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,sudeepa}@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.

PVLDB Reference Format:
Xiao Hu, Shouzhuo Sun, Shweta Patwa, Debmalya Panigrahi, and Sudeepa Roy . Aggregated Deletion Propagation for Counting Conjunctive Query Answers. PVLDB, 14(2): 228-240, 2021.
doi:10.14778/3425879.3425892

PVLDB Availability Tag:
The source code of this research paper has been made publicly available at https://github.com/ssz1997/GDP.git.

1 INTRODUCTION

The problem of *view update* (e.g., [2, 9]) – 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.

*This work has been supported in part by NSF awards IIS-1552538, IIS-1703431, CCF-1750140, IIS-1814493, CCF-1955703, CCF-2007556, and NIH award R01-EB025021.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 2 ISSN 2150-8097.
doi:10.14778/3425879.3425892

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 1.1. *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 1.2. *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 1.3. *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 [25]) 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 primal-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, 9]). 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 [7]. 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 [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 [8], responsibility [20], or database repair [19]. Another problem related to ADP is *reverse data management* and *how-to* queries [21, 22]. 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 [23, 24, 26], 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 [10] (see also [11] 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 $k = |Q(D)|$. [10] 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 [10] as a special case of our algorithmic and structural characterization for ADP and discuss the resilience problem further in subsequent sections.

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 $Q_2(D)$.

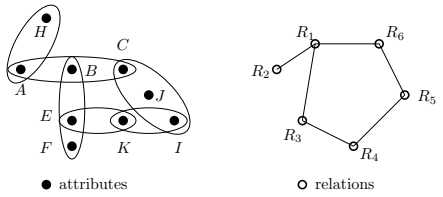

Figure 2: Hypergraph (left) and graph (right) representation for an example CQ $Q(A, C, F, K) : -R_1(A, B, C), R_2(A, H), R_3(B, E, F), R_4(E, K), R_5(K, I), R_6(C, I, J)$.

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 $A \in \texttt{attr}(R_i)$, $\pi_A t \in \texttt{dom}(A)$ 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

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

where $\boldsymbol{A} \subseteq \mathbb{A}$ denotes the *output attributes* and $\mathbb{A} - \boldsymbol{A}$ 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 $\boldsymbol{A} = \mathbb{A}$, such a CQ query is known as *full CQ* which represents the natural join among the given relations. If $\boldsymbol{A} = \emptyset$, 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 3.1. *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 [25] 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. The formal proof is given in the full version [13].

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 gives a decidability result of the ADP problem on boolean CQ:

Theorem 3.2 ([10]). *On a boolean CQ Q, the poly-time solvability (in data complexity) of the* $\mathtt{ADP}(Q, D, 1)$ *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 3.3. *For a CQ Q, if there exists some vacuum relation, the* $\mathtt{ADP}(Q, D, k)$ *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 $\mathtt{ADP}(Q, D, k)$ 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 [13].

4 POLY-TIME DECIDABILITY

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

Theorem 4.1. *On a CQ Q, IsPtime(Q) can decide poly-time solvability of the* $\mathtt{ADP}(Q, D, k)$ *problem, which runs in polynomial time.*

The procedure IsPtime(Q) is illustrated in Figure 3. Note that when IsPtime(Q) returns `true`, the $\mathtt{ADP}(Q, D, k)$ problem is poly-time solvable, and NP-hard otherwise. The algorithmic description of IsPtime is given in full version [13]. 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 3.2 and Lemma 3.3.

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 4.2. *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* $\mathtt{ADP}(Q, D, k)$ *is* NP-hard.

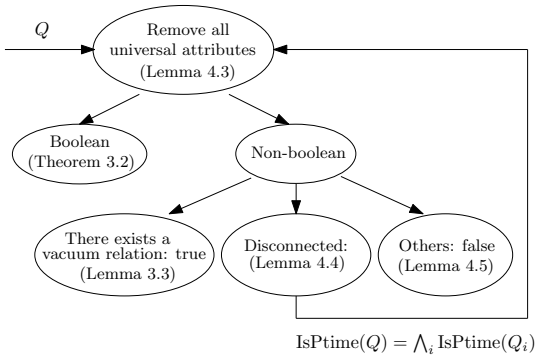

Figure 3: Procedure IsPtime(Q).

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 4.3 and Lemma 4.4. 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 4.1 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 3.2 and Lemma 3.3. 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 4.5. Putting everything together, the correctness for Theorem 4.1 then follows from induction over the size of the query.

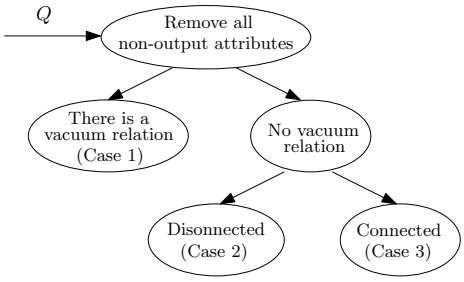

Figure 4: Proof plan of Lemma 4.5.

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

Lemma 4.4. *Let $Q_1, Q_2, \cdots, Q_s$ be the connected subqueries of Q for $s \geq 2$. The* $\mathtt{ADP}(Q, D, k)$ *problem is NP-hard if and only if there exists some Q_i for which the* $\mathtt{ADP}(Q_i, D, k)$ *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. The detailed proofs of these lemmas are deferred to the full version [13].

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 4.5.

Lemma 4.5. *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* $\mathtt{ADP}(Q, D, k)$ *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 4.5 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 as follows:

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

Careful inspection reveals that these queries have a common property: w.l.o.g., we can assume that an optimal solution of $\mathtt{ADP}(Q, D, k)$ 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 $(b) \in R_3$, which follows immediately from the notion of "domination" in [10]. 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 4.6).

Lemma 4.6. *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 hardness of problem (2) and (3) is established from the *k-minimum coverage* (KMC) problem and the *clique in regular graph* problem, with detailed proofs in [13].

4.2.2 Hardness Preserving Mapping. The high-level idea of relating an arbitrary query Q characterized by Lemma 4.5 to the core queries is to divide the attributes in $\mathtt{attr}(Q)$ 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 4.7 (Query Mapping). *Suppose we are given a function $f : \mathtt{attr}(Q_1) \to \mathtt{attr}(Q_2) \cup \{*\}$. Let*

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

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

In the definition above, if $g(R_i) = \mathtt{attr}(R_j)$ for relations $R_i \in \mathtt{rels}(Q_1)$ and $R_j \in \mathtt{rels}(Q_2)$, then $R_i \in \mathtt{rels}(Q_1)$ is said to be *mapped* to relation $R_j \in \mathtt{rels}(Q_2)$. The next lemma, whose proof is deferred to the full version [13], shows that query mappings preserve hardness of the ADP problem.

Lemma 4.8. *If there is a mapping from a CQ Q_1 to another CQ Q_2, and* $\mathtt{ADP}(Q_2, D, k)$ *is NP-hard, then* $\mathtt{ADP}(Q_1, D, k)$ *is also NP-hard.*

4.2.3 Mapping to the core. To prove the NP-hardness of the ADP problem on a query Q, it suffices to show a mapping to any core query, implied by Lemma 4.8. The high-level idea is that for any query characterized by Lemma 4.5, we find a partition of attributes

[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.

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. The mapping constructed for each case with examples are given in [13].

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 [10]. 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 5.1. *For a CQ Q, $\texttt{ADP}(Q, k, D)$ 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 5.1 and give some intuition for why they make the ADP problem NP-hard. The proof of Theorem 5.1 is given in [13].

5.1 Boolean CQ Revisited

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

Theorem 5.2 ([10]). *On a boolean CQ Q without self-joins, the problem $\texttt{ADP}(Q, D, 1)$ 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:

Definition 5.3 (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 $\texttt{attr}(Q) - \texttt{attr}(R_3)$.*

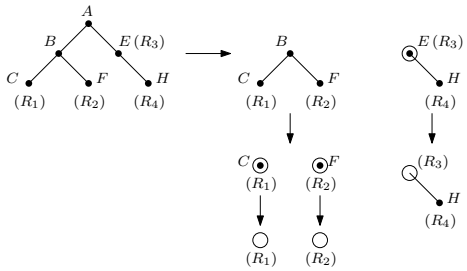

Figure 5: An example of hierarchical join $Q(A, B, C, E, F, H) :- R_1(A, B, C), R_2(A, B, F), R_3(A, E), R_4(A, E, H)$, and an illustration of applying procedure IsPtime on it.

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 first 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 5.4 (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 5.5 (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 5.6 (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 5.7. *For a full CQ Q, the $\texttt{ADP}(Q,D,k)$ 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 5.8 (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 [10], as we need a more fine-grained characterization of exogenous relations for ADP. Moreover, Lemma 3.3 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 $\texttt{ADP}(Q,D,k)$ is poly-time solvable by Theorem 5.1.

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 5.9 (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_ℓ 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_ℓ, 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.

5.3 Sketch of Proof of Theorem 5.1

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 5.1; the detailed proof is in the full version [13]. This proof uses Theorem 4.1 by mapping each of the NP-hard cases in Theorem 4.1 to the existence of a hard structure as defined by Theorem 5.1, and vice-versa. But, interestingly, this mapping is not one-one in the sense that multiple cases in the procedural dichotomy established by Theorem 4.1 map to same case in the structural dichotomy of Theorem 5.1, 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 5.10. *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 5.11. *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

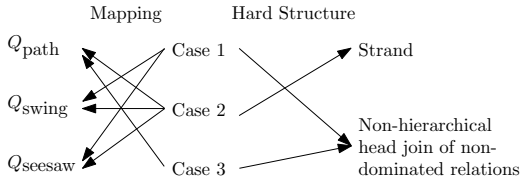

Figure 6: Correspondence between the three cases of hard query on which IsPtime falling into "other" bucket in Figure 4, the core query it maps to (the left) and the hard structure it contains (the right).

join of non-dominated relations in Q_{path} is non-hierarchical, and both Q_{swing} and Q_{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 $\texttt{ADP}(Q, D, k)$ 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 6.1. *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 $\texttt{ADP}(Q, D, k)$, 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 the $\texttt{ADP}(Q, D, k)$ instance, then every element belongs to at most p sets. (A formal description of this reduction and approximation-preserving property are left to the full version [13].)

It is known that the PSC problem admits greedy and primal-dual algorithms with approximation factors of $O(\log k)$ and p respectively [12]. Hence, we get the same results for the ADP problem.

Theorem 6.2. *For a full CQ Q with p relations, any instance D and integer k, $\texttt{ADP}(Q, k, D)$ admits $O(\log k)$ and p-approximations.*

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:

Algorithm 1: 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);

Lemma 6.3. *Under some mild cryptographic assumptions, the $\texttt{ADP}(Q_{\text{swing}}, D, k)$ 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 $\texttt{ADP}(Q_{\text{swing}}, D, k)$ via a reduction from the *k-minimum coverage* (KMC) problem. As shown in the full version [13], this reduction is also approximation-preserving, which implies the above lemma via known hardness results for the KMC problem [1, 5, 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 1. It builds upon the algorithm for the Resilience problem [10], 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 4.1 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 1 also generates a feasible 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. All omitted proofs and pseudocodes are in [13].

7.1 Singleton

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

Definition 7.1 (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 CQ or contains vacuum relation, and each internal node corresponds to one simplification step. On this recursion tree, we observe that for a poly-time solvable CQ Q, each leaf (not root) node containing a vacuum relation must have an ancestor that is a singleton query. So, it suffices to replace the vacuum relation base case with the singleton. The detailed proof, algorithm, and pseudocode are given in [13].

7.2 Universe and Decompose

Decompose(Q, D, k). Assume Q is disconnected, with$s$ connected subqueries $Q_1, Q_2, \cdots, Q_s$. The divide-and-conquer strategy will first compute a subproblem $\texttt{ADP}(Q_i, D, k_i)$ 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 $\textsc{Opt}[i][j]$ denote the minimum number of input tuples that can remove at least j output tuples from subquery $\times_{j=1}^{i} Q_j(D)$, which 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 1 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 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 $\texttt{ADP}(Q_i, D, k_2)$ 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 can 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 1 is revoked for solving the $\texttt{ADP}(Q, D_i, m)$ 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 $\texttt{ADP}(Q, D_i, j)$ 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.

7.3 Greedy Heuristics

GreedyForCQ(Q, D, k): For many simple queries, the ADP problem is NP-hard, and even hard to approximate implied by the results in Section 6. The prime-dual approximation algorithm [12] 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 (pseudocode is in [13]). 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 no theoretical guarantees on the approximation ratio when projection exists.

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, pseudocode in [13]). 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 is not equivalent to the profit of their union.

7.4 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

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

where θ_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 θ_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 7.2, whose proof is in [13].

Lemma 7.2. *For a CQ Q and selection predicates θ, the* $\mathsf{ADP}(Q, D, k)$ *is NP-hard if and only if* $\mathsf{ADP}(Q_{-\mathbb{A}_\theta}, D, k)$ *is NP-hard, where $Q_{-\mathbb{A}_\theta}$ is the residual query after removing selected attributes $\mathbb{A}_\theta$ from Q.*

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.

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 ρ% trading records can be restricted. (2) The same query but for the specific PartKey = 13370.* They can be characterized by two problems $\mathsf{ADP}(Q_1, D, k)$ and $\mathsf{ADP}(\sigma_\theta Q_1, D, k_\theta)$ 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 ρ fraction of outputs are removed.

As shown in Lemma 7.2, the $\mathsf{ADP}(\sigma_\theta Q_1, D, k)$ is poly-time solvable with exact optimal solution returned, while the $\mathsf{ADP}(Q_1, D, k)$ 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 for them.

8.2 Scalability

Poly-time query: We evaluate $\mathsf{ADP}(\sigma_\theta Q_1, D, k_\theta)$ 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 ρ. 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 1), 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 $\mathsf{ADP}(Q_1, D, k_\theta)$ 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 $\mathsf{ADP}(Q_1, D, k)$ 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 $\rho = 0.2$.

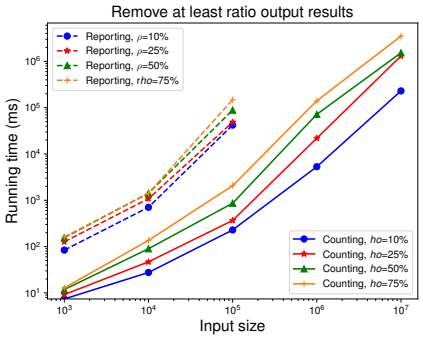

Figure 7: Running Time: $\sigma_\theta Q_1$ (easy) exactly (count/report).

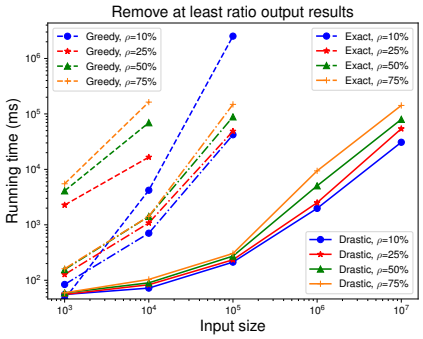

Figure 8: Running Time: reporting $\sigma_\theta Q_1$ (easy) by heuristics.

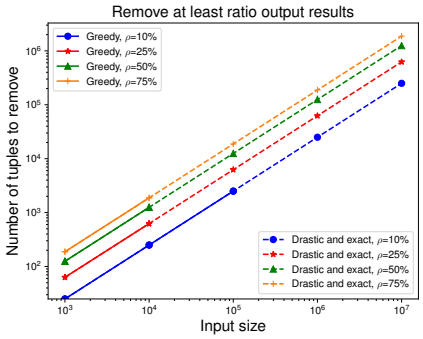

Figure 9: Quality: $\sigma_\theta Q_1$ (easy) by heuristics.

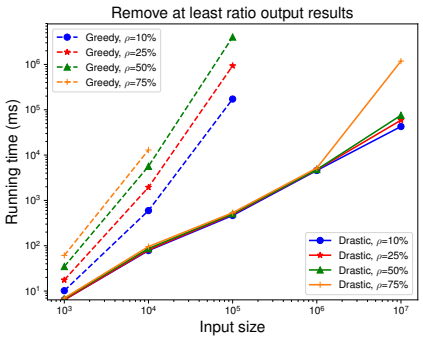

Figure 10: Running Time: reporting Q_1 (hard) by heuristics.

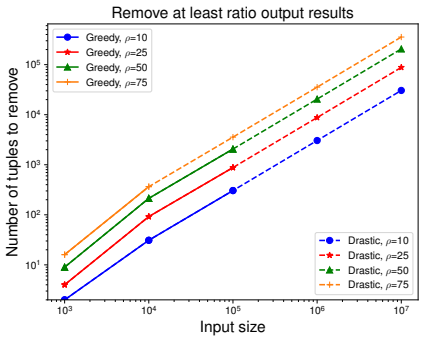

Figure 11: Quality: Q_1 (hard) by heuristics.

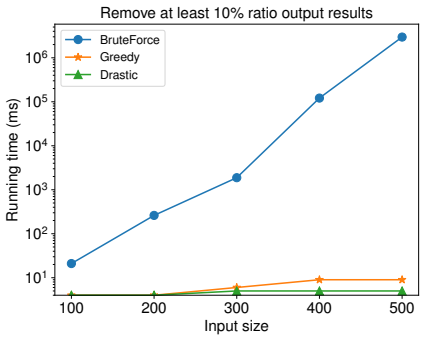

Figure 12: Running Time: brute-force v.s. heuristics for Q_1 (hard).

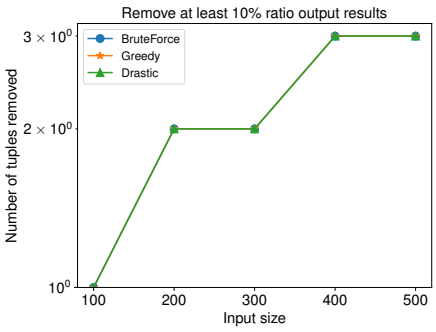

Figure 13: Quality: brute-force v.s. heuristics for Q_1 (hard).

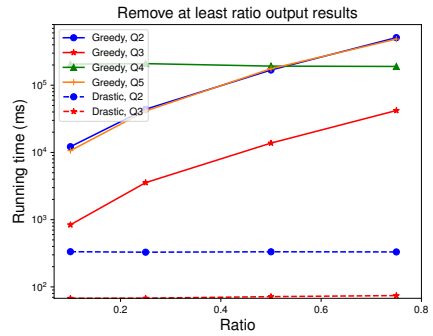

Figure 14: Running Time: Q_2, Q_3, Q_4, Q_5 (hard) by heuristics.

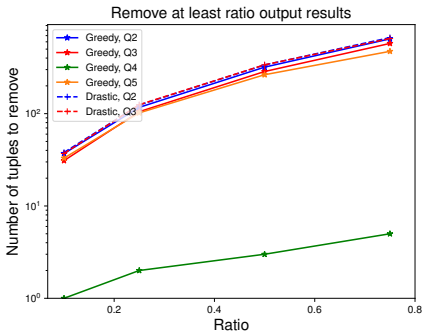

Figure 15: Quality: Q_2, Q_3, Q_4, Q_5 (hard) by heuristics.

8.3 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 ρ) 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.

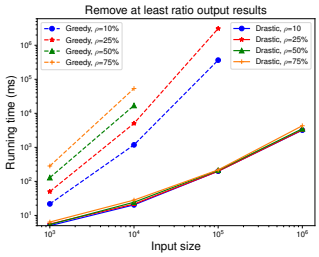

Figure 16: $\alpha = 0$ **(hard)**

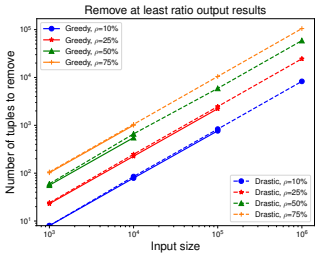

Figure 17: $\alpha = 0$ **(hard)**

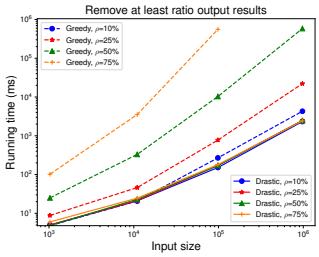

Figure 18: $\alpha = 1$ **(hard)**

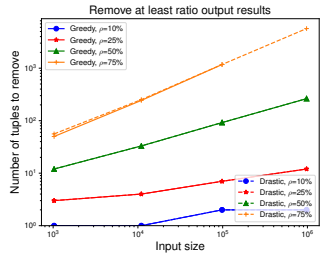

Figure 19: $\alpha = 1$ **(hard)**

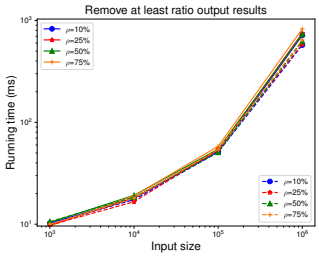

Figure 20: $\alpha = 0$ **(easy)**

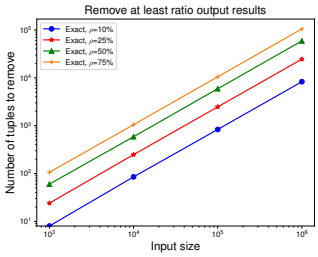

Figure 21: $\alpha = 0$ **(easy)**

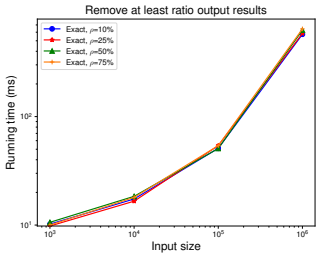

Figure 22: $\alpha = 1$ **(easy)**

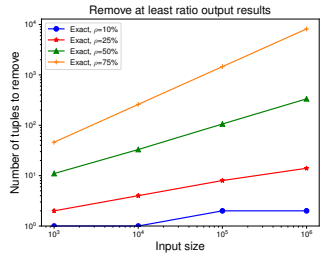

Figure 23: $\alpha = 1$ **(easy)**

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 α 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 α. We evaluate both Q_6 and Q_{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_{path} are shown in Figure 16–19, and those for Q_6 are shown in Figure 20–23. We also tested other values of α, which are reported in the full version [13].

For every fixed value of α, the running time as well as the size of solutions returned by any algorithm increase with the input size and the value of ρ. If both the input size and ρ are fixed, the size of the solution decreases with increasing α. 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 α.

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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] X. Hu, S. Patwa, S. Sun, D. Panigrahi, and S. Roy. Aggregated deletion propagation for counting conjunctive query answers. *https://arxiv.org/pdf/2010.08694.pdf*, 2020.

[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] 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.

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

[22] 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.

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

[24] 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.

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

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