Size Bound–Adorned Datalog

CHRISTIAN FATTEBERT, EPFL, Switzerland
ZHEKAI JIANG, EPFL, Switzerland
CHRISTOPH KOCH, EPFL, Switzerland
REINHARD PICHLER, TU Wien, Austria
QICHEN WANG, Nanyang Technological University, Singapore

We introduce EDB-bounded datalog, a framework for deriving upper bounds on intermediate result sizes and the asymptotic complexity of recursive queries in datalog. We present an algorithm that, given an arbitrary datalog program, constructs an EDB-bounded datalog program in which every rule is adorned with a (non-recursive) conjunctive query that subsumes the result of the rule, thus acting as an upper bound. From such adornments, we define a notion of width based on (integral or fractional) edge-cover widths. Through the adornments and the width measure, we obtain, for every IDB predicate, worst-case upper bounds on their sizes, which are polynomial in the input data size, given a fixed program structure. Furthermore, with these size bounds, we also derive fixed-parameter tractable, output-sensitive asymptotic complexity bounds for evaluating the entire program. Additionally, by adapting our framework, we obtain a semi-decision procedure for datalog boundedness that efficiently rewrites most practical bounded programs into non-recursive equivalent programs.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**.

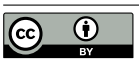
Additional Key Words and Phrases: Datalog, Cost-based query optimization, Result size estimation

ACM Reference Format:
Christian Fattebert, Zhekai Jiang, Christoph Koch, Reinhard Pichler, and Qichen Wang. 2026. Size Bound–Adorned Datalog. *Proc. ACM Manag. Data* 4, 2 (PODS), Article 97 (May 2026), 27 pages. https://doi.org/10.1145/3801893

1 Introduction

Datalog engines [6, 7, 20, 22, 24, 41, 48] are starting to show their potential in a wide range of data-intensive applications, from static program analysis [1, 30, 36, 39, 45] to large-scale graph processing [6, 9, 26, 34, 37]. With ever-growing data sizes, cost-based optimization is essential for fast, scalable execution in such engines. However, accurate cost-based optimization relies heavily on the ability to estimate the cardinalities of intermediate results generated during query evaluation [23]. In traditional join processing, there have been significant research efforts on cardinality estimation [4, 8, 11, 15, 47], cost-based optimization [27, 33, 40], and theoretical complexity bounds [12, 16, 21, 43, 46]. But estimating result sizes is significantly more challenging when recursion is involved, which is an inherent feature of datalog. This is also the case for recursive SQL [14], which is starting to gain popularity [10, 31, 32]. In particular, there is currently no reliable method for computing size bounds on the final outputs of recursively-defined relations, except for a few very specific datalog programs [35]. Without such bounds, optimizers are often forced to

Authors' Contact Information: Christian Fattebert, EPFL, Lausanne, Vaud, Switzerland, christian.fattebert@epfl.ch; Zhekai Jiang, EPFL, Lausanne, Vaud, Switzerland, zhekai.jiang@epfl.ch; Christoph Koch, EPFL, Lausanne, Vaud, Switzerland, christoph.koch@epfl.ch; Reinhard Pichler, TU Wien, Vienna, Austria, pichler@dbai.tuwien.ac.at; Qichen Wang, Nanyang Technological University, Singapore, Singapore, qichen.wang@ntu.edu.sg.



ACM 2836-6573/2026/5-ART97
https://doi.org/10.1145/3801893

rely on (1) heuristics, as is the case for the popular datalog engine Soufflé [5, 22], or (2) runtime optimization techniques [17, 18]. Such methods often lead to suboptimal execution plans or higher runtime overhead for complex workloads.

The present paper aims to improve this situation by developing techniques for computing (asymptotic) bounds on the result sizes of recursive datalog programs. To this end, we introduce a framework for adorning datalog rules (and specifically IDB atoms) with conjunctive queries over the input database, the union of which subsumes the result of the IDB predicate, thus acting as a form of upper bound. We refer to such adorned programs as *EDB-bounded datalog programs*. We give a simple illustration with a variant of the well-known transitive closure program:

Example 1.1. Consider the datalog program Π with the following rules:

$$R_1 : \mathsf{TC}(\bar{x}, \bar{y}) \leftarrow e(\bar{x}, \bar{y}),$$
$$R_2 : \mathsf{TC}(\bar{x}, \bar{y}) \leftarrow \mathsf{TC}(\bar{x}, \bar{z}), e(\bar{z}, \bar{y}),$$

where e is an EDB relation, each of $\bar{x}$, $\bar{y}$, and $\bar{z}$ is a vector of m variables, and e and TC hence have arity $2m$. This program is *unbounded*, as the depth of recursion depends on the size of the input. Therefore, there does not exist a finite non-recursive program (or, finite union of conjunctive queries (UCQ)) that is equivalent to Π. However, assuming that the number of tuples in e is N, we can in fact see that the number of tuples in the final output of TC is bounded by N^2, even though the arity $2m$ may be greater than 2. This is because, at maximum, we choose (1) one tuple from e and assign the first m attributes to $\bar{x}$, and (2) one tuple from e and assign the last m attributes to $\bar{y}$.

In this paper, we generalize this idea by deriving size bounds based on a notion of width analogous to edge cover numbers in conjunctive queries. For example, following our method, the program Π above would be rewritten to the following program Π':

$$\begin{aligned}
R'_1 :&\quad \mathsf{TC}_{\mathsf{TC}(\bar{x},\bar{y})\leftarrow e(\bar{x},\bar{y})}(\bar{x},\bar{y}) && \leftarrow e(\bar{x},\bar{y}),\\
R'_2 :&\quad \mathsf{TC}_{\mathsf{TC}(\bar{x},\bar{y})\leftarrow e(\bar{x},_),e(_,\bar{y})}(\bar{x},\bar{y}) \leftarrow \mathsf{TC}_{\mathsf{TC}(\bar{x},\bar{y})\leftarrow e(\bar{x},\bar{y})}(\bar{x},\bar{z}), && e(\bar{z},\bar{y}),\\
R'_3 :&\quad \mathsf{TC}_{\mathsf{TC}(\bar{x},\bar{y})\leftarrow e(\bar{x},_),e(_,\bar{y})}(\bar{x},\bar{y}) \leftarrow \mathsf{TC}_{\mathsf{TC}(\bar{x},\bar{y})\leftarrow e(\bar{x},_),e(_,\bar{y})}(\bar{x},\bar{z}), e(\bar{z},\bar{y}).
\end{aligned}$$

In Π', each atom of the IDB predicate TC is adorned by a conjunctive query which shows the "provenance" of the tuples involved. More importantly, the rule in the adornment of the head atom of each rule subsumes the results that will be produced by the rule. Therefore, Π' is equivalent to Π in the sense that the tuples that will be in the final output of the IDB relation TC in the original program Π is exactly the same as the union of the two adorned variants of TC in Π', namely $\mathsf{TC}_{\mathsf{TC}(\bar{x},\bar{y})\leftarrow e(\bar{x},\bar{y})}$ and $\mathsf{TC}_{\mathsf{TC}(\bar{x},\bar{y})\leftarrow e(\bar{x},\_),e(\_,\bar{y})}$. In such a case, we say the edge-cover width of TC is 2, and we can then easily tell that the size of TC is bounded by N^2. With such bounds on the output sizes, we can further derive output-sensitive, fixed-parameter tractable asymptotic complexity bounds for evaluating the entire program, which, in this case of TC, is $O(f(\Pi) \cdot N^2)$, where f is a function that depends only on the structure of the program Π.

Our approach is thus based on rewriting—or, more precisely, adorning—datalog programs with non-recursive queries that serve as upper bounds on the IDB relations defined by the program. While a datalog engine might use such bounding queries for cardinality estimation, in this paper, we focus on proving asymptotic bounds based on structural properties of the datalog programs—specifically, integral and fractional edge-cover width.

As a central result, we show that our program rewriting technique can be used to find the minimal integral edge-cover width (which immediately yields an asymptotic bound on output sizes) across all datalog programs equivalent to a given input program. In other words, determining the worst-case sizes of the IDB relations of a datalog program, or any datalog program equivalent to

it, is decidable. This may appear somewhat surprising, as semantic properties such as (program) boundedness or equivalence of recursive datalog programs are generally undecidable [19, 38].

We demonstrate the tightness of integral edge-cover widths and our size bounds by constructing database instances where they are reached.

We show that output sizes of IDB relations of a program Π have worst-case upper bounds of the form $f(\Pi) \cdot N^w$, which is polynomial in the input data size N for a fixed structure of Π.

Based on our size bounding machinery, we obtain corresponding output-sensitive, fixed-parameter tractable complexity bounds for evaluating entire programs. These results are based on insights into incrementally evaluating our rewritten programs. We present results on general datalog as well as tighter bounds on a number of datalog fragments—linear, simple chain, and adornment-groundable programs (a newly defined class).

Our adornment rewriting algorithm is fundamentally a bottom-up fixpoint procedure that is guaranteed to terminate given suitable methods for comparing and relaxing (simplifying) adornments. A variation of this procedure unrolls recursive datalog programs into (possibly infinite) UCQs, and terminates if the program is bounded—a semi-decision procedure for datalog boundedness. We show in this paper how our adornment algorithm can be instrumented to search for non-recursive rewritings (for bounded programs) with a memory budget, gracefully degrading to our EDB-bounded datalog programs (as in Example 1.1) when the input program is not bounded or cannot be proven to be within the memory budget.

The structure of the paper is as follows:

- We start by laying out the necessary background in Section 2.
- In Section 3, we present our technique and algorithm for adorning IDB atoms in a datalog program with conjunctive queries, and show its correctness (conservativity). We present this approach as a generic framework that may be adapted for different use cases.
- In Section 4, we revisit the problem of datalog boundedness. We show its semi-decidability, and give one instantiation of our framework to either (1) rewrite most practical bounded programs into equivalent UCQs by inlining, or (2) give an equivalent EDB-bounded program that is recursive but still allows for reasoning about size bounds and complexity.
- In Section 5, we obtain size bounds on IDB relations via a notion of edge-cover widths based on the adornments in EDB-bounded programs.
- We discuss the optimization and minimization of our program rewritings in Section 6.
- Section 7 presents our fixed-parameter tractable complexity bounds for general datalog as well as tighter results for specific fragments of datalog.

Due to space constraints, all omitted proofs in the rest of the paper can be found in the full version of the paper [13].

2 Preliminaries

Datalog. In this paper, we consider standard positive datalog programs and adopt the standard naming conventions. A datalog program Π consists of a set of *rules* of the form

$$q(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_\mu(\bar{s}_\mu), \quad \text{or, equivalently,} \quad q(\bar{s}) \leftarrow \bigwedge_{i=1}^{\mu} p_i(\bar{s}_i),$$

where $q(\bar{s})$ is called the *head atom* and all $p_i(\bar{s}_i)$ are *body atoms*. Each atom $p(\bar{t})$ consists of a predicate symbol p and a tuple of terms $\bar{t} = (t_1, \ldots, t_\alpha)$. We use the same set of symbols, such as $p, q, \ldots$, to denote both a predicate symbol and the relation corresponding to that predicate. A predicate p belongs to the intensional database (IDB) if it is in the head of a rule with a non-empty body. Otherwise, it belongs to the extensional database (EDB). Without loss of generality, we

assume that each rule is ordered so that the first λ atoms (for some $0 \leq \lambda \leq \mu$) have IDB predicates and the rest have EDB predicates. Within a tuple $\bar{t}$ of an atom $p(\bar{t})$, each term t_i is either a variable or a constant. We write $\text{vars}(\bar{t})$ to denote the set of variables occurring in $\bar{t}$. The wildcard "_" is employed to denote a freshly introduced, existentially quantified variable. The number of terms α is called the *arity* of the predicate p, generally denoted by $\text{ar}(p)$. A datalog rule is defined as *safe* if every variable present in the head also appears at least once in the body. The notations #EDBs and #IDBs represent the total number of EDB predicates and IDB predicates, respectively, within a datalog program. Given a datalog program Π, its number of rules is denoted by $|\Pi|$.

For an IDB predicate q in Π and a database instance D, we write $[\![q(\Pi, D)]\!]$ to denote the IDB relation q that can be derived with the program Π from the database instance D. That is, $[\![q(\Pi, D)]\!]$ contains all tuples $\bar{s}$, such that the fact $q(\bar{s})$ is contained in the least fixpoint of the immediate consequence operator defined by the program Π, when starting with the EDB D.

We say a datalog rule R_0 is *subsumed* by a rule R_1, if there is a homomorphism h from R_1 to R_0, such that h restricted to the head variables is a variable renaming. That is, all the tuples produced by rule R_0 are also produced by rule R_1.

We use *inlining* to refer to the process of replacing an IDB atom $q(\bar{t})$ in some rule R with the body of a rule $R' : q(\bar{s}) \leftarrow B_1, \ldots, B_n$ that has q as the head predicate, under a substitution σ such that $\sigma(\bar{s}) = \bar{t}$, which yields instantiated body atoms $\sigma(B_1), \ldots \sigma(B_n)$, where σ must not rename any variable that is not in the head $\bar{s}$ to the name of some variable in the original rule R. For instance, in the following program

$$R_1 : p_0(x, y) \leftarrow e(x, z), e(z, y),$$
$$R_2 : \ q(x, y) \leftarrow p_0(x, y), p_1(z),$$

we can obtain $q(x, y) \leftarrow e(x, z_0), e(z_0, y), p_1(z)$ by one step of inlining. Note that, while doing so, z in R_1 is renamed to z_0 to avoid collision with the variable z in the original rule R_2.

Unification. A *unifier* of two tuples $\bar{s}$ and $\bar{t}$ of terms is a substitution σ such that, when applied to the tuples, σ makes them syntactically equal, i.e., $\sigma(\bar{s}) = \sigma(\bar{t})$. When such a unifier exists, we say that the two tuples unify. Given a set of pairs of tuples $\{(\bar{s}_i, \bar{t}_i) \mid 1 \leq i \leq \mu\}$, a *simultaneous unifier* is a unifier such that $\sigma(\bar{s}_i) = \sigma(\bar{t}_i)$ for all i. A unifier is called a *most general unifier* (mgu) if every other unifier can be obtained from it by further instantiating variables. mgus can be computed in time linear in the size of the input set [25]. For instance, $\sigma = \{x \mapsto x, y \mapsto y, z \mapsto z, w \mapsto z\}$ is a most general unifier for the tuples (x, y, z, z) and (x, y, z, w).

Proof trees. Given a datalog program Π and an EDB D, a *proof tree* for a ground atom A in the IDB is a finite, rooted tree whose root is labeled by A, whose leaves are EDB atoms in D, and whose edges correspond to rule applications. More specifically, if a node is labeled by an atom A and its children are labeled $B_1, \ldots, B_\mu$, then the edge from A to each B_i is labeled by a rule R of the form $A' \leftarrow B'_1, \ldots, B'_\mu$, such that $A = \sigma(A')$ and $B_i = \sigma(B'_i)$ for some *ground substitution* σ, i.e., a substitution that replaces every variable with a constant.

Derived atoms. Given a rule R with head predicate q in a datalog program Π, and given an EDB D, we say that a ground atom $q(\bar{t})$ is derived by rule R over D if there exists some proof tree whose root is labeled by $q(\bar{t})$ and such that the edges from the root are labeled by the rule R.

3 EDB-Adorned and EDB-Bounded Datalog Programs

In this section, we begin to introduce our framework to provide bounds for datalog programs via adornments added to IDB atoms. These adornments themselves are datalog rules with EDB atoms only, and they subsume the adorned IDB predicate. We formally define the related notations

and give a generic algorithm to construct such adornments. The algorithm is parameterized and customizable for different use cases, as we will show in the following sections.

3.1 Basic Definitions

We start from the most basic building blocks of our framework.

Definition 3.1 (EDB-adorned atom). An *EDB-adorned atom* is an atom of the form $q_\rho(\bar{t})$, such that q is an IDB predicate of the given schema of arity α, $\bar{t} = (t_1, \ldots, t_\alpha)$ is a tuple of the same arity, such that each t_i is either a variable or a constant, and the *adornment* ρ is a safe datalog rule $A \leftarrow B_1, \ldots, B_\mu$ satisfying the following properties:

(1) A is an atom with predicate symbol q,
(2) every body atom B_i has as an EDB predicate and each argument in B_i is either a variable or the underscore "_".
(3) $q(\bar{t})$ is an instance of A—that is, there exists a substitution σ such that $\sigma(A) = q(\bar{t})$.

Definition 3.2 (EDB-adorned datalog rule). An *EDB-adorned datalog rule* R is a datalog rule in which the head and every body atom is either an EDB atom or an EDB-adorned IDB atom. A datalog program with only EDB-adorned rules is called an *EDB-adorned datalog program.*

The *intended meaning* of the adornment ρ of an EDB-adorned IDB atom $q_\rho(\bar{t})$ is to indicate how instances of the atom $q_\rho(\bar{t})$ could be directly derived from EDB atoms via ρ. The rule ρ thus corresponds to iterated inlining, and the EDB atoms in the rule body of an adornment constitute restrictions on the tuples that can possibly be derived by the adorned IDB predicate. Of course, such a restriction is useless in the case of an EDB atom. We thus only consider adornments for IDB predicates. We formalize this idea by the following definitions of EDB-bounded rules and programs.

Definition 3.3 (EDB-bounded rule). Given an EDB-adorned datalog program Π, let R be a rule in Π with head predicate q_ρ. We say that rule R *is bounded* by the adornment ρ, if for every EDB D, the following property holds: If an atom $q_\rho(\bar{t})$ is derived by rule R when evaluating Π over D, then $q(\bar{t})$ is also derived when evaluating the single-rule program consisting of the rule ρ over D.

Definition 3.4 (EDB-bounded program). Given an EDB-adorned datalog program Π, we call Π an *EDB-bounded* program if every rule of Π is bounded by the adornment of its head predicate.

To give an intuition, we illustrate again with the transitive closure program Π in Example 1.1.

Example 3.5. Consider the transitive closure program Π in Example 1.1 again. For simplicity, from this point, we assume $\bar{x}$, $\bar{y}$, and $\bar{z}$ each contain one term only and hence drop the bars. The following datalog program Π'' is a valid EDB-bounded program:

$$R_1'' : \quad \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,y)}(x,y) \qquad \leftarrow e(x,y),$$
$$R_2'' : \quad \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,z),e(z,y)}(x,y) \leftarrow \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,y)}(x,z), e(z,y).$$

One can easily verify that the adornment of R_1'', $\mathsf{TC}(x,y) \leftarrow e(x,y)$, subsumes the rule R_1''—in fact, they are equivalent. The predicate $\mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,y)}$ contains only tuples that are produced by R_1'', i.e., the pairs (x,y) that are connected via some edge in e. Similarly, the adornment of R_2'' subsumes, and is equivalent to, the rule R_2'' as well, if we replace the atom $\mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,y)}(x,z)$ by $e(x,y)$. Notice that the adornments in this case simply correspond to inlinings of the original program. We could do one step further and add the following rule

$$R_3'' : \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,z'),e(z',z),e(z,y)}(x,y) \leftarrow \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,z),e(z,y)}(x,z), e(z,y).$$

Again, this is a valid EDB-bounded rule. However, these adorned versions of the predicate TC in Π'' only produce pairs of x and y with distance at most 3, but the original program Π is unbounded.

Hence, Π'' is not equivalent to Π (unless we iterate this inlining-style rewriting of rules infinitely). However, in the next section, we will introduce means to get meaningful, finite adorned programs that allow us to bound the IDB predicates of the original program.

3.2 Construction of EDB-Bounded Datalog Programs

In Example 3.5, we have seen an intuition of EDB adornments by viewing them as possible inlinings of the original program and how this may become infinite if we want to bound all tuples produced by the original program. It is easy to verify that program Π' in Example 1.1 is also an EDB-bounded program. In contrast to program Π'', the adornments of the predicate TC in Π' actually do allow us to bound the IDB relation TC of the original program Π: Intuitively, the adornments of TC in Π' capture the observation that a tuple (x, y) generated by TC in program Π could either be (1) two endpoints of the same edge, hence $e(x, y)$, or (2) the source of one edge and target of another, hence $e(x, \_), e(\_, y)$. Alternatively, we can consider these adornments as taking all possible inlinings, but "relaxing" the rules by replacing variables with "_" or dropping body atoms. We generalize this idea in the following definition of *adornment relaxation functions*.

Definition 3.6 (adornment relaxation function). An *adornment relaxation function* g is one that takes as input an adornment $\rho : p(\bar{t}) \leftarrow e_1(\bar{t}_1), \ldots, e_\mu(\bar{t}_\mu)$, where all body atoms $e_1(\bar{t}_1), \ldots, e_\mu(\bar{t}_\mu)$ are EDB atoms, and performs a sequence of operations, each of which either removes a body atom or replaces an attribute in a body atom by "_", such that $g(\rho)$ remains a safe datalog rule.

Clearly, if a rule $g(\rho)$ results from "relaxing" a rule ρ, then $g(\rho)$ subsumes ρ:

Proposition 3.7. *For any adornment relaxation function g and safe datalog rule ρ with only EDB atoms in the body, $g(\rho)$ subsumes ρ.*

Example 3.8. We show a few examples of adornment relaxation functions:

- *id*: The identity function, which does not modify the rule at all.
- g_{out}: For each body atom $e(\bar{t})$ and for each attribute t_i in $\bar{t}$, we replace t_i by _ if t_i does not occur in the head of the rule (i.e., if t_i is not in the "output"). Finally, we remove all body atoms $e(\bar{t})$ that impose less restrictions than some other body atom $e(\bar{t}')$. That is, for each $i = 1, \ldots, \text{ar}(e)$, either $t_i = t'_i$ or $t_i = \_$. Note that, in particular, this removes all body atoms $e(\bar{t})$ that have only "_"s remaining in $\bar{t}$. Example 3.9 illustrates the application of this function, and this function will be used to derive edge-cover widths and size bounds in Sections 5 and 7.
- g_k: A function that acts as the identity function when the input rule contains fewer than k atoms and applies g_{out} otherwise. This function will be used in Section 4 to try to rewrite a bounded program into a non-recursive equivalent version with some storage budget.
- g_{min}: This function retains only a minimal subset of body atoms from the original rule such that all head variables are contained within the set. Then, similar to g_{out}, it replaces all variables that are not in the head by "_"s. Furthermore, any duplicate variable occurrences in the body are replaced by "_", so that each head variable appears only once in the resulting rule. This function will be used in Section 6 to reduce the number of rules in the transformed program.

Intuitively, different adornment relaxation functions have different levels of relaxation. Moreover, in combination with the inlining-style adornments in Example 3.5, this notion of adornment relaxation function suggests a way in which adornments can be propagated algorithmically.

Example 3.9. Consider the transitive closure program Π from Example 1.1 and suppose that we generate adorned rules as in Example 3.5. But now we apply the adornment relaxation function g_{out} to every rule thus generated. We initialize $\Pi' = \emptyset$. The first rule R_1 in Π gives rise to the following

EDB-adorned rule which we add to Π':

$$R_1' : \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,y)}(x,y) \leftarrow e(x,y).$$

Next, consider rule R_2 in Π, but replace the IDB atom in the body of R_2 by the adorned head atom of the rule $R_1' \in \Pi'$. Moreover, to get the adorned rule R_2', we apply g_{out} to the adornment $\mathsf{TC}(x,y) \leftarrow e(x,z), e(z,y)$, replacing z (which is not a head variable) by _. We thus obtain the rule

$$R_2' : \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,\_),e(\_,y)}(x,y) \leftarrow \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,y)}(x,z), e(z,y),$$

which we add to Π'.

We can now add the adornment $\mathsf{TC}(x,y) \leftarrow e(x,\_), e(\_,y)$ to the IDB predicate in the rule body of R_2. That is, we assume that, when executing program Π' on a database D, the body atom will be generated by firing rule R_2'. Hence, in the first place, we have to rename the variables in R_2' apart from the variables in R_2. However, we then need to unify the head of R_2' with the body atom in R_2. Then, inlining results in an adornment of the form $\mathsf{TC}(x,y) \leftarrow e(x,\_), e(\_,u), e(u,y)$, which we transform into $\mathsf{TC}(x,y) \leftarrow e(x,\_), e(\_,\_), e(\_,y)$ and further into $\mathsf{TC}(x,y) \leftarrow e(x,\_), e(\_,y)$ by applying g_{out}. So we add to Π' the rule

$$R_3' : \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,\_),e(\_,y)}(x,y) \leftarrow \mathsf{TC}_{\mathsf{TC}(x,y)\leftarrow e(x,\_),e(\_,y)}(x,z), e(z,y).$$

In principle, we could now try to use the newly created rule R_3' to generate yet another adorned version of R_2 by unifying the head of R_3' with the IDB body atom of R_2. It is easy to verify that the same process as described above would again yield an adorned rule that is exactly the same as R_3'. Hence, the process of generating new EDB-adorned rules has reached a fixpoint, and we can take $\Pi' = \{R_1', R_2', R_3'\}$, which was already shown in Example 1.1, as our final EDB-bounded program.

As was illustrated in Example 3.9, the use of an adornment relaxation function allows us to delete redundant body atoms from an adornment. However, in the context of an adorned program, an entire rule may be redundant in that its deletion does not alter the semantics of the adorned program. To detect certain forms of redundancy of an adorned rule for potentially different use cases, we introduce "membership checking functions" next:

Definition 3.10 (membership checking function). A *membership checking function* is a function h which takes as input an EDB-bounded rule R with head predicate q_ρ, and a datalog program Π, and outputs either true or false. In this work, we consider two membership checking functions:

- h_{eq}: A function that returns true if there exists some $R' \in \Pi$ that is identical to R, up to variable renaming, and false otherwise. Except for the discussion in Section 4, this function is used throughout the remainder of the paper for deriving size bounds.
- h_{cont}: A function that returns true if either (1) q_ρ is not recursive in Π and there exists a predicate $q_{\rho'}$ such that ρ' subsumes ρ, or (2) there exists some $R' \in \Pi$ that is identical to R up to variable renaming. Otherwise, it returns false. This function will be used in Section 4 in the semi-decision procedure for boundedness.

3.2.1 Algorithm to Construct EDB-Bounded Programs. We now present Algorithm 1, which constructs an EDB-bounded datalog program based on a given program using a given adornment relaxation function g and membership checking function h. The idea of the algorithm aligns with the intuition given in our previous example. Namely, it iteratively tries, until a fixpoint is reached, to turn the rules in the original program Π into adorned rules in Π' by replacing each IDB body atom of the original rule by an adorned atom already in Π', and then generating the adornment for the head atom using the adornment relaxation function g. The adorned rule is then added to Π' subject to the membership checking function h.

Algorithm 1: Construction of an EDB-bounded datalog program from a given program

input : A datalog program Π
parameters: An adornment relaxation function g, a membership checking function h
output : An EDB-bounded datalog program Π'

Initialize Π' as an empty datalog program
repeat
  foreach *rule* R *in* Π **do**
    W.l.o.g. assume that R has the form $q(\bar{s}) \leftarrow (\bigwedge_{i=1}^{\lambda} p_i(\bar{s})) \wedge \mathcal{E}$, where $p_1(\bar{s}_1), \ldots, p_\lambda(\bar{s}_\lambda)$ are IDB atoms, and $\mathcal{E}$ is a conjunction of EDB atoms
    foreach $(\rho_1, \ldots, \rho_\lambda)$, *such that for all* $i = 1, \ldots, \lambda$, *there exists a rule* R_i *in* Π' *with head atom* $p_{i\,\rho_i}(\bar{v}_i)$ *with adornment* ρ_i **do**
      Apply a variable renaming to all head atoms $p_{i\,\rho_i}(\bar{v}_i)$ to obtain $p_{i\,\rho_i'}(\bar{v}_i')$ so that R and all head atoms in all R_i are variable-disjoint
      Let ρ_i' be of the form $p_i(\bar{v}_i') \leftarrow \mathcal{B}_i$
      $\sigma \leftarrow$ some mgu$\{(\bar{s}_i, \bar{v}_i') \mid 1 \le i \le \lambda\}$
      if *no* σ *exists* **then**
        Continue
      $\rho_0 :=$ the rule $q(\sigma(\bar{s})) \leftarrow (\bigwedge_{i=1}^{\lambda} \sigma(\mathcal{B}_i)) \wedge \sigma(\mathcal{E})$
      $\rho := g(\rho_0)$
      $R' :=$ the rule $q_\rho(\sigma(\bar{s})) \leftarrow (\bigwedge_{i=1}^{\lambda} p_{i\,\rho_i}(\sigma(\bar{s}_i))) \wedge \sigma(\mathcal{E})$
      if $h(R', \Pi')$ *is false* **then**
        Add R' to Π'
until *fixpoint for* Π'
return Π'

We give a detailed explanation of the algorithm using the adornment relaxation function g_{out} and membership checking function h_{eq}. We examine the behavior of the algorithm by distinguishing rules (in Π) with only EDB body atoms and rules containing at least one IDB atom in the body.

Base case. Let $R\colon q(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_\mu(\bar{s}_\mu)$ be a rule in Π with only EDB atoms in the body. Then we construct a rule $R'\colon q_\rho(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_\mu(\bar{s}_\mu)$ to be added to Π'. To obtain the adornment ρ, let ρ_0 be a rule of the form $\rho_0\colon q(\bar{s}) \leftarrow p_1(\bar{t}_1), \ldots, p_\mu(\bar{t}_\mu)$ with $t_{ij} = s_{ij}$ if s_{ij} is a head variable in R (i.e., $t_{ij} \in \text{vars}(\bar{s})$) and $t_{ij} = \_$ otherwise. We then obtain ρ from ρ_0 by applying the adornment relaxation function g_{out}.

Adornment propagation. Let $R\colon q(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_\mu(\bar{s}_\mu)$ be a rule in Π, where at least one p_i is an IDB predicate. Then, for every i, such that p_i is an IDB predicate, choose one rule R_i in Π' with head atom $p_{i\,\rho_i}(\bar{v}_i)$ for some adornment ρ_i. If, for some i, no such rule exists yet, then no new EDB-adorned datalog rule can be created from rule R at this point. Otherwise, let us assume w.l.o.g. that R has the IDB body atoms $p_1(\bar{s}_1), \ldots, p_\lambda(\bar{s}_\lambda)$ for some $\lambda \le \mu$ and that the remaining body atoms are EDB atoms.

Now apply to all head atoms $p_{i\,\rho_i}(\bar{v}_i)$ of the rules $R_1, \ldots, R_\lambda$ a variable renaming to make sure that rules R and the head atoms of the rules $R_1, \ldots, R_\lambda$ are variable-disjoint. Note that, for each atom $p_{i\,\rho_i}(\bar{v}_i)$, we now apply the same renaming of the variables $\bar{v}_i$ also to the adornment ρ_i. Hence, by this variable renaming, the new head atoms of the rules $R_1, \ldots, R_\lambda$ are now $p_{i\,\rho_i'}(\bar{v}_i')$.

Next, we compute the simultaneous, most general unifier σ of $\{(\bar{s}_1, \bar{v}'_1), \ldots, (\bar{s}_\lambda, \bar{v}'_\lambda)\}$. If no such simultaneous unifier exists, then no new EDB-adorned datalog rule can be created from rule R with this choice of EDB-adorned rules $R_1, \ldots, R_\lambda$. Otherwise, we add to Π' an adorned rule

$$R' \colon q_\rho(\sigma(\bar{s})) \leftarrow p_{1_{\rho_1}}(\sigma(\bar{s}_1)), \ldots, p_{\lambda_{\rho_\lambda}}(\sigma(\bar{s}_\lambda)), p_{\lambda+1}(\sigma(\bar{s}_{\lambda+1})), \ldots, p_\mu(\sigma(\bar{s}_\mu)),$$

where we first define the rule ρ_0 and then obtain ρ by applying on ρ_0 the adornment relaxation function g_{out}. The rule ρ_0 is constructed as follows:

- The head of ρ_0 is $q(\sigma(\bar{s}))$, i.e., the same as the unadorned head of the new rule R';
- for every EDB atom $p_i(\bar{s}_i)$ in the body of R, we add the atom $p_i(\bar{t}_i)$ to the body of ρ_0 with $t_{ij} = \sigma(s_{ij})$ if $t_{ij} \in \text{vars}(\sigma(\bar{s}))$ (i.e., $\sigma(s_{ij})$ is a head variable of R'), and $t_{ij} = \_$ otherwise;
- for every $i \in \{1, \ldots, \lambda\}$ and for every atom $e(\bar{a})$ in the body of ρ'_i, we add to the body of ρ_0 the atom $e(\bar{b})$ with $b_j = \sigma(a_j)$ if $\sigma(a_j) \in \text{vars}(\sigma(\bar{s}))$ (i.e., $\sigma(a_j)$ is a head variable of R'), and $b_j = \_$ otherwise.

Membership checking. Suppose that a new adorned rule is generated as explained above. Then our algorithm only adds this new rule to Π' if the membership checking function h_{eq} returns false—i.e., the new rule is not just a renaming of an already existing rule.

Let us comment on two particular aspects of the construction of Π' in Algorithm 1:

(1) It should be noted that, as adornment of an IDB body atom $p_{i_{\rho_i}}(\sigma(\bar{s}_i))$, we take the original adornment ρ_i of the head atom of rule R_i. In contrast, when transferring body atoms from the adornment of the head atom of rule R_i to the body of the adornment ρ_0, we first take the adornment ρ'_i resulting from the variable renaming and then also apply the mgu σ to these body atoms. We then apply g_{out} to make sure that all variables in an EDB atom in the adornment ρ_0 also occur in the head of rule R' (after applying the mgu σ to the rule R) and that redundant EDB atoms are removed from the rule body of ρ_0.
(2) Note that we have explicitly mentioned the base case here for the sake of readability. Clearly, the case of a rule with only EDB atoms in the body is a special case of the adornment propagation where $\lambda = 0$. This is how rules with only EDB atoms in the body are handled in Algorithm 1.

3.2.2 Requirement of Simultaneous Unifiers in Algorithm 1. Note that, in Algorithm 1, we have to compute a simultaneous unifier between each IDB body atom of a rule R and the rule heads of the adorned rules $R_1, \ldots, R_\lambda$. We show why this is required by applying the construction of Algorithm 1 to a slightly more complex datalog program as an example.

Example 3.11. Suppose that we apply Algorithm 1 with functions g_{out} and h_{eq}. We consider the following datalog program

$$p(v, v) \leftarrow e(v, w).$$
$$q(x, y, z) \leftarrow p(x, y), p(y, z).$$

Note that, in this case, we in fact have $x = y$ and $y = z$. Therefore, we should ultimately get the following adorned program:

$$p_{p(v,v) \leftarrow e(v,\_)}(v, v) \leftarrow E(v, w).$$
$$q_{q(x,x,x) \leftarrow e(x,\_)}(x, x, x) \leftarrow p_{p(v,v) \leftarrow e(v,\_)}(x, x), p_{p(v,v) \leftarrow e(v,\_)}(x, x).$$

Since the two body atoms in the second rule are identical, we can actually delete one and get

$$q_{q(x,x,x) \leftarrow e(x,\_)}(x, x, x) \leftarrow p_{p(v,v) \leftarrow e(v,\_)}(x, x).$$

Now, how should we define the adornment propagation in this case—taking variable renaming and unification into account, and being able to reason about the equality of variables across different atoms (e.g., x, y, and z in this particular example)?

First, we choose a rule for each of the two IDB body atoms. In this case, $R_1 = R_2$. So the adornments of the two IDB body atoms are fixed, namely $p(v, v) \leftarrow e(v, \_)$ for both. However, before we can compute the simultaneous mgu of each body atom with the head atom of the corresponding rule, we have to rename the variables of all rules apart to avoid collisions, i.e., we get

$$R_1' : p_{p(v,v)\leftarrow e(v,\_)}(v, v) \leftarrow e(v, w).$$
$$R_2' : p_{p(v',v')\leftarrow e(v',\_)}(v', v') \leftarrow e(v', w').$$

Note that, for computing the adornment of the head atom of the new rule, we have also applied the variable renaming in rule R_2 to the adornment. This is in contrast to the adornment of the second body atom of the new rule, where we have left the adornment of R_2 unchanged.

Now we have to compute $\sigma = \text{mgu}(\{(p(x, y), p(v, v)), (p(y, z), p(v', v'))\})$. Clearly, σ sets all variables x, y, z, v, v' equal. Nevertheless, there are several ways of expressing σ, e.g.

$$\sigma_1 = \{x \mapsto v, y \mapsto v, z \mapsto v, v' \mapsto v\}.$$
$$\sigma_2 = \{y \mapsto x, z \mapsto x, v \mapsto x, v' \mapsto x\}.$$

We now have to apply the mgu to all atoms (head atom and all body atoms) of the rule R, i.e., depending on the choice of mgu, we get:

$$\text{With mgu } \sigma_1: \quad q_\rho(v, v, v) \leftarrow p_{\rho_1}(v, v), p_{\rho_2}(v, v).$$
$$\text{With mgu } \sigma_2: \quad q_\rho(x, x, x) \leftarrow p_{\rho_1}(x, x), p_{\rho_2}(x, x).$$

Since $\rho_1 = \rho_2$, we may drop one of the two body atoms in each of the above two rules.

Finally, we have to compute the adornment of the head atom of the new rule. We thus construct a rule whose head atom is the same as the head atom of the new rule; as body atoms, we collect the body atoms of the adornments of the head atoms in the rules R_1 and R_2, but first apply the mgu to these body atoms. We thus get the following adornments:

$$\text{With mgu } \sigma_1: \quad \rho = q(v, v, v) \leftarrow e(v, \_), e(v, \_).$$
$$\text{With mgu } \sigma_2: \quad \rho = q(x, x, x) \leftarrow e(x, \_), e(x, \_).$$

In both rules, the body consists of two identical atoms. Hence, in each rule, we can delete one of the two body atoms, and we arrive at the rule $q_{q(x,x,x)\leftarrow e(x,\_)}(x, x, x) \leftarrow p_{p(v,v)\leftarrow e(v,\_)}(x, x)$, as mentioned above.

3.2.3 Equivalence of EDB-Bounded Programs Produced by Algorithm 1. We now establish the close relationship between the original datalog program Π and the adorned program Π' resulting from Algorithm 1. More specifically, we show that, by an appropriate choice of the functions g and h, the two programs are "equivalent", i.e., for any EDB, they compute "the same" IDB relations modulo adornment. This notion of "equivalence" is formally defined below.

Definition 3.12 (equivalent EDB-bounded program). Let Π be a datalog program and let Π' be an EDB-bounded datalog program. We say that Π *and* Π' *are equivalent*, if for every EDB D and for every IDB predicate q occurring in Π, the following condition holds:

$$[\![q(\Pi, D)]\!] = \bigcup_\rho [\![q_\rho(\Pi', D)]\!]$$

That is, the set of tuples in the IDB relation q that can be derived by program Π from the database D coincides with the union over all sets of tuples in any adorned IDB relation q_ρ that can be derived by program Π' from the database D.

We next show that the EDB-adorned program Π' constructed from an arbitrary datalog program Π according to Algorithm 1, with g_{out} and h_{eq}, is indeed an *equivalent EDB-bounded* program.

Theorem 3.13. *Given an arbitrary datalog program* Π, *let* Π' *be the EDB-adorned program obtained from* Π *according to Algorithm 1 with adornment relaxation function* g_{out} *and membership checking function* h_{eq}. *Then* Π' *is an EDB-bounded program equivalent to* Π.

Proof. Let Π and Π' be according to the statement of the theorem. Moreover, let D be an arbitrary database over the EDB predicates of Π (and, hence, also of Π'). We have to prove (1) that the two programs lead to the same IDB relations (modulo adornments) and (2) that every adorned rule in Π' is indeed bounded by the adornment of the head predicate.

Equivalence. Let q be an arbitrary IDB predicate occurring in Π. We have to show that $[\![q(\Pi, D)]\!] = \bigcup_\rho [\![q_\rho(\Pi', D)]\!]$ holds.

The inclusion $[\![q(\Pi, D)]\!] \supseteq \bigcup_\rho [\![q_\rho(\Pi', D)]\!]$ is seen as follows: Suppose that Π' contains a rule $R' \colon q_\rho(\sigma(\bar{s})) \leftarrow p_{1_{\rho_1}}(\sigma(\bar{s}_1)), \ldots, p_{\mu_{\rho_\mu}}(\sigma(\bar{s}_\mu))$, then, by the construction in Algorithm 1, Π contains the rule $R \colon q(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_\mu(\bar{s}_\mu)$. Hence, any proof tree $\mathcal{T}'$ of a ground IDB atom $q_\rho(\bar{t})$ with program Π' can be turned into a proof tree $\mathcal{T}$ of $q(\bar{t})$ by simply replacing every rule R' by the corresponding rule R and removing all adornments.

It remains to prove the inclusion $[\![q(\Pi, D)]\!] \subseteq \bigcup_\rho [\![q_\rho(\Pi', D)]\!]$. That is, for every $\bar{t} \in [\![q(\Pi, D)]\!]$, we have to show that there exists an adorned predicate q_ρ with $\bar{t} \in [\![q_\rho(\Pi', D)]\!]$. We prove this property by induction on the depth of a proof tree of $q(\bar{t})$ of program Π over the database D.

Base case. Suppose that the depth of a proof tree of $q(\bar{t})$ is $j = 1$, i.e., $q(\bar{t})$ is derived by applying a rule of the form $q(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_\mu(\bar{s}_\mu)$, such that all body atoms are EDB atoms. Then Π' contains a rule $q_\rho(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_\mu(\bar{s}_\mu)$, where ρ is constructed according to the base case of Algorithm 1. In particular, ρ is such an adornment with $\bar{t} \in [\![q_\rho(\Pi', D)]\!]$.

Induction step. Now suppose that the depth of a proof tree $\mathcal{T}$ of $q(\bar{t})$ is $j > 1$. That is, the proof tree $\mathcal{T}$ has $q(\bar{t})$ as root node, which is connected via edges with some label R to its child nodes. Let $p_1(\bar{t}_1), \ldots, p_\mu(\bar{t}_\mu)$ be the ground atoms labeling the child nodes of the root in $\mathcal{T}$ and let the rule R have the form $q(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_\mu(\bar{s}_\mu)$. Hence, R fires by (simultaneously) instantiating each body atom $p_i(\bar{s}_i)$ to $p_i(\bar{t}_i)$, i.e., there exists a substitution η, such that $\eta(\bar{s}_i) = \bar{t}_i$ for every $i \in \{1, \ldots, \mu\}$ and $\eta(\bar{s}) = \bar{t}$.

Clearly, the ground atoms $p_1(\bar{t}_1), \ldots, p_\mu(\bar{t}_\mu)$ are either EDB atoms or they have a proof tree of depth $< j$. Since we are assuming that the depth of $\mathcal{T}$ is > 1, there is at least one IDB atom in the body of R. W.l.o.g., let $1 \leq \lambda \leq \mu$ such that $p_1, \ldots, p_\lambda$ are IDB predicates and $p_{\lambda+1}, \ldots, p_\mu$ are EDB predicates. Hence, by the induction hypothesis, we can derive adorned atoms $p_{i_{\rho_i}}(\bar{t}_i)$ for some adornments ρ_i also in Π' with $1 \leq i \leq \lambda$. In particular, this means that Π' contains rules $R'_i, \ldots, R'_\lambda$ with rule heads $p_{i_{\rho_i}}(\bar{u}_i)$, such that $p_{i_{\rho_i}}(\bar{t}_i)$ is a ground instance of $p_{i_{\rho_i}}(\bar{u}_i)$. W.l.o.g., suppose that $R'_i, \ldots, R'_\lambda, R$ are variable disjoint (otherwise apply variable renamings to $R'_i, \ldots, R'_\lambda$ as in Algorithm 1). Clearly, $\bar{t}_i$ is a common ground instance of $\bar{u}_i$ and $\bar{s}_i$, for every $i \in \{1, \ldots, \lambda\}$. Hence, we can simultaneously unify $\bar{u}_i$ and $\bar{s}_i$ for all $1 \leq i \leq \lambda$. Let $\sigma = \text{mgu}\{(\bar{u}_1, \bar{s}_1), \ldots, (\bar{u}_\lambda, \bar{s}_\lambda)\}$. Then there exists a substitution τ, such that $\tau\sigma(\bar{u}_i) = \tau\sigma(\bar{s}_i) = \bar{t}_i = \eta(\bar{s}_i)$ for every $i \in \{1, \ldots, \lambda\}$. Note that η possibly instantiates additional variables compared to $\eta(\sigma(\cdot))$, since it also maps all tuples $\bar{s}_i$ with $i \in \{\lambda + 1, \ldots, \mu\}$ to $\bar{t}_i$. Hence, there exists a substitution θ (equal to η or extending η), such that $\theta(\sigma(\bar{s}_i)) = \eta(\bar{s}_i) = \bar{t}_i$ for every $i \in \{1, \ldots, \mu\}$ and $\theta(\sigma(\bar{s})) = \eta(\bar{s}) = \bar{t}$.

By Algorithm 1, Π' contains the rule $R' \colon q_\rho(\sigma(\bar{s})) \leftarrow p_{1_{\rho_1}}(\sigma(\bar{s}_1)), \ldots, p_{\mu_{\rho_\mu}}(\sigma(\bar{s}_\mu))$, where ρ is constructed according to the adornment propagation in Algorithm 1. By the above considerations, θ maps each body atom $p_{i_{\rho_i}}(\sigma(\bar{s}_i))$ to $p_{i_{\rho_i}}(\theta(\sigma(\bar{s}_i))) = p_{i_{\rho_i}}(\bar{t}_i)$ and it maps the head atom $q_\rho(\sigma(\bar{s}))$ to $q_\rho(\theta(\sigma(\bar{s}))) = q_\rho(\bar{t})$. Since we are assuming that all the EDB-adorned atoms $p_{i_{\rho_i}}(\bar{t}_i)$ for $i \in \{1, \ldots, \lambda\}$ can be derived by Π' from the EDB D, and D contains the ground atoms $p_i(\bar{t}_i)$ for $i \in \{\lambda + 1, \ldots, \mu\}$, we conclude that $q_\rho(\bar{t})$ can indeed be derived by Π' from the EDB D.

Boundedness. We have to show that every rule in Π' is bounded by the adornment of its head predicate. That is, let R' with head predicate q_ρ be a rule in Π' and let $q_\rho(\bar{t})$ be derived by rule R' when evaluating Π' over an EDB D. We have to show that $q(\bar{t})$ is also derived when evaluating the single-rule program consisting of rule ρ over the EDB D. The proof is by induction on the depth of a proof tree in Π' of a fact $q_\rho(\bar{t})$.

Base case. Suppose that $q_\rho(\bar{t})$ is derived by a proof tree $\mathcal{T}$ of depth $j = 1$ and that rule R' with head predicate q_ρ is the top-most rule in $\mathcal{T}$. Then R' is of the form $q_\rho(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_1(\bar{s}_\mu)$, where all p_i are EDB predicates. By the base case in Algorithm 1, R' is constructed from a rule R of the form $q(\bar{s}) \leftarrow p_1(\bar{s}_1), \ldots, p_1(\bar{s}_\mu)$ in Π, such that ρ is obtained from R by leaving the head atom unchanged and replacing in every body atom all arguments that are not head variables by "_" and then deleting redundant body atoms. Recall that the occurrences of "_" in body atoms are treated like pairwise distinct, existentially quantified variables. Hence, whenever R fires, so does ρ, since the body of ρ has been constructed by starting off with the body of R and possibly "weakening" (i.e., replacing some arguments by "_") or even deleting some of the body atoms.

Induction step. Suppose that $q_\rho(\bar{t})$ is derived by a proof tree $\mathcal{T}$ of depth $j > 1$ and that rule R' with head predicate q_ρ is the top-most rule in $\mathcal{T}$. That is, $\mathcal{T}$ has $q_\rho(\bar{t})$ as root node, which is connected via edges with label R' to its child nodes. Moreover, since $j > 1$, R' contains at least one IDB atom in the body. By the the adornment propagation in Algorithm 1, rule R' is of the form

$$R' \colon q_\rho(\sigma(\bar{s})) \leftarrow p_{1_{\rho_1}}(\sigma(\bar{s}_1)), \ldots, p_{\lambda_{\rho_\lambda}}(\sigma(\bar{s}_\lambda)), p_{\lambda+1}(\sigma(\bar{s}_{\lambda+1})), \ldots, p_\mu(\sigma(\bar{s}_\mu)).$$

Hence, the child nodes of the root in $\mathcal{T}$ are labeled by ground atoms $p_{1_{\rho_1}}(\bar{t}_1), \ldots, p_{\lambda_{\rho_\lambda}}(\bar{t}_\lambda), p_{\lambda+1}(\bar{t}_{\lambda+1}), \ldots, p_\mu(\bar{t}_\mu)$, such that $p_{1_{\rho_1}}, \ldots p_{\lambda_{\rho_\lambda}}$ are IDB predicates, and $p_{\lambda+1}, \ldots, p_\mu$ are EDB predicates. Importantly, all tuples $\sigma(\bar{s}_i)$ with $i \in \{1, \ldots, \mu\}$ can be simultaneously instantiated to $\bar{t}_i$. Hence, there exists a substitution τ, such that $\bar{t}_i = \tau(\sigma(\bar{s}_i))$ for every $i \in \{1, \ldots, \mu\}$ and $\bar{t} = \tau(\sigma(\bar{s}))$.

Moreover, there exist rules $R_1, \ldots, R_\lambda$ in Π' with head predicates $p_{1_{\rho_1}}, \ldots p_{\lambda_{\rho_\lambda}}$, which were used in the construction of rule R'. Recall from the propagation of adornments in Algorithm 1 that σ is actually obtained as mgu when simultaneously unifying, for every $i \in \{1, \ldots, \lambda\}$, the arguments $\bar{u}_i$ of the head atom $p_{i_{\rho_i}}(\bar{u}_i)$ of rule R_i with the arguments $\bar{s}_i$ of the corresponding body atom $p_i(\bar{s}_i)$ of the rule R in Π. (Here we are assuming w.l.o.g., that the rules $R_1, \ldots, R_\lambda, R$ are pairwise variable-disjoint; this can always be achieved by appropriate renaming of variables.) Hence, $\sigma(\bar{s}_i) = \sigma(\bar{u}_i)$ and, therefore, we also have $\bar{t}_i = \tau(\sigma(\bar{u}_i))$ for every $i \in \{1, \ldots, \lambda\}$.

In the proof tree $\mathcal{T}$, each $p_{i_{\rho_i}}(\bar{t}_i)$ has itself a proof tree $\mathcal{T}_i$ of depth $< j$, and the root node of $\mathcal{T}_i$ is connected with edges labeled by R_i with its child nodes. Hence, by the induction hypothesis, every $p_i(\bar{t}_i)$ is also derived when evaluating the single-rule program consisting of rule ρ_i only. By the construction of Π' in Algorithm 1, rule R_i and the rule ρ_i adorning the head predicate of R_i have, apart from the adornment, the same head atom. That is, the head atom of ρ_i is $p_i(\bar{u}_i)$. Moreover, as has been argued above, we have $p_i(\bar{t}_i) = p_i(\tau(\sigma(\bar{u}_i)))$. That is, the rule ρ_i fires on the EDB D by applying a substitution of the form $\eta_i \cup \tau(\sigma(\cdot))$ to ρ_i, where η_i extends the substitution $\tau(\sigma(\cdot))$ to the variables only occurring in the body of ρ_i. Of course, for any two indices $i \neq i'$, the substitutions η_i and $\eta_{i'}$ cannot interfere with each other, since we are assuming that any two rules out of $R_1, \ldots, R_\lambda$ are variable-disjoint and, therefore, also the adornments $\rho_1, \ldots, \rho_\lambda$ (which may

only contain variables occurring in the head of the corresponding rules $R_1, \ldots, R_\lambda$) are pairwise variable-disjoint.

Recall from the adornment propagation in Algorithm 1 that the adornment ρ of the head atom $q_\rho(\sigma(\bar{s}))$ of rule R' is obtained as follows:

- The head atom of ρ is the same as the head atom of R' without the adornment, i.e., $q(\sigma(\bar{s}))$;
- in every EDB atom of the body of R', the variables not occurring in $\sigma(\bar{s})$ are replaced by "_", and the resulting atom is added to the body of ρ;
- in every body atom of every rule ρ_i, we first apply the substitution σ and then again replace the variables not occurring in $\sigma(\bar{s})$ by "_", before we add the resulting atom to the body of ρ.

Actually, some of the aforementioned body atoms of rule ρ may possibly be deleted by the redundancy elimination step. Anyway, we show that all of the above mentioned (potential) body atoms of rule ρ can be simultaneously matched into the EDB D by a substitution that extends $\tau(\sigma(\cdot))$.

First consider some body atom of ρ obtained from an EDB atom $p_i(\sigma(\bar{s}_i))$ of the body of R'. Even without replacing some of the arguments in $\sigma(\bar{s}_i)$ by "_" (i.e., by a fresh existentially quantified variable), we have that $p_i(\tau(\sigma(\bar{s}_i)))$ is a ground atom from the EDB D. Hence, there clearly exists an extension of $\tau(\sigma(\cdot))$ to the fresh, existentially quantified variables so that the atom obtained from $p_i(\sigma(\bar{s}_i))$ by introducing "_" is mapped to a ground atom in the EDB.

Now consider a body atom of ρ that is obtained from some body atom of some rule ρ_i. As has been shown above, even without replacing some of the arguments in this atom by "_" (i.e., by a fresh existentially quantified variable), $\tau(\sigma(\cdot))$ can be extended by a substitution η_i that maps every body atom of ρ_i to a ground atom from the EDB D. Hence, we may again conclude that there exists an extension of $\tau(\sigma(\cdot))$ to the fresh, existentially quantified variables so that every body atom of ρ_i is mapped to an atom from the EDB D.

In other words, the rule ρ fires on the EDB D and produces the atom $q(\bar{t}) = q(\tau(\sigma(\bar{s})))$. □

4 Semi-Deciding and Rewriting Bounded Datalog Programs

In this section, we revisit the problem of datalog boundedness. We first show the semi-decidability of boundedness by (potentially infinite) inlining. Then, as one use case of our framework, we give an instantiation that rewrites most practical bounded datalog programs into equivalent UCQs by simple inlining. In case the equivalent UCQ is too large, or the program is unbounded, the algorithm applies adornment relaxation functions to obtain an equivalent finite EDB-bounded program.

We start by formalizing this idea of potentially infinite inlining of datalog programs:

Definition 4.1 (unrolling). The *unrolling* of a datalog program Π is defined as the non-recursive, potentially infinite equivalent program derived by repeatedly inlining the rules of Π.

It has also been well known that

Theorem 4.2 ([29, Theorem 3]). *Given two UCQs Q_1 and Q_2, then $Q_1 \subseteq Q_2$ if and only if every CQ in Q_1 is contained in a CQ in Q_2.*

Notice that Theorem 4.2 can be easily generalized to infinite UCQs and to non-recursive datalog programs, which are equivalent to finite sets of UCQs. This leads directly to the following corollary:

Corollary 4.3. *Given a datalog program Π, let Π^* be the unrolling of Π. Let Π_b be a non-recursive (finite) datalog program equivalent to Π. Then Π_b is equivalent to a finite subset of rules from Π^*.*

Proof. A set of datalog rules sharing the same head predicate defines a UCQ. Consequently, every CQ derived from rules in Π_b must be contained within a CQ from the rules in Π^*. Since Π_b has a finite set of rules, the set of CQs equivalent to Π_b in Π^* is also finite. □

Next, we show an application of this result within our framework to obtain a non-recursive program that is equivalent to a given bounded datalog program.

Proposition 4.4. *Given a datalog program* Π, *let* Π' *be the (potentially infinite) output of applying Algorithm 1 on* Π *with the identity function id as the adornment relaxation function, and let* Π^* *be the non-recursive unrolling of* Π. *Then* Π' *is equivalent to* Π^*.

Proof. The head adornments form the unrolling of Π (each of which is simply an inlining), and every rule in Π^* is equivalent to its head adornment. □

Furthermore, by applying h_{cont} as the membership checking function, which attempts not to add an adornment if it is subsumed by another adornment, then the algorithm always terminates when the input program is bounded:

Proposition 4.5. *Algorithm 1, with id as the adornment relaxation function and* h_{cont} *as the membership checking function, terminates on a datalog program* Π *if and only if* Π *is bounded.*

Proof. Let Π_b be a non-recursive datalog program equivalent to Π and let Π' be the output of the new algorithm. By Propositions 4.3 and 4.4, Π_b is equivalent to a subset S of Π'. Every rule in $S^{-1} = \Pi' \setminus S$ must be contained by S. Since every rule in Π' is equivalent to a CQ, every rule in S^{-1} is equivalent to exactly one rule in S. It must then be the case that $S^{-1} = \emptyset$ by definition of the use of the h_{cont} membership checking function. □

We illustrate the process with the following examples.

Example 4.6. Consider the following program:

$$R_1 : r(y) \leftarrow e(x, y).$$
$$R_2 : r(y) \leftarrow r(x), e(x, y).$$

This program is bounded and is in fact equivalent to one with only a single rule R_1. Running the algorithm with *id* and h_{cont}, we first generate the adorned rule $R_1' : r_{r(y) \leftarrow e(x,y)}(y) \leftarrow e(x, y)$. Then, the algorithm would attempt to add $\rho_1 : r(y) \leftarrow e(x, y)$ as the adornment of $r(x)$ in the second rule, which then generates the adornment $\rho_2 : r(y) \leftarrow e(x', x), e(x, y)$ for the head atom. However, ρ_2 *is subsumed by* ρ_1, so the algorithm terminates and outputs a program with one rule R_1'.

Example 4.7. We take the following program from [2, Example 12.5.6]:

$$R_1 : \mathsf{Buys}(x, y) \leftarrow \mathsf{Likes}(x, y).$$
$$R_2 : \mathsf{Buys}(x, y) \leftarrow \mathsf{Trendy}(x), \mathsf{Buys}(\_, y).$$

This program is bounded. After running the algorithm, we obtain the following equivalent non-recursive program:

$$R_1' : \mathsf{Buys}_{\mathsf{Buys}(x,y) \leftarrow \mathsf{Likes}(x,y)}(x, y) \leftarrow \mathsf{Likes}(x, y).$$
$$R_2' : \mathsf{Buys}_{\mathsf{Buys}(x,y) \leftarrow \mathsf{Trendy}(x), \mathsf{Likes}(\_,y)}(x, y) \leftarrow \mathsf{Trendy}(x), \mathsf{Buys}_{\mathsf{Buys}(x,y) \leftarrow \mathsf{Likes}(x,y)}(\_, y).$$

If we continue and attempt to add the adornment $\rho_2 : \mathsf{Buys}(x, y) \leftarrow \mathsf{Trendy}(x), \mathsf{Likes}(\_, y)$ of the head atom of R_2' as an adornment of $\mathsf{Buys}(\_, y)$, we generate the adornment $\rho_3 : \mathsf{Buys}(x, y) \leftarrow \mathsf{Trendy}(x), \mathsf{Trendy}(x'), \mathsf{Likes}(\_, y)$. However, ρ_3 is subsumed by ρ_2, so the algorithm terminates.

From Proposition 4.5, we have that, even though boundedness is undecidable,

Corollary 4.8. *Bounded datalog programs are recursively enumerable.*

We note that this does not have to be shown using our framework of adornment. The essence is to perform inlining (which generates rules with only EDB atoms in the body) and eliminate rules that are subsumed by another rule in the flavor of the membership checking function h_{cont}.

Nevertheless, the algorithm does not terminate on an unbounded datalog program. To address this, we propose relaxing the procedure using the adornment relaxation function g_k (instead of id). This function introduces a budget k, retaining all EDB atoms in the adornment (like the identity function does) until there are more than k, at which point it starts to behave as g_{out}. This guarantees termination for all programs but loses the guarantee that all bounded datalog programs can be rewritten into non-recursive equivalents when a rule in the output program requires more than k EDB atoms in an adornment. However, one can choose a concrete value of k based on the time and space budget allocated for the procedure. If the program is unbounded, or if the equivalent non-recursive program is too large (exceeding the specified budget), the algorithm gracefully degrades to use g_{out} instead, as it is either impossible or practically unbeneficial to find the non-recursive equivalent. Otherwise, the algorithm will successfully rewrite all bounded datalog programs that fit within the budget k into non-recursive forms. We illustrate this idea by the following example.

Example 4.9. We take g_2 (g_k with $k = 2$) as the adornment relaxation function and consider again the program in Example 4.6, but with R_1 replaced by $r(x) \leftarrow b(x)$, where b is an EDB relation. This program is no longer bounded. The algorithm starts by generating two EDB-bounded rules:

$$R_1' : r_{r(x)\leftarrow b(x)}(x) \leftarrow b(x).$$
$$R_2' : r_{r(y)\leftarrow b(x),e(x,y)}(y) \leftarrow r_{r(x)\leftarrow b(x)}(x), e(x,y).$$

Then, when the algorithm attemps to add $r(y) \leftarrow b(x), e(x,y)$ as the adornment of $r(x)$ in the body of R_2, it generates the adornment $r(y) \leftarrow b(x_0), e(x_0,x), e(x,y)$, which now has more than 2 body atoms. Therefore, the adornment relaxation function would now behave as g_{out} and transform it to $r(y) \leftarrow e(\_, y)$. We generate the following two additional rules before the procedure terminates:

$$R_3' : r_{r(y)\leftarrow e(\_,y)}(y) \leftarrow r_{r(y)\leftarrow b(x),e(x,y)}(x), e(x,y).$$
$$R_4' : r_{r(y)\leftarrow e(\_,y)}(y) \leftarrow r_{r(y)\leftarrow e(\_,y)}(x), e(x,y).$$

5 Towards FPT Upper Bounds on the Sizes of IDB Relations

As a second use case of our framework, we begin our investigation on upper bounds on the size of the IDB relations produced by Π over an arbitrary (extensional) database instance D. For a given IDB predicate p in Π, we aim at computing an edge-cover width w in terms of EDB predicates. More specifically, if we have an upper bound N on the size of a single EDB relation in D and the size $|\Pi|$ of the program Π, then we want to conclude that the size of the IDB relation p resulting from the evaluation of Π on D is bounded by $f(\Pi) \cdot N^w$. In other words, over datalog programs with bounded widths, the output size is fixed-parameter tractable with respect to the size of the program. For this purpose, we again use the adornment relaxation function g_{out} and membership checking function h_{eq}, which allowed us to establish the equivalence between an original datalog program Π and the EDB-bounded program Π' resulting from Algorithm 1 in Theorem 3.13.

5.1 Edge-Cover Width

Towards our goal of providing upper bounds on sizes of IDB relations via adornments, we use a notion analogous to edge cover numbers for conjunctive queries, which is naturally related to size bounds. We start with the following simple example of an EDB-bounded datalog program.

Example 5.1. Consider the datalog program Π with the following rules:

$$p(x, y, z) \leftarrow e(x, y, w), e(x, z, w), e(y, z, w).$$
$$q(x, y) \leftarrow p(x, y, \_),$$

where e is an EDB relation. Π is equivalent to the following EDB-bounded datalog program Π':

$$p_{p(x,y,z)\leftarrow e(x,y,\_),e(x,z,\_),e(y,z,\_)}(x, y, z) \leftarrow e(x, y, w), e(x, z, w), e(y, z, w),$$
$$q_{q(x,y)\leftarrow e(x,y,\_)}(x, y) \leftarrow p_{p(x,y,z)\leftarrow e(x,y,\_),e(x,z,\_),e(y,z,\_)}(x, y, \_).$$

We note that this EDB-bounded datalog program Π' has the following property: if (a, b) is a tuple in the IDB relation q, then there exists some c, such that the EDB relation e contains the tuple (a, b, c). In other words, the adornment yields a restriction on the values that can possibly occur in an IDB tuple, and it also yields an upper bound on the size of the IDB relation. In this simple example, the size of the IDB relation q_ρ is clearly bounded by the size of the EDB relation e.

The upper bounds on the sizes of IDB relations is captured by the notion of "edge-cover width" from a hypergraph, which we define next.

Definition 5.2 (hypergraph associated with a rule). For every rule $q(\bar{s}) \leftarrow p_1(\bar{t}_1), \cdots, p_\mu(\bar{t}_\mu)$, analogously to the case for conjunctive queries, we define a hypergraph associated with the rule as $H = (V(H), E(H))$, where the set of vertices $V(H) = \bigcup_{i=1}^{\mu} (\text{vars}(\bar{t}_i))$, i.e., the set of variables, and the set of hyperedges $E(H) = \{\text{vars}(\bar{t}_i) \mid 1 \leq 1 \leq \mu\}$. We let $V_{\text{out}} = \text{vars}(\bar{s})$ denote the set of output variables, i.e., the set of variables in the head atom $q(\bar{s})$.

In the case of a conjunctive query q with associated hypergraph $H = (V(H), E(H))$ and output variables V_{out}, the simplest size bound is known to be given by (integral or fractional) edge covers of V_{out} using the hyperedges in $E(H)$, formally defined as follows.

Definition 5.3. An *(integral/fractional) edge cover* of $H = (V(H), E(H))$ with output attributes V_{out} is any assignment $(x_e)_{e\in E(H)}$ that is a feasible solution to the following linear program:

$$\begin{aligned}
\kappa \;=\; \min \quad & \sum_{e\in E(H)} x_e \\
\text{s.t.} \quad & \sum_{e\in E(H):\, v\in e} x_e \;\geq\; 1 && \forall v \in V_{\text{out}}, \\
& x_e \in \{0, 1\} && \forall e \in E(H), \text{ for integral edge cover, or} \\
& 0 \leq x_e \leq 1 && \forall e \in E(H), \text{ for fractional edge cover.}
\end{aligned}$$

κ is called the *(integral/fractional) edge cover number.*

A *full (integral/fractional) edge cover* of $H = (V(H), E(H))$ is an (integral/fractional) edge cover of H with output attributes $V_{\text{out}} = V(H)$.

With the (integral or fractional) edge cover, we can further calculate a size bound of q:

$$\begin{aligned}
|q| \;\leq\; \min \quad & \prod_{e\in E(H)} |e|^{x_e} \\
\text{s.t.} \quad & \sum_{e\in E(H):\, v\in e} x_e \;\geq\; 1 && \forall v \in V_{\text{out}}, \\
& x_e \in \{0, 1\} && \forall e \in E(H), \text{ for integral edge cover, or} \\
& 0 \leq x_e \leq 1 && \forall e \in E(H), \text{ for fractional edge cover.}
\end{aligned}$$

Assuming all EDB relations have at most N tuples, the number of tuples in the final result is bounded by N^κ, where κ is the edge cover number.

For general datalog programs, we define an analogous measure called "edge-cover width" based on the adornments, as follows.

Definition 5.4 (edge-cover width). Let $q_\rho(\bar{s})$ be an IDB relation in an EDB-bounded datalog program Π'. Let H_ρ be the hypergraph associated with ρ and V_{out} be the set of output variables of ρ. Then, the (integral or fractional) *edge-cover width* $\text{ew}(\rho)$ of the adornment ρ is defined by the edge cover number κ of H_ρ to cover the output variables V_{out}.

The edge-cover width $\text{ew}(q_\rho)$ of the adorned predicate q_ρ is defined as $\text{ew}(q_\rho) = \text{ew}(\rho)$.

Moreover, the *edge-cover width* $\text{ew}(\Pi')$ *of an EDB-bounded datalog program* Π' is given by the maximum edge-cover width $\text{ew}(q_\rho)$ among all EDB-adorned IDB relations q_ρ in Π'.

Below, we generalize the observations from Example 5.1. That is, in an EDB-bounded program, the edge-cover width allows us to establish an upper bound on the sizes of IDB relations and a restriction on the values that can possibly occur in tuples of an IDB.

Theorem 5.5. *Let* Π *be an EDB-bounded datalog program, and suppose that we evaluate* Π *over a database instance* D *of size* $|D| = N$. *Let* $k = \text{ew}(q_\rho)$ *denote the fractional edge cover width of* q_ρ. *Then every IDB relation* q_ρ *has size* $|q_\rho| \leq N^k$.

The above theorem is an immediate consequence of the famous AGM bound [8]. For the following restrictions on the values that may possibly occur in an IDB tuple, we have to switch from the fractional to the integral version of ew.

Proposition 5.6. *Let* Π *be a datalog program, let* Π' *be an EDB-bounded program equivalent to* Π, *and let* q_ρ *be an IDB predicate in* Π', *and let* $k = \text{ew}(q_\rho)$ *denote the integral edge-cover width. Then there exist* k *EDB relations* $e_1, \ldots e_k$ *with the following property: for every tuple* $\bar{s}$ *in the IDB relation* q_ρ, *there exist tuples* $\bar{t}_1 \in e_1, \ldots, \bar{t}_k \in e_k$, *such that every value* s_ℓ *in* $\bar{s}$ *occurs in one of the tuples* $\bar{t}_1, \ldots, \bar{t}_k$, *i.e., there exist* i, j *with* $s_\ell = t_{ij}$.

Proof. Let q_ρ be an IDB predicate in Π' and let $S = \{e_1, \ldots, e_k\}$ be an integral edge cover of ρ. We are assuming that Π is an EDB-bounded datalog program. In particular, this means that the adornment ρ bounds all rules with head predicate q_ρ, i.e., every tuple in the relation q_ρ derived by program Π over a database D is also derived by applying ρ to D. Let ρ' denote the rule obtained from ρ by deleting all body atoms that are not in S. Clearly, every tuple derived by applying ρ to D is also derived by applying ρ' to D. In other words, the IDB relation q_ρ is obtained by joining the EDB relations $e_1, \ldots, e_k$ followed by projection. In particular, this means that every tuple $\bar{s}$ in the IDB relation q_ρ gets its values from tuples $\bar{t}_1 \in e_1, \ldots, \bar{t}_k \in e_k$, i.e., for every value s_n in $\bar{s}$, there exist i, j with $s_n = t_{ij}$. □

By taking the equivalence of a datalog program with an EDB-bounded program into account, we immediately get a further corollary.

Proposition 5.7. *Let* Π *be a datalog program and let* Π' *be an EDB-bounded program equivalent to* Π. *Moreover, let* q *be an IDB predicate in* Π *and let* $k = \text{ew}(q)$ *denote the integral edge-cover width. Then, for every tuple* $\bar{s}$ *in the IDB relation* q, *there exist* k *tuples* $\bar{t}_1, \ldots, \bar{t}_k$ *in EDB relations* $e_1, \ldots, e_k$, *respectively, such that every value* s_ℓ *in* $\bar{s}$ *occurs in some tuple* $\bar{t}_i \in e_i$, *i.e., there exist* i, j, *s.t.* $t_{ij} = s_\ell$.

Proof. Let $\bar{s}$ be an arbitrary tuple in the IDB relation q. By the equivalence of Π and the EDB-bounded program Π', there exists an adornment ρ, such that $\bar{s}$ is also a tuple in the IDB relation q_ρ. By the definition of the edge-cover width, we have $k = \text{ew}(q) \geq \text{ew}(q_\rho)$. By Proposition 5.6, there

exist (at most) k EDB relations $e_1, \ldots, e_k$, such that every value s_n in $\bar{s}$ occurs in some tuple $t_i \in e_i$ with $i \in \{1, \ldots, k\}$. □

We have just seen that the edge-cover width $\mathrm{ew}(q_\rho)$ of the IDB-predicate q_ρ in an EDB-bounded program Π' allows us to bound the size of the IDB relation q_ρ and to restrict the possible values occurring in q_ρ when evaluating Π' over a given EDB D. Hence, to make the bound as tight as possible, it would be desirable to get $\mathrm{ew}(q_\rho)$ as small as possible. Below we show that the EDB-adorned program Π' according to Algorithm 1 is optimal in this sense, i.e., it minimizes $\mathrm{ew}(q)$ for every IDB predicate q of the original datalog program Π.

Theorem 5.8. *Let Π be a datalog program and let Π' be the EDB-adorned program Π' produced by Algorithm 1, when executed with relaxation function g_{out} and membership checking function h_{eq}. Then, for every IDB predicate q in Π, the integral edge-cover width $\mathrm{ew}(q)$ for the program Π' is minimal over all EDB-bounded programs equivalent to Π. More precisely, let Π'' be another EDB-bounded program equivalent to Π. Moreover, let $\mathrm{ew}_{\Pi'}(q)$ and $\mathrm{ew}_{\Pi''}(q)$ denote the integral edge-cover width of q in Π' and Π'', respectively. Then $\mathrm{ew}_{\Pi'}(q) \leq \mathrm{ew}_{\Pi''}(q)$ holds.*

Proof sketch. If Algorithm 1 gives an edge-cover width $\mathrm{ew}_{\Pi'}(q) = k$ for a predicate q, we show a way to construct an EDB instance such that, in some output tuple, there are k pairwise distinct attribute values that have to be covered by k different tuples from this EDB instance. □

Theorem 5.8 allows us to refer to the integral edge-cover width of the EDB-bounded datalog program Π' constructed by Algorithm 1 with functions g_{out} and h_{eq} as the integral *edge-cover width of program* Π. We note that, in Theorem 5.8, it is crucial that we define equivalence between the original program Π and the adorned EDB-bounded program Π' in terms of *all* IDB predicates in Π and Π'. If we were only interested in the behavior of Π and Π' on some distinguished *output predicate* q, the edge-cover width of the EDB-bounded program produced by Algorithm 1 is not guaranteed to be minimal over all datalog programs which agree with Π on q but may possibly use a completely different set of additional IDB predicates. This is illustrated by the next example. Of course, by the undecidability of datalog equivalence [38], we cannot expect to avoid such situations.

Example 5.9. Consider the following datalog program Π:

$$p(w, x, y, z) \leftarrow e(w), e(x), e(y), e(z),$$
$$q(w) \leftarrow p(w, \_, \_, \_).$$

and assume that q is the distinguished output predicate. Applying Algorithm 1 with functions g_{out} and h_{eq}, we obtain an EDB-bounded datalog program Π' with the following rules:

$$p_{p(w,x,y,z) \leftarrow e(w), e(x), e(y), e(z)}(w, x, y, z) \leftarrow e(w), e(x), e(y), e(z),$$
$$q_{q(w) \leftarrow e(w)}(w) \leftarrow p_{p(w) \leftarrow e(w), e(x), e(y), e(z)}(w, \_, \_, \_).$$

The adorned IDB relation $p_{p(w,x,y,z) \leftarrow e(w), e(x), e(y), e(z)}$ and thus the program Π' have $\mathrm{ew}(\Pi') = 4$. However, the edge-cover width of the output predicate $q_{q(w) \leftarrow e(w)}$ is only 1. If we perform a simple inlining of the first rule of Π into the second before applying Algorithm 1, we obtain a program Π'' with only one rule $q_{q(w) \leftarrow e(w)}(w) \leftarrow e(w)$, and hence get edge-cover width $\mathrm{ew}(\Pi'') = 1$.

This example illustrates that the width defined on the overall datalog program *with a dedicated output predicate* q may change if inlining is employed to eliminate an auxiliary IDB relation with a large edge-cover width (in this case, p). Nevertheless, the edge-cover width of q remains invariant and the algorithm still correctly returns the minimum in either case.

As a final remark to conclude this subsection, notice that in the Propositions 5.6 and 5.7 as well as in Theorem 5.8, it was crucial to take the *integral* edge-cover width. From now on, unless explicitly specified otherwise, "edge cover" will always refer to the fractional one.

5.2 Size Bounds

We have seen that, for each IDB relation q in some datalog program Π, in the equivalent EDB-bounded datalog program Π' constructed by Algorithm 1, there are some EDB-bounded IDB relations in the form of q_ρ in Π'. In Theorem 5.5, we have shown that, for each q_ρ in Π', $|q_\rho| \leq N^{\mathrm{ew}(q_\rho)}$. Theorem 3.13 further suggests that the tuples in q will be exactly the union of the tuples in all q_ρ. Therefore, we immediately get the following form of upper bound on the size of q:

Proposition 5.10. *For an IDB relation q in a datalog program Π, let Π' be the equivalent EDB-bounded datalog program constructed by Algorithm 1 with functions g_{out} and h_{eq}. Then, $|q| \leq f(\Pi) \cdot N^{\mathrm{ew}(q)}$, where $f(\Pi)$ is the number of IDB relations of the form q_ρ in Π' (or, the number of different adornments ρ that q takes in Π').*

Note that Algorithm 1 does not rely on any property of the data in the EDB. Therefore, the coefficient $f(\Pi)$ depends solely on the schema of the original program Π. But, how can we determine this coefficient, i.e., the number of adornments each IDB relation q in Π can take? We first show that, due to the complexity of this task, it is unlikely that we can get an exact closed-form formula for this number without explicitly generating Π':

Theorem 5.11. *Given an IDB relation q in a datalog program Π, the problem of counting the number of adorned relations in the form of q_ρ in the equivalent EDB-bounded datalog program Π' generated by Algorithm 1 with g_{out} and h_{eq}, denoted by* #ADORNMENTS*, is* #P*-hard.*

Proof sketch. We use a reduction from the problem of counting the satisfying valuations of a positive 2-CNF, which is known to be #P-hard [28]. We create one EDB relation per variable. Then, for each disjunction $x \vee y$, we create three rules that have the same head $p(\bar{s})$ but will have adornments corresponding to (1) x, (2) y, and (3) both x and y, respectively. Finally, we create a rule with head $q(\bar{s})$ that has a conjunction of all $p(\bar{s})$ as the body. Each adornment of q corresponds precisely to one satisfying assignment of variables in the formula. □

Nevertheless, we can derive an upper bound of $f(\Pi)$ as follows:

Theorem 5.12. *For an IDB relation q in a datalog program Π, let Π' be the equivalent EDB-bounded datalog program constructed by Algorithm 1 with functions g_{out} and h_{eq}. Let* #EDBs *denote the number of EDB predicates and let* ear *denote the maximum arity among the set of EDB predicates. Then, $f(\Pi)$, the number of IDB relations of the form q_ρ in Π', is bounded by*

$$f(\Pi) \leq (\mathrm{ar}(q) + |\Pi|)^{\mathrm{ar}(q)} \cdot 2^{\#\mathrm{EDBs} \cdot ((\mathrm{ar}(q)+1)^{\mathrm{ear}}-1)}.$$

Proof sketch. The factor $(\mathrm{ar}(q) + |\Pi|)^{\mathrm{ar}(q)}$ counts the number of possible head atoms: The relation is fixed to be q, and each attribute can be either a variable or a constant. There are at most $\mathrm{ar}(q)$ different variables, and the number of possible constants is bounded by the size of the program, $|\Pi|$. Then, we count all subsets of the set of EDB atoms that can possibly be part of the body of the adornment. There are #EDBs possible relations, and there are at most ear attributes per atom. For each attribute, we choose either one of the $\mathrm{ar}(q)$ variables from the head, or the underscore, and we eliminate the one case where all attributes are underscores. □

Note that this bound overcounts, as we miss an additional requirement that the EDB atoms in the adornment cover all variables in the head atom. However, if we use a different argument, we

can get better bounds on the size of an IDB q. We start by introducing, as a crucial tool for this proof, the Stirling partition number.

Definition 5.13 (Stirling partition number). The *Stirling partition number*, or *Stirling number of the second kind*, denoted by $S(n,k)$, counts the number of ways to partition a set of n *labeled* elements into k *non-empty, unlabeled* sets. It can be calculated by $S(n,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^{k-i}\binom{k}{i}i^n$, based on the inclusion-exclusion principle.

This concept aligns with the idea of edge covers for the IDB attributes. We then give the following much tighter bounds.

Theorem 5.14. *For an IDB relation q in a datalog program Π, let Π' be the equivalent EDB-bounded datalog program constructed by Algorithm 1 with g_{out} and h_{eq}. Then,*

$$|q| \le \sum_{k=1}^{\text{ew}(q)} S(\text{ar}(q),k)\cdot P(\#\text{EDBs}\cdot N,k)\cdot \text{ear}^{\text{ar}(q)} \tag{1}$$

$$\le (\#\text{EDBs}\cdot\text{ear}\cdot\text{ar}(q))^{\text{ar}(q)}\cdot N^{\text{ew}(q)}, \tag{2}$$

where $P(n,k)$ is the permutation number, and $S(n,k)$ is the Stirling partition number.

Proof sketch. For bound (1), we first enumerate k, the number of partitions of a tuple. For each k, $S(\text{ar}(q),k)$ gives the number of possible ways to partition, and $P(\#\text{EDBs}\cdot N,k)$ gives the number of ways to assign one distinct EDB tuple (out of at most $\#\text{EDBs}\cdot N$) to each of the k partitions. Then, each of the $\text{ar}(q)$ attributes can take one of the ear values in the EDB tuple that is assigned to the partition the attribute is in. (1) could be relaxed to obtain (2). □

Note that, in bound (2), the coefficient before $N^{\text{ew}(q)}$ depends only on the schema of the original program Π. This gives a tighter upper bound $f(\Pi) \le (\#\text{EDBs}\cdot\text{ear}\cdot\text{ar}(q))^{\text{ar}(q)}$.

Below, we show multiple examples, in increasing order of generality, to illustrate the correctness and tightness of the bounds.

Example 5.15. Consider the following datalog program. We have an EDB relation e with tuples $e(1,2)$ and $e(3,4)$. And we have the following rules:

$$R_1: q(x,y,z) \leftarrow e(x,y), e(z,w) \qquad R_2: q(x,y,z) \leftarrow e(x,y), e(w,z)$$
$$R_3: q(x,y,z) \leftarrow e(x,w), e(y,z) \qquad R_4: q(x,y,z) \leftarrow e(w,x), e(y,z)$$
$$R_5: q(x,y,z) \leftarrow q(y,z,x) \qquad R_6: q(x,y,z) \leftarrow q(y,x,z) \qquad R_7: q(x,x,z) \leftarrow q(x,y,z)$$

This program has only 1 EDB relation e with arity 2 and one IDB relation q with arity 3. The edge-cover width of the program is 2. In this particular case, q will have all tuples (x,y,z) where each of the three values x,y,z can take any value from the domain $\{1,2,3,4\}$—Rules R_1 to R_4 takes three arbitrary values, and rules R_5 to R_7 generate all permutations that also allow duplicates. So the size of q is $4^3 = 64$. The bound (1) gives $|q| \le 2^3\cdot\sum_{k=1}^{2} S(3,k)\cdot P(1\cdot 2,k) = 8\cdot(1\times 2+3\times 2) = 64$, which is exactly tight here. Nevertheless, if we use the relaxed version (2), we get $|q| \le (\#\text{EDBs}\cdot\text{ear}\cdot\text{ar}(q))^{\text{ar}(q)}\cdot N^{\text{ew}(q)} = (1\times 2\times 3)^3\times 2^2 = 864$, which is still correct but overcounts.

Example 5.16. We slightly extend Example 5.15. We keep the datalog program as-is, but the EDB relation e now has three tuples: $e(1,2)$, $e(3,4)$, and $e(5,6)$. How many tuples does q generate then?

We start with cases where all the three attributes (x,y,z) come from the same tuple in e. This is the case when, e.g., we generate $q(1,2,1)$ from $e(1,2)$ and $e(1,2)$ using the rule R_1. For each such tuple (a,b) in e, it can be verified that q will indeed contain all tuples (x,y,z) such that each value

$x, y, z \in \{a, b\}$. Since no value overlaps in the EDB relation e, we have a total of $3 \times 2^3 = 24$ tuples formed this way.

Now we continue to the tuples (x, y, z) in q where there are actually two different tuples in e that contribute some value. There will be one tuple (a, b) in e that gives two values and one tuple (c, d) that gives one value to this tuple in q. So we (1) choose the tuple (a, b) that gives two values (3 choices), (2) choose the tuple (c, d) that gives one value (2 choices), (3) choose one of the three positions in the tuple (x, y, z) in q that has a value coming from (c, d) (3 choices, x or y or z), (4) choose the value to fill in the position we chose in step (3) (2 choices, c or d), and finally (5) choose the values of the remaining two positions ($2 \times 2 = 4$ choices, as each can take either a or b, and duplicates are allowed). So we have $3 \times 2 \times 3 \times 2 \times 4 = 144$ tuples this way.

In total, we get $24 + 144 = 168$ tuples in q in this program.

Now, if we use the size bound (1):

$$|q| \le 2^3 \cdot \sum_{k=1}^{2} S(3,k) \cdot P(1 \times 3, k) = 8(S(3,1) \cdot P(3,1) + S(3,2) \cdot P(3,2)) = 8 \times (1 \times 3 + 3 \times 6) = 168,$$

which again is exactly tight.

If we use the relaxed version (2), we get

$$|q| \le (\#\mathsf{EDBs} \cdot \mathsf{ear} \cdot \mathsf{ar}(q))^{\mathsf{ar}(q)} \cdot N^{\mathsf{ew}(q)} = (1 \times 2 \times 3)^3 \times 3^2 = 1944,$$

which again is correct but overcounts.

Example 5.17. We can further generalize this example. In fact, given any edge-cover width $\mathsf{ew}(q) = \omega$, IDB arity $\mathsf{iar} = \mu$, EDB arity $\mathsf{ear} = \nu$, number of EDB relations $\#\mathsf{EDBs} = m$, and maximum data size N in each EDB relation, we can generate a datalog program that matches exactly the size bound (1), which shows its tightness.

We start by creating m EDB relations $e_1, \ldots, e_m$, each with arity ν. In each EDB relation e_i, we add N data tuples

$$\begin{array}{llll}
(N\nu(i-1)+1, & N\nu(i-1)+2, & \ldots, & N\nu(i-1)+\nu), \\
(N\nu(i-1)+\nu+1, & N\nu(i-1)+\nu+2, & \ldots, & N\nu(i-1)+2\nu), \\
\vdots & & & \\
(N\nu(i-1)+(N-1)\nu+1, & N\nu(i-1)+(N-1)\nu+2, & \ldots, & N\nu),
\end{array}$$

where each value occurs in exactly one attribute of exactly one tuple in exactly one EDB relation. There is no duplicate value. For example, in e_1, we have $(1, 2, \ldots, \nu)$, $(\nu+1, \nu+2, \ldots, 2\nu)$, $\ldots$, $((N-1)\nu+1, (N-1)\nu+2, \ldots, N\nu)$. In e_2, we start with $(N\nu+1, N\nu+2, \ldots, (N+1)\nu)$, etc.

Then, for each $k = 1, \ldots, \omega$, for each $(e_{i_1}, \ldots, e_{i_k})$ where each $e_{i_\ell} \in \{e_1, \ldots, e_m\}$, for each $(x_{j_1}, \ldots, x_{j_\mu})$ where each $x_{j_\ell} \in \{x_1, \ldots, x_{k\nu}\}$, we add a rule

$$q(x_{j_1}, \ldots, x_{j_\mu}) \leftarrow e_{i_1}(x_1, \ldots, x_\nu), e_{i_2}(x_{\nu+1}, \ldots, x_{2\nu}), \ldots, e_{i_k}(x_{(k-1)\nu+1}, \ldots, x_{k\nu}).$$

In essence, this rule chooses k EDB atoms to form an IDB atom in q, using μ values from these k EDB atoms (duplicates allowed). It should be clear that this program has exactly the edge cover width ω. Since all attributes in all tuples in all EDB relations are distinct, the number of tuples that can be generated is exactly the bound (1), by the same argument as in the proof of Theorem 5.14.

To conclude this section, we would like to highlight that, even though the coefficient $f(\Pi)$ appears to have a large upper bound (in Theorems 5.12 and 5.14), we expect the exact $f(\Pi)$ to be small for most practical datalog programs that are not artificially made to be adversarial in this respect (e.g., Examples 5.15–5.17, which contain multiple shifting, swapping, and duplication of

variables). It is small for, e.g., many classic examples such as transitive closure (Example 1.1, for which $f(\Pi) = 2$), Example 4.6 from [2, Example 12.5.6], and same generation [2, Section 13.4]. Furthermore, in real-world scenarios, we very often need to assign types to attributes, and each attribute has a semantic meaning. In such cases, variables have very limited possibilities to shift or swap. Therefore, the actual number of adornments tends to be small, and this theoretical worst-case upper bound seems, in general, a gross over-estimation.

6 Minimizing EDB-Bounded Datalog Programs

In the previous section, we have seen that, given a datalog program Π, if we simply follow Algorithm 1 to construct an equivalent EDB-bounded program, the number of adornments of each predicate q (the coefficient in Theorem 5.12) can actually be much larger than the coefficient in Theorem 5.14. In this section, we propose a way to *minimize* the EDB-bounded datalog program to obtain an equivalent EDB-bounded datalog program with far fewer adornments and rules, so that the number of adornments per IDB predicate matches the coefficient in Theorem 5.14.

We first note that the large number of adornments is partially because the same variable may appear in multiple EDB atoms in the adornment. Therefore, to minimize, we enforce that each variable appears exactly once in the body of the adornment.

Definition 6.1 (minimal adornment and minimal EDB-bounded program). A *minimal* adornment is an adornment in which every variable v in the head of the rule appears in exactly one atom of the body of the adornment. No atom in the adornment consists solely of underscores.

A *minimal* EDB-bounded datalog program Π'' is an EDB-bounded datalog program for which every rule has a minimal adornment.

Recall the adornment relaxation function g_{min} mentioned in Example 3.8, which retains a minimal set of atoms required to cover the head of a rule and preserves only one occurrence of each variable in the body. Assuming that, given an adornment, there is a consistent, deterministic way to choose which atom the single occurrence of each variable should be in, we show that applying g_{min} on an EDB-bounded program yields an equivalent program with minimal integral edge-cover width.

Theorem 6.2. *Given an EDB-bounded datalog program Π'. Let Π'' be the program constructed by applying g_{min} on every adornment of Π', then Π'' is minimal and equivalent to Π'. Furthermore, Π'' remains an EDB-bounded datalog program such that the integral edge-cover width $\mathsf{ew}(\Pi'') = \mathsf{ew}(\Pi')$.*

Proof. By definition of g_{min}, every adornment in Π'' is minimal. It is established that renaming invented predicates results in a new program that contains the original one [42]. Thus, $\Pi' \subseteq \Pi''$. Similarly to the proof of Theorem 3.13, restoring the original program Π from Π'' is possible by renaming all predicates to their non-adorned version. Therefore, $\Pi' \subseteq \Pi'' \subseteq \Pi$. Since Π' is equivalent to Π, it follows that Π'' is equivalent to both Π' and Π. Finally, by construction, Π'' is an EDB-bounded datalog program, since every head adornment is further relaxed to an adornment that still subsumes the rule. The *integral* edge-cover width of the output program cannot change from $\mathsf{ew}(\Pi')$ since g_{min} only keeps the minimal sets of EDB atoms needed for the cover. □

Example 6.3. The following is a minimal EDB-bounded program equivalent to Π in Example 5.1. It has the same minimal integral edge-cover width of 2.

$$p_{p(x,y,z)\leftarrow e(x,y,\_),e(\_,z,\_)}(x,y,z) \leftarrow e(x,y,w), e(x,z,w), e(y,z,w),$$
$$q_{q(x,y)\leftarrow e(x,y,\_)}(x,y) \leftarrow p_{p(x,y,z)\leftarrow e(x,y,\_),e(\_,z,\_)}(x,y,\_).$$

Note, however, that this process of removing duplicate occurrences may change the *fractional* edge-cover width. In this example, the adornments in Example 5.1 gives a fractional edge-cover width of 1.5, whereas the minimal program above has a fractional edge-cover width of 2.

We then give the following upper bound of the number of adornments in a minimal EDB-bounded program, which now matches the coefficient $f(\Pi)$ in bound (2) of Theorem 5.14.

Theorem 6.4. *Given a datalog program Π, let Π'' be the minimal equivalent EDB-bounded datalog program obtained by applying Algorithm 1 with g_{out} and h_{eq} followed by minimization with g_{min}. Then, for each IDB relation q in Π, the number of IDB relations of the form q_ρ in Π'' is bounded by*

$$\sum_{k=1}^{\text{ew}(q)} (S(\text{ar}(q), k) \cdot \#\text{EDBs}^k \cdot \text{ear}^{\text{ar}(q)}) \leq (\#\text{EDBs} \cdot \text{ear} \cdot \text{ar}(q))^{\text{ar}(q)} = f(\Pi).$$

7 FPT Complexity Bounds of Evaluation

In this section, we use EDB adornments and the resulting size bounds to provide fixed-parameter tractable complexity bounds for datalog evaluation. In addition to the size bounds that we have shown above, we also need some width notions from hypertree decompositions, as we will define next, to reason about the complexity of evaluation.

We start by defining the hypergraphs associated with datalog rules, which are analogous to the case for conjunctive queries as in [16].

Definition 7.1 (hypergraphs associated with datalog rules). Given a datalog program Π, for each rule $R : q(\bar{s}) \leftarrow p_1(\bar{t}_1), \ldots, p_\mu(\bar{t}_\mu)$, we define the hypergraph H associated with the rule R as one with hyperedges $E(H) = \{\text{vars}(\bar{t}_i) \mid 1 \leq i \leq \mu\}$ and vertices $V(H) = \cup E(H)$.

The complexity bounds that will follow are based on results of incremental view maintenance, which fundamentally relies on *free-connex* tree decompositions and related width measures [21, 44]:

Definition 7.2 (free-connex generalized hypertree decompositions). A *hypertree* (for a hypergraph H) is a triple $\mathcal{T} = (T, \chi, \lambda)$, where T is a tree, $\chi : V(T) \to 2^{V(H)}$ is a function s.t. $\bigcup_{p \in V(T)} \chi(p) = V(H)$, and $\lambda : V(T) \to 2^{E(H)}$. We call $\mathcal{T}$ a *free-connex generalized hypertree decomposition (FCGHD)* of H if

- (H_1) For every $e \in E(H)$ there exists some $p \in V(T)$ with $e \subseteq \chi(p)$.
- (H_2) For every $v \in V(H)$, the set $\{p \in V(T) \mid v \in \chi(p)\}$ induces a connected subtree of T.
- (H_3) For every $p \in V(T)$, we have $\chi(p) \subseteq V(\lambda(p))$.
- (H_4) There is a connected subtree S of $\mathcal{T}$ such that $\bigcup_{v \in V(S)} \chi(v) = V_{\text{out}}$, i.e., the union of attributes that appear in the χ-labels in S is exactly the output attributes.

(H_2) is generally called the *connectedness condition*, and (H_4) is called the *connex condition*. The set $\chi(p)$ for each tree vertex $p \in V(T)$ is conventionally called a *bag*.

Definition 7.3 ((integral/fractional) free-connex generalized hypertree width). Given a hypertree decomposition $\mathcal{T}$, for each node $p \in V(\mathcal{T})$, let $V_{\text{full}}(p) = \bigcup_{e \in \lambda(p)} e$, i.e., the set of all variables in the hyperedges in the λ-label. We define the hypergraph associated with the node p as $H_p = (V_{\text{full}}(p), \lambda(p))$. The (integral or fractional) *width* $w(\mathcal{T})$ of a hypertree $\mathcal{T} = (T, \chi, \lambda)$ is the maximum (integral or fractional) edge cover width to cover all variables in $V_{\text{full}}(p)$, among all nodes $p \in V(T)$. The (integral or fractional) *free-connex generalized hypertree width*, denoted by $\text{fcghw}(p)$, is the minimal width among all possible free-connex generalized hypertree decompositions $\mathcal{T}$.

Using the notion of free-connex generalized hypertree width, as well as the size bounds given by the edge-cover width that we introduced earlier, we have

Theorem 7.4. *A datalog program Π over a database instance with at most N tuples per EDB relation can be evaluated in time $O((f(\Pi))^{\text{fcghw}(\Pi)} \cdot |\Pi| \cdot N^{\text{ew}(\Pi) \cdot \text{fcghw}(\Pi)})$, where $\text{fcghw}(\Pi)$ is the free-connex generalized fractional hypertree width.*

Proof sketch. We assume a semi-naïve-style evaluation on the program Π and use techniques of incremental view maintenance to evaluate the program. We start by evaluating rules whose body consists of EDB atoms only, simply as conjunctive queries. Then, in each following iteration, for each rule, we update the relation in the head atom based on the delta from the previous iteration. For this, we view the rule again as a conjunctive query, where the relations have sizes at most $f(\Pi) \cdot N^{\mathrm{ew}(\Pi)}$. Using results of incremental view maintenance [3, 44], we can obtain the bound by performing change propagations on a free-connex hypertree decomposition, which leads to a constant amortized delay for result enumeration. □

If we restrict the scope and consider only *simple chain datalog programs*, i.e., those where each rule contains at most two body atoms, the free-connex hypertree width is at most 2, and we therefore obtain the following bound as a corollary.

Corollary 7.5. *A* simple chain datalog program Π *over a database instance with at most N tuples per EDB relation can be evaluated in time $O((f(|\Pi|))^2 \cdot |\Pi| \cdot N^{2\,\mathrm{ew}(\Pi)})$.*

For *linear* datalog programs, i.e., those where each rule has at most an IDB atom in the body, and the rest are all EDB atoms, each with size at most N, we have

Theorem 7.6. *A* linear *datalog program* Π *(where each rule has at most one IDB atom in the body) over a database instance with at most N tuples per EDB relation can be evaluated in time $O(f(\Pi) \cdot |\Pi| \cdot N^{\mathrm{ew}(\Pi)+\mathrm{fcghw}(\Pi)-1})$.*

Proof sketch. The strategy is similar to that of Theorem 7.4. The difference is that, for linear datalog programs, if each EDB relation has size at most N and each IDB relation has size at most $f(\Pi) \cdot N^{\mathrm{ew}(\Pi)}$, then each intermediate relation materialized for each bag of the FCGHD has size at most $f(\Pi) \cdot N^{\mathrm{ew}(\Pi)+\mathrm{fcghw}(\Pi)-1}$, since there are at least $(\mathrm{fcghw}(\Pi) - 1)$ EDB relations each with size at most N, and the remaining one is either IDB or EDB, but the size is at most $f(\Pi) \cdot N^{\mathrm{ew}(\Pi)}$. □

In addition, we present the complexity bound for a new class of programs:

Definition 7.7. A datalog program is called *adornment groundable*, if, for every rule,

(1) every EDB atom must either have a variable that is not shared with any other atom and appears in the rule's head, or every variable in the EDB atom must appear in the rule's head; and
(2) each IDB variables must appear in the head or in some EDB atom.

This definition implies that, for every rule of the equivalent EDB-bounded program, selecting a grounding for the head adornment implies a unique grounding for the entire rule. This class includes classic examples such as transitive closure and reachability.

Theorem 7.8. *An adornment groundable datalog program* Π *over a database instance with at most N tuples per EDB relation can be evaluated in time $O(f(\Pi) \cdot |\Pi| \cdot N^{\mathrm{ew}(\Pi)})$.*

Due to space constraints, we leave more details about this class to the full version of the paper [13].

Acknowledgments

Qichen Wang is supported by the Ministry of Education, Singapore, through the Academic Research Fund Tier 1 grant (RS32/25), and by the Nanyang Technological University Startup Grant. Part of this work was completed while Qichen Wang was at EPFL. Reinhard Pichler is supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013, 10.47379/NXT22018].

References

[1] [n. d.]. CTADL 0.11.2 documentation. https://ctadl.readthedocs.io/en/latest/

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison Wesley, Reading, Massachusettes.

[3] Mahmoud Abo Khamis, Ahmet Kara, Dan Olteanu, and Dan Suciu. 2024. Insert-Only versus Insert-Delete in Dynamic Query Evaluation. *Proceedings of the ACM on Management of Data* 2, 5 (Nov. 2024), 1–26. doi:10.1145/3695837

[4] Mahmoud Abo Khamis, Vasileios Nakos, Dan Olteanu, and Dan Suciu. 2024. Join Size Bounds using ℓ_p-Norms on Degree Sequences. *Proceedings of the ACM on Management of Data* 2, 2 (May 2024), 1–24. doi:10.1145/3651597

[5] Samuel Arch, Xiaowen Hu, David Zhao, Pavle Subotić, and Bernhard Scholz. 2022. Building a Join Optimizer for Soufflé. In *Logic-Based Program Synthesis and Transformation*, Alicia Villanueva (Ed.). Vol. 13474. Springer International Publishing, Cham, 83–102. doi:10.1007/978-3-031-16767-6_5 Series Title: Lecture Notes in Computer Science.

[6] Molham Aref, Paolo Guagliardo, George Kastrinis, Leonid Libkin, Victor Marsault, Wim Martens, Mary McGrath, Filip Murlak, Nathaniel Nystrom, Liat Peterfreund, Allison Rogers, Cristina Sirangelo, Domagoj Vrgoc, David Zhao, and Abdul Zreika. 2025. Rel: A Programming Language for Relational Data. In *Companion of the 2025 International Conference on Management of Data*. 283–296. doi:10.1145/3722212.3724450 arXiv:2504.10323 [cs].

[7] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1371–1382. doi:10.1145/2723372.2742796

[8] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* (Aug. 2013). doi:10.1137/110859440 Publisher: Society for Industrial and Applied Mathematics.

[9] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog system: datalog-based reasoning for knowledge graphs. *Proceedings of the VLDB Endowment* 11, 9 (May 2018), 975–987. doi:10.14778/3213880.3213888

[10] Marta J. Burzańska, Piotr Wiśniewski, and Krzysztof Stencel. 2024. Recursive Queries: Twenty-Five Years After SQL:1999. In *2024 IEEE International Conference on Big Data (BigData)*. 8272–8279. doi:10.1109/BigData62323.2024.10825088 ISSN: 2573-2978.

[11] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam Netherlands, 18–35. doi:10.1145/3299869.3319894

[12] Shaleen Deep, Hangdong Zhao, Austen Z. Fan, and Paraschos Koutris. 2024. Output-sensitive Conjunctive Query Evaluation. *Proceedings of the ACM on Management of Data* 2, 5 (Nov. 2024), 1–24. doi:10.1145/3695838

[13] Christian Fattebert, Zhekai Jiang, Christoph Koch, Reinhard Pichler, and Qichen Wang. 2026. Size Bound–Adorned Datalog. arXiv:2603.15425 [cs.DB] https://arxiv.org/abs/2603.15425

[14] International Organization for Standardization (ISO). 1999. Information technology – Database languages – SQL Part 1: Framework (SQL/Framework). https://www.iso.org/standard/26196.html

[15] Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. 2012. Size and Treewidth Bounds for Conjunctive Queries. *J. ACM* 59, 3 (June 2012), 1–35. doi:10.1145/2220357.2220363

[16] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree Decompositions and Tractable Queries. *J. Comput. System Sci.* 64, 3 (May 2002), 579–627. doi:10.1006/jcss.2001.1809

[17] Anna Herlihy, Anastasia Ailamaki, and Martin Odersky. 2025. Static Typing Meets Adaptive Optimization: A Unified Approach to Recursive Queries. In *Proceedings of the 19th International Symposium on Database Programming Languages*. ACM, Berlin Germany, 1–6. doi:10.1145/3735106.3736533

[18] Anna Herlihy, Guillaume Martres, Anastasia Ailamaki, and Martin Odersky. 2024. Adaptive Recursive Query Optimization. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 368–381. doi:10.1109/ICDE60146.2024.00035

[19] Gerd G. Hillebrand, Paris C. Kanellakis, Harry G. Mairson, and Moshe Y. Vardi. 1995. Undecidable Boundedness Problems for Datalog Programs. *J. Log. Program.* 25, 2 (1995), 163–190. doi:10.1016/0743-1066(95)00051-K

[20] Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. 2011. μZ – An Efficient Engine for Fixed Points with Constraints. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Vol. 6806. Springer Berlin Heidelberg, Berlin, Heidelberg, 457–462. doi:10.1007/978-3-642-22110-1_36 Series Title: Lecture Notes in Computer Science.

[21] Xiao Hu. 2025. Output-Optimal Algorithms for Join-Aggregate Queries. *Proceedings of the ACM on Management of Data* 3, 2 (June 2025), 1–27. doi:10.1145/3725241

[22] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Vol. 9780. Springer International Publishing, Cham, 422–430. doi:10.1007/978-3-319-41540-6_23 Series Title: Lecture Notes in Computer Science.

[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (Nov. 2015), 204–215. doi:10.14778/2850583.2850594

[24] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic* 7, 3 (July 2006), 499–562. doi:10.1145/1149114.1149117

[25] Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (1982), 258–282. doi:10.1145/357162.357169

[26] Kristóf Marussy, Attila Ficsor, Oszkár Semeráth, and Dániel Varró. 2024. Refinery: Graph Solver as a Service: Refinement-based Generation and Analysis of Consistent Models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. ACM, Lisbon Portugal, 64–68. doi:10.1145/3639478.3640045

[27] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 539–552. doi:10.1145/1376616.1376672

[28] J. Scott Provan and Michael O. Ball. 1983. The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected. *SIAM J. Comput.* 12, 4 (Nov. 1983), 777–788. doi:10.1137/0212053

[29] Yehoshua Sagiv and Mihalis Yannakakis. 1980. Equivalences Among Relational Expressions with the Union and Difference Operators. *J. ACM* 27, 4 (Oct. 1980), 633–655. doi:10.1145/322217.322221

[30] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, Barcelona Spain, 196–206. doi:10.1145/2892208.2892226

[31] Maximilian Emanuel Schüle, Alfons Kemper, and Thomas Neumann. 2022. Recursive SQL for Data Mining. In *Proceedings of the 34th International Conference on Scientific and Statistical Database Management (SSDBM '22)*. Association for Computing Machinery, New York, NY, USA, 1–4. doi:10.1145/3538712.3538746

[32] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2022. Recursive SQL and GPU-support for in-database machine learning. *Distributed and Parallel Databases* 40, 2 (Sept. 2022), 205–259. doi:10.1007/s10619-022-07417-7

[33] Patricia Griffiths Selinger, Morton Michael Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas Gordon Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data (SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. doi:10.1145/582095.582099

[34] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. 2013. Distributed socialite: a datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment* 6, 14 (Sept. 2013), 1906–1917. doi:10.14778/2556549.2556572

[35] Srinivasan Seshadri and Jeffrey F. Naughton. 1995. On the Expected Size of Recursive Datalog Queries. *J. Comput. System Sci.* 51, 2 (Oct. 1995), 137–148. doi:10.1006/jcss.1995.1057

[36] Qingkai Shi, Xiaoheng Xie, Xianjin Fu, Peng Di, Huawei Li, Ang Zhou, and Gang Fan. 2025. Datalog-Based Language-Agnostic Change Impact Analysis for Microservices. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 78–89. doi:10.1109/ICSE55347.2025.00115 ISSN: 1558-1225.

[37] Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. 2013. Graph queries in a next-generation Datalog system. *Proceedings of the VLDB Endowment* 6, 12 (Aug. 2013), 1258–1261. doi:10.14778/2536274.2536290

[38] Oded Shmueli. 1993. Equivalence of DATALOG Queries is Undecidable. *J. Log. Program.* 15, 3 (1993), 231–241. doi:10.1016/0743-1066(93)90040-N

[39] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded*, Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Vol. 6702. Springer Berlin Heidelberg, Berlin, Heidelberg, 245–251. doi:10.1007/978-3-642-24206-9_14 Series Title: Lecture Notes in Computer Science.

[40] Mihail Stoian and Andreas Kipf. 2024. DPconv: Super-Polynomially Faster Join Ordering. *Proc. ACM Manag. Data* 2, 6 (Dec. 2024), 234:1–234:26. doi:10.1145/3698809

[41] Terrance Swift and David S. Warren. 2012. XSB: Extending Prolog with tabled logic programming. *Theory Pract. Log. Program.* 12, 1-2 (Jan. 2012), 157–187. doi:10.1017/S1471068411000500

[42] Aalok Thakkar, Nathaniel Sands, George Petrou, Rajeev Alur, Mayur Naik, and Mukund Raghothaman. 2023. Mobius: Synthesizing Relational Queries with Recursive and Invented Predicates. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 271 (Oct. 2023), 24 pages. doi:10.1145/3622847

[43] Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. 2025. Yannakakis+: Practical Acyclic Query Evaluation with Theoretical Guarantees. *Proc. ACM Manag. Data* 3, 3 (June 2025), 28. doi:10.1145/3725423

[44] Qichen Wang, Xiao Hu, Binyang Dai, and Ke Yi. 2023. Change Propagation Without Joins. *Proceedings of the VLDB Endowment* 16, 5 (Jan. 2023), 1046–1058. doi:10.14778/3579075.3579080

[45] Xiuheng Wu, Chenguang Zhu, and Yi Li. 2021. DIFFBASE: a differential factbase for effective software evolution management. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium*

on the Foundations of Software Engineering. ACM, Athens Greece, 503–515. doi:10.1145/3468264.3468605

[46] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB '81: Proceedings of the seventh international conference on Very Large Data Bases*, Vol. 7. 82–94.

[47] Haozhe Zhang, Christoph Mayer, Mahmoud Abo Khamis, Dan Olteanu, and Dan Suciu. 2025. LpBound: Pessimistic Cardinality Estimation Using ℓ_p-Norms of Degree Sequences. *Proceedings of the ACM on Management of Data* 3, 3 (June 2025), 1–27. doi:10.1145/3725321

[48] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 468–492. doi:10.1145/3591239

Received December 2025; accepted February 2025