Universal Finiteness and Satisfiability

(Extended Abstract)

Inderpal Singh Mumick
AT&T Bell Laboratories
mumick@research.att.com

Oded Shmueli
Technion
oshmu@cs.Technion.ac.il

Abstract

The problem of determining whether, for every extensional database, a given predicate in a given program has a finite number of derivations is called the universal finiteness problem. The problem of determining whether a given predicate in a given program has a non-empty extension for some extensional database is called the satisfiability problem. We show that the universal finiteness problem can be reduced to the satisfiability problem. Thus all decidability results for satisfiability can be applied to universal finiteness - for example, we can infer that the universal finiteness problem is decidable for Datalog extended with negation on base predicates. The satisfiability problem can be easily reduced to the universal finiteness problem, so that all undecidability results for satisfiability can be applied to universal finiteness. For example we can infer that the universal finiteness problem is undecidable for Datalog extended with stratified negation.

Many recursive programs have infinite number of derivations only when edb relations have data cycles. It is thus of particular interest to study universal finiteness in the presence of acyclicity constraints on the edb relations. We define acyclicity constraints in terms of non-satisfiability of a specific recursive program. We show that both the problems of universal finiteness and satisfiability of Datalog in the presence of acyclicity constraints (on one or more edb relations) remain decidable for a language L whenever the problems are decidable for language L in absence of such constraints. We also show that the problems are undecidable for arbitrary constraints expressed in terms of non-satisfiability of a recursive program.

1 Introduction

1.1 Motivation

The query language SQL works with tables, or multisets. The number of copies of a tuple is important. The duplicate semantics of an SQL query has been defined by counting the number of derivation trees for each fact derived by the query. SQL is being extended with recursion [ISO93], and the duplicate semantics of recursive SQL queries has also been defined by counting the number of derivation trees [MPR90, Mum91, MS93]. Recursive SQL queries are very similar to Datalog with an extension to keep track of the number of derivations for each tuple. As the following example shows, the number of derivation trees of a fact derived by a recursive SQL or Datalog program may be infinite.

EXAMPLE 1.1 (Path Query) We are given an $\mathtt{edge}(X, Y)$ relation, and need to find the number of paths between every pair of nodes X and Y. A $\mathtt{path}(X, Y)$ table can be defined in SQL extended with recursion (we use the syntax of Starburst [MFPR90]) as follows:

```
(N):  CREATE VIEW path(X,Y) AS
          ((SELECT X, Y FROM edge)
           UNION ALL
           (SELECT e.X, p.Y FROM edge e, path p
            WHERE e.Y = p.X)).

(Q):  SELECT X, Y, COUNT (*) FROM path GROUPBY X, Y.
```

The `path` view in SQL is equivalent to the following definition in Datalog if we interpret `path` as a multiset predicate.

$(N1)$: $\mathtt{path}(X, Y)$:- $\mathtt{edge}(X, Y)$.
$(N2)$: $\mathtt{path}(X, Y)$:- $\mathtt{edge}(X, Z)$ & $\mathtt{path}(Z, Y)$.

If the `edge` relation has a cycle then the number of derivation trees for some `path` fact will be infinite. For instance, given $\mathtt{edge} = \{(1,2),(2,3),(3,2)\}$, the fact $\mathtt{path}(1,3)$ has an infinite number of derivation trees. Duplicate semantics requires that the fact $\mathtt{path}(1,3)$ be derived with a multiplicity of *infinity*, so that the query Q derives the fact $(1, 3, \infty)$. □

Given a predicate p in program P, the problem of determining whether, for all possible edb relations, the number of derivation trees for all tuples of predicate p is finite is called the *universal finiteness problem.* The universal finiteness problem is an important problem for query evaluation in commercial database systems, and for certain advanced applications. A solution would mean that (1) We can determine whether a given recursive query has a finite answer under multiset semantics to be used in commercial database systems based on SQL3 [ISO93]. The results are also important in defining a multiset semantics for SQL extended with recursion. (2) We can determine whether an evaluation method that does not eliminate duplicates (such as not-so-naive [MR89] or duplicate semi-naive [Mum91]) will always terminate. (3) Incremental view maintenance algorithms [GKM92, GMS93] can rely on the number of derivations. The applicability of such algorithms can be determined by checking the universal finiteness property.

This paper identifies several classes of programs for which the universal finiteness problem is solvable, and other classes for which it is undecidable. We relate the universal finiteness problem to the well known *satisfiability problem* [Shm87, CGKV88, KKR90, vdM92, LMSS93]: Given a predicate p in program P, is there an edb for which there exists a derivation for a tuple (fact) of predicate p.

Acyclicity Constraits: In Example 1.1, if the given `edge` relation is acyclic, the number of derivation trees of `path` will be finite. Acyclicity constraints are very important in practical applications such as bill-of-materials. Further, several common recursive programs, such as the transitive closure family of programs, have a finite number of derivations when the edb relations are restricted to be acyclic, but have an infinite number of derivations under cyclic edb relations. It is thus important to consider the universal finiteness problem in the presence of acyclicity constraints on the edb relations. We show that if the satisfiability problem, and thus the universal finiteness problem, are decidable for a language L, then both problems remain decidable for language L in the presence of acyclicity constraints.

We also consider more general constraints on edb relations that can be expressed in terms of the non-satisfiability of an idb predicate. We show that the satisfiability and universal finiteness problems are generally undecidable under such constraints.

1.2 Multiset Semantics

The languages we use are all Datalog derivatives. Rules are required to be safe (i.e. each rule variable must appear in a positive body atom). We consider safe stratified negation. Stratification is done via numbered strata (the edb is at stratum 0).

Definition 1.1 Derivation tree (Informal): Derivation trees are defined inductively over stratum numbers. Each edb atom forms a derivation tree for itself. Derivation trees of stratum $i+1$ are obtained from rules of stratum $i+1$, and derivation trees of stratum i or less. If all the non-negated idb and edb subgoals in the body of a rule can be simultaneously unified with roots of already constructed trees so that all constraints, if any, are satisfied, and so that no derivation tree was generated earlier for a negative subgoal, then a new derivation tree is generated for the ground head atom of the rule, with the head atom as the root, and the trees whose roots unified with the subgoals as children of the root. □

A *relation* is a set of tuples, where each tuple occurs exactly once. A *table* is a multiset of tuples, where each tuple can occur multiple times. The multiple copies of a tuple in a table are called *duplicates*. A database system that supports tables is said to support duplicates. We now define a variant of Datalog that uses duplicates.

Definition 1.2 DDatalog: Duplicate Datalog, or DDatalog, is safe Datalog with all idb predicates interpreted as multisets, and all edb predicates interpreted as sets. The multiplicity of each tuple derived in DDatalog is important. The semantics for DDatalog is as defined in [Mum91, MS93]. Briefly, each distinct derivation tree for a fact adds one to its multiplicity in the derived set. □

The formal duplicate semantics can be understood as follows: if duplicates are not eliminated when evaluating rules defining a predicate, say p, then p may have several occurrences of a given tuple, say t. Thus, p is a multiset of tuples, rather than a set. The corresponding set of tuples in p is easily understood by ignoring duplicates; the subtle part is in defining the multiplicity of each tuple in the multiset p. Intuitively, a tuple t appears in the multiset as often as it is *derived*. The number of times that a tuple is derived in SQL is not arbitrary: a tuple is derived exactly once for each distinct *derivation tree* that supports it.

Outline: We start with definitions of the finiteness, universal finiteness, and the satisfiability problems in Section 2. Then, in Section 3, we reduce the universal finiteness problem to the satisfiability problem. The reduction enables us to infer decidability of the universal finiteness problem for all cases where satisfiability is known to be decidable. Section 4 introduces acyclicity constraints, and argues that the universal finiteness problem in the presence of acyclic constraints is an important problem. For languages for which the satisfiability problem is decidable, both the satisfiability and the universal finiteness problems are shown to be decidable in the presence of acyclicity constraints on the database. The universal finiteness problem is related to the boundedness problem in Section 5, and we conclude in Section 6. Related work also is discussed in Section 6.

2 The Basics

In the following definitions, we use language L to mean a language that is obtained from Datalog by various extensions and restrictions. We only consider finite edbs E.

2.1 Finiteness

Given a specific edb, we would like to check whether all tuples of a predicate have a finite number of derivations.

Definition 2.1 Finiteness Problem for a Language L [MS93]: Given a program P in language L, a distinguished predicate p, and an instance E of the extensional database, does every tuple of p derivable from edb E have a finite number of derivation trees? □

Lemma 2.1 *[MS93] The finiteness problem is decidable, in polynomial time, for Datalog extended with safe stratified negation, constants, and inequalities ($\leq, <, >, \geq, \neq$).* □

2.2 Universal Finiteness

The finiteness property of a program depends upon the edb; hence it cannot usually be checked at compile time. Further, the run time check of [MS93] involves as much work as computing the program. Thus, the finiteness property cannot be used at compile time for optimization of the program. One would really like to solve a universal finiteness problem: check whether a program is guaranteed to have a finite number of derivation trees for all possible edbs.

Definition 2.2 Universal Finiteness Problem for a Language L [MS93]: Given a program P in language L, with a distinguished predicate q, is it the case that for all possible extensional databases E, every derivable tuple of q has a finite number of derivation trees?

If every tuple of q does have a finite number of derivation trees, then the program (P, q) is said to have the universal finiteness property. □

EXAMPLE 2.1 The `path` program N of Example 1.1 does not have a finite number of derivation trees for the edb $E = \{(1,2),(2,1)\}$. Thus Program N does not have the universal finiteness property. □

2.3 Satisfiability

Definition 2.3 Satisfiability Problem for a Language L: Given an idb predicate p of a program P in language L, does there exist an edb E such that P defines a nonempty relation for predicate p?

If such an edb does exist, then p is said to be satisfiable in program P. □

Satisfiability is well recognized as being important for query optimization. We summarize the known decidability/undecidability results for various extensions of Datalog in Table 1.

Lemma 2.2 *The satisfiability problem can be reduced to the universal finiteness problem.* □

Proof: Given the satisfiability problem instance T: Is (Q, q) satisfiable, derive program P by adding the following rule to program Q:

$$q(X_1, \ldots, X_n) \text{ :- } q(X_1, \ldots, X_n).$$

Then, (P, q) has a finite number of derivation trees iff (Q, q) is unsatisfiable. □

From Lemma 2.2 it follows that each undecidable case (extension or restriction) for the satisfiability problem (as listed in Table 1) induces undecidability for that case for the universal finiteness problem.

3 Universal Finiteness

We wish to check whether a given program can have an infinite number of derivation trees for some edb E. Lemma 2.2 implies that such a check is not possible for programs with most types of stratified negation. What about programs without negation? What about programs where negation is applied only to edb relations? In this section we will first show that universal finiteness can be reduced to satisfiability; then, from Table 1 it follows that universal finiteness can be checked for the above classes of programs.

3.1 Reducing universal finiteness to satisfiability

Let P be the original program. Let the distinguished query predicate be q.

We make the following basic observation. If there is a database D on which there are infinitely many proof trees over P, then there is a derivation tree T over D and some n-ary predicate symbol, say p, and constants $c_1, \ldots, c_n$ such that fact $p(c_1, \ldots, c_n)$ repeats in T along a path from the root to a leaf.

Conversely, once we have a derivation tree T for q in P in which fact $p(c_1, \ldots, c_n)$ repeats, we can generate infinitely many derivation trees from T by cutting and pasting at $p(c_1, \ldots, c_n)$. Here, we cut and paste trees having constants and so no contradictions may arise.

Lemma 3.1 *Consider safe stratified Datalog. A predicate q in program P has an infinite number of derivation trees for q on a finite edb E if and only if there exists a predicate p, and a derivation tree T for q, such that a tuple $p(a_1, \ldots, a_m)$ occurs at least twice along a path from the root of tree T to a leaf of tree T.* □

Datalog Extension/Restriction	Satisfiability Problem
Unary idb's, negation on nonrecursive predicates, 1 recursive rule, $\neq$	Undecidable [LMSS93]
Unary recursive idb's, binary nonrecursive idbs, negation on nonrecursive predicates, 1 recursive rule, no $\neq$	Undecidable [LMSS93]
Dense order constraints	Decidable [KKR90, LS92]
Dense order constraints, negation on edbs	Decidable [LMSS93]
Unary edb's, stratified negation	Decidable [LMSS93]
$\neq$, functional dependencies in edb	Undecidable [LMSS93]

Table 1: Known Results for Satisfiability.

Thus, the universal finiteness problem is to determine if there is some fact $p(a_1, \ldots, a_n)$ that repeats in some derivation tree for q in P over some edb. So, for each idb predicate p in P we check whether a derivation tree for q in P can be generated with two identical occurrences of a p fact. The checking is done by modifying the program P into another program P' such that program P' is satisfiable if and only if p repeats itself in a derivation tree for q. In modifying P we enlarge the arities of all idb predicates.

Let us use an example to illustrate the technique. Consider the following program:

Stratum 1:

$(P1)$: $r(X,Y) \coloneq u(Y) \,\&\, v(X,Y).$

Stratum 2:

$(P2)$: $p(X,Y) \coloneq r(X,Y) \,\&\, \neg u(X).$

$(P3)$: $p(X,Y) \coloneq p(X,Z) \,\&\, p(Z,Y) \,\&\, r(W,Y).$

Stratum 3: (Note that we increase stratum numbers even in the absence of negation)

$(P4)$: $q(X,Y,Z) \coloneq p(X,Z) \,\&\, p(Z,Y).$

$(P5)$: $q(X,Y,Z) \coloneq q(X,Z,W) \,\&\, q(W,Y,Z).$

q is the query predicate. u and v are edb predicates. r, p, and q are idb predicates in strata 1, 2, and 3 respectively. When we are considering repetition of a p ground atom in the derivation tree for a q ground atom, we will map predicate p and each predicate in a higher stratum (in this case, predicate q) onto predicates p' and q'. Each of the new predicates has three more attributes than the original predicate. Intuitively, two additional attributes correspond to a tuple of p used in the derivation, and the third place is an indicator, telling us whether or not this tuple has been repeated at least twice in the derivation tree. Thus, $p'(X,Y,U,V,I)$ means that $p(X,Y)$ is derivable, and there exists a derivation tree for $p(X,Y)$ that uses the atom $p(U,V)$ along a path from the root to a leaf. Similarly, $q'(X,Y,Z,U,V,I)$ means that $q(X,Y,Z)$ is derivable, and there exists a derivation tree for $q(X,Y,Z)$ that uses the atom $p(U,V)$ along a path from the root to a leaf. I can only have one of two constant values: If $I =$ 'repeated', then the atom $p(U,V)$ is used at least twice along one path from the root to a leaf in some derivation tree for some fact. If $I =$ 'once', then the atom $p(U,V)$ is used at least once along some path from the root to a leaf in some derivation tree for some fact. A third value $I =$ 'nil' is used to handle special cases. $I =$ 'nil' means that $U = V =$ 'nil', and that no p atom is used to derive the atom $q(X,Y,Z)$.

Suppose that there is only one predicate (p) in the stratum for p. (The case where there are multiple predicates in the stratum for p is discussed later in this subsection.) With each rule in program P defining predicate p we associate (1) one *initializer* rule, (2) g *passing* rules, and (3) g *duplicator* rules, where g is the number of occurrences of predicate p in the body of the rule. Also, with each rule in program P defining predicate q (in general, with each rule for all predicates in strata higher than predicate p), we associate h *passing* rules, where h is the number of occurrences of predicates p and q in the body of the rule. (In general, h would be the number of occurrences of predicates in same or higher strata than p in the body of the rule).

Our example program P is transformed into P' as follows. Predicate r is in a lower stratum than p, so rule $P1$ is left untouched:

$(P'1)$: $r(X,Y) \coloneq u(Y) \,\&\, v(X,Y).$

Next, we transform rules $P2$ and $P3$ for p. One *initializer* rule is created from each of $P2$ and $P3$:

$(P'2_i)$: $p'(X,Y,X,Y,\text{'once'}) \coloneq r(X,Y) \,\&\, \neg u(X).$

$(P'3_i)$: $p'(X,Y,X,Y,\text{'once'}) \coloneq p'(X,Z,U1,V1,I1) \,\&\, p'(Z,Y,U2,V2,I2) \,\&\, r(W,Y).$

Rules $P'2_i$ $P'3_i$ capture the fact that whenever a

$p(X,Y)$ tuple is derived in P, there is a derivation tree for $p(X,Y)$ using $p(X,Y)$ at least once, regardless of what other tuples are being used in the derivation tree.

Rule $P'2_i$ does not generate *passing* or *duplicator* rules since it has no subgoals in the same stratum as p. Rule $P'3_i$ has two subgoals in the same stratum as p, so it generates two *passing* rules and two *duplicator* rules:

$(P'3_{p1})$: $p'(X,Y,U1,V1,I1)$:- $p'(X,Z,U1,V1,I1)$ & $p'(Z,Y,U2,V2,I2)$ & $r(W,Y)$.

$(P'3_{p2})$: $p'(X,Y,U2,V2,I2)$:- $p'(X,Z,U1,V1,I)$ & $p'(Z,Y,U2,V2,I2)$ & $r(W,Y)$.

$(P'3_{d1})$: $p'(X,Y,X,Y,\text{'repeated'})$:- $p'(X,Z,X,Y,\text{'once'})$ & $p'(Z,Y,U2,V2,I2)$ & $r(W,Y)$.

$(P'3_{d2})$: $p'(X,Y,X,Y,\text{'repeated'})$:- $p'(X,Z,U1,V1,I)$ & $p'(Z,Y,X,Y,\text{'once'})$ & $r(W,Y)$.

Rule $P'3_{p1}$ captures the fact that if the fact $p(X,Z)$ is used in deriving a $p(X,Y)$ tuple, then any p fact used in the subtree of $p(X,Z)$ is used at least the same number of times in the derivation tree of $p(X,Y)$. In other words, (1) if there is a derivation tree for $p(X,Z)$ that uses $p(U1,V1)$ at least once along some path, then there is a derivation tree for $p(X,Y)$ that uses $p(U1,V1)$ at least once along some path. And (2), if there is a derivation tree for $p(X,Z)$ that uses $p(U1,V1)$ at least twice along some path, then there is a derivation tree for $p(X,Y)$ that uses $p(U1,V1)$ at least twice along some path. Rule $P'3_{p2}$ is similar.

Rule $P'3_{d1}$ captures the fact that if the fact $p(X,Z)$ is used in deriving a $p(X,Y)$ tuple, and if the fact $p(X,Y)$ is itself used in the subtree of $p(X,Z)$, then the fact $p(X,Y)$ is used at least twice in a path from the root to a leaf in the derivation tree of $p(X,Y)$. Thus, if there is a derivation tree for $p(X,Z)$ that uses $p(X,Y)$ at least once along a path, then there is a derivation tree for $p(X,Y)$ that uses $p(X,Y)$ at least twice. Rule $P'3_{d2}$ is similar.

Next, we transform rules $P4$ and $P5$ in the stratum above p. Each of these rules generates two *passing* rules:

$(P'4_{p1})$: $q'(X,Y,Z,U1,V1,I1)$:- $p'(X,Z,U1,V1,I1)$ & $p'(Z,Y,U2,V2,I2)$.

$(P'4_{p2})$: $q'(X,Y,Z,U2,V2,I2)$:- $p'(X,Z,U1,V1,I1)$ & $p'(Z,Y,U2,V2,I2)$.

$(P'5_{p1})$: $q'(X,Y,Z,U1,V1,I1)$:- $q'(X,Z,W,U1,V1,I1)$ & $q'(W,Y,Z,U2,V2,I2)$.

$(P'5_{p2})$: $q'(X,Y,Z,U2,V2,I2)$:- $q'(X,Z,W,U1,V1,I1)$ & $q'(W,Y,Z,U2,V2,I2)$.

Rule $P'4_{p1}$ captures the fact that if the fact $p(X,Z)$ is used in deriving a $q(X,Y,Z)$ tuple, then any p fact used in the subtree of $p(X,Z)$ is used at least the same number of times in the derivation tree of $q(X,Y,Z)$. Rules $P'4_{p2}$, $P'5_{p1}$, $P'5_{p2}$ are similar.

Finally, a new query predicate h is defined as:

$(P'6)$: $h(X,Y,Z)$:- $q'(X,Y,Z,U,V,\text{'repeated'})$.

We can show that the following properties hold between the modified program P' and the original program P:

Lemma 3.2 *For all edbs E*

1. *If the tuple $p(c,d)$ is used in some derivation tree for $p(a,b)$ in program P on edb E, the tuple $p'(a,b,c,d,\text{'once'})$ is derived in program P' on edb E.*
2. *If a tuple $p(c,d)$ appears twice or more along a root to leaf path in some derivation tree for the tuple $p(a,b)$ in program P on edb E, then the tuple $p'(a,b,c,d,\text{'repeated'})$ is derived in program P' on edb E.*

□

Proof: By induction on the height of the derivation tree for $p(a,b)$. □

Lemma 3.3 *For all edbs E*

1. *If the tuple $p'(a,b,c,d,\text{'once'})$ is derived in program P' on edb E, then the tuple $p(c,d)$ is used in some derivation tree for $p(a,b)$ in program P on edb E.*
2. *If the tuple $p'(a,b,c,d,\text{'repeated'})$ is derived in program P' on edb E, then the tuple $p(c,d)$ appears at least twice on a path from a leaf to the root in some derivation tree for $p(a,b)$ in program P on edb E.*

□

Proof: By induction on the height of the derivation trees for p' tuples. □

Similarly, we can show that (1) a tuple

$$q'(X,Y,Z,U,V,\text{'once'})$$

is derived in program P' iff the tuple $q(X,Y,Z)$ has a derivation tree in program P using the tuple $p(U,V)$ at least once, and (2) a tuple $q'(X,Y,Z,U,V,\text{'repeated'})$ is derived in program P' iff the tuple $q(X,Y,Z)$ has a derivation tree in program P using the tuple $p(U,V)$ at least twice on a path from the root to a leaf. We thus arrive at:

Lemma 3.4 *The predicate h in program P' is satisfiable iff there exists an edb E and constants c,d such that the tuple $p(c,d)$ repeats along a path from the root to a leaf in a derivation tree for predicate q in program P over edb E.* □

Multiple Predicates in one stratum In the example and the description of the reduction from universal finiteness to satisfiability above, each stratum had only one predicate. Now, consider the case where there are multiple predicates in the stratum for p. Then, during the reduction to satisfiability with respect to a predicate p, we associate, with each rule for p, (1) one *initializer* rule, (2) h *passing* rules, and (3) h *duplicator* rules, where h is the number of occurrences of p and predicates in the same stratum as p in the body of the rule.

Also, for each predicate $s \neq p$ defined in the same stratum as p, we associate, with each rule for s, (1) one *initializer* rule, (2) h *passing* rules, and no *duplicator* rules, where h is the number of occurrences of p and predicates in the same stratum as p in the body of the rule. The single *initializer* rule for predicate s is generated with the value $I =$ 'nil', indicating that no p tuple may be assumed to be used in the derivation tree. The arguments corresponding to the p tuple used are also given the value 'nil'.

When strata above the stratum of p have multiple predicates, their transformation is the same as discussed in the example.

Negated Subgoals During the reduction to satisfiability with respect to a predicate p, if a negated subgoal belongs to a stratum below p, it needs no special handling. If a negated predicate s belongs to the stratum of or above p, it must appear in a rule in a stratum above p. In addition to the new predicate s', define the predicate s as a projection of s' onto its original arguments. Then, the transformation is modified so as to use the predicate s in the negated subgoal instead of the predicate s'. No *passing* rules are generated from the negated subgoal occurrences.

3.2 Impact on Universal Finiteness

In the above construction, we checked for derivation trees of q that have a repeated occurrence of a p tuple. To eliminate all possibilities of generating an infinite number of derivation trees, we must also check for derivation trees of q that have a repeated occurrence of a q tuple. In fact, for a program with n idb predicates, n reductions must be done, one with respect to each idb predicate. Each reduced program is checked for satisfiability, and if all the reduced programs are unsatisfiable, then the original program has the universal finiteness property. Since the n instances of satisfiability can be combined into a single satisfiability problem instance, we have:

Theorem 3.1 *Let L be a language that does not restrict the arity of predicates or the number of subgoals in a rule.*
The universal finiteness problem for the language L can be reduced to a single instance of the satisfiability problem for language L. □

Proof: From Lemmas 3.1 and 3.4, and the argument given above. Note that we do not require the language L to include constants. The use of the constants 'nil', 'once', and 'repeated' in the above reductions can be avoided by using specialized predicates p_{nil}, p_{once}, and p_{repeated}. □

Corollary 3.2 *The universal finiteness problem is decidable for the following classes of programs: (1) Datalog with constants, dense order constraints, and safe negation on edb predicates, (2) Datalog with constants, dense order constraints, and stratified negation, provided all edb predicates are restricted to be unary.* □

4 Acyclicity Constraints

The `edge` relation in program N of Example 1.1 represents edges connecting nodes in a graph.

A graph is cyclic if there is a path from some node to itself; otherwise the graph is acyclic. The `edge` relation is said to be acyclic if the graph it represents is acyclic. In fact any binary relation can be viewed as representing a graph, and we can then define an acyclicity condition on the binary relation. We can also generalize the acyclicity condition to an arbitrary relation $u(W_1, \ldots, W_n)$ of arity $n \geq 2$. Consider two distinguished sets of attributes $X_1, \ldots, X_k$ and $Y_1, \ldots, Y_k$ from amongst $W_1, \ldots, W_n$. Now draw a graph as follows: The nodes are k-tuples. For each tuple $(w_1, \ldots, w_n)$ in relation u include nodes $(x_1, \ldots, x_k)$ and $(y_1, \ldots, y_k)$, and draw an edge from $(x_1, \ldots, x_k)$ to $y_1, \ldots, y_k)$. The relation u is acyclic if the graph so constructed is acyclic.

Definition 4.1 Acyclicity Constraint: Let $u(X_1, \ldots, X_k, Y_1, \ldots, Y_k, Z_1, \ldots, Z_m)$ be a given edb relation. Let $s(X_1, \ldots, X_k, Y_1, \ldots, Y_k)$ and q be idb predicates defined as

$s(X_1, \ldots, X_k, Y_1, \ldots, Y_k)$:-
  $u(X_1, \ldots, X_k, Y_1, \ldots, Y_k, Z_1, \ldots, Z_m)$.

$s(X_1, \ldots, X_k, Y_1, \ldots, Y_k)$:-
  $u(X_1, \ldots, X_k, U_1, \ldots, U_k, Z_1, \ldots, Z_m)$ &
  $s(U_1, \ldots, U_k, Y_1, \ldots, Y_k)$.

q :- $s(X_1, \ldots, X_k, X_1, \ldots, X_k)$.

Let the edb relation u be constrained such that, for any legal value of u, the idb predicate q evaluates to false. Then, the relation u is said to satisfy the acyclicity constraint with respect to the partition $(X_1, \ldots, X_k), (Y_1, \ldots, Y_k), (Z_1, \ldots, Z_m)$. □

Acyclicity constraints are very common in practical applications (for example, bill-of-materials), and hence our interest.

4.1 Satisfiability in the Presence of Acyclicity Constraints

Definition 4.2 Acyclic Satisfiability Problem for Language L: Given an idb predicate p of a program P in language L, and given a set of acyclicity constraints on edb relations, does there exist an edb E satisfying the given constraints such that P defines a nonempty relation for predicate p? □

We will derive the following result:

Theorem 4.1 *Let L be a language that includes disequalities ($\neq$). The acyclic satisfiability problem for language L is decidable whenever the satisfiability problem for language L is decidable.* □

The idea is to rewrite the program and test the rewritten program for ordinary satisfiability. Let us take a simple example to explain the ideas. Let program T be:

$(T1)$: $p(X,Y)$:- $u(X,Y)$.
$(T2)$: $p(X,Y)$:- $p(X,Z)$ & $p(Y,Z)$.

The single binary edb relation u is constrained to be acyclic. We shall increase the arity of each idb predicate to reflect reachability information, and as derivations progress we'll insure that no cycles are forced to exist in the edb.

To transform program T, we shall add 4 more argument positions to p and think of them as entries in a 2×2 reachability matrix R (in general, if p had n arguments, we would add n^2 arguments, viewed as an $n \times n$ matrix). Intuitively, $R[1,2] = 1$ says that there is a path from X to Y in the directed graph induced by u, and $R[1,2] = 0$ says that there is no path from X to Y; similarly, $R[2,1] = 1$ says there is a path from Y to X. We'll modify the rules to disallow derivations that would create (in facts for rule heads) new reachability matrices R such that $R[1,1] = 1$ or $R[2,2] = 1$. The information for filling 0 or 1 in each of these new "matrix entries" comes from reachability information in the body of the rule (obtained from the idb predicates) together with the particular equalities and usages of u in the body. For example, from rule $T1$ we derive:

$(T'1)$: $p'(X,Y,XX,XY,YX,YY)$:- $u(X,Y)$ &
$X \neq Y$ & $XX = 0$ & $XY = 1$ & $YY = 0$ &
$YX = 0$.

The deduction of the p' tuple is permitted only if $X \neq Y$, and the derived tuple indicates that $XY = 1$, meaning that a path from X to Y has been forced by the derivation. To understand the transformed rules derived from rule $T2$, first realize that there are 5 cases of equalities between the variables X, Y, and Z in rule $T2$ – (1) $X \neq Y, X \neq Z, Z \neq Y$, (2) $X = Z, X \neq Y$, (3) $Y = Z, X \neq Y$, (4) $X = Y, X \neq Z$, and (5) $X = Y = Z$. Rules deriving p' are written for each of these cases. For example, here is a rule derived from case (1).

$(T'2_{(1)(9)})$: $p'(X,Y,XX,XY,YX,YY)$:-
/* CASE 1 */ $X \neq Y$ & $X \neq Z$ & $Z \neq Y$ &
$p'(X,Z,XX,XZ,ZX,ZZ)$ &
$p'(Y,Z,YY,YZ,ZY,ZZ)$ &
/*MUST HOLD*/ $XX = 0$ & $YY = 0$ & $ZZ = 0$ &
/*SUBCASE 9*/ $XZ = 1$ & $ZX = 0$ & $YZ = 0$ & $ZY = 1$ &
/* DEDUCE*/ $XY = 1$ & $YX = 0$.

The first three subgoals establish case (1). We then consider possible matrix values used in deriving the two p' subgoals. The variable XX indicates the presence of a path from X to X in the derivation of the first p subgoal. Clearly, we don't want this cycle to exist, so we force $XX = 0$. Similarly for $YY = 0$ and $ZZ = 0$. Then, we are left with 16 possibilities for the values of the four variables (XZ, ZX, YZ, ZY). One possibility, corresponding to $(XZ, ZX, YZ, ZY) = (1, 0, 0, 1)$ which when looked upon as a binary number is decimal 9 (called subcase 9) is shown in the rule, and leads to a new p tuple whose derivation tree has a path from X to Y, no path from Y to X, and no cycle. The path information involving Z is forgotten, and is not relevant since Z does not appear in the new p tuple.

As another example, consider a modification of the above rule for the subcase $(XZ, ZX, YZ, ZY) = (1, 0, 1, 1) = 11$. This would mean that the second p subgoal already uses a cycle –a path from Y to Z and a path from Z to Y. Therefore, the rule causes an invalid derivation that must be disallowed. In fact a p' fact matching this subgoal would never be generated. Consequently, this rule is dropped. In general, we never create a rule that would lead to a matrix entry $R[i,i]$ being set to 1.

Note that all possible cases of equalities between variables of a rule, and all subcases of 0's and 1's assigned to matrix positions can be determined at compile time from the form of the u facts being used. Consequently only the new rules that derive a p fact only if it uses an acyclic subset of relation u are written out. What we have done is written a meta-interpreter that checks for acyclicity in base relations as it goes along.

Lemma 4.1 *The transformed program T' obtained as above is satisfiable for predicate p' iff the original program T is satisfiable for predicate p on a database conforming to the acyclicity constraints.* □

Proof: (Sketch) (if) Suppose that T is satisfiable for predicate p on an edb E satisfying the acyclicity constraint. Consider a derivation tree for a p fact. Label each derivation tree node with (1) the rule r used to derive it, and (2) a matrix encoding paths information corresponding to argument positions in the head of r. If the head of r has n arguments, we have an n by n matrix. Matrix entry (i,j) is 1 if there is a path along u facts from the constant in position i to the constant in position j in the part of edb E used in the subtree rooted at the node, and 0 otherwise. Since E is acyclic all diagonal matrix entries in all nodes are 0.

Construct a derivation tree over program T' for the same p' fact. For each node v in the tree, labeled by a rule r in program T, use the rule r' that was derived from r during the construction of program T', such that r' uses (for its head and subgoals) matrices that are compatible with the matrices that label node v and its children in the derivation tree. A matrix R' in rule r' is compatible to a matrix R in the derivation tree if $R' = R$, or if R' has a zero at every position where R has a zero. Since E is acyclic, v and its children all display compatible matrices that faithfully describe the structure of E, hence the rule r' exists and can be used. This proves that T' is satisfiable for p' on D.

(only if) First, suppose that the original program contains no constants. (If it does we can rewrite it so that each idb predicate has an argument for each constant in the program and also a matrix row and column for each such constant.) Suppose the transformed program T' is satisfiable for p' on some edb E. Let t be a derivation tree for a p' fact on E. Abstract this tree by replacing constants with the variables they would match with in a top-down expansion of the p atom that would generate the derivation tree. All existential variables are renamed in every rule application during the top-down expansion. Let t' be the resulting tree with variables. Instantiate the variables in the tree t' to constants so that for each variable X a unique constant x is created and for 0 and 1, 0 and 1 are created. Let the resulting tree be t''. Let E' be the edb composed of all edb facts in t''. Clearly, program T' is satisfiable for predicate p' on edb E'. What we need show is that there is no cycle for the edb relation u in E'. The proof is by induction on the height h of a derivation tree that is a subtree of tree t''. The induction hypothesis is that: (1) There is no cycle composed of u facts in the subtree, and (2) The matrix component in the root fact correctly describes paths between the corresponding constants in the u facts used in the subtree.

The critical observation in the proof for $h > 1$ is that a cycle involving u edges in more than one subtree of the tree will identify at least two constants in the root say c and d such that there is a u cycle passing through c and d. (Because by top-down expansion, any constant appearing in more than one subtree must appear in the root.) This is impossible because the rule "allowing the cycle to happen" would not have been generated at all during compilation. (The information that such a cycle would be formed was available at compilation time.) □

General Acyclicity: The above example and the Lemma 4.1 use acyclicity constraints on binary relations. The reduction can be generalized according to Definition 4.1 for arbitrary arity (k in the definition). The reachability matrix that is added to each predicate has a greater dimension. For example, for $k = 2$, and an idb predicate of arity a, the matrix has the dimension $a^2 \times a^2$ (in general, $a^k \times a^k$), and the idb predicate will get a^4 extra arguments.

4.2 Universal Finiteness in the Presence of Acyclicity Constraints

The acyclic universal finiteness problem, defined analogously to Definition 4.2, can be reduced to several instances of the acyclic satisfiability problem using the techniques of Section 3. The acyclicity constraints are not modified by the reduction. If each of the reduced programs is unsatisfiable under the given constraints, the original program has the acyclic universal finiteness property.

Corollary 4.2 *Let L be a language that includes disequalities ($\neq$) and does not restrict the arity of predicates or the number of subgoals in a rule. The acyclic universal finiteness problem for a language L is decidable whenever the universal finiteness problem for the language L is decidable.* □

4.3 Emptiness Constraints

Whereas satisfiability under acyclicity constraints is solvable, such is not the case for satisfiability under arbitrary *emptiness constraints*. Define an *emptiness constraint* to be a 0-ary idb predicate defined via a Datalog program. The constraint is that a database is *admissible* if the 0-ary predicate is false. So, for example, acyclicity is the 0-ary predicate q in Definition 4.1.

Lemma 4.2 *The query containment problem can be reduced to satisfiability in the presence of emptiness constraints.* □

Proof: (Sketch) Consider two Datalog program P and Q with idb predicates p and q respectively. Let h be a new edb predicate. Define a new program Q' by adding the rule

$$q' \text{ :- } p(X,Y) \;\&\; q(X,Y) \;\&\; h(X,Y).$$

to program Q, and a new program P' by adding the rule

$$p'(X,Y) \text{ :- } p(X,Y) \;\&\; h(X,Y).$$

to program P. Consider q' to be an emptiness constraint on the evaluation of program P'.

One can show that (P,p) is contained in (Q,q) iff, when restricted to databases that satisfy the emptiness constraint q', (P',p') is not satisfiable. □

From the above theorem and the result that containment is undecidable for Datalog [Shm87], it follows that:

Corollary 4.3 *The satisfiability and universal finiteness problems for Datalog in the presence of emptiness constraints are undecidable.* □

5 Relationship to the Boundedness Problem

Given a Datalog program P and a predicate p, the *(predicate) boundedness problem* [Abi89, GMSV87, Gue90, Nau86, NS87, Var88, vdM90, HKMV91] is to determine whether there exists a constant c such that whenever there is a derivation tree for an atom a of predicate p in program P, there exists a derivation tree of height at most c for atom a over the same database. In other words, the boundedness problem asks whether a given program P is equivalent (in set semantics) to some non-recursive Datalog program P'.

In this section we show that boundedness and universal finiteness are unrelated in general; however for (pure) Datalog without any extensions, universal finiteness implies boundedness.

5.1 Does Boundedness imply Universal Finiteness?

Consider the following (pure) Datalog program P:

$p(X)$:- $p(X)$.
$p(X)$:- $r(X)$.

Clearly, program P is bounded (the first rule is superfluous), but P is not universally finite because, due to the first rule, there are infinitely many derivation trees for the fact $p(1)$ on a database containing the single fact $r(1)$. We thus obtain:

Lemma 5.1 *Boundedness of a Datalog program does not imply universal finiteness of the same program.* □

5.2 Does Universal Finiteness imply Boundedness?

We now consider whether universal finiteness of a program P implies boundedness of program P. The answer here depends upon whether we consider pure Datalog or its extensions.

Pure Datalog: Consider a program P in pure Datalog. If P is universally finite then P is finite on the particular extensional database E that contains, for each edb predicate, a single tuple made of constant a. This means that on edb E there are only some k derivation trees for the target predicate p. Any predicate q not appearing in any of these k trees is useless and all rules in which predicate q appears can be erased, while preserving equivalence to program P, because if q were to appear in any derivation tree on any extensional database whatsoever, q would appear in one of these k trees. So, without loss of generality, assume that all predicates in P appear in some of these k derivation trees.

This implies that there is no recursive chain in P. A recursive chain is a sequence of predicates $p = p_1 \ldots p_n = p$ such that for $1 \leq i < n$, there is a rule in P with p_i appearing in its body and p_{i+1} appearing in its head. Had there been such a recursive chain, it could be used to generate infinitely many derivation trees on edb E.

Thus, all the relevant predicates in P are non-recursive (a predicate is *recursive* if it appears in some recursive chain). Hence, P is equivalent to a non-recursive Datalog program and P is therefore bounded.

Datalog Extensions: Consider the following program P in Datalog extended with inequalities ($>, \geq, <, \leq, \neq$):

$p(X,Y)$:- $r(X,Y)$ & $X > Y$.
$p(X,Y)$:- $p(X,Z)$ & $r(Z,Y)$ & $X > Z$ & $Z > Y$.

In any derivation tree of program P, (X, Y) values on a leaf to root path are increasing and therefore such a path is bounded in length (the bound depends on the extensional database). Hence, given an edb, there can only be finitely many derivation trees for a derived fact. So, program P is universally finite. However, program P is not bounded as the depth of recursion is clearly data dependent.

The above example uses inequalities to introduce acyclicity in the database. Given acyclicity constraints on the database (Section 4), a pure Datalog program can be universally finite, but may not be bounded. As an example, the program P above, with the inequalities removed (i.e., a transitive closure computation) is universally finite if the relation r is constrained to be acyclic; however P is still not bounded.

Lemma 5.2 *Universal finiteness of a pure Datalog program on an unconstrained database implies boundedness of the same program. Universal finiteness of a Datalog program with inequalities, or of a pure Datalog program with acyclicity constraints on the database, does not imply boundedness of the same program.* □

6 Conclusions

We define two problems that are related to checking finiteness of number of derivation trees: universal finiteness and universal finiteness in the presence of acyclicity constraints. We also consider the problem of satisfiability in the presence of acyclicity constraints.

The two major results in this paper are:

1. We show that the universal finiteness problem can be reduced to the satisfiability problem, allowing us to map known decidability results on satisfiability onto universal finiteness. For instance, universal finiteness is decidable for Datalog extended with constants, inequalities, and negation on edbs. The problem is decidable for Datalog extended with safe stratified negation if all edb's are restricted to be unary.

2. We show that decidable cases of the satisfiability and the universal finiteness problems stay decidable in the presence of acyclicity constraints. The result is important since several common recursive programs in the transitive closure family have a finite number of derivations when the edb relations are restricted to be acyclic, but have an infinite number of derivations under cyclic edb relations.

In addition, we obtained undecidability results. We showed that both satisfiability and universal finiteness are undecidable in the presence of emptiness constraints, and that universal finiteness is undecidable whenever satisfiability is undecidable. Thus, universal finiteness is undecidable for the very limited case of stratified negation where only nonrecursive predicates are negated, all recursive predicates are unary, all nonrecursive predicates are binary, and there is only one recursive rule. It is also undecidable in the presence of functional dependencies on the edb relations.

Our results are useful for analysis and evaluation of recursive SQL3 [ISO93] queries. We can check whether query evaluation can be optimized to use an evaluation method that does not eliminate duplicates (such as not-so-naive [MR89] or duplicate semi-naive [Mum91]). The counting algorithm for incremental view maintenance [GKM92, GMS93] is applicable only if the number of derivation trees is finite. Our techniques can be used to check whether the counting algorithm is applicable.

The acyclicity constraints are a very special case of emptiness constraints for which the satisfiability and universal finiteness problems remain decidable. It would be interesting to define a class of emptiness constraints that is both non-trivial and contains acyclicity constraints, and that can be handled by a meta-interpretation method similar to the one we used for acyclicity constraints.

Related Work: The satisfiability problem is related to the equivalence and containment problems [LMSS93]. Our work adds upon the known results on satisfiability [Shm87, CGKV88, UVG88, KKR90, vdM92, LMSS93]. The finiteness problem was first introduced in [MS93], and a polynomial time algorithm to check finiteness for a given edb was presented. Preliminary work on universal finiteness was also reported there – it was shown that universal finiteness is decidable for Datalog with constants, disequalities ($\neq$), and stratified negation provided all edbs are unary, and was undecidable for full stratified negation. However, the important connection between satisfiability and universal finiteness was not made. The connection, made in this paper, strengthens both undecidability and decidability results for universal finiteness. Our techniques for reducing universal finiteness to satisfiability and for reducing satisfiability under acyclicity constraints to normal satisfiability can be viewed as meta-interpretation [SS86] techniques to check that the derivations made by the programs satisfy certain constraints (e.g., do not repeat a tuple, do not use a cycle).

Acknowledgements

We thank the program committee for suggesting that we relate the universal finiteness problem to the boundedness problem.

References

[Abi89] S. Abiteboul. Boundedness of a single rule program is undecidable. Information Processing Letters, 32 (1989), 281-289.

[CGKV88] S. S. Cosmadakis, H. Gaifman, Paris C. Kanellakis, and Moshe Y. Vardi. Decidable optimization problems for database logic programs. In *Proceedings of the Twentieth Symposium on Theory of Computing*, pages 477–490, 1988.

[GKM92] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting solutions to the view maintenance problem. In *Proceedings of the Workshop on Deductive Databases, Joint International Conference and Symposium on Logic Programming*, Washington D. C., USA, November 14 1992.

[GMSV87] H. Gaifman, H. Mairson, Y. Sagiv and M. Y. Vardi. Undecidable optimization problems for database logic programs. In *Proceedings 2'nd IEEE LICS* , pages 106-115, 1993.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, Washington, DC, May 26-28 1993.

[Gue90] G. Guessarian. Deciding boundedness for uniformly connected Datalog programs. In *Proceedings 3rd ICDT , Springer-Verlag LNCS 470*, pages 395-409, 1990.

[HKMV91] G. G. Hillebrand, P. C. Kanellakis, H. G. Mairson and M. Y. Vardi. Tools for Datalog boundedness. In *Proceedings of the Tenth Symposium on Principles of Database Systems (PODS)*, pages 1–12, Denver, CO, 1991.

[ISO93] ISO-ANSI. ISO-ANSI working draft: Database language SQL3, February 1993.

[KKR90] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Proceedings of the Ninth Symposium on Principles of Database Systems (PODS)*, pages 299–313, Nashville, TN, April 2-4 1990.

[LMSS93] Alon Levy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Equivalence, query-reachability, and satisfiability in Datalog extensions. In *Proceedings of the Twelfth Symposium on Principles of Database Systems (PODS)*, pages 109–122, Washington, DC, May 25-27 1993.

[LS92] Alon Levy and Yehoshua Sagiv. Constraints and redundancy in Datalog. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 67–80, San Diego, CA, June 2-4 1992.

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is relevant. In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, pages 247–258, Atlantic City, NJ, May 23-25 1990.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 264–277, Brisbane, Australia, August 13-16 1990.

[MR89] Michael Maher and Raghu Ramakrishnan. Déjàvu in fixpoints of logic programs. In *North American Conference on Logic Programming (NACLP)*, October 16-20 1989.

[MS93] Inderpal Singh Mumick and Oded Shmueli. Finiteness properties of database queries. In Maria E. Orlowska and Michael Papazoglou, editors, *Advances in Database Research: Proceedings of the Fourth Australian Database Conference (ADC)*, pages 274–288, Brisbane, Australia, February 1-2 1993. World Scientific Publishing Co.

[Mum91] Inderpal Singh Mumick. *Query Optimization in Deductive and Relational Databases*. PhD thesis, Stanford University, Stanford, CA 94305, USA, December 1991. Technical Report No. STAN-CS-91-1400.

[Nau86] J. F. Naughton. Data independent recursion in deductive databases. *JCSS*, 38 259-289, 1990.

[NS87] J. F. Naughton and Y. Sagiv. A decidable class of bounded recursions. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS)*, pages 227–236, San Diego, CA, 1987.

[Shm87] Oded Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the Sixth Symposium on Principles of Database Systems (PODS)*, pages 237–249, San Diego, CA, March 1987.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog. Advanced Programming Techniques*. MIT Press, Cambridge, Massachusetts, USA, 1986.

[UVG88] Jeffrey D. Ullman and Allen Van Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, January 1988.

[vdM90] Ronald van der Meyden. Predicate boundedness of linear monadic Datalog is in PSPACE. Unpublished manuscript, 1990.

[vdM92] Ronald van der Meyden. *The Complexity of Querying Indefinite Information: Defined Relations, Recursion, and Linear Order*. PhD thesis, Rutgers, The State University of New Jersey, New Brunswick, NJ, October 1992.

[Var88] M. Y. Vardi. Decidability and undecidability results of linear recursive queries. In *Proceedings of the Seventh Symposium on Principles of Database Systems (PODS)*, pages 341–351, Austin, TX, March 1988.