solution for $\texttt{ADP}(Q, D, k' \cdot L^{s-1})$ with $x'_i = x_i$ for $i \notin \{2, 3\}$, $x'_2 = x_2 + x_3$, and $x'_3 = 0$, which is no worse. This is because:

$$f(x_1, x_2 + x_3, 0, x_4, \cdots, x_s) \ge f(x_1, x_2, \cdots, x_s).$$

After applying this argument repeatedly, we can obtain a solution for $\texttt{ADP}(Q, D, k' \cdot L^{s-1})$ that removes $x_1$ tuples from relations in $Q_1$ and $x_2$ tuples from relations in $Q_2$, where $x_1 + x_2 \le c$, with $\ge k$ results removed from $Q(D)$.

For (iii), suppose $y_1 < k'$. As $x_1 + x_2 \le c$, there comes

$$f(x_1, c - x_1, 0, \cdots, 0) \ge f(x_1, x_2, 0, \cdots, 0) \ge k$$

Expanding $f(x_1, c - x_1, 0, \cdots, 0)$ and $k$, we get:

$$|Q_1(D')| \cdot L^{s-1} - (|Q_1(D')| - y_1)(L - c + x_1) \cdot L^{s-2} \ge k' \cdot L^{s-1}$$

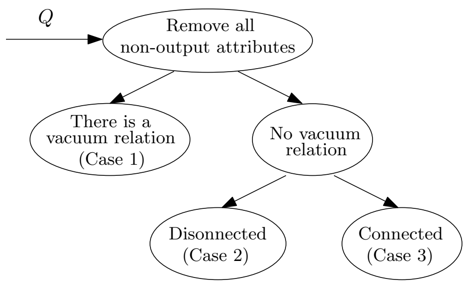

Q
Remove all non-output attributes
There is a vacuum relation (Case 1)
No vacuum relation
Disonnected (Case 2)
Connected (Case 3)

Figure 4: Proof plan of Lemma 4.

Rearranging this inequality, we will get

$$(|Q_1(D')| - k') \cdot L \geq (|Q_1(D')| - y_1)(L - c + x_1) \geq (|Q_1(D')| - y_1)(L - |D'|)$$

where the last inequality is implied by the fact that $c \leq |D'|$. We can further rewrite the inequality above as

$$(|Q_1(D')| - y_1) \cdot |D'| \geq (k' - y_1) \cdot L > L \text{ (since } y_1 < k').$$

This contradicts: $(|Q_1(D')| - y_1) \cdot |D'| \leq |Q_1(D')| \cdot |D'| = L$.

Thus, there exists a solution for $\texttt{ADP}(Q, D, k' \cdot L^{s-1})$ of size $\leq c$ such that $y_1 \geq k'$. Removing those $x_1$ tuples from relations in $Q_1$ is a solution for $\texttt{ADP}(Q_1, D', k')$ of size $\leq c$.

**The "only-if" direction.** Assume that for each $Q_i$, there is a poly-time algorithm $\mathcal{A}_i$ for computing $\texttt{ADP}(Q_i, D, k)$ for any instance $D$ and integer $k$. We next present another poly-time algorithm $\mathcal{A}$ for $\texttt{ADP}(Q, D, k)$. Consider an arbitrary input instance $D$ and integer $k$. Let $|Q_i(D)| = m_i$. Note that if removing $k_i$ output tuples from $Q_i(D)$, there are $m_i - k_i$ remaining output tuples in $Q_i(D)$, which together form $\prod_{i=1}^{s}(m_i - k_i)$ output results overall. In other words, $\prod_{i=1}^{s} m_i - \prod_{i=1}^{s}(m_i - k_i)$ output tuples are removed from $Q(D)$ in total. Therefore, the overall optimal solution is given by:

$$\texttt{ADP}(Q, D, k) = \min_{(k_1, k_2, \cdots, k_s) \in K} \sum_{i=1}^{s} \texttt{ADP}(Q_i, D, k_i) \tag{2}$$

where $K = \{(k_1, k_2, \cdots, k_s) : \prod_{i=1}^{s} m_i - \prod_{i=1}^{s}(m_i - k_i) \geq k, k_i \in \mathbb{Z}^+, \forall i \in \{1, 2, \cdots, s\}\}$. Note that the $\texttt{ADP}(Q_i, D, k_i)$ is solved in polynomial time by algorithm $\mathcal{A}_i$. Note that there are at most $k^s = O(|Q(D)|^s)$ different combinations of $k_1, k_2, \cdots, k_s$, which is still polynomial in terms of data complexity. Overall, the running time of $\mathcal{A}$, which simply enumerates all these options and chooses the best one, is polynomial. □

## 4.2 NP-Hardness for "Others"

In this part, we prove the hardness of the class of queries characterized by "others" bracket in Figure 3, as stated in Lemma 4.

**Lemma 4.** *For a CQ $Q$, if* IsPtime$(Q)$ *goes to "others" in Figure 3, i.e., if (1) $Q$ contains no universal attributes; (2) $Q$ is non-boolean; (3) $Q$ contains no vacuum relations; and (4) $Q$ is connected, then* $\texttt{ADP}(Q, D, k)$ *is NP-hard.*

We start by identifying three simple but **NP-hard** queries for the ADP problem that will be at the core of showing the above lemma. Then we present a general framework of proving the hardness for a given CQ by *mapping* it to another query on which the ADP problem is known (or has been proven) to be NP-hard. Finally, we classify all queries in Lemma 4 into three groups using the flowchart in Figure 4, and give a mapping from queries ending up in each leaf of the flowchart to a core query identified at the beginning.

### 4.2.1 Core Queries

The three queries we focus on are the following:

$Q_{\text{cover}}(A,B) :- R_1(A), R_2(A,B), R_3(B).$

$Q_{\text{swing}}(A) :- R_2(A,B), R_3(B).$

$Q_{\text{seesaw}}(A) :- R_1(A), R_2(A,B), R_3(B).$

Careful inspection reveals that these queries have a common property: w.l.o.g., we can assume that an optimal solution of $\texttt{ADP}(Q,D,k)$ won't remove any tuples from relation $R_2(A,B)$. The effect of the removal of any tuple $(a,b) \in R_2$ can also be achieved by removing tuple $(a) \in R_1$ or $(b) \in R_3$. (The formal proof is in Appendix A.) Therefore, an optimal solution for ADP on any one of these three queries could be restricted to removing tuples only from $R_1(A)$ and $R_3(B)$. In this way, the ADP problem on these queries can be interpreted as optimization problems on bipartite graphs, which turn out to be *NP-hard* (Lemma 5).

**Lemma 5.** *Given an undirected bipartite graph $G(A \cup B, E)$ where $E$ is the set of edges between two sets of vertices $A$ and $B$, and an integer $k$, each of the following problems is NP-hard:*

*(1) Remove the minimum number of vertices in $A \cup B$ such that at least $k$ edges in $E$ are removed.*[1]

*(2) Remove the minimum number of vertices in $B$ such that at least $k$ vertices in $A$ are removed;*

*(3) Remove the minimum number of vertices in $A \cup B$ such that at least $k$ vertices in $A$ are removed;*

Problem (1) is exactly *partial vertex cover for bipartite graphs*, which is known to be NP-hard [**4**]. The NP-hardness proofs for (2) and (3) are deferred to Appendix B.

### 4.2.2 Hardness Preserving Mapping

The high-level idea of relating an arbitrary query $Q$ characterized by Lemma 4 to the core queries is to divide the attributes in $\texttt{attr}(Q)$ into two groups, one mapped to $A$ and the other mapped to $B$. In this way, each relation in $Q$ plays the role of $R_1(A)$, $R_2(A,B)$ or $R_3(B)$ in the core queries. The notion of "query mapping" is formally defined below:

**Definition 2** (Query Mapping)**.** *Suppose we are given a function $f : \texttt{attr}(Q_1) \to \texttt{attr}(Q_2) \cup \{*\}$. Let*

$$g(R_i) = \{Y \in \texttt{attr}(Q_2) : \exists X \in \texttt{attr}(R_i) \text{ s.t. } f(X) = Y\}.$$

*$f$ is said to be a query mapping if the following properties hold: (i) for every relation $R_i \in \texttt{rels}(Q_1)$, there is a (unique) relation $R_j \in \texttt{rels}(Q_2)$ such that $g(R_i) = \texttt{attr}(R_j)$. (ii) for every relation $R_j \in \texttt{rels}(Q_2)$, there exists at least one relation $R_i \in \texttt{rels}(Q_1)$ such that $g(R_i) = \texttt{attr}(R_j)$.*

In the definition above, if $g(R_i) = \texttt{attr}(R_j)$ for relations $R_i \in \texttt{rels}(Q_1)$ and $R_j \in \texttt{rels}(Q_2)$, then $R_i \in \texttt{rels}(Q_1)$ is said to be *mapped* to relation $R_j \in \texttt{rels}(Q_2)$. The next lemma shows that query mappings preserve hardness of the ADP problem.

**Lemma 6.** *If there is a mapping from a CQ $Q_1$ to another CQ $Q_2$, and $\texttt{ADP}(Q_2,D,k)$ is NP-hard, then $\texttt{ADP}(Q_1,D,k)$ is also NP-hard.*

*Proof.* Assume $Q_1$ is mapped to $Q_2$ under the mapping function $f$. Given any instance $D_2$ for $Q_2$, we construct an instance $D_1$ for $Q_1$ as follows. Consider an arbitrary relation $R_i \in \texttt{rels}(Q_1)$ that is mapped to relation $R_j \in \texttt{rels}(Q_2)$ under $f$. If there is a tuple $t' \in R_j$, we create a tuple $t \in R_i$ such that for any $X \in \texttt{attr}(R_i)$, $\pi_X t = \pi_{f(X)} t'$ if $f(X) \in \texttt{attr}(R_j)$, and $\pi_X t = *$ otherwise. Overloading notation, we will say that $t$ is also mapped to $t'$. Note that there is a one-to-one correspondence between the output tuples in $Q_1(D_1)$ and $Q_2(D_2)$.

We next show that the problem $\texttt{ADP}(Q_1,D_1,k)$ has a solution of size $\leq c$ if and only if $\texttt{ADP}(Q_2,D_2,k)$ has a solution of size $\leq c$.

**The "only-if" direction.** Suppose we are given a solution $\mathcal{S}_1$ for $\texttt{ADP}(Q_1,D_1,k)$ has of size $\leq c$. We next construct a solution $\mathcal{S}_2$ for $\texttt{ADP}(Q_2,D_2,k)$ as follows. For any relation $R_i \in \texttt{rels}(Q_1)$, if tuple $t \in R_i$ is removed by $\mathcal{S}_1$, then tuple $t' \in R_j$ is removed by $\mathcal{S}_2$, where $R_i \in \texttt{rels}(Q_1)$ is mapped to $R_j \in \texttt{rels}(Q_2)$ and

[1]A **remove** procedure on a graph is defined as: (1) when a vertex is removed, all the incident edges are also removed; (2) when all the incident edges on a vertex are removed, this vertex is also removed.

$t$ is mapped to $t'$. Since multiple tuples from different relations in $D_1$ could be mapped to $t'$, $|\mathcal{S}_2| \leq |\mathcal{S}_1| \leq c$. As a result, if an output tuple from $Q_1(D_1)$ is removed, its corresponding tuple from $Q_2(D_2)$ will also be removed. Thus, $\mathcal{S}_2$ removes at least $k$ results from $Q_2(D_2)$, with size $\leq c$.

**The "if" direction.** Suppose we are given a solution $\mathcal{S}_2$ for $\texttt{ADP}(Q_2, D_2, k)$ of size $\leq c$. We next construct a solution $\mathcal{S}_1$ for $\texttt{ADP}(Q_1, D_1, k)$ as follows. Consider any relation $R_j \in \texttt{rels}(Q_2)$ with some tuples removed by $\mathcal{S}_2$. Let $R_i \in \texttt{rels}(Q_1)$ be any one relation mapped to $R_j$ under $f$. If $t' \in R_j$ is removed, remove the tuple $t$in $\mathcal{S}_1$ that is mapped to $t'$. Clearly, $|\mathcal{S}_1| = |\mathcal{S}_2| \leq c$. As a result, if an output tuple from $Q_2(D_2)$ is removed, its corresponding tuple from $Q_1(D_1)$ will also be removed. Thus, $\mathcal{S}_1$ removes at least $k$ results from $Q_1(D_1)$, with size $\leq c$. $\square$

### 4.2.3 Mapping to the core

To prove the NP-hardness of the $\texttt{ADP}$ problem on a query $Q$, it suffices to show a mapping to any core query, implied by Lemma 6. The high-level idea is that for any query characterized by Lemma 4, we find a partition of attributes in $Q$ as $(\mathbb{I}, \mathbb{J}, \texttt{attr}(Q) - \mathbb{I} - \mathbb{J})$ where $\mathbb{I} \cap \mathbb{J} = \emptyset$ and define the mapping function $f : X \to \{A, B, *\}$ as follows:

$$f(X) = \begin{cases} A & \text{if } X \in \mathbb{I} \\ B & \text{if } X \in \mathbb{J} \\ * & \text{otherwise} \end{cases}$$

Then it remains to show that $f$ is a mapping from $Q$ to one of the three core queries. As mentioned, we distinguish $Q$ into three cases in Figure 4, and identify the mapping for each case separately.

Note that any query in Lemma 4 is connected and does not have any universal attribute or vacuum relation. For simplicity, *head join* is defined as the residual query after removing all non-output attributes from all relations in $Q$, denoted as $Q_{\text{head}}$. In a CQ $Q$, a *path* between a pair of attributes $A, B \in \texttt{attr}(Q)$, is a sequence of relations starting with some $R_i \in \texttt{rels}(A)$ and $R_j \in \texttt{rels}(B)$ such that each consecutive pair of relations share a common attribute.

**Case 1: Head join has at least one vacuum relation.** In this case, observe that there must exist some relation $R_i \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_i) \subseteq \texttt{attr}(Q) - \texttt{head}(Q)$. Let $\mathbb{I} = \texttt{head}(Q)$ and $\mathbb{J} = \texttt{attr}(Q) - \texttt{head}(Q)$. We next show that $f$ is a valid mapping from $Q$ to $Q_{\text{swing}}$ if there exists some relation $R_j \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_j) \subseteq \texttt{head}(Q)$, and to $Q_{\text{seesaw}}$ otherwise.

Note that every relation $R_i \in \texttt{rels}(Q)$ is mapped to $R_1(A)$, $R_2(A, B)$, or $R_3(B)$. Crucially, there is at least one relation that is mapped to $R_3(B)$, e.g., $R_i$. Moreover, there is at least one relation that is mapped to $R_2(A, B)$; otherwise attributes in $\mathbb{I}$ and $\mathbb{J}$ are not connected, contradicting the fact that $Q$ is connected. (Note that $Q$ is connected irrespective of whether the head join is connected or not.) If there exists some relation $R_j \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_j) \subseteq \texttt{head}(Q)$, then $R_j$ will be mapped to $R_1(A)$; and $f$ is a valid mapping from $Q$ to $Q_{\text{seesaw}}$. Otherwise, $f$ is a valid mapping from $Q$ to $Q_{\text{swing}}$.

**Case 2: Head join is disconnected (and no vacuum relation).** In this case, we can always identify a pair of attributes $X, Z \in \texttt{head}(Q)$ such that there is no path between $X, Z$ in $Q_{\text{head}}$. As $Q$ is connected, every path between $X, Z$ in $Q$ uses at least one attribute in $\texttt{attr}(Q) - \texttt{head}(Q)$. In other words, removing $\texttt{attr}(Q) - \texttt{head}(Q)$ decomposes $Q$ into multiple connected subqueries, where $X, Z$ are in different ones. Let $\mathbb{I}$ be the set of attributes appearing in the connected subquery containing $X$. Note that $\texttt{head}(Q) - \mathbb{I} \neq \emptyset$ since $X, Z$ are in different connected subqueries.

Observe that there must exist a relation $R_\ell \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_\ell) \cap \mathbb{I} \neq \emptyset$ and $\texttt{attr}(R_\ell) \cap (\texttt{attr}(Q) - \texttt{head}(Q)) \neq \emptyset$; otherwise, there is no path between $X$ and any non-output attribute, contradicting the fact that $Q$ is connected. Applying a similar argument to the connected subquery that doesn't contain $X$, there must exist a relation $R_h \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_h) \cap (\texttt{head}(Q) - \mathbb{I}) \neq \emptyset$ and $\texttt{attr}(R_h) \cap (\texttt{attr}(Q) - \texttt{head}(Q)) \neq \emptyset$. Depending on whether there exists some relation $R_i \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_i) \subseteq \mathbb{I}$ and some relation $R_j \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_j) \subseteq \texttt{head}(Q) - \mathbb{I}$, we have two different cases.

**Case 2.1:** Both relations $R_i$ and $R_j$ as described above exist. Set $\mathbb{J} = \texttt{attr}(Q) - \mathbb{I}$. On one hand, each relation in $Q$ is mapped to any one of $R_1(A)$, $R_2(A, B)$ or $R_3(B)$. On the other hand, relations $R_i, R_\ell, R_j$ are mapped to $R_1, R_2, R_3$ respectively. Thus, $f$ is a valid mapping from $Q$ to $Q_{\text{path}}$.

**Case 2.2:** At least one of $R_i, R_j$ doesn't exist, say $R_j$. Set $\mathbb{J} = \texttt{attr}(Q) - \texttt{head}(Q)$. In this mapping, no relation has all of its attributes mapped to $*$; otherwise, $R_j$ exists, which is a contradiction. So, each relation in $Q$ is mapped to any one of $R_1(A)$, $R_2(A,B)$ or $R_3(B)$. On the other hand, relations $R_\ell, R_h$ are mapped to $R_2, R_3$ respectively. If $R_i$ exists, it will be mapped to $R_1(A)$ and $f$ is a valid mapping from $Q$ to $Q_{\text{seesaw}}$. Otherwise, $f$ is a valid mapping from $Q$ to $Q_{\text{swing}}$.

**Case 3: Head join is connected (and no vacuum relation).** In this case, the head join is connected but has no vacuum relation. We further distinguish $Q$ into two cases: (3.1) there exists a pair of relations $R_i, R_j \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_i) \cap \texttt{attr}(R_j) \cap \texttt{head}(Q) = \emptyset$; (3.2) for each pair of relations $R_i, R_j \in \texttt{rels}(Q)$, we have $\texttt{attr}(R_i) \cap \texttt{attr}(R_j) \cap \texttt{head}(Q) \neq \emptyset$.

**Case 3.1.** Set $\mathbb{I} = \texttt{attr}(R_i) \cap \texttt{head}(Q)$ and $\mathbb{J} = \texttt{head}(Q) - \texttt{attr}(R_i)$. In this mapping, no relation has its all attributes mapped to $*$; otherwise, there is a vacuum relation in the head join, which is a contradiction. So, each relation in $Q$ is mapped to any one of $R_1(A)$, $R_2(A,B)$ or $R_3(B)$. Moreover, $R_i, R_j$ are mapped to $R_1(A), R_3(B)$ respectively. Note that there must also exist some relation mapped to $R_2(A,B)$; otherwise, $R_i$ is a single connected subquery of the head join, contradicting the fact that the head join is connected. Thus, $f$ is a valid mapping from $Q$ to $Q_{\text{path}}$.

**Case 3.2.** In this case, we first observe that $|\texttt{attr}(R_i) \cap \texttt{head}(Q)| \geq 2$ for any relation $R_i \in \texttt{rels}(Q)$. Suppose not, say $\texttt{attr}(R_i) \cap \texttt{head}(Q) = \{C\}$. Since $\texttt{attr}(R_i) \cap \texttt{attr}(R_j) \cap \texttt{head}(Q) \neq \emptyset$ for any $R_j \in \texttt{rels}(Q)$, then $C$ is a universal attribute of $Q$, which is a contradiction. For simplicity, assume no pair of relations in the head join have exactly the same attributes; otherwise, we just keep one of them in the mapping construction.

We label all relations in an increasing order of the number of output attributes, as $R_1, R_2, \cdots, R_p$, breaking ties arbitrarily. For simplicity, denote $\texttt{attr}(R_i) \cap \texttt{attr}(R_j) \cap \texttt{head}(Q)$ as $\mathbb{A}_{ij}$ with ordering $(i,j)$ if $i < j$, and $\mathbb{A}_{ji}$ with ordering $(j,i)$ otherwise. Let $R_i, R_j$ be the pair of relations whose intersection contains smallest number of output attributes. Without loss of generality, assume $i < j$. If there are multiple pairs with the same number of attributes in their intersection, we just break ties by their lexicographical order. We further distinguish the mappings into two cases as follows.

Case 3.2.1: $i > 1$. We observe that $\mathbb{A}_{1i} - \mathbb{A}_{1j} \neq \emptyset$ and $\mathbb{A}_{1j} - \mathbb{A}_{1i} \neq \emptyset$. Suppose not, say $\mathbb{A}_{1i} - \mathbb{A}_{1j} = \emptyset$. This implies $\mathbb{A}_{1i} \subseteq \mathbb{A}_{1j} \subseteq \mathbb{A}_{ij}$, contradicting the fact that $\mathbb{A}_{ij}$ has smaller number of attributes than $\mathbb{A}_{1i}$. (Note that $(1,i)$ is lexicographically earlier than $(i,j)$ in the case of a tie.) Similarly, we can also show that $\mathbb{A}_{1j} - \mathbb{A}_{1i} \neq \emptyset$. Moreover, there exists no relation $R_\ell \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_\ell) \cap \texttt{head}(Q) \subseteq \mathbb{A}_{ij}$. This is because of the fact that no pair of relations have exactly the same attributes, which in combination with $\texttt{attr}(R_\ell) \cap \texttt{head}(Q) \subseteq \mathbb{A}_{ij}$ would imply that $\texttt{attr}(R_\ell) \cap \texttt{head}(Q) \subsetneq R_i \cap \texttt{head}(Q)$. This would in turn imply $\ell < i$, and consequently, that $\mathbb{A}_{\ell i}$ has smaller number of attributes than $\mathbb{A}_{ij}$ (or is lexicographically earlier in the case of a tie), which is a contradiction.

Set $\mathbb{I} = (\texttt{attr}(R_i) \cap \texttt{head}(Q)) - \texttt{attr}(R_j)$ and $\mathbb{J} = \texttt{head}(Q) - \texttt{attr}(R_i)$. In this mapping, no relation gets all attributes mapped to $*$, since there is no relation $R_\ell$ such that $\texttt{attr}(R_\ell) \cap \texttt{head}(Q) \subseteq \mathbb{A}_{ij}$ as discussed above. So, each relation in $Q$ is mapped to any one of $R_1(A)$, $R_2(A,B)$ or $R_3(B)$. Moreover, relations $R_i, R_1, R_j$ are mapped to $R_1, R_2, R_3$ respectively. Thus, $f$ is a valid mapping from $Q$ to $Q_{\text{path}}$.

Case 3.2.2: $i = 1$. For any attribute $C \in \mathbb{A}_{1j}$, there must exist a relation $R_\ell$ such that $C \notin \texttt{attr}(R_\ell)$; otherwise, $C$ is an universal attribute, which is a contradiction. W.l.o.g., assume $\ell < j$. We claim that $\mathbb{A}_{\ell j} - \texttt{attr}(R_1) \neq \emptyset$; otherwise, $\mathbb{A}_{\ell j} \subseteq \mathbb{A}_{1j}$. Since $C \in \mathbb{A}_{1j} - \mathbb{A}_{\ell j}$, $|\mathbb{A}_{\ell j}| < |\mathbb{A}_{1j}|$, contradicting the fact that $R_1, R_j$ share the smallest number of output attributes among all pair of relations. Moreover, there exists no relation $R_h \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_h) \cap \texttt{head}(Q) \subseteq \mathbb{A}_{1\ell}$. Otherwise, either $\texttt{attr}(R_h) \cap \texttt{head}(Q) \subsetneq \texttt{attr}(R_1) \cap \texttt{head}(Q)$ which contradicts the fact that $h > 1$, or $\texttt{attr}(R_h) \cap \texttt{head}(Q) = \texttt{attr}(R_1) \cap \texttt{head}(Q)$ which contradicts the fact that no pair of relations have exactly the same output attributes.

Set $\mathbb{I} = (\texttt{attr}(R_1) \cap \texttt{head}(Q)) - \texttt{attr}(R_\ell)$ and $\mathbb{J} = \texttt{head}(Q) - \texttt{attr}(R_1)$. In this mapping, no relation gets all attributes mapped to $*$, since there exists no relation $R_h$ such that $\texttt{attr}(R_h) \cap \texttt{head}(Q) \subseteq \mathbb{A}_{1\ell}$ as discussed above. So, each relation in $Q$ is mapped to any one of $R_1(A)$, $R_2(A,B)$ or $R_3(B)$. Moreover, $R_1, R_j, R_\ell$ are mapped to $R_1, R_2, R_3$ respectively. Thus, $f$ is a valid mapping from $Q$ to $Q_{\text{path}}$.

We show examples for each case in Figure 4 separately.

**Example 5.** *Consider an example query $Q_1(A,C,F) : -R_1(A,C), R_2(B), R_3(B,C), R_4(C,E,F)$, with a vacuum relation $R_2$ in head join $Q_1'(A,C,F) : -R_1(A,C), R_2(), R_3(C), R_4(C,F)$. In this example, we map attributes $A, C, F$ to $A$ and $B, C$ to $B$, yielding a new query $Q_1''(A) : -R_1(A), R_2(B), R_3(B), R_4(A,B)$, i.e., the $Q_{\text{seesaw}}$ query. If $R_1(A,C)$ does not appears in $Q_1$, the same mapping yields another query $Q_1'''(A) : -R_2(B), R_3(B), R_4(A,B)$, i.e., the $Q_{\text{swing}}$ query.*

**Example 6.** *Consider an example query $Q_2(A,B) : -R_1(A), R_2(A,C), R_3(C,B), R_4(B)$, where the head join $Q_2'(A,B) : -R_1(A), R_2(A), R_3(B), R_4(B)$ is disconnected. For one connected subquery containing $A$, we can identify relation $R_2$ such that $A \in \texttt{attr}(R_2)$ and $\texttt{attr}(R_2) \cap (\texttt{attr}(Q) - \texttt{head}(Q)) \neq \emptyset$. Similarity, for the other connected subquery containing $B$, we can identify relation $R_3$ such that $B \in \texttt{attr}(R_3)$ and $\texttt{attr}(R_3) \cap (\texttt{attr}(Q) - \texttt{head}(Q)) \neq \emptyset$. In this case, we map attributes $B, C$ to $B$, yielding a new query $Q_2'(A,B) : -R_1(A), R_2(A,B), R_3(B)$, i.e., the $Q_{\text{path}}$ query. If $R_4(B)$ does not appear in $Q_2$, we map $B$ to $*$, yielding a new query $Q_2''(A) : -R_1(A), R_2(A,C), R_3(C)$, i.e., the $Q_{\text{seesaw}}$ query. If both $R_1(A), R_4(B)$ does not appear in $Q_2$, we map $B$ to $*$, yielding a new query $Q_2'''(A) : -R_2(A,C), R_3(C)$, i.e., the $Q_{\text{swing}}$ query.*

**Example 7.** *We show two examples for (3.1) and (3.2) separately. In (3.1), there is a pair of relations $R_i, R_j \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_i) \cap \texttt{attr}(R_j) = \emptyset$. Consider a full CQ $Q_3(A,B,C,E) : -R_1(A,C), R_2(C,E), R_3(E,B)$. There is a pair of relations $R_1, R_3$ such that $\texttt{attr}(R_1) \cap \texttt{attr}(R_3) \neq \emptyset$. We map attributes $A, C$ to attribute $A$ and $B, E$ to attribute $B$, yielding a new query $Q_3'(A,B) : -R_1(A), R_2(A,B), R_3(B)$, i.e., the $Q_{\text{path}}$ query. In (3.2), for every pair of relations $R_i, R_j \in \texttt{rels}(Q)$, $\texttt{attr}(R_i) \cap \texttt{attr}(R_j) \cap \texttt{head}(Q) \neq \emptyset$. Consider an example full CQ $Q_4(A,B,C,E,F) : -R_1(A,B,C,E,F), R_2(B,C,E), R_3(A,C)$. We map attributes $C, E, F$ to $*$ and obtain a new query $Q_4'(A,B) : -R_1(A,B), R_2(B), R_3(A)$, i.e., the $Q_{\text{path}}$ query.*

## 5 Structural Characterization

In the last section, we provided a simple poly-time algorithm IsPtime to decide the poly-time solvability of the ADP problem for CQs without self-join. However, this algorithm does not provide structural insight into what makes the ADP problem NP-hard or poly-time solvable for individual queries. Namely, it does not provide a structural characterization for solvability of the ADP problem, such as the one shown for the special case of the resilience problem in [**11**]. To rectify this shortcoming and complement the procedural dichotomy established in the last section, we provide, in this section, a *structural dichotomy* of the ADP problem for CQs. Interestingly, it turns out that the procedural and structural dichotomies do not have a one-one mapping; namely, distinct cases of the IsPtime procedure map to same case in the structural characterization, and vice-versa. Our main theorem in this section is the following:

**Theorem 3.** *For a CQ $Q$, ADP$(Q,k,D)$ is NP-hard if and only if one of the following happens:*

- *$Q$ contains a "triad-like" structure,*
- *$Q$ contains a "strand" structure, or*
- *the head join of non-dominated relations is non-hierarchical.*

In the rest of this section, we explain the the three "hard structures" in Theorem 3 and give some intuition for why they make the ADP problem NP-hard. The proof of Theorem 3 is given in Appendix D.

### 5.1 Boolean CQ Revisited

As mentioned earlier, a complete characterization of boolean CQs for the ADP problem is known from previous work:

**Theorem 4** ([**11**])**.** *On a boolean CQ $Q$ without self-joins, the problem ADP$(Q,D,1)$ is poly-time solvable if there is no triad structure, and NP-hard otherwise.*

To explain this result, we introduce some new terminology. In a CQ $Q$, a relation $R_j \in \texttt{rels}(Q)$ is *exogenous* if there exists another relation $R_i \neq R_j \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_i) \subsetneq \texttt{attr}(R_j)$, and *endogenous* otherwise. If there is more than one relation defined on the same set of attributes, we just consider any one of them as *endogenous* and the remaining ones as *exogenous*. For example, in the boolean CQ $Q : -R_1(A), R_2(A,B), R_3(B,C), R_4(B,C), R_5(B,C)$, there are two endogenous relations: $R_1$ and any one of $R_3$, $R_4$, $R_5$. Next, we define a *path* between a pair of relations $R_i, R_j \in \texttt{rels}(Q)$ as a path between any pair of attributes $A, B$ for $A \in \texttt{attr}(R_i)$ and $B \in \texttt{attr}(R_j)$. This brings us to the definition of the *triad* structure:

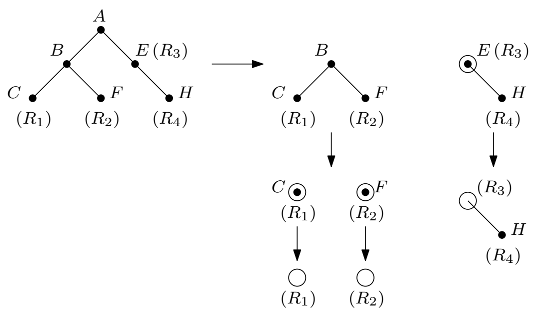


Figure 5: An example of hierarchical join $Q(A,B,C,E,F,H) : -R_1(A,B,C), R_2(A,B,F),\ R_3(A,E),\ R_4(A,E,H)$, and an illustration of applying procedure IsPtime on it.

**Definition 3** (triad)**.** *A triad is a triple of endogenous relations $R_1, R_2, R_3$ such that for each pair of relations, say $R_1, R_2$, there is a path from $R_1$ to $R_2$ only using any attributes in $\texttt{attr}(Q) - \texttt{attr}(R_3)$.*

Two examples of boolean CQs containing a triad structure are $Q_\triangle : -R_1(A,B), R_2(B,C), R_3(C,A)$ and $Q_T : -R_1(A,B,C),\ R_2(A),\ R_3(B),\ R_4(C)$, on which the `ADP` problem is NP-hard.

## 5.2 Hard Structures for General CQs

A natural question for general CQs is how the existence of output attributes changes the hardness of `ADP` problem. We will explore this question starting with three hard structures.

### 5.2.1 Triad-like

We observe that adding output attributes to a hard boolean CQ maintains the NP-hardness of the `ADP` problem. For example, the CQ $Q(E,F,G) : -R_1(A,B,E), R_2(B,C,F), R_3(C,A,G)$ is NP-hard (since IsPtime returns `false`), which contains the $Q_\triangle$. We extend the notion of triad to capture this class of hard queries:

**Definition 4** (triad-like)**.** *A triad-like structure is a triple of endogenous relations $R_1, R_2, R_3$ such that for each pair of relations, say $R_1, R_2$, there is a path from $R_1$ to $R_2$ only using attributes in $\texttt{attr}(Q) - (\texttt{head}(Q) \cup \texttt{attr}(R_3))$.*

This takes care of our first case: if there is a triad-like structure (in the non-output attributes), the CQ is NP-hard.

### 5.2.2 Non-hierarchical Join

The situation becomes more complicated when we add output attributes to a poly-time solvable boolean CQ. For example, on a boolean CQ $Q : -R_1(C,E), R_2(E,F), R_3(F,H)$, adding a universal attribute $A$ leads to a poly-time solvable query $Q(A) : -R_1(A,C,E), R_2(A,E,F), R_3(A,F,H)$, but adding attributes $A, B$ selectively to some of the relations (e.g., $Q(A,B) : -R_1(A,C,E), R_2(A,B,E,F), R_3(B,F,H)$) can result in an NP-hard query. So, our goal is to understand how the addition of output attributes changes the complexity of the `ADP` problem. For simplicity, the *head join* for a CQ $Q$ denotes the residual query after removing all non-output attributes from all relations in $Q$. We start with the class of full CQs, i.e., without non-output attributes. A nice connection between *hierarchical join* and our previously defined procedure IsPtime can be observed.

**Definition 5** (Hierarchical Join)**.** *A full CQ $Q$ is hierarchical if for each pair of attributes $A, B \in \texttt{attr}(Q)$, $\texttt{rels}(A) \subseteq \texttt{rels}(B)$, $\texttt{rels}(B) \subseteq \texttt{rels}(A)$, or $\texttt{rels}(A) \cap \texttt{rels}(B) = \emptyset$, and non-hierarchical otherwise.*

Note that a hierarchical CQ can be organized into a tree structure, where each relation is a root-to-node path. An example is given in Figure 5. Moreover, each relation ends up vacuum by alternately applying the two simplification steps in IsPtime on this tree. In this way, if $Q$ is hierarchical, IsPtime($Q$) always returns `true`. However, the converse is not necessarily true. For example, $Q(A,B,E) :$

$-R_1(A,E), R_2(A,B,E), R_3(B,E), R_4(E)$ is non-hierarchical but IsPtime($Q$) returns `true` (after removing the universal attribute $E$, relation $R_4$ becomes vacuum). We focus on non-hierarchical CQs in the rest of this discussion.

The previous result on boolean CQs only considers endogenous relations. Unfortunately, this is insufficient for a full CQ in general; for example, removing the exogenous relation $R_2$ would make $Q_{\text{path}}(A,B) : -R_1(A), R_2(A,B), R_3(B)$ poly-time solvable. So, we need a more fine-grained notion than exogenous/endogenous relations in characterizing the complexity of non-boolean CQs.

**Definition 6** (Dominated Relation in Full CQs)**.** *In a full CQ $Q$, relation $R_j$ is dominated by relation $R_i$ if (1) $\texttt{attr}(R_i) \subseteq \texttt{attr}(R_j)$; and (2) for any relation $R_k$ with $\texttt{attr}(R_i) - \texttt{attr}(R_k) \neq \emptyset$, $\texttt{attr}(R_j) \cap \texttt{attr}(R_k) \subseteq \texttt{attr}(R_i)$.*

We say that a relation is *dominated* if it is dominated by any other relation, and *non-dominated* otherwise. Note that a dominated relation must be exogenous, but all exogenous relations may not be dominated. A structural dichotomy for full CQs based on dominated relations is given by:

**Lemma 7.** *For a full CQ $Q$, the ADP$(Q,D,k)$ problem is NP-hard if and only if the non-dominated relations are non-hierarchical.*

Note that full CQs do not have any non-output attributes. But, fortunately, the above hardness continues to hold even on adding output attributes. To make this formal, we need to extend the notion of dominated relations to general CQs.

**Definition 7** (Dominated Relation in CQs)**.** *In a CQ $Q$, relation $R_j$ is dominated by relation $R_i$ if (1) $\texttt{attr}(R_i) \subseteq \texttt{attr}(R_j)$; (2) for any relation $R_k$ with $\texttt{attr}(R_i) - \texttt{attr}(R_k) \neq \emptyset$, $\texttt{attr}(R_j) \cap \texttt{attr}(R_k) \subseteq \texttt{attr}(R_i) \cap \texttt{head}(Q)$; (3) $\texttt{attr}(R_i) \subseteq \texttt{head}(Q)$ or $\texttt{head}(Q) \subseteq \texttt{attr}(R_i)$.*

If there is more than one relation defined on the same attributes, i.e., $\texttt{attr}(R_i) = \texttt{attr}(R_j)$, then we just consider any one of them as *non-dominated* and the remaining ones as *dominated*. We can now use this extended definition to claim our second hard case: if the head join of non-dominated relations is non-hierarchical, then the CQ is NP-hard. Note that these definitions of "domination" are different from [**11**], as we need a more fine-grained characterization of exogenous relations for `ADP`. Moreover, Lemma 1 can be easily interpreted as follows: If there is a vacuum relation $R_i$ in a CQ $Q$, then every remaining relation must be dominated by $R_i$, therefore `ADP`$(Q,D,k)$ is poly-time solvable by Theorem 3.

### 5.2.3 Strand

The remaining case is one where on the output attributes, the non-dominated relations are hierarchical *and* on the non-output attributes, there is no triad-like structure. These two conditions guarantee poly-time solvability for full and boolean CQs respectively. But, interestingly, when appearing together in a general CQ, they no longer guarantee poly-time solvability. For example, the CQ $Q(A,B,C) :- R_1(A,B,E), R_2(A,C,E)$ is NP-hard while both $Q(A,B,C) :- R_1(A,B), R_2(A,C)$ and $Q() : -R_1(E), R_2(E)$ are poly-time solvable. To characterize this class of queries, we introduce our third hard structure that we call a *strand*:

**Definition 8** (strand)**.** *A strand is a pair of non-dominated relations $R_i, R_j \in \texttt{rels}(Q)$ such that (1) $\texttt{head}(Q) \cap \texttt{attr}(R_i) \neq \texttt{head}(Q) \cap \texttt{attr}(R_j)$; (2) $(\texttt{attr}(R_i) \cap \texttt{attr}(R_j)) - \texttt{head}(Q) \neq \emptyset$.*

The reason why the strand structure makes the `ADP` problem hard can be explained by the procedure IsPtime. Consider any CQ with such a strand structure with $R_i, R_j$. After applying two simplification steps, $R_i, R_j$ will be in the same connected subquery $Q_0$, since attributes in $(\texttt{attr}(R_i) \cap \texttt{attr}(R_j)) - \texttt{head}(Q)$ are not universal and therefore couldn't have been removed by IsPtime. Moreover, $Q_0$ is non-boolean, since $\texttt{attr}(R_i) \cap \texttt{head}(Q) \neq \texttt{attr}(R_j) \cap \texttt{head}(Q)$ and therefore, there is at least one non-universal output attribute. Next, we prove that there is no vacuum relation in $Q_0$. Suppose $R_\ell$ becomes vacuum in $Q_0$. Observe that $\texttt{attr}(R_\ell) \subseteq \texttt{head}(Q)$ and $\texttt{attr}(R_\ell) \subseteq \texttt{attr}(R_h)$ for every relation $R_h \in \texttt{attr}(Q_0)$. Since $R_i$ is not dominated by $R_\ell$, there must exist another relation $R_k \in \texttt{rels}(Q) - \{R_i, R_j\}$ such that $\texttt{attr}(R_\ell) - \texttt{attr}(R_k) \neq \emptyset$ and $(\texttt{attr}(R_i) \cap \texttt{attr}(R_k)) - \texttt{attr}(R_\ell) \neq \emptyset$. Note that $R_k$ is not in $Q_0$; otherwise, $\texttt{attr}(R_\ell) - \texttt{attr}(R_k) = \emptyset$. In this case, $(\texttt{attr}(R_i) \cap \texttt{attr}(R_k)) - \texttt{attr}(R_\ell) = \emptyset$, coming to a contradiction. Therefore, the IsPtime algorithm will go to "others", and return `false` for $Q_0$, as well as for $Q$. This allows us to claim our third hard case: if a strand exists, then CQ is NP-hard.

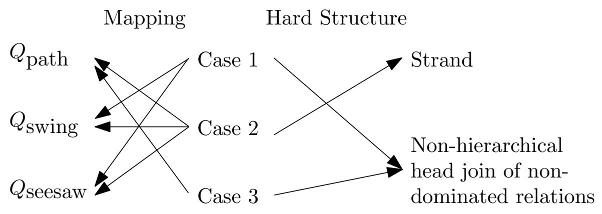


Figure 6: Correspondence between the three cases of CQs on which IsPtime falling into "other" bucket in Figure 4, the core query it maps to (the left) and the hard structure it contains (the right).

## 5.3 Sketch of Proof of Theorem 3

So far, we have defined three hard structures for general CQs, any one of which makes the `ADP` problem NP-hard. We now sketch the main ideas in the proof of Theorem 3; the detailed proof is in Appendix. This proof uses Theorem 2 by mapping each of the NP-hard cases in Theorem 2 to the existence of a hard structure as defined by Theorem 3, and vice-versa. But, interestingly, this mapping is not one-one in the sense that multiple cases in the procedural dichotomy established by Theorem 2 map to same case in the structural dichotomy of Theorem 3, and vice-versa. This lends further credence to our assertion that the procedural dichotomy of the previous section is not sufficient by itself to explain the structural reasons behind the NP-hardness or poly-time solvability of the `ADP` problem for individual CQs.

We first point out that the two simplification steps in the IsPtime procedure preserve the existence of hard structures.

**Lemma 8.** *Let $A$ be a universal attribute in $Q$. Then, there is a hard structure in $Q$ if and only if there is a hard structure in $Q_{-A}$.*

**Lemma 9.** *Let $Q_1, Q_2, \cdots, Q_s$ be the connected subqueries of $Q$. Then, there is a hard structure in $Q$ if and only if there is a hard structure in $Q_i$ for some $i \in \{1, 2, \cdots, s\}$.*

When neither of the simplification steps can be applied, IsPtime$(Q)$ ends up with three cases. If there is a vacuum relation in $Q$, say $R_i$, IsPtime$(Q)$ returns `true`. In this case, $Q$ does not contain any hard structure as $R_i$ is the only endogenous and non-dominated relation. If $Q$ is boolean, IsPtime$(Q)$ returns `false` if and only if it contains a triad. Then, we are left with the case when IsPtime$(Q)$ goes into the "Others" bucket. Each core query shown in Section 4.2.1 contains hard structure; more specifically, the head join of non-dominated relations in $Q_{\text{path}}$ is non-hierarchical, and both $Q_{\text{swing}}$ and $Q_{\text{seesaw}}$ contain a strand. In general, we can show the existence of hard structures for $Q$ falling into one of the three cases in Figure 4. The correspondence between different cases of the procedural and structural characterizations are shown in Figure 6.

# 6 Approximations

In this section, we discuss approximations for the `ADP`$(Q, D, k)$ problem when it is NP-hard.

## 6.1 Full CQs

We first consider full CQs, on which `ADP` problem can be related to the *Partial Set Cover problem* (PSC).

**Definition 9.** *Given a set of elements $\mathcal{U}$, a family of subsets $\mathcal{S} \subseteq 2^U$, and a positive integer $k'$, the goal of the Partial Set Cover problem is to pick a minimum collection of sets from $\mathcal{S}$ that covers at least $k'$ elements in $\mathcal{U}$.*

Observe that `ADP`$(Q, D, k)$, where the goal is to pick the smallest number of input tuples that intervene on at least $k$ output tuples, can be modeled as a PSC problem as follows. Sets correspond to input tuples from relations in the body of $Q$ and elements to output tuples in $Q(D)$. The set corresponding to an input tuple comprises all elements corresponding to output tuples that are deleted on the deletion of the input tuple. Also, $k' = k$. Additionally, if there are $p$ relations in $Q$, then every element belongs to at most $p$ sets. It is known that the PSC problem admits greedy and primal-dual algorithms with approximation factors of $O(\log k)$ and $p$ respectively [**13**]. Hence, we get the same results for the `ADP` problem.

**Theorem 5.** *For a full CQ $Q$ with $p$ relations, any instance $D$ and integer $k$, `ADP`$(Q, k, D)$ admits $O(\log k)$ and $p$-approximations.*

*Proof.* We prove that the reduction preserves the approximation guarantee in two steps: 1) given an instance of `ADP`$(Q, k, D)$, how to construct an instance of $k'$-PSC, and 2) given a solution to $k'$-PSC, how to recover a solution to `ADP`$(Q, k, D)$.

Given the full CQ $Q$ containing $p$ relations in its body, namely $R_1, R_2, \cdots, R_p$, we create a set per input tuple in the $p$ relations, and an element per output tuple in $Q(D)$. Each set contains elements that correspond to the output tuples resulting from the join between the associated input tuple and tuples from other relations in $Q$. It is well-known that the natural join on $R_1, R_2, \cdots, R_p$ can be computed in poly-time. Moreover, exactly one tuple in each of the $p$ relations participates in the join operation that produces a particular output tuple. Therefore, each element in the $k'$-PSC instance belongs to exactly $p$ sets. As a result, the size of the $k'$-PSC instance that we create is polynomial in the data complexity of `ADP`$(Q, k, D)$. Moreover, there is a one-on-one correspondence between instances of the two problems.

Lastly, given a $p$-approximate solution to $k'$-PSC, we recover a solution to `ADP`$(Q, D, k)$ by picking the tuples associated with the sets in the solution, say $I$. Observe that the sets in $I$ cover $k' = k$ elements in $U$. Thus, removing the corresponding input tuples from `ADP`$(Q, D, k)$ will intervene on at least $k$ output tuples.

Note that this implies that if the query has constant size, i.e., $p$ is a constant, full CQs admit a constant-factor approximation for the `ADP` problem. □

This implies that if the query has constant size, i.e., $p$ is a constant, full CQs admit a constant-factor approximation for the `ADP` problem.

## 6.2 Inapproximability of General CQs

The situation, however, is quite different for general CQs. We first observe that obtaining even sub-polynomial approximations for the `ADP` problem in general is unlikely. In particular, on $Q_{\text{swing}}(A) : R_2(A, B), R_3(B)$, which is the core hard query in Section 4.2.1, we show the following hardness:

**Lemma 10.** *Under some mild cryptographic assumptions, the `ADP`$(Q_{\text{swing}}, D, k)$ problem with $|D| = n$ is hard to approximate within $\Omega(n^{\epsilon})$ factor for some constant $\epsilon > 0$.*

Recall that we established NP-hardness of `ADP`$(Q_{\text{swing}}, D, k)$ via a reduction from the *k-minimum coverage* (`KMC`) problem. As shown in Appendix B, his reduction is also approximation-preserving, which implies the above lemma via known hardness results for the `KMC` problem [**1, 7, 6**]. While this rules out the possibility of approximation algorithms in general for the `ADP` problem, there are several query classes on which we had shown NP-hardness of the problem but their approximability is still open. This includes simple CQs such as $Q_{\text{seesaw}}(A) : R_1(A), R_2(A, B), R_3(B)$. We leave the precise classification of query classes according to approximability of the `ADP` problem as an interesting direction for future work.

## 7 Algorithms and Optimizations

The framework of our poly-time algorithm, which returns the exact solution for "easy" queries and a heuristic for hard queries, is described as ComputeADP in Algorithm 2. It builds upon the algorithm for the Resilience problem [**11**], which is a special case of the `ADP` problem. Our algorithm recursively calls itself through Universal and Decompose procedures. For poly-time solvable CQs, it only uses the first four cases: this follows the proof of Theorem 2 by applying the two simplifications repeatedly until it becomes a boolean query or contains a vacuum relation. Our first optimization is to include a new base case that we call *singleton*. If the conditions of this case (we describe them below) are satisfied, then a simple algorithm Singleton is directly applied instead of continuing to apply the two simplification steps. In addition to computing the optimal solution for poly-time solvable CQs, Algorithm 2 also generates a feasible

**Algorithm 2:** ComputeADP$(Q, D, k)$

```
1 If $Q$ is Boolean return Boolean$(Q, D, k)$;
2 ElseIf $Q$ is a singleton return Singleton$(Q, D, k)$;
3 ElseIf $Q$ has universal attribute then Universe$(Q, D, k)$;
4 ElseIf $Q$ is disconnected then Decompose$(Q, D, k)$;
5 Else return GreedyForCQ$(Q, D, k)$;
```

solution for NP-hard CQs. In this case, it alternately applies these two simplification steps until it becomes boolean or goes to the "others" category in Figure 3. We eventually invoke an approximate procedure GreedyForCQ on the non-boolean CQ when neither simplification step can be applied any more. Our second optimization is a smarter way of solving the recurrent formula for these two simplification steps, as shown in Universe$(Q, D, k)$ and Decompose$(Q, D, k)$. Note that the simplification steps involve large dynamic programs; so, this optimization provides significant scalability in practice. Both poly-time solvable and NP-hard queries benefit from the improvement of two simplification steps.

In the recursion tree of ComputeADP, each leaf node (Boolean, Singleton and GreedyForCQ) can be computed in poly-time and internal node (Universe and Decompose) can be built upon its children in poly-time. Also, there are $O(1)$ nodes in this recursion tree, since the query size (in terms of number of attributes and relations) is constant and each recursive call decreases the query by at least one relation or attribute. Hence, we get an poly-time algorithm overall.

## 7.1 Boolean

In [**11**], a poly-time algorithm was proposed for boolean CQs without a triad structure. A boolean query is *linear* if its relations may be arranged in linear order such that each attribute occurs in a contiguous sequence of atoms. It is proved that every boolean query without a triad structure can be transformed into a query of equivalent complexity that is linear. Thus, we only provide the algorithm for computing the `ADP` problem on an arbitrary linear query.

**Boolean**$(Q, D, k)$. We first label relations in linear ordering $R_1, R_2, \cdots, R_p$ and then build a network construct a network $G$ as follows. Note that $G$ is an $(p+1)$-partite graph consists of vertices $V = \{x\} \cup V_1 \cup V_2 \cup \cdots \cup V_{p-1} \cup \{y\}$, where $V_i = \texttt{attr}(R_i) \cap \texttt{attr}(R_{i+1})$. There is an edge $e = (u, v)$ for $u \in V_i, v \in V_{i+1}$ if there exists a tuple $t \in R_{i+1}$ with $\pi_{V_i} t = u$ and $\pi_{V_{i+1}} t = v$. Moreover, there is an edge between every vertex in $V_1$ and $x$, and every vertex in $V_{p-1}$ and $y$. Each edge has weight 1.

A minimum cut of $G$ is exactly the solution for $\texttt{ADP}(Q, D, k)$, which can be computed using the standard Edmonds–Karp algorithm with time complexity $O(|D|^3)$.

## 7.2 Singleton

We first lay out the conditions of this new base case for a poly-time solvable CQ:

**Definition 10** (Singleton)**.** *A CQ $Q$ is singleton, if there exists a relation $R_i \in \texttt{rels}(Q)$ such that (1) $\texttt{attr}(R_i) \subseteq \texttt{attr}(R_j)$ holds for every other relation $R_j \in \texttt{rels}(Q)$; and (2) either $\texttt{attr}(R_i) \subseteq \texttt{head}(Q)$ or $\texttt{head}(Q) \subseteq \texttt{attr}(R_i)$.*

Note that the execution of IsPtime can also be modeled as *recursion tree*, where each leaf node is either a Boolean query or contains vacuum relation, and each internal node corresponds to one simplification step. On this recursion tree, we point out an important property for singleton structure, as stated in Lemma 11.

**Lemma 11.** *For a CQ $Q$ on which* IsPtime$(Q)$ *returns* `true`*, each leaf (not root) node containing a vacuum relation must have an ancestor that is a singleton query.*

*Proof.* Note that each node $v$ in the recursive tree is associated with a query $Q_v$. Let $v$ be a leaf node in the recursion tree containing a vacuum relation $R_i$. Let $u$ be the parent node of $v$. Observe that $u$ doesn't contain a vacuum relation; otherwise, $u$ itself is a leaf. If $u$ generates $v$ by decomposing a disconnected, then $R_i$ is also a vacuum relation in $Q_u$, coming to a contradiction. If $u$ generates $v$ by removing an universal attribute $A$, $\texttt{attr}(R_i) = \{A\}$ in query $Q_v$. As $A$ is an universal attribute in $Q_u$, $Q_u$ is a singleton by Definition 10. $\square$

So, it suffices to replace the vacuum relation base case with the singleton.

**Algorithm 3:** SINGLETON$(Q, k, D)$

```
1  $R_i \leftarrow \arg\min_{R_j \in \texttt{rels}(Q)} |\texttt{attr}(R_j)|$;
2  if $\texttt{attr}(R_i) \subseteq \texttt{head}(Q)$ then
3      foreach tuple $t \in R_i$ do
4          $p_t \leftarrow \pi_{\texttt{attr}(R_i)=t} Q(D)$;
5      Sort all $p_t$'s in decreasing order as $p_1, p_2, \cdots, p_m$;
6      Find index $i$ such that $\sum_{j=1}^{i-1} p_j < k \leq \sum_{j=1}^{i} p_j$;
7      return $i$;
8  else
9      Remove all dangling tuples in $R_i$;
10     foreach $t \in Q(D)$ do
11         $c_t \leftarrow |\pi_{\texttt{head}(Q)=t} R_i|$;
12     Sort all $c_t$'s in increasing order as $c_1, c_2, \cdots, c_m$;
13     return $\sum_{j=1}^{k} c_j$;
```

**Singleton**$(Q, D, k)$. Let $R_i$ be the relation with the minimum number of attributes. By definition, either $\texttt{head}(Q) \subseteq \texttt{attr}(R_i)$ or $\texttt{attr}(R_i) \subseteq \texttt{head}(Q)$.

Case 1: $\texttt{attr}(R_i) \subseteq \texttt{head}(Q)$. We compute the number of output tuples that inherent attribute values from a tuple $t \in R_i$ and call it the "profit" of $t$, denoted as $p_t$. Then, we sort the tuples by their profits and choose greedily in decreasing order until their sum exceeds $k$. These chosen tuples form an optimal solution.

Case 2: $\texttt{head}(Q) \subseteq \texttt{attr}(R_i)$. We first remove all *dangling tuples*[2] in $R_i$, i.e., those don't participate in the full join result of the body of $Q$. Then we count for each output tuple $t \in Q(D)$, the number of tuples in $R_i$ whose projection on attributes $\texttt{head}(Q)$ is equivalent to $t$, and call it the "cost" of $t$, denoted by $c_t$. Finally, we sort the output tuples by cost and choose in increasing order the first $k$ tuples. The optimal solution is now obtained as the set of input tuples in $R_i$ whose removal deletes $k$ output tuples.

This algorithm takes $O(|D|^{|Q|})$ time since computing full join results dominates the complexity.

## 7.3 Universe and Decompose

We show some optimization for DECOMPOSE and UNIVERSE procedures respectively.

**Decompose**$(Q, D, k)$. Assume $Q$ has $s$ connected subqueries, $Q_1, Q_2, \cdots, Q_s$. The divide-and-conquer strategy will first compute $\texttt{ADP}(Q_i, D, k_i)$ for each subquery $Q_i$ over $k_i$, and then find an optimal combination of $k_1, k_2, \cdots, k_s$ by enumeration over $\Theta(k^s)$ solutions, which becomes expensive for large $s$. We give an optimized algorithm.

Let OPT$[i][j]$ denote the minimum number of input tuples to remove at least $j$ output tuples from subquery $\times_{j=1}^{i} Q_j(D)$. OPT$[i][j]$ can be computed using the following dynamic program:

$$\textsc{Opt}[i][j] = \min_{k_1, k_2 \in K(i,j)} \textsc{Opt}[i-1][k_1] + \textsc{ComputeADP}(Q_i, D, k_2)$$

where $K(i,j) = \{k_1, k_2 : k_1|Q_i(D)| + k_2 \prod_{\ell=1}^{i-1} |Q_\ell(D)| - k_1 k_2 \geq j, k_1, k_2 \in \mathbb{Z}^+\}$ and Algorithm 2 is invoked for solving $\texttt{ADP}(Q_i, D, k_2)$. To remove at least $j$ output tuples from $\times_{j=1}^{i} Q_j(D)$, we remove $k_1$ output

[2]A tuple is *dangling* if it doesn't participate in any full join result, and *non-dangling* otherwise. For $R_j \in \texttt{rels}(Q)$, its non-dangling tuples can be obtained by projecting full join results on $\texttt{attr}(R_i)$. This can be done in poly-time.

**Algorithm 4:** UNIVERSE$(Q, D, k)$

```
1  $A \leftarrow \mathtt{head}(Q) \cap \left(\bigcap_{R \in \mathtt{rels}(Q)} \mathtt{attr}(R)\right)$;
2  Label all possible combinations over $A$ as $\{a_1, a_2, \cdots, a_g\}$;
3  foreach $i \in \{1, 2, \cdots, g\}$ do
4      $D_i \leftarrow \{\sigma_{\pi_A t = a_i} R_i : \forall R_i \in \mathtt{rels}(Q)\}$;
5  foreach $j \in \{1, 2, \cdots, k\}$ do
6      $\textsc{Opt}[1][j] \leftarrow \textsc{ComputeADP}(Q, D_1, j)$;
7  foreach $i \in \{2, \cdots, g\}$ do
8      foreach $j \in \{1, 2, \cdots, k\}$ do
9          $\textsc{Opt}[i][j] \leftarrow \textsc{Opt}[i-1][j]$;
10         for $m = 1$ to $j - 1$ do
11             $c_{i,m} \leftarrow \textsc{ComputeADP}(Q, D_i, m)$;
12             if $\textsc{Opt}[i][j] > \textsc{Opt}[i-1][j-m] + c_{i,m}$ then
13                 $\textsc{Opt}[i][j] \leftarrow \textsc{Opt}[i-1][j-m] + c_{i,m}$;
14 return $\textsc{Opt}[g][k]$;
```

**Algorithm 5:** DECOMPOSE$(Q, D, k)$

```
1  Let $Q_1, Q_2, \cdots, Q_s$ be the connected subquery of $Q$;
2  $Q_\alpha \leftarrow Q_1$;
3  foreach $j \in \{1, 2, \cdots, k\}$ do
4      $\textsc{Opt}[1][j] \leftarrow \textsc{ComputeADP}(Q, D_1, j)$;
5  foreach $i \in \{2, 3, \cdots, s\}$ do
6      $m_1 \leftarrow \prod_{\ell=1}^{i-1} |Q_\ell(D)|$, $m_2 \leftarrow |Q_i(D)|$;
7      foreach $j \in \{1, 2, \cdots, k\}$ do
8          $\textsc{Opt}[i][j] \leftarrow +\infty$;
9          foreach $(k_1, k_2) \in \{0, 1, \cdots, j\} \times \{0, 1, \cdots, j\}$ do
10             if $k_1 m_2 + k_2 m_1 - k_1 k_2 \geq j$ then
11                 $c_{i,k_2} \leftarrow \textsc{ComputeADP}(Q_i, D, k_2)$;
12                 if $\textsc{Opt}[i][j] > \textsc{Opt}[i-1][k_1] + c_{i,k_2}$ then
13                     $\textsc{Opt}[i][j] \leftarrow \textsc{Opt}[i-1][k_1] + c_{i,k_2}$;
14     $Q_\alpha \leftarrow Q_\alpha \times Q_i$;
15 return $\textsc{Opt}[s][k]$;
```

tuples from first $i-1$ queries and $k_2$ output tuples from $Q_i(D)$, the total number of results removed is $k_1|Q_i(D)| + k_2 \prod_{\ell=1}^{i-1} |Q_\ell(D)| - k_1 k_2$ since results across subqueries are joined by Cartesian product. Thus, after recursively computing the solution to $\mathtt{ADP}(Q_i, D, k_2)$ for each subquery $Q_i$ over all values of $k_2$, the recurrence formula can be solved in $O(s \cdot k^3) = O(|Q| \cdot k^3)$ time since there are $O(sk)$ cells in the two-dimensional data structure $\textsc{Opt}[i][j]$ and each can be computed in $O(k^2)$ time.

**Universe**$(Q, D, k)$. Let $A$ be an universal attribute in $Q$. The input instance $D$ is partitioned into $D_1, D_2, \cdots, D_g$ corresponding to possible combinations of values $a_1, a_2, \cdots, a_g$ over $A$. In $D_i$, each tuple $t$ has $\pi_A t = a_i$. Note that the query result $Q(D)$ is a disjoint union of the subquery results $Q(D_1), Q(D_2), \cdots, Q(D_i)$.

Let $\textsc{Opt}[i][s]$ denote the minimum number of input tuples that have to be removed in order to remove at least $s$ output tuples from $Q(D)$, under the constraint that the input tuples can only be chosen from $D_1$ to $D_i$. Using this notation, we can now write the following dynamic program:

$$\textsc{Opt}[i][s] = \min_{m=0}^{s} \Big\{ \textsc{Opt}[i-1][s-m] + \textsc{ComputeADP}(Q, D_i, m) \Big\}.$$

where Algorithm 2 is revoked for solving the $\mathtt{ADP}(Q, D_i, m)$ over $1 \leq i \leq g$ and $0 \leq m \leq s$.

When there are more than one universal attributes, they should be removed as one "combined" attribute, instead of one by one. Let $A_1, A_2, \cdots, A_h$ be the universal attributes in $Q$. Assume all subproblems

ADP$(Q, D_i, j)$ over $1 \leq i \leq g$ and $1 \leq j \leq k$ have been computed. Then, removing $A_1$, $A_2$, $\cdots$, $A_h$ one by one takes $O(k \cdot |\pi_{A_1,A_2,\cdots,A_h} Q(D)|)$ time while removing them as whole (say in index ordering) takes $O(k \cdot \sum_{\ell=1}^{h} |\pi_{A_1,\cdots,A_\ell} Q(D)|)$ time. Our experiments show this difference in practice.

## 7.4 Greedy Heuristics

Clearly, we cannot hope for a poly-time algorithm on NP-hard CQs for all input instances $D$ and integers $k$. We provide the following greedy heuristics for computing a feasible solution to ADP$(Q, D, k)$ when it is NP-hard.

**GreedyForCQ**$(Q, D, k)$: For many simple queries, the ADP problem is NP-hard, and is even hard to approximate implied by the results in Section 6. The prime-dual approximation algorithm [**13**] for full CQs mentioned in Section 6.1 is not scalable since the size of linear programming would become very large, and not applicable to CQs with projections. So, we give a greedy heuristic for handling all NP-hard CQs when neither simplification steps can be applied.It greedily chooses a tuple which removes the maximum number of output tuples among the remaining ones in every step (like the approximation algorithm for the set cover problem). Moreover, we can narrow our scope to tuples in endogenous relations in the greedy algorithm. Note that GreedyForCQ achieves $O(\log k)$-approximation for full CQs, but there is no theoretical guarantees on the approximation ratio when projection exists.

**Algorithm 6:** GreedyForCQ$(Q, D, k)$

```
1 $S \leftarrow \emptyset$;
2 while $k > 0$ do
3     $t' \leftarrow null$, $p(t') \leftarrow 0$;
4     foreach tuple $t$ from an endogenous relation do
5         $p(t) = |Q(D - S)| - |Q(D - S - t)|$;
6         if $p(t) \geq p(t')$ then
7             $t' \leftarrow t$, $p(t') \leftarrow p(t)$;
8     $S \leftarrow S \cup \{t'\}$, $k \leftarrow k - p(t')$;
9 return $S$;
```

**DrasticGreedyForFullCQ**$(Q, D, k)$: In the heuristic above, however, computing the "profit" for all input tuples from endogenous relations after every one input tuple is removed is expensive in practice. For full CQs, we propose a more 'drastic' greedy solution where we remove input tuples only from one endogenous relation (goes over all endogenous relations and picks the one giving smallest cost). This significantly improves the efficiency in our experiments, since the profits are computed for all input tuples only once (since different tuples in the same relation remove disjoint full join results), but theoretically the approximation ratio is no longer guaranteed. Moreover, this strategy fails on CQs with projection. The reason is that input tuples from the same relation do not necessarily remove distinct query results, thus adding their individual profits together is not equivalent to the profit of their union.

**Algorithm 7:** DrasticGreedyForFullCQ$(Q, D, k)$

```
1 $S \leftarrow \emptyset$;
2 foreach endogenous relation $R(e)$ do
3     foreach $t \in R(e)$ do
4         $p(t) = |Q(D)| - |Q(D - t)|$;
5     Sort $R(e)$ by $p(t)$ decreasingly, as $t_1, t_2, \cdots, t_{|R(e)|}$;
6     Find the smallest $i$ such that $\sum_{j=1}^{i} p(t_j) \geq k$;
7     if $i \leq |S|$ then
8         $S \leftarrow \{t_j \in R(e) : j \leq i\}$;
9 return $S$;
```

## 7.5 Supporting Selection Operator

So far, we focused on the class of CQs only with *project* and *join* operators. In fact, our algorithm also supports a larger class of CQs involving selection operator (when the domain of some of the attributes is restricted to be constant). The class of *conjunctive queries with selections* can be described as

$$Q(\mathbf{A}) :- \sigma_{\theta_1} R_1(\mathbb{A}_1), \sigma_{\theta_2} R_2(\mathbb{A}_2), \cdots, \sigma_{\theta_p} R_p(\mathbb{A}_p)$$

where $\theta_i$ is a set of predicates each in form of $A = a$ for some attribute $A \in \mathbb{A}$ and value $a$. The result of $\sigma_{\theta_i} R_i(\mathbb{A}_i)$ is the set of tuples in $R_i$ satisfying all predicates in $\theta_i$. Note that we do not have any selection in the head, since any selection in the head can be pushed down to relations in the query body. An attribute is *selected* if it appears in any selection; and *unselected* otherwise. Let $\mathbb{A}_\theta \subseteq \mathbb{A}$ be the set of *selected attributes* in $Q$. Here, we also don't include any self-joins, i.e., each $R_i$ in $Q$ is distinct.

Interestingly, for the ADP problem, the polynomial solvability of a CQ with selections is equivalent to that of the residual query on the unselected attributes. This is formally stated in Lemma 12.

**Lemma 12.** *For a CQ $Q$ with selection predicates $\theta$, the ADP$(Q, D, k)$ is NP-hard if and only if ADP$(Q_{-\mathbb{A}_\theta}, D, k)$ is NP-hard, where $Q_{-\mathbb{A}_\theta}$ is the residual query after removing selected attributes $\mathbb{A}_\theta$ from $Q$.*

*Proof.* We will first show that if ADP$(Q_{-\mathbb{A}_\theta}, D, k)$ is NP-hard, then ADP$(Q, D, k)$ is also NP-hard. For an arbitrary instance $D_\theta$ for $Q_{-\mathbb{A}_\theta}$, we construct another instance $D$ for $Q$ by setting a single value $*$ in the domain of every attribute $A \in \mathbb{A}_\theta$ and the related predicate as $A = *$. It can be easily checked that any solution for ADP$(Q, D, k)$ with selections is also a solution for ADP$(Q_{-\mathbb{A}_\theta}, D, k)$. If there is an poly-time algorithm for ADP$(Q, D, k)$, ADP$(Q_{-\mathbb{A}_\theta}, D, k)$ is also poly-time solvable, coming to a contradiction. Thus, the problem ADP$(Q, D, k)$ is NP-hard.

Next we show that if there is a poly-time algorithm $\mathcal{A}$ for ADP$(Q_{-\mathbb{A}_\theta}, D, k)$ over all instances $D$ and integer $k$, then there is also an poly-time algorithm $\mathcal{A}_\theta$ for ADP$(Q, D, k)$. Consider an arbitrary instance $D$ for query $Q$. Let $D'$ be the residual instance of applying predicates to $D$. Observe that the solution for ADP$(Q, D', k)$ is exactly that for ADP$(Q, D, k)$ since tuples in $D$ violating any predicate will not be removed. Moreover, tuples in $D'$ have the same value on every attribute $A \in \mathbb{A}_\theta$. Let $D''$ be the instance of removing attributes $\mathbb{A}_\theta$ from $D'$. The solution for ADP$(Q_{-\mathbb{A}_\theta}, D'', k)$ is also the solution for ADP$(Q, D', k)$, and can be computed in poly-time. Thus, ADP$(Q, D, k)$ is also poly-time solvable for any instance $D$ and integer $k$. □

# 8 Experiments

In this section, we evaluate the running time, scalability, and quality of ComputeADP algorithm, and compare it with other baselines.

**Algorithms:** In our plots, we call the exact algorithm using ComputeADP for easy (poly-time) queries as "Exact". For hard queries, and also for easy queries for scalability, we have implemented two versions of ComputeADP embedded with GreedyForCQ and DrasticGreedyForFullCQ separately, shorted as "Greedy" and "Drastic". We also implemented a baseline brute-force algorithm called "BruteForce", which enumerates all subsets of input tuples, computes the number of query results that can be removed by each subset (by invoking a SQL query), and finds the minimum one among which removes at least $k$ results.

**Reporting vs. counting versions:** Wherever applicable and feasible, we report the running time for both *counting version*, when the goal is to only count the minimum number of input tuples to remove to achieve the desired effect, and the *reporting version*, which reports the actual input tuples in one such solution. Note that for some of our motivating examples, e.g., for understanding robustness, the counting version suffices.

**Setup:** We implemented our algorithms in JavaSE-1.8 with the database stored in PostgreSQL 10.12. The experiments were performed on MacOS, with 16GB of RAM and Intel Core i7 2.9 GHz processor. We run the experiment 10 times and present the average results (metric) of the 10 runs.

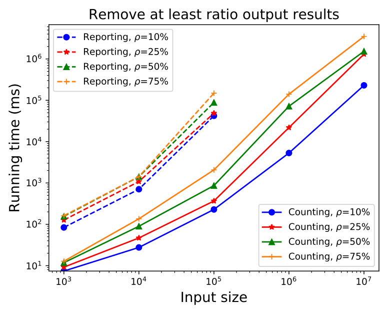


Figure 7: Running Time: $\sigma_\theta Q_1$ (easy) exactly (count/report).

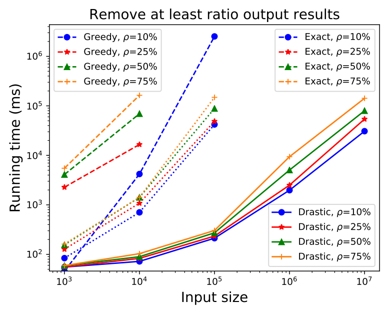


Figure 8: Running Time: reporting $\sigma_\theta Q_1$ (easy) by heuristics.

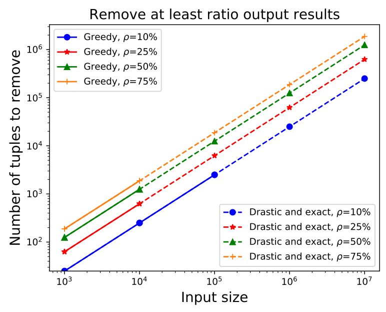


Figure 9: Quality: $\sigma_\theta Q_1$ (easy) by heuristics.

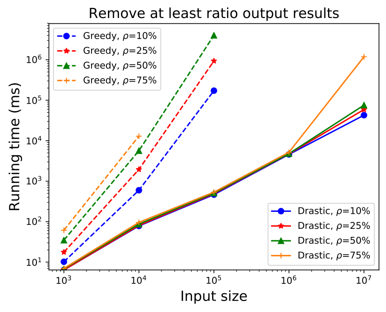


Figure 10: Running Time: reporting $Q_1$ (hard) by heuristics.

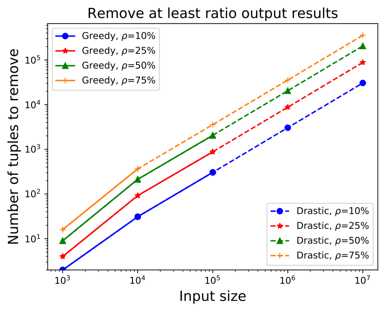


Figure 11: Quality: $Q_1$ (hard) by heuristics.

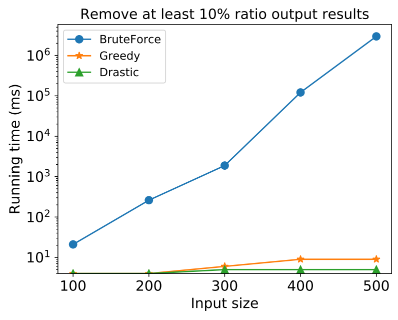


Figure 12: Running Time: brute-force v.s. heuristics for $Q_1$ (hard).

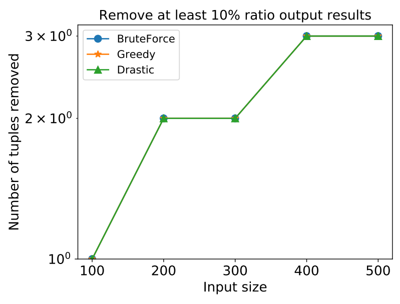


Figure 13: Quality: brute-force v.s. heuristics for $Q_1$ (hard).

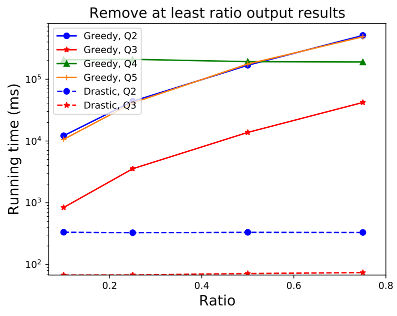


Figure 14: Running Time: $Q_2$, $Q_3$, $Q_4$, $Q_5$ (hard) by heuristics.

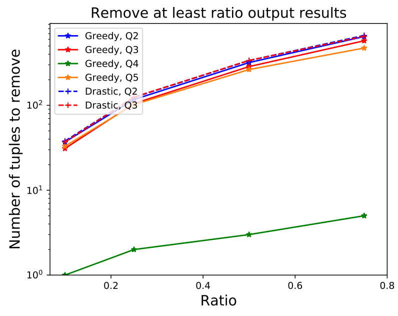


Figure 15: Quality: $Q_2$, $Q_3$, $Q_4$, $Q_5$ (hard) by heuristics.

## 8.1 Datasets and Queries

**TPC-H dataset and queries:** The TPC-H dataset has three relations: Supplier(S:NK, SK), PartSupp(PS:SK, PK), LineItem(L:OK, SK, PK). Consider the following two queries: *(1) Remove least number of orders or suppliers so that at least $\rho$% trading records can be restricted. (2) The same query but for the specific PartKey = 13370.* They can be characterized by two problems $\texttt{ADP}(Q_1, D, k)$ and $\texttt{ADP}(\sigma_\theta Q_1, D, k_\theta)$ respectively, where

- $Q_1$(NK, SK, PK, OK):-Supplier(S: NK, SK), PartSupp(PS: SK, PK), LineItem(L: OK, PK), $\theta : PK = 13370$, $k_\theta = \rho \cdot |\sigma_\theta Q(D)|$ and $k = \rho \cdot |Q(D)|$, where $\rho$ fraction of outputs are removed.

As shown in Lemma 12, the $\texttt{ADP}(\sigma_\theta Q_1, D, k)$ is poly-time solvable with exact optimal solution returned, while the $\texttt{ADP}(Q_1, D, k)$ is NP-hard with only heuristic solution returned, by ComputeADP.

**SNAP dataset and queries:** We use the common ego-networks from SNAP (Stanford Network Analysis Project) [**17**] for Facebook, where an ego-network of a user is a set of "social circles" formed by this user's friends [**18**]. This dataset consists 10 ego-networks, 4233 circles, 4039 nodes, and 88234 edges. We choose the network around user 414 which consists of 7 circles, 150 nodes and 3386 edges. We further create tables $R_i(A, B)$ for $i \in [4]$ and insert $E_j$ into $R_i$ if the rank of $E_j \mod 4 = i$. All edges are bi-directed. We evaluate three different queries on this dataset as below:

- $Q_2(A, B, C, D) : -R_1(A, B), R_2(B, C), R_3(C, D)$
- $Q_3(A, B, C) : -R_1(A, B), R_2(B, C), R_3(C, A)$
- $Q_4(A, C, E, G) : -R_1(A, B), R_2(B, C), R_3(E, F), R_4(F, G).$
- $Q_5(A, B, C) : -R_1(A, E), R_2(B, E), R_3(C, E)$

which are commonly used in community detection or friend recommendation over social networks. For instance, $Q_2$ finds a path of length three, $Q_3$ finds a triangle, $Q_4$ finds a pair of length-2 connection, and $Q_5$ captures a common friend. All of them are NP-hard, so ComputeADP only returns heuristic results.

## 8.2 Scalability

**Poly-time query:** We evaluate $\texttt{ADP}(\sigma_\theta Q_1, D, k_\theta)$ on the TPC-H dataset with different input sizes $N$ =1k, 10k, 100k, 1M, 10M, which denotes the number of survived tuples after selection. We use different fractions $\rho = 0.1, 0.25, 0.5, 0.75$. Figure 7 display the results for both reporting and counting versions. The running time increases with increase of input data size and the $\rho$. Since the counting version only performs computation on numbers in dynamic programming, it uses much less memory and behaves much more scalable than the reporting version does. Moreover, as a remedy for reporting results when the data size becomes large, we also test the Greedy and Drastic on $\sigma_\theta Q_1$ (by directly invoking Line 5 in Algorithm 2), whose running time is much smaller than the exact algorithm as shown in Figure 8. Meanwhile, we also show the quality of these three techniques in Figure 9. All of them coincide due to the data distribution for $\sigma_\theta Q_1$, which implies that Greedy and Drastic also find optimal solutions. But Greedy is not as scalable as Drastic to larger dataset with input size 100K or more.

**Hard query:** We next evaluate $\texttt{ADP}(Q_1, D, k_\theta)$ on the TPC-H dataset with different input sizes $N$ =1k, 10k, 100k, 1M, 10M and $\rho = 0.1, 0.25, 0.5, 0.75$ using Greedy and Drastic separately. Since Drastic only computes the "profit" for all input tuples through a SQL query once, while Greedy needs to update these statistics once an input tuple is removed. Thus, Drastic takes much less time than Greedy, as shown in Figure 10. We also compare the quality of solutions returned by these two heuristics, as shown in Figure 11. Due to the data distribution (which is varied in Section 8.4), Greedy and Drastic have the same quality when data size is smaller than 100K. However, Greedy is not scalable to larger dataset and quality results are only shown for Drastic in Figure 11.

**Comparison with brute-force:** Next, we evaluate the BruteForce algorithm on the TPC-H dataset for the NP-hard query $\texttt{ADP}(Q_1, D, k)$ with input size $N = 500$ and $\rho = 0.1$. The straightforward brute-force implementation does not work even on such a small dataset, since it iterates over all subsets of input tuples and issues as many as $2^{500}$ SQL queries in total. We use an optimization here by iterating all subsets in increasing order of their sizes, until a feasible solution (removing at least $k$ query results) is found.

We compare the optimized BruteForce with two heuristics. All three algorithms have their quality coinciding for this small dataset, as shown in Figure 13. But heuristics significantly improve the running time of BruteForce, as shown in Figure 12. The BruteForce did not stop in several hours for $N = 1000$ or $\rho = 0.2$.

## 8.3 Complexity of Queries

For each of $Q_2, Q_3, Q_4, Q_5$, we ran our experiments on the SNAP dataset and varied the fraction of query results to be removed (denoted as $\rho$) over $\{0.1, 0.25, 0.5, 0.75\}$. We evaluated Greedy and Drastic as follows. First, we invoked GreedyForCQ directly on $Q_2, Q_3, Q_5$ since neither of the simplification steps can be applied to these queries. For $Q_4$, Greedy first decomposes it into two subqueries as $Q_{41}(A, C) : -R_1(A, B), R_2(B, C)$ and $Q_{42}(E, G) : -R_3(E, F), R_4(F, G)$ using Decompose, and handles them using GreedyForCQ separately. Next, we invoked DrasticGreedyForFullCQ on $Q_2, Q_3$ directly. All running times are displayed in Figure 14. As Drastic cannot be applied to $Q_4, Q_5$ with projection, these are not in Figure 14. The quality of these heuristics is displayed in Figure 15.

The running time of Drastic depends on (i) the number of endogenous relations, (ii) computing the profits for all tuples in an endogenous relation by SQL queries, (iii) sorting the tuples by profit, and (iv) finding tuples with largest profits whose profits add up to at least $k$. Note that $Q_2, Q_3$ are executed on the same dataset and the number of input tuples to be removed are almost the same (see Figure 15). So Figure 14 displays the difference in runtimes for executing the SQL queries for $Q_2, Q_3$.

The running time of Greedy depends on (i) the number of iterations of the while loop, which is equal to the number of input tuples to be removed, (ii) the number of SQL queries for each iteration of the while loop, which is the number of endogenous relations, and (iii) the time for executing one SQL query. On $Q_2, Q_3, Q_5$, Greedy removes almost the same number of tuples as shown in Figure 15. So, Figure 14 displays the difference in running time for executing SQL queries for $Q_2, Q_3, Q_5$ respectively. Note that Greedy needs to solve a dynamic program in Decompose as well as a large number of sub-problems for both $Q_{41}, Q_{42}$, which is only relevant to the sizes of their own query results, so $Q_4$ has a larger and stable running time even though it removes much fewer input tuples.

## 8.4 Data Distribution

We study the performance of ComputeADP for a poly-time solvable singleton query $Q_6(A, B) : -R_1(A)$, $R_2(A, B)$ and an NP-hard query $Q_{\text{path}}(A, B) : -R_1(A), R_2(A, B), R_3(B)$ on various data distributions, where the degrees of values from $A$ or $B$ in relation $R_2(A, B)$ is varied according to to obtain the different distributions. We used the Zipfian distribution, where the frequency of the $i$-th distinct key is proportional to $i^{-\alpha}$. The parameter $\alpha \geq 0$ controls the skewness of the distribution: larger $\alpha$ means larger skew. We fix the distribution of degrees for values in $B$ as uniform and vary the skewness of degrees of values in $A$ by varying $\alpha$. We evaluate both $Q_6$ and $Q_{\text{path}}$ on our synthetic dataset with different input sizes $N = 1k, 10k, 100k, 1M$ and $0.2N$ distinct values in $A$ and $B$ separately. The results for $Q_{\text{path}}$ are shown in Figure 16–19, and those for $Q_6$ are shown in Figure 20–23. We also tested other values of $\alpha$, which are reported in Figures 24, 25, 26, 27.

For every fixed value of $\alpha$, the running time as well as the size of solutions returned by any algorithm increase with the input size and the value of $\rho$. If both the input size and $\rho$ are fixed, the size of the solution decreases with increasing $\alpha$. This is because on a skewed instance, the same number of output tuples can be removed by removing fewer input tuples. The running time for Drastic and Exact stays almost the same since computing the profits for input tuples is the most costly step, independent of the size of the solution. However, the running time of Greedy decreases with the size of the solution, which is affected by $\alpha$.

## 8.5 Optimizations

Next, we evaluate our optimizations on synthetic datasets. We use the following two queries: a singleton query $Q_5$ (attributes in $R_1$ are universal) and a disconnected query $Q_6$ (that can be decomposed into three easy queries).

- $Q_7(A, B, C, D, E, F, G) : -R_1(A, B, C), R_2(A, B, C, D, E), R_3(A, B, C, D, G), R_4(A, B, C, F)$

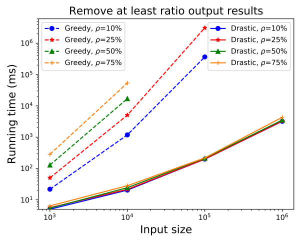


Figure 16: $\alpha = 0$ (hard)

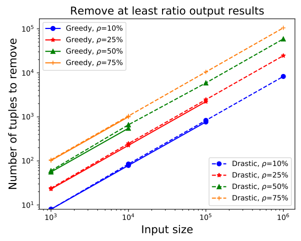


Figure 17: $\alpha = 0$ (hard)

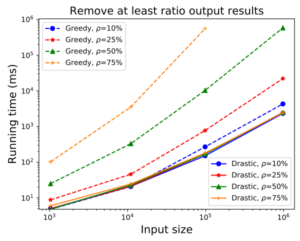


Figure 18: $\alpha = 1$ (hard)

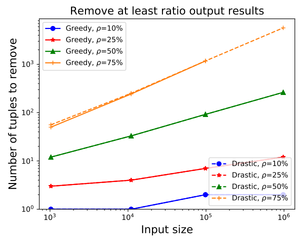


Figure 19: $\alpha = 1$ (hard)

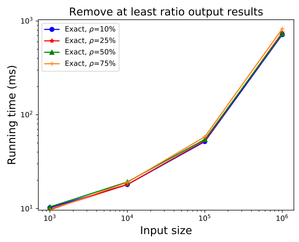


Figure 20: $\alpha = 0$ (easy)

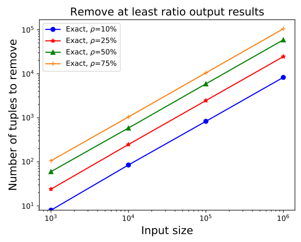


Figure 21: $\alpha = 0$ (easy)

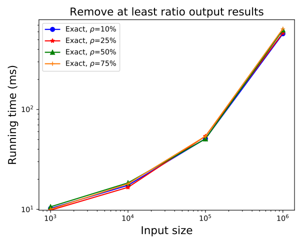


Figure 22: $\alpha = 1$ (easy)

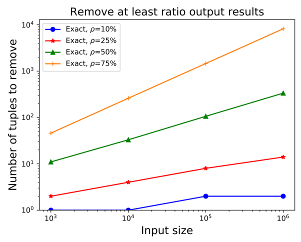


Figure 23: $\alpha = 1$ (easy)

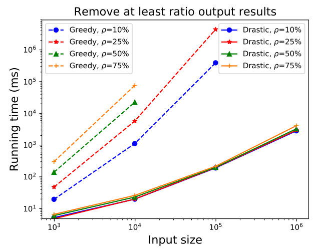


Figure 24: $\alpha = 0.25$ (hard)

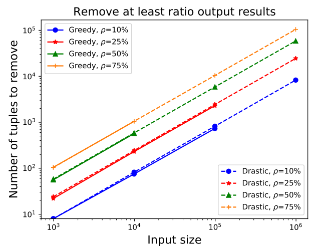


Figure 25: $\alpha = 0.25$ (hard)

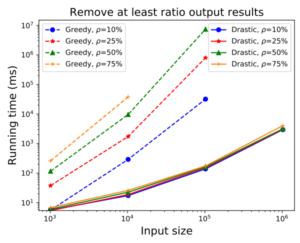


Figure 26: $\alpha = 0.5$ (hard)

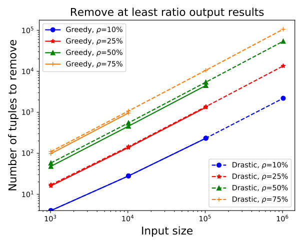


Figure 27: $\alpha = 0.5$ (hard)

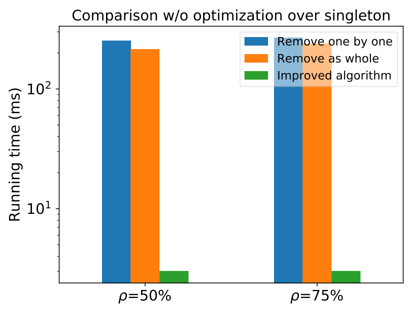


Figure 28: $Q_7$.

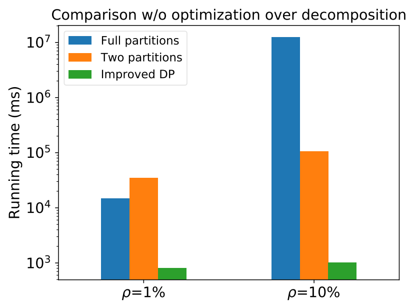


Figure 29: $Q_8$.

- $Q_8(A_1, \cdots, C_3) :- R_{11}(A_1), R_{12}(A_1, B_1), R_{21}(A_2), R_{22}(A_2, B_2), R_{31}(A_3), R_{32}(A_3, B_3)$

We generate relatively small synthetic datasets, as the non-optimized algorithm would take prohibitively long time on larger ones. For $Q_7$, each relation has 500 input tuples and each tuple is randomly generated with a combination of integers between 1 and 100; for $Q_8$, $R_{11}, R_{21}, R_{31}$ each has 25 input tuples and $R_{12}, R_{22}, R_{32}$ each has 50. Each input tuple is randomly generated with a combination of integers between 1 and 100. For $\texttt{ADP}(Q_7, D, k)$, we compare three different strategies: (1) removing universal attributes $A, B, C$ one by one, (2) removing $A, B, C$ together, and (3) invoking procedure $\textsc{Singleton}(Q_7, D, k)$ based on sorting; the results are shown in Figure 28. For $\texttt{ADP}(Q_8, D, k)$, we compare three different strategies: (1) decompose into 3 partitions at once, (2) decompose into 2 partitions each time, and (3) improved dynamic programming; the results are shown in Figure 29. Note that all these strategies will compute all subproblems $\texttt{ADP}(Q_i, D, k)$ for each subquery $Q_i(A_i, B_i, C_i) :- R_{i1}(A_i, B_i), R_{i2}(A_i, B_i)$, but only differ how the solutions for each subquery are used to construct the optimal solution for the $\texttt{ADP}(Q_7, D, k)$ problem. Figures 28 and 29 show that optimizations improve the running time significantly.

## 9 Future Work

Several open questions remain. First, it would be interesting to study the ADP problem beyond CQs. In particular, many natural queries involve self-joins and/or aggregates like *sum*, for which the observations of this paper do not apply. It is also natural to consider scenarios where all input tuples are not equivalent in terms of the cost of removing them. As a first step, one might want to consider a scenario where only a subset of input tuples can be removed, and the remaining input tuples cannot be deleted. Investigating the approximability of the ADP problem is another interesting research direction. Although we showed some preliminary results in this context, obtaining an exact characterization of the approximability of this problem for individual queries, even for the special case of the Resilience problem, remains open. A related question is that of the parameterized complexity of ADP with respect to $k$ for full CQs. While we showed that ADP admits a poly-time algorithm for fixed $k$, obtaining an FPT algorithm for the problem remains open.

## References

[**1**] B. Applebaum. Pseudorandom generators with long stretch and low locality from random local one-way functions. *SIAM Journal on Computing*, 42(5):2008–2037, 2013.

[**2**] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, Dec. 1981.

[**3**] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 150–158, 2002.

[**4**] B. Caskurlu, V. Mkrtchyan, O. Parekh, and K. Subramani. Partial vertex cover and budgeted maximum coverage in bipartite graphs. *SIAM J. Discrete Math.*, 31(3):2172–2184, 2017.

[**5**] J. Chen and I. A. Kanj. Constrained minimum vertex cover in bipartite graphs: complexity and parameterized algorithms. *Journal of Computer and System Sciences*, 67(4):833–847, 2003.

[**6**] E. Chlamtác, M. Dinitz, C. Konrad, G. Kortsarz, and G. Rabanca. The densest k-subhypergraph problem. *SIAM Journal on Discrete Mathematics*, 32(2):1458–1477, 2018.

[**7**] E. Chlamtáč, M. Dinitz, and Y. Makarychev. Minimizing the union: Tight approximations for small set bipartite vertex expansion. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 881–899. SIAM, 2017.

[**8**] G. Cong, W. Fan, and F. Geerts. Annotation propagation revisited for key preserving views. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management*, CIKM '06, pages 632–641, 2006.

[**9**] N. N. Dalvi and D. Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *J. ACM*, 59(6):30:1–30:87, 2012.

[**10**] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, Sept. 1982.

[**11**] C. Freire, W. Gatterbauer, N. Immerman, and A. Meliou. The complexity of resilience and responsibility for self-join-free conjunctive queries. *PVLDB*, 9(3):180–191, 2015.

[**12**] C. Freire, W. Gatterbauer, N. Immerman, and A. Meliou. New results for the complexity of resilience for binary conjunctive queries with self-joins. *arXiv preprint arXiv:1907.01129*, 2019.

[**13**] R. Gandhi, S. Khuller, and A. Srinivasan. Approximation algorithms for partial covering problems. *Journal of Algorithms*, 53(1):55–84, 2004.

[**14**] B. Kimelfeld. A dichotomy in the complexity of deletion propagation with functional dependencies. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 191–202, 2012.

[**15**] B. Kimelfeld, J. Vondrák, and R. Williams. Maximizing conjunctive views in deletion propagation. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2011, June 12-16, 2011, Athens, Greece*, pages 187–198, 2011.

[**16**] B. Kimelfeld, J. Vondrák, and D. P. Woodruff. Multi-tuple deletion propagation: Approximations and complexity. *PVLDB*, 6(13):1558–1569, 2013.

[**17**] J. Leskovec and A. Krevl. Snap datasets: Stanford large network dataset collection. *http://snap.stanford.edu/data/*, June 2014.

[**18**] J. Leskovec and J. J. Mcauley. Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547, 2012.

[**19**] E. Livshits, B. Kimelfeld, and S. Roy. Computing optimal repairs for functional dependencies. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 225–237, 2018.

[**20**] L. Mathieson and S. Szeider. The parameterized complexity of regular subgraph problems and generalizations. In *Proceedings of the fourteenth symposium on Computing: the Australasian theory-Volume 77*, pages 79–86, 2008.

[**21**] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1):34–45, 2010.

[**22**] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(12):1490–1493, 2011.

[**23**] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 337–348, 2012.

[**24**] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. *PVLDB*, 9(4):348–359, 2015.

[**25**] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *International Conference*

*on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1579–1590, 2014.

[**26**] M. Y. Vardi. The complexity of relational query languages. In *STOC*, pages 137–146, 1982.

[**27**] S. A. Vinterboa. A note on the hardness of the k-ambiguity problem. 2002.

[**28**] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

## A Endogenous relations

To compare our definitions with those from [**11**], we need to introduce the following terminologies. In a CQ $Q$, relation $R_j \in \texttt{rels}(Q)$ is *exogenous* if there exists another relation $R_i \neq R_j \in \texttt{rels}(Q)$ such that $\texttt{attr}(R_i) \subset \texttt{attr}(R_j)$, and *endogenous* otherwise. It should be noted that if there are more than one relation defining on the same attributes, i.e., $\texttt{attr}(R_i) = \texttt{attr}(R_j)$, then we just consider arbitrary one of them as *endogenous* and the remaining as *exogenous*. In $Q() : -R_1(A), R_2(A,B), R_3(B,C), R_4(B,C), R_5(B,C)$, there are two endogenous relations $R_1$ and any one of $R_3, R_4, R_5$. We generalize their observation on endogenous relations in [**11**] to the ADP problem, as stated in Lemma 13, which will be used in this paper.

**Lemma 13.** *For any CQ $Q$, if ADP$(Q, D, k)$ problem is poly-time solvable, there exists a solution which only removes input tuples from endogenous relations.*

*Proof.* Consider an arbitrary solution $\mathcal{S}$ for $\texttt{ADP}(Q, D, k)$. By contradiction, assume tuple $t \in R_j$ is removed by $\mathcal{S}$ where $R_j$ is an exogenous relation. Let $R_i \in \texttt{rels}(Q)$ be the endogenous relation such that $\texttt{attr}(R_i) \subset \texttt{attr}(R_j)$, and $t'$ be the tuple such that $\pi_{\texttt{attr}(R_i)} t = t'$. If $t' \in \mathcal{S}$, we observe that $\mathcal{S} - \{t\}$ also removes at least $k$ results from $Q(D)$, contradicting the optimality of $S$. Otherwise, $t' \notin \mathcal{S}$. Then we claim that $\mathcal{S} - \{t\} + \{t'\}$ is also an optimal solution for $\texttt{ADP}(Q, D, k)$. Applying this argument to each tuple removed from exogenous relation, we will obtain an optimal solution which only removes tuples from endogenous solution. Thus adding the restriction on $Q$ doesn't change the minimum number of tuples to be removed for $\texttt{ADP}(Q, D, k)$. □

## B Proof of Lemma 5

We show the NP-hardness of each problem in Lemma 5 separately.

**Hardness Proof of Problem (1).** With an equivalent definition, problem (1) is exactly the *Partial Vertex Cover for Bipartite Graphs* (PVCB) problem, which is known to be NP-hard [**4**].

**Definition 11.** *The input to the problem is an undirected bipartite graph $G(A, B, E)$ where $E$ is the set of edges between two sets of vertices $A$ and $B$, and an integer $k$. The goal is to find a subset $S \subseteq A \cup B$ of minimum size such that at least $k$ edges from $E$ have at least one endpoint in $S$.*

**Hardness Proof of Problem (2).** It is easy to relate problem (2) to the *k-Minimum Coverage* (KMC) problem, which is known to be NP-hard [**27**].

**Definition 12.** *Given a universe $\mathcal{U}$, a family $\mathcal{S}$ of subsets of $\mathcal{U}$ and an integer $k$, find $k$ subsets from $\mathcal{S}$ such that the size of their union is minimized.*

We give a reduction from the KMC problem that takes as input $(\mathcal{U}, \mathcal{S})$ and $k$, denoted as $\texttt{KMC}(\mathcal{U}, \mathcal{S}, k)$. Moreover, it can be easily checked that this reduction preserves the approximation, i.e., if there is an $\alpha$-approximation algorithm for the $\texttt{ADP}(Q_{\text{swing}}, D, k)$ problem, then there must exist an $\alpha$-approximation algorithm for the KMC problem.

Given an instance of the KMC problem, we construct a bipartite graph $G = (A, B, E)$ as follows. For each element $u \in \mathcal{U}$, we include a vertex $b_u \in B$. For each subset $S \in \mathcal{S}$, we include a vertex $a_S \in A$. If $u \in S$, we add an edge $(a_S, b_u) \in E$. Next we show that the problem $\texttt{KMC}(\mathcal{U}, \mathcal{S}, k)$ has a solution of size $\leq c$ if and only if the problem (2) has a solution of size $\leq c$.

**The"only-if" direction.** Suppose we are given a solution $\mathcal{S}' \subseteq \mathcal{S}$ for problem $\texttt{KMC}(\mathcal{U}, \mathcal{S}, k)$ of size $\leq c$. We then construct a solution for problem (2) as follows. If $u \in \bigcup_{S \in \mathcal{S}'} S$, then we remove $b_u$ from $B$. This solution removes at most $c$ vertices from $B$ since $|\bigcup_{S \in \mathcal{S}'} S| \leq c$. Moreover, every vertex $a_S \in A$ is removed as long as $S \in S'$. The total number of vertices removed from $A$ is at least $k$, thus this is exactly a solution for problem (2) of size $\leq k$.

**The "if" direction.** Suppose we are given a solution for problem (2) of size $\leq c$. We choose $k$ arbitrary vertices from $A$ which is removed because of the removal of vertices in $B$, denoted as $A'$. We then construct a solution for $\texttt{KMC}(\mathcal{U}, \mathcal{S}, k)$ as $\{S : a_S \in A'\}$. It can be easily argued that $|\bigcup_{S : a_S \in A'} S| \leq c$. Suppose not, there must exist at least one vertex $b_u$ for $u \in \bigcup_{S:a_S \in A'}$ not removed. In this way, at least one vertex in $A'$ cannot be removed, coming to a contradiction.

**Hardness Proof of Problem (3).** However, to our knowledge, there is no existing result directly implying the hardness of problem (3). We first elaborate it as the *Sided-Constrained Vertex Cover in Bipartite Graphs* (`SVCB`).

**Definition 13.** *The input to the problem is an undirected bipartite graph $G(A, B, E)$ where $E$ is the set of edges between two sets of vertices $A$ and $B$, and an integer $c$. The goal is to find a subset $S \subseteq A \cup B$ of minimum size such that each edge from $E$ have at least one endpoint in $S$ and at least $c$ vertices in $A$ are included in $S$.*

A related problem that has been studied is the *constrained minimum vertex cover*, which is known to be NP-complete [**5**], but with a different settings from `SVCB`. It asks to find a minimum vertex cover $S \subseteq A \cup B$ such that $|A \cap S| \leq k_1$ and $|B \cap S| \leq k_2$ for some input integer $k_1, k_2$. As a side product, we also first show that `SVCB` problem is NP-hard in Lemma 14, whose proof is given in Appendix C of independent interest.

**Lemma 14.** *The `SVCB` problem is NP-hard.*

We give a reduction from the decision version of `SVCB` problem that takes input $G = (A, B, E)$ and an integer $c \leq |A|$, denoted as `SVCB`$(G, c)$. For simplicity, assume each vertex in $G$ is incident to at least one edge in $E$. Given an instance of the `SVCB` problem, we have the same bipartite graph $G$ for problem (3). Next we show that `SVCB`$(G, c)$ has a solution of size $\leq c'$ if an only if the problem (3) with parameter $k = |A| - c$ has a solution of size $\leq c' - c$.

**The"only-if" direction.** Suppose we are given a vertex cover $\mathcal{C}$ for the problem `SVCB`$(G, c)$ of size $\leq c'$. Let $B_1 = \mathcal{C} \cap B$ and $A_1 = \mathcal{C} \cap A$, where $|A_1| \geq c$. As a complement of $\mathcal{C}$, $(A - A_1, B - B_1)$ form an independent set of $G$. This implies that for each vertex $a \in A - A_1$, if $(a, b) \in E$, then $b \in B_1$.

We construct a solution $\mathcal{S}$ for problem (3) as follows. We choose arbitrary $|A_1| - c$ vertices from $A_1$, denoted as $A_2$. Let $\mathcal{S} = A_2 \cup B_1$. The size of $\mathcal{S}$ can be bounded as $|A_1| + |B_1| - c = |\mathcal{C}| - c' \leq c' - c$. Moreover, it can be easily checked that $\mathcal{S}$ is a valid solution for problem (3). Each vertex $a \in A - A_1$ will be removed, since all of its neightbors are in $B_1$, which have been removed already. Additional $|A_1| - c$ vertices are also removed from $A_2$. Thus, the total number of vertices removed from $A$ is $|A - A_1| + |A_1| - c = |A| - c$.

**The "if" direction.** Suppose we are given a solution $\mathcal{S}$ for the problem (3) with parameter $k = |A| - c$, of size $\leq c' - c$. Let $B_1 = \mathcal{S} \cap B$ and $A_2 = \{a \in A : (a, b) \notin E \ \forall b \in B - B_1\}$. We mention two important properties on $\mathcal{S}$ first. (i) If $|A_2| \geq |A| - c$, then $\mathcal{S} = B_1$ with size $\leq c' - c$. (ii) If $|A_2| < |A| - c$, there must be $|\mathcal{S} \cap (A - A_2)| \geq |A| - |A_2| - c$. In this case, $c' - c \geq |\mathcal{S}| \geq |\mathcal{S} \cap (A - A_2)| + |B_1| \geq |A| - |A_2| - c + |B_1|$, thus $c' \geq |A| - |A_2| + |B_1|$.

We construct a solution $\mathcal{C}$ for the problem `SVCB`$(G, c)$ as follows. If $|A_2| \geq |A| - c$, choose arbitrary $|A_2| - |A| + c$ vertices from $A_2$ as $A_3$ and set $\mathcal{C} = (A - A_2 + A_3, B_1)$. Otherwise, set $\mathcal{C} = (A - A_2, B_1)$.

Observe that $\mathcal{C}$ is a valid vertex cover since $(A_2, B - B_1)$ is an independent set of $G$. It remains to show that $|\mathcal{C}| \leq c'$ and $|\mathcal{C} \cap A| \geq c$. Note that if $|A_2| \geq |A| - c$, we have $|\mathcal{C} \cap A| = |A| - |A_2| + |A_3| = c$ and $|\mathcal{C}| = |A| - |A_2| + |A_3| + |B_1| \leq c + c' - c = c'$, implied by (i). Otherwise, $|\mathcal{C} \cap A| = |A| - |A_2| \geq c$. Moreover, $|\mathcal{C}| = |A| - |A_2| + |B_1| \leq c'$, implied by (ii).

## C Proof of Lemma 14

In this part, we prove the NP-hardness of `SVCB` problem by showing that the NP-complete problem of `CLIQUE` in a regular graph [**20**] is polynomial time reducible to it, denoted as `REGULAR-CLIQUE`.

Recall that the input is an undirected bipartite graph $G(A, B, E)$ where $E$ is the set of edges between two sets of vertices $A$ and $B$, and an integer $c \leq |A|$. The goal is to find a subset $S \subseteq A \cup B$ of minimum size such that each edge from $E$ have at least one endpoint in $S$ and at least $c$ vertices in $A$ are included by $S$.

**Instance Construction.** Let $G' = (V', E')$ be a $d$-regular graph, where $|V'| = n$ and $|E'| = m$. Let $5 \leq q \leq \frac{n-1}{2}$ be an integer. The `CLIQUE` problem asks whether there exists a set of $q$ vertices in $V'$ such that each pair of vertices chosen are connected by an edge in $E'$. We construct an instance $G = (A \cup B, E)$

with $k_1, k_2$ as follows. Each vertex $u \in V'$ defines a vertex-block, in forms of a biclique $A_u \times B_u$, where $A_u \subseteq A$ contains $\lambda_1$ vertices and $B_u \subseteq B$ contains $\lambda_2$ distinct vertices. Moreover, $\lambda_1 - \lambda_2 \geq d$. Each edge $e \in E'$ defines a vertex $b_e \in B$. If vertex $u$ is the endpoint of edge $e$ in $G'$, we just add one edge from $b_e$ to one vertex in $A_u$ with degree $\lambda_2$. This is always possible since $\lambda_1 > d$. In our constructed graph, there is $|A| = \lambda_1 n$, $|B| = \lambda_2 n + m$ and $|E| = \lambda_1 \lambda_2 n + 2m$. Set $c = \lambda_1 q$.

Any $\lambda_1, \lambda_2, d$ satisfying the following the constraints work for this proof, say, $\lambda_1 = 2q(q+1), \lambda_2 = 2q^2, d = 2q$.

1. $\lambda_1 > \max\{2q-1, \frac{1}{2}q(q-1)\}$;
2. $\lambda_1 - \lambda_2 \geq d \geq 2q$;
3. $(\lambda_1 - \lambda_2)(n-q) + \frac{1}{2}q(q-1) \geq m$;
4. $\lambda_1 \geq (q-1) \cdot d$;
5. $\lambda_2 + \frac{1}{2}(q-1) > \frac{1}{2}\lambda_1$;
6. $\lambda_2 > \frac{1}{2}\lambda_1 + \frac{1}{2}(q-1)(d-q)$;
7. $2m = nd$;
8. $d \leq n-1$.

But for generality, we still use $\lambda_1, \lambda_2, d$ for analysis. We will show that the original graph $G' = (V', E')$ has a clique of size $q$ if and only if the bipartite graph $G = (A \cup B, E)$ has a vertex cover $J$ such that $|J \cap A| \geq \lambda_1 q$ and $|J| \leq \lambda_1 q + \lambda_2(n-q) + m - \frac{1}{2}q(q-1)$.

**"Yes" instance:** If there exists a clique of size $q$ in $G'$, we construct the vertex cover as follows. If a vertex $u \in V'$ is in the clique, choose $A_u$; otherwise, choose $B_u$. For an edge $e = (u, u') \in E'$, if at least one of $u, u'$ is not in the clique, choose $e_u$. It can be easily checked that each edge is covered, so this is a valid vertex cover. Moreover, $|J \cap A| = \lambda_1 q$ and $|J| = \lambda_1 q + \lambda_2(n-q) + m - \frac{1}{2}q(q-1)$.

**"No" instance:**If there exists no clique of size $q$ in $G'$, every vertex cover $J$ of $G$ with $|J \cap A| \geq \lambda q$, must have its size strictly larger than $\lambda_1 q + \lambda_2(n-q) + m - \frac{1}{2}q(q-1)$. Let $J^*$ be the minimum one among the class of vertex covers with $|J \cap A| \geq \lambda_1 q$.

The first observation is that $|J^* \cap A| = \lambda_1 q$. By contradiction, assume $|J^* \cap A| > \lambda_1 q$. If we can find some $u \in V'$ with $A_u \subsetneq J^*$, then $A_u \subseteq J^*$; for each vertex $a \in A_u \cap J^*$, we remove $a$ from $J^*$ and add $b_e$ to $J^*$ if there is a edge block $b_e$ connected to $v$. Otherwise, for each $u \in V'$ with $A_u \cap J^* \neq \emptyset$, there is $A_u \cap J^* = A_u$. In this case, $|J^* \cap A| = \lambda_1 q''$ with $q'' > q$. For an arbitrary $u \in V'$ with $A_u = J^*$, we remove $A_u$ from $J^*$ and add $B_u \cup (\bigcup_{e \in E' : u \in e} b_e)$ to $J^*$. Note that $|B_u \cup (\bigcup_{e \in E' : u \in e} b_e)| = |B_u| + |\bigcup_{e \in E' : u \in e} b_e| = \lambda_2 + d \leq \lambda_1$. In this way, we can get a better (at least not worse) vertex cover while maintaining the constraint that $|J^* \cap A| \geq \lambda_1 q$.

Based on $J^*$, we divide vertices in $V'$ into three subsets:

$$A_1 = \{u \in V' : A_u - J^* = \emptyset\};$$

$$A_2 = \{u \in V' : A_u - J^* \neq \emptyset, A_u \cap J^* \neq \emptyset\};$$

$$A_3 = \{u \in V' : A_u \cap J^* = \emptyset\};$$

Note that $J^*$ has to pick the $B_u$ for every $u \in A_2 \cup A_3$. We further consider two cases: (1) $|A_1| = q$; (2) $|A_1| \leq q-1$. Both cases are built on the following common observations. Consider an edge block $b_e$ with $e = (u, u')$. Let $a_{eu} \in A_u$ and $a_{eu'} \in A_{u'}$ be the two vertices incident to $b_e$ in $G$. Note that $b_e \notin J^*$ if and only if $a_{eu} \in J^*$ and $a_{eu'} \in J^*$.

Case 1: $|A_1| = q$. In this case, $|A_2| + |A_3| = n - q$, and $A_2 = \emptyset$ since $|J^* \cap A| = \lambda_1 q$. For any edge $e = (u, u') \in E$, $b_e \notin J^*$ if and only if $u \in A_1$ and $u' \in A_1$. Since there is no $q$-clique in $G'$, $J^*$ has size at least $\lambda_1 q + \lambda_2(n-q) + m - \frac{1}{2}q(q-1) + 1$.

Case 2: $|A_1| \leq q-1$. Consider each edge $e = (u, u') \in G'$. Observe that if one of $u, u'$ is in $A_3$, there must be $b_e \in J^*$. We further distinguish three more cases for $e$ when $b_e \notin J^*$. (i) both $u, u' \in A_1$, $b_e \notin J^*$.

Let $\alpha$ be the number of edges falling into this case. (ii) $u, u' \in A_2$, then $J^*$ has to choose both $a_{eu}, a_{eu'}$ for only exempting $e_u$. (iii) one of $u, u'$ is in $A_1$ and the other in $A_2$, say $u \in A_1, u' \in A_2$, then $J^*$ has to choose $a_{eu'}$ for exempting $b_e$; and the number of such edges is at most $|A_1| \cdot d - 2\alpha$. Note that $J^*$ will exempt as many as edge blocks as possible. With the additional budget of $\lambda_1(q - |A_1|)$ vertices in $A_2$, it will firstly exempt as many edge blocks in (iii) as possible; and then exempt edge blocks in (ii). Under the parameter constraint (1), $\lambda_1(q - |A_1|) \geq |A_1| \cdot d \geq |A_1| \cdot d - 2\alpha$ for any $|A_1| \in \{1, 2, \cdots, q-1\}$. So the number of exempted edge blocks is at most

$$\begin{aligned} f(|A_1|) &= \alpha + |A_1| \cdot d - 2\alpha + \frac{1}{2}\big(\lambda_1(q - |A_1|) - (|A_1| \cdot d - 2\alpha)\big) \\ &= \frac{1}{2}|A_1| \cdot d + \frac{1}{2}\lambda_1(q - |A_1|) \end{aligned}$$

In this case, $J^*$ has size at least $\lambda_1 q + \lambda_2(n - |A_1|) + m - f(|A_1|)$. To show why it is always strictly larger than $\lambda_1 q + \lambda_2(n-q) + m - \frac{1}{2}q(q-1)$, it suffices to show that

$$\lambda_2(q - |A_1|) - f(|A_1|) + \frac{1}{2}q(q-1) > 0$$

for any $|A_1| \in \{0, 1, 2, \cdots, q-1\}$. Rearranging the inequality, this is equivalent to show

$$(\lambda_2 - \frac{1}{2}\lambda_1)(q - x) + \frac{1}{2}q(q-1) - \frac{1}{2}xd > 0$$

holds for any $x \in [0, q-1]$. Note that this is a monotone function, so it holds for the whole interval $[0, q-1]$ as long as it holds for both endpoints. For $x = 0$, it holds if $\lambda_2 + \frac{1}{2}(q-1) > \frac{1}{2}\lambda_1$. For $x = q-1$, it holds if $\lambda_2 > \frac{1}{2}\lambda_1 + \frac{1}{2}(q-1)(d-q)$. Both constraints are implied by the parameter settings.

## D Proof of Theorem 3

We will prove Theorem 3 by drawing an equivalence to Theorem 2. For simplicity, when there is a triad-like or strand structure, or the head join of non-dominated relations is non-hierarchical in $Q$, $Q$ is referred to *contain hard structure*.

We first show that these two simplification steps in procedure IsPtime preserve the hard structures (Lemma 8 and Lemma 9).We then investigate three base cases. Note that when $Q$ is boolean, there is no triad structure since $\texttt{head}(Q) \cap \texttt{attr}(R_i) = \emptyset$ for any $R_i \in \texttt{rels}(Q)$. The head join of $Q$ has no attributes, thus always being hierarchical. On boolean CQ, Theorem 3 degenerates to Theorem 4 directly. So, it remains to consider the case when there is a vacuum relation in $Q$ (Lemma 15) or IsPtime($Q$) goes to "other" in Figure 3 (Lemma 16).

*Proof Lemma 8.* For each relation $R_i \in \texttt{rels}(Q)$, let $R'_i$ be the corresponding relation in $Q_{-A}$, with $\texttt{attr}(R'_i) = \texttt{attr}(R_i) - \{A\}$. We first mention two important observations for $Q, Q_{-A}$: (1) there is a one-to-one correspondence of non-dominated (resp. endogenous) relations in $Q$ and $Q_{-A}$, i.e., $R_i$ is non-dominated (resp. endogenous) if and only $R'_i$ is non-dominated (resp. endogenous); (2) for a full CQ, $Q$ is hierarchical if and only if $Q_{-A}$ is hierarchical. Both can be easily checked by definition. .

**The "only-if" direction.** Suppose $Q$ contains hard structure, and we prove each case separately.

If there is a triad-structure with a triple of endogenous relations $R_1, R_2, R_3 \in \texttt{rels}(Q)$ such that for each pair of relations, say $R_1, R_2$, there exists a path between $R_1, R_2$ only using attributes in $\texttt{attr}(Q) - \texttt{head}(Q) - \texttt{attr}(R_3)$. Obviously, $A$ doesn't appear on this path since $A \in \texttt{attr}(R_3)$. Correspondingly, this path between $R'_1, R'_2$ only uses attributes in $\texttt{attr}(Q_{-A}) - \texttt{head}(Q_{-A}) - \texttt{attr}(R_3) = \texttt{attr}(Q) - \texttt{head}(Q) - \texttt{attr}(R'_3)$. Similar argument applies for $R'_1, R'_3$ and $R'_2, R'_3$. Thus, $R'_1, R'_2, R'_3$ form a triad in $Q_{-A}$.

If there is a strand with a pair of non-dominated relations $R_1, R_2 \in \texttt{rels}(Q)$ such that (1) $\texttt{head}(Q) \cap \texttt{attr}(R_1) \neq \texttt{head}(Q) \cap \texttt{attr}(R_2)$; (2) $\texttt{attr}(R_i) \cap \texttt{attr}(R_j) - \texttt{head}(Q) \neq \emptyset$. It can be easily checked that $\texttt{head}(Q) \cap \texttt{attr}(R'_1) \neq \texttt{head}(Q) \cap \texttt{attr}(R'_2)$, and $\texttt{attr}(R'_i) \cap \texttt{attr}(R'_j) - \texttt{head}(Q_{-A}) = \texttt{attr}(R_i) \cap \texttt{attr}(R_j) - \texttt{head}(Q) \neq \emptyset$. Thus, $R'_1, R'_2$ form a strand in $Q_{-A}$.

If the head join of non-dominated relations in $Q$ is non-hierarchical, removing a universal attribute $A$ from all relations doesn't change this property. Thus, the head join of non-dominated relations in $Q_{-A}$ is also non-hierarchical.

**The "if" direction.** Suppose $Q_{-A}$ contains hard structure. This direction can be argued similarly with the "only-if" direction. $\square$

*Proof of Lemma 9.* We first mention two important observations for a disconnected query: (1) the set of non-dominated (resp. endogenous) relations in $Q$ is just the disjoint union of non-dominated (resp. endogenous) relations in each subquery; (2) a full join is hierarchical, if each of its connected subqueries is hierarchical. Both can be easily checked by definition.

**The "only-if" direction.** Suppose $Q$ contains hard structure, and we prove each case separately.

If there is a triad-structure with a triple of endogenous relations $R_1, R_2, R_3 \in \texttt{rels}(Q)$, they must come from the same subquery, say $Q_i$, since there exists a path between any pair of them by definition. It can be easily checked that $R_1, R_2, R_3$ still form a triad in $Q_i$.

Similarly, if there is a strand with a pair of endogenous relations $R_1, R_2 \in \texttt{rels}(Q)$, they must come from the same subquery, say $Q_i$, since they are connected. It can be easily checked that $R_1, R_2$ still form a strand in $Q_i$.

If the head join of non-dominated relations in $Q$ is non-hierarchical, we can identify two attributes $A, B$ and three non-dominated relations $R_1, R_2, R_3$ such that $A \in \texttt{attr}(R_1) \cap \texttt{attr}(R_2) - \texttt{attr}(R_3)$ and $B \in \texttt{attr}(R_3) \cap \texttt{attr}(R_2) - \texttt{attr}(R_1)$. In this way, $R_1, R_2, R_3$ must come from the same subquery, say $Q_i$. It can be easily checked that this condition still holds in $Q_i$, thus being non-hierarchical.

**The "if" direction.** Suppose $Q_i$ contains hard structure. It can be easily checked that any hard structure in $Q_i$ also exists in $Q$. $\square$

**Lemma 15.** *For a CQ $Q$, if there is a vacuum relation, then $Q$ doesn't contain any hard structure.*

*Proof.* Let $R_i$ be the vacuum relation. By definition, every remaining relation $R_j \in \texttt{rels}(Q) - \{R_i\}$ is dominated by $R_i$. Thus, there is neither triad-like nor strand structure in $Q$. The head join of non-dominated relations in $Q$ only includes $R_i$, thus always being hierarchical. Overall, $Q$ doesn't contain any hard structure. $\square$

**Lemma 16.** *For a CQ $Q$, if* IsPtime($Q$) *goes to "other" in Figure 3, then $Q$ contains hard structure.*

*Proof.* We follow the same proof plan for Lemma 4, by distinguishing the class of CQs characterized by Lemma 16 into three cases, as illustrated in Figure 4. Recall that any query characterized by Lemma 4 is connected, without any universal attribute and vacuum relation. we show that $Q$ falling into any one case contains hard structure.

**Case 1: head join contains at least one vacuum relation.** Let $R_i \in \texttt{rels}(Q)$ be the relation such that $\texttt{attr}(R_i) \neq \emptyset$ and $\texttt{attr}(R_i) \subseteq \texttt{attr}(Q) - \texttt{head}(Q)$. We start from any non-output attribute $B \in \texttt{attr}(R_i)$ and do a binary search until we find an output attribute $A$. Let $R_1, R_2$ be the consecutive pair of relations on this path between $A, B$, such that $A \in \texttt{attr}(R_1)$. Note that $\texttt{attr}(R_2) \subseteq \texttt{attr}(Q) - \texttt{head}(Q)$; otherwise, $R_2$ would be the first relation containing output attributes in our search. Moreover, $R_2$ is non-dominated since there is no vacuum relation in $Q$, and $R_1$ is also non-dominated since $\texttt{attr}(R_1) \cap \texttt{attr}(R_2) \subseteq \texttt{attr}(Q) - \texttt{head}(Q)$. In this way, $R_1, R_2$ form a stand in $Q$.

**Case 2: head join is disconnected (and no vacuum relation).** As there is no vacuum relation in head join, $\texttt{head}(Q) \cap \texttt{attr}(R_i) \neq \emptyset$ holds for each relation $R_i \in \texttt{rels}(Q)$. Moreover, we can always identify a pair of attributes $X, Z \in \texttt{head}(Q)$ such that there is no path between $X, Z$ in the head join. As $Q$ is connected, every path between $X, Z$ in $Q$ uses at least one non-output attribute.

Consider any path between $X, Z$ in $Q$, in which there is a pair of consecutive relations $R_1, R_2$ such that $\texttt{attr}(R_1) \cap \texttt{attr}(R_2) \subseteq \texttt{attr}(Q) - \texttt{head}(Q)$; otherwise, $X, Z$ are connected in the head join. Obviously, $\texttt{attr}(R_1) \cap \texttt{attr}(R_2) \cap \texttt{head}(Q) = \emptyset$. We claim that both $R_1, R_2$ are non-dominated. Suppose not, say $R_1$ is dominated by $R_i$. By definition, $\texttt{attr}(R_i) \subseteq \texttt{attr}(R_1)$. Observe that $\texttt{attr}(R_i) - \texttt{attr}(R_2) \neq \emptyset$ since $\texttt{attr}(R_i) - \texttt{attr}(R_2) \supseteq \texttt{attr}(R_i) \cap \texttt{head}(Q) - \texttt{attr}(R_2) \supseteq \texttt{attr}(R_i) \cap \texttt{attr}(R_1) \cap \texttt{head}(Q) - \texttt{attr}(R_2) \neq \emptyset$. Implied by Definition 7, $\texttt{attr}(R_1) \cap \texttt{attr}(R_2) \subseteq \texttt{attr}(R_i) \cap \texttt{head}(Q)$, coming to a contradiction. Applying a similar argument, we can show that $R_2$ is non-dominated. Moreover, $\texttt{attr}(R_1) \cap \texttt{head}(Q) \neq \emptyset$, $\texttt{attr}(R_2) \cap$

$\texttt{head}(Q) \neq \emptyset$, and $\texttt{attr}(R_1) \cap \texttt{attr}(R_2) \cap \texttt{head}(Q) = \emptyset$, thus $\texttt{attr}(R_1) \cap \texttt{head}(Q) \neq \texttt{attr}(R_2) \cap \texttt{head}(Q)$. In this way, $R_1, R_2$ form a strand in $Q$.

**Case 3: head join is connected (and no vacuum relation).** As there is no vacuum relation in head join, $\texttt{head}(Q) \cap \texttt{attr}(R_i) \neq \emptyset$ holds for each relation $R_i \in \texttt{rels}(Q)$. Note that there exists no universal attribute in $Q$.

We claim that the head join of non-dominated relations in $Q$ is also connected. Suppose not, there is a pair of attributes $A, B \in \texttt{head}(Q)$ which becomes disconnected in the head join of non-dominated relations. Consider any path $P$ between $A, B$ in the head join of $Q$, a sequence of relations where each pair of consecutive relations share at least one output attribute. We construct another path $P'$ as follows. For each relation $R_j \in P$, if it is dominated by $R_i \in \texttt{rels}(Q)$, then we just replace $R_j$ by $R_i$ in $P'$. Let $R_1 \in P, R'_1 \in P'$ be the first relation in each path respectively. If $A \notin \texttt{attr}(R'_1)$, then add an arbitrary non-dominated relation $R'_0 \in \texttt{rels}(Q)$ such that $A \in \texttt{attr}(R'_0)$ before $R'_1$. The similar operation is applied for $B$. We next argue that $P'$ is a valid path between $A, B$. It suffices to show that for each pair of consecutive relations in $P'$, they share at least one output attribute.

If $R'_0$ exists, we first show that $\texttt{attr}(R'_0) \cap \texttt{attr}(R'_1) \cap \texttt{head}(Q) \neq \emptyset$. In this case, $R_1 \neq R'_1$; otherwise, $A \in \texttt{attr}(R_1)$. Observe that $\texttt{attr}(R'_1)\texttt{attr}(R'_0) \neq \emptyset$, then $A \in \texttt{attr}(R_1) \cap \texttt{attr}(R'_0) \subseteq \texttt{attr}(R'_1) \cap \texttt{head}(Q) \subseteq \texttt{attr}(R'_1)$, coming to a contradiction. Otherwise, $\texttt{attr}(R'_1) \subseteq \texttt{attr}(R'_0)$, thus

$$\texttt{attr}(R'_0) \cap \texttt{attr}(R'_1) \cap \texttt{head}(Q) = \texttt{attr}(R'_1) \cap \texttt{head}(Q) \neq \emptyset.$$

The symmetric case when such a relation for $B$ is added can be argued similarly.

Consider any pair of consecutive relations $R_1, R_2 \in P$. Let $R'_1, R'_2$ be the corresponding relations in $P'$. By contradiction, assume $R'_1 \cap R'_2 \cap \texttt{head}(Q) = \emptyset$. If $R_1 = R'_1, R_2 = R'_2$, it comes to a contradiction. Otherwise, we further distinguish two cases. If only one of $R_1 = R'_1$ and $R_2 = R'_2$ holds, say $R_1 \neq R_1, R_2 = R'_2$. Since $\texttt{attr}(R'_2) - \texttt{attr}(R'_1) \neq \emptyset$, then $\texttt{attr}(R_1) \cap \texttt{attr}(R_2) = \texttt{attr}(R_1) \cap \texttt{attr}(R'_2) \subseteq \texttt{attr}(R'_1) \cap \texttt{head}(Q)$, which implies $\texttt{attr}(R'_1) \cap \texttt{attr}(R'_2) \cap \texttt{head}(Q)$, coming to a contradiction. Otherwise, $R_1 \neq R_1, R_2 \neq R'_2$, which can be argued similarly.

Note that if a full CQ is connected without a universal attribute, it must be non-hierarchical, implied by the definition of hierarchical join. In this way, the head join of non-dominated relations in $Q$ is non-hierarchical.

When IsPtime($Q$) goes to "others", some hard structure has been identified in $Q$ in each case, thus completing the whole proof. $\square$