Finiteness Properties of Database Queries

(Extended Abstract)

Inderpal Singh Mumick
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA
mumick@research.att.com

Oded Shmueli*
AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974, USA
oshmu@cs.Technion.AC.IL

Abstract

We investigate the problem of checking whether the number of derivation trees of a Datalog program with duplicate semantics is finite or not. We show that given a safe stratified query and an edb, it is possible to check, in polynomial time, whether the query has a finite number of derivation trees. However, it is undecidable to check whether a safe stratified query has a finite number of derivation trees for every possible edb. We identify two classes of queries for which checking finiteness over all edbs is decidable. We also define duplicate semantics for Datalog (and thereby for the newly proposed variants of recursive SQL) queries with stratified negation. Since evaluation involving counting of duplicates may not terminate in recursive SQL, this problem is of significant practical importance.

1 Introduction

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.

The relational and deductive database models are built on relations and do not support duplicates[3, 14, 2]. The semantics of a query in such relational and deductive

*Current address: Computer Science Department, Technion, Haifa, Israel 32000.

databases can be defined in terms of derivations — a fact is in the answer to a query if it can be *derived* from the given database relations. The idea has been formalized through derivation trees[8] (a fact is in the answer if there exists one or more derivation tree for it).

Many commercially available relational database systems support duplicates, and need a duplicate semantics for recursive queries. There is also a need for duplicates in many database applications, and the question of duplicate semantics in the presence of recursion is an important one.

The semantics for recursive and nonrecursive queries with duplicates has also been defined in terms of derivation trees[11, 9]. Given a query, a fact is defined to be in the answer with multiplicity m if there are m derivation trees for the fact. A definition and enumeration of derivation trees for *Datalog with tables*, a language similar to SQL without the difference and aggregation operators, was given by Mumick[9]. A duplicate semi-naive evaluation algorithm to evaluate duplicate semantics was also given.

It is possible for a fact to have an infinite number of derivations. Consider the following example:

EXAMPLE 1.1 (Path Query:) We are given an $\texttt{edge}(X,Y)$ relation, and need to find the number of paths between every pair of nodes X and Y. A $\texttt{path}(X,Y)$ table can be defined in Starburst SQL[10] 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 Starburst SQL is equivalent to the following definition in Datalog if we interpret path as a multiset predicate.

$(N1)$: $\texttt{path}(X,Y) \mathrel{:-} \texttt{edge}(X,Y)$.

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

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

Note that if the given `edge` relation is acyclic, both the duplicate semi-naive and the not-so-naive evaluation methods would terminate and correctly compute the answer. It can be shown that the duplicate semi-naive and the not-so-naive evaluation methods work as long as the number of derivation trees is finite. However, there is no known algorithm that would recognize that a fact would be derived infinitely many times. The duplicate semi-naive evaluation[9] would not terminate on the above program given a cyclic `edge` relation. An evaluation under ordinary set semantics without eliminating duplicates, such as the not-so-naive evaluation[8], would also not terminate. Due to the cost of duplicate elimination, such evaluation techniques are very common for nonrecursive queries in commercial databases, and could be useful for certain recursive queries.

A solution to the problem of determining tuple multiplicity would enable the following:

1. We can determine whether a given query has a finite answer under multiset semantics used in many commercial database systems, such as those based on SQL (DB2, SQL/DS, Nonstop SQL). The results are also important in defining a meaningful duplicate semantics for SQL extended with recursion.

2. We can determine whether an evaluation method that does not eliminate duplicates will terminate. Duplicate elimination is an expensive operation, so that even when multiset semantics is not required, relational database systems often evaluate queries without eliminating duplicates.

3. Incremental view maintenance algorithms[4] can rely on the number of derivations. The applicability of such algorithms can be determined by our techniques. Gupta, Katiyar, and Mumick[4] have proposed an algorithm for incremental view maintenance by keeping track of the number of derivation trees for each fact. The method is applicable to SQL queries, as well as to deductive databases that do not use duplicate semantics. However, their algorithm works only if the number of derivation trees is finite.

Thus, for all of the above applications, we need to detect for a given program whether or not it has an infinite number of derivations. If it has an infinite number of derivations, then the duplicate semi-naive evaluation, the not-so-naive evaluation, and the incremental evaluation methods cannot be used.

We define two problems relating to checking finiteness of number of derivation trees. The first problem asks for finiteness under one given edb instance. This problem is shown to be decidable in polynomial time for Datalog extended with safe stratified negation. The second problem asks for finiteness under all possible values for the edb. This problem is shown to be undecidable for Datalog extended with safe stratified negation, and decidable for Datalog extended with constants, disequalities ($\neq$), and negation on unary edb's.

To work with programs with negation, we need to define derivation trees and duplicate semantics for programs with safe stratified negation. We extend the definition of Mumick[9] (for positive programs) to safe stratified programs.

The paper is organized as follows. Section 2 outlines some of the notation and definitions used in the rest of the paper. Section 3 presents duplicate semantics for programs with safe stratified negation. In Section 4 we discuss the finiteness problem under one given edb. We give a polynomial time algorithm for deciding finiteness. The universal finiteness problem is introduced in Section 5, and we present various decidability and undecidability results in the subsections. Conclusions are presented in Section 6.

2 Notation

We use Datalog, mostly as defined in Ullman[14]. A mapping between a class of SQL queries and Datalog with extensions for negation and aggregation is given in Mumick[9]. We do not consider aggregation in this paper. Predicates in a Datalog program are of two types: An **edb** predicate is one for which a relation is stored in the database. An edb predicate does not appear in the head of any rule. An **idb** predicate is defined by rules in the program, and no relation is stored in the database for an idb predicate.

The results in this paper are applicable to SQL extended with recursion and to Datalog. We could choose to use either Datalog syntax or SQL syntax. We have chosen to use Datalog syntax since many concepts are easier to explain in Datalog syntax. However, all definitions and results extend to SQL with recursion.

Definition 2.1 Range Restricted Datalog: Range restricted Datalog is Datalog in which for every rule, every head variable appears in a positive body literal. We consider only range restricted Datalog and extensions thereof. □

Definition 2.2 DDatalog: Duplicate Datalog, or DDdatalog, is 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 by Mumick[9]. Briefly, each distinct derivation tree for a fact adds one to its multiplicity in the derived set. □

Definition 2.3 Safe Negation: Given a rule r with negated subgoals, the negation is defined to be safe if every variable in the negated subgoals also appears in a positive subgoal of rule r. □

Definition 2.4 Stratum Numbers: Assume an assignment of stratum numbers as follows. Let the *dependency graph* of the program have predicate names as nodes; a directed edge (p, q) is in the edge set if there is a rule headed by a q atom and having in its body a p literal. Construct a *reduced dependency graph* of the given program by collapsing every strongly connected component of the dependency graph to a single node. A reduced graph is guaranteed to be acyclic. A topological sort[1] of the reduced dependency graph assigns a stratum number to each node. If a node represents a strongly connected component, all predicates in the strongly connected component are assigned the stratum number of the node. By convention, the base predicates are assigned stratum number $= 0$. □

Definition 2.5 Stratified Negation: Given a rule r with negated subgoals, the negation is defined to be stratified if every negated subgoal belongs to a lower stratum. □

3 Duplicate Semantics for safe stratified programs

We first define derivation trees for a DDatalog program P extended with safe stratified negation. The following definition captures all derivations for each fact derived by program P, so that duplicate semantics of program P is obtained by counting the number of derivation trees.

Definition 3.1 Derivation trees for safe stratified DDatalog programs: Derivation trees for predicates of stratum i are defined inductively over i.

For $i = 0$, that is, for edb predicates, each atom in the given edb relations forms one derivation tree for itself.

Inductively, assume that it is known which atoms of predicates of stratum i or less have a derivation tree, and all the derivation trees of atoms of predicates of stratum i or less have been defined.

For each rule r for a predicate of stratum $i + 1$ of the form

(r): $(h \text{ :- } b_1, b_2, \ldots, b_k, \neg c_1, \neg c_2, \ldots, \neg c_j)$.

with $k > 0$, and for each k-tuple

$$(t_1, t_2, \ldots, t_k)$$

of derivation trees for ground atoms

$$(d_1, d_2, \ldots, d_k)$$

that unify with subgoals

$$(b_1, b_2, \ldots, b_k)$$

with θ as the (necessarily, most general) unifier, such that the ground atoms

$$c_1\theta, c_2\theta, \ldots, c_j\theta$$

do not have any derivation tree, there is a derivation tree for $h\theta$, with $h\theta$ as the root label, and

$$(t_1, t_2, \ldots, t_k, \neg c_1\theta, \neg c_2\theta, \ldots, \neg c_j\theta)$$

as the children, in that order, and r as the edge label from the root to the children. □

Duplicate Semantics The duplicate semantics of a program is given by prescribing multisets to the derived relations. Given a DDatalog program P with safe stratified negation and an edb E, let a be a fact for predicate p and let there be m_a distinct derivation trees for a in E. Then, the multiset extension of p has m_a copies of fact a.

4 Finiteness

Definition 4.1 Finiteness Problem for a Language L: Given a program P in language L, a distinguished predicate q, and an instance E of the extensional database, does every derivable tuple of q have a finite number of derivation trees? □

In this section we show that the finiteness problem is decidable for DDatalog with safe stratified negation. We then give a polynomial time (in the size of the database) algorithm to check finiteness.

4.1 *Decidability of the Finiteness Property*

Given a safe stratified program P and an instance E of the extensional database, one can use several evaluation methods, amongst them the semi-naive bottom-up method[14], to compute the set of derivable atoms. For finiteness to be decidable for safe stratified programs, we need to show that it is possible to mark each derivable atom to indicate whether it has a finite or infinite number of derivation trees. An atom is marked f if it has a finite number of derivation trees, and is marked ∞ if it has an infinite number of derivation trees.

We give an inductive description of a semi-naive marking algorithm (Algorithm 4.1) for marking derivable atoms of a safe stratified program P using edb E. We use the following notation: An atom a is said to be of stratum i if it is an atom of a predicate of stratum i. We first state two lemmas.

Lemma 4.1 *Given a safe stratified program P and an edb E with k constants, there exists a constant G such that the number of derivable atoms is less than G. Further, G is a polynomial in k.* □

Proof: Let A be the maximum arity of any predicate in program P, let there be m predicates in program P, and let there be k constants in edb E. Then, the number of possible atoms is $G = mk^A$. Thus the number of derivable atoms is less than G, a polynomial in k. □

Lemma 4.2 *Given a program P, an edb E, and an integer constant $h \geq 0$, the number of derivation trees of height $h+1$ or less is, at most*

$$(r^{(n^h-1)/(n-1)})(mk^A)^{n^h}$$

where $r, n.m, k$, and A depend on program P. □

The *semi-naive marking algorithm* proceeds stratum by stratum. At each stratum we compute all derivable facts, along with a marking for finiteness or infiniteness. Computation within a stratum is done in a manner similar to the semi-naive bottom-up evaluation method[14]. The semi-naive marking algorithm is polynomial in the size of the database. Further optimizations to the semi-naive marking algorithm are possible, but we have not considered them since our intention is to show polynomial complexity.

For each atom a computed by the semi-naive algorithm, the semi-naive marking algorithm computes a triple

$$(a, \mathsf{mark}(a), \mathsf{deriving}(a))$$

where $\mathsf{mark}(a)$ is the finiteness/infiniteness marking of atom a, and $\mathsf{deriving}(a)$ is the set of all atoms of the same stratum as atom a used to derive atom a. The $\mathsf{deriving}$ set is maintained only if $\mathsf{mark}(a)$ is finite.

Algorithm 4.1 Semi-Naive Marking Algorithm: The algorithm works stratum by stratum, starting from stratum 0 and going up one stratum at a time.
Stratum 0: All edb atoms are marked f.
Stratum $i+1$: Inductively assume that all derivable atoms of stratum i or less have been derived and marked. We derive and mark the atoms of stratum $i+1$.

We do a semi-naive evaluation, modified to keep track of the mark field and the $\mathsf{deriving}$ sets. The mark of an atom can change from finite to infinite during the evaluation, but it cannot change from infinite to finite. For each derived atom a of stratum $i+1$ with a finite mark, the set $\mathsf{deriving}(a)$ consists of all atoms of stratum $i+1$ used to derive atom a.

In the first iteration of the semi-naive evaluation all derived atoms of stratum $i+1$ are derived using only facts of stratum i or less. If all positive atoms (of stratum i or less) used to derive fact a are marked finite, mark atom a as finite, else mark atom a as infinite. The $\mathsf{deriving}$ sets are all empty.

During a subsequent iteration of the semi-naive evaluation, consider, in some arbitrary order, each atom a derived in the iteration. Let the atom a be derived using some rule r and the previously derived atoms $g_1, \ldots, g_k$ of stratum $i+1$ and lower. Further processing is driven by the following conditions:

1. If atom a had been derived earlier, and has an infinite marking, do nothing. Specifically, the atom a is not included in the new δ relations of the semi-naive evaluation.

2. If atom a had not been derived earlier, or if atom a had been derived earlier but had a finite marking, define a new deriving set, $\mathsf{deriving}^{\nu}(a)$, for atom a that includes the old $\mathsf{deriving}(a)$, the atoms $g_1, \ldots, g_k$, and the $\mathsf{deriving}$ sets of atoms $g_1, \ldots, g_k$:

$$\mathsf{deriving}^{\nu}(a) = \mathsf{deriving}(a) \cup \{g_1\} \cup \ldots \cup \{g_k\} \cup \mathsf{deriving}(g_1) \cup \ldots \cup \mathsf{deriving}(g_k)$$

Note that the old $\mathsf{deriving}(a)$ is empty if atom a had not been derived earlier. Further processing depends on $\mathsf{deriving}^{\nu}(a)$:

(a) If any atom in $\mathsf{deriving}^{\nu}(a)$ is marked infinite, or if $a \in \mathsf{deriving}^{\nu}(a)$, change $\mathsf{mark}(a)$ to infinite, and consider atom a to be a new fact (include it in the δ relations of the semi-naive evaluation).

(b) If Case 2a does not apply, and $\mathsf{deriving}^{\nu}(a) \neq \mathsf{deriving}(a)$, change $\mathsf{deriving}(a)$ to $\mathsf{deriving}^{\nu}(a)$, and consider atom a to be a new fact (include it in the δ relations of the semi-naive evaluation).

(c) If Cases 2a and 2b do not apply, do nothing. Specifically, the atom a is not included in the new δ relations of the semi-naive evaluation.

EXAMPLE 4.1 Consider the `path` program N of Example 1.1. Let the edb be $E_2 = \{(1,1)\}$.

In the first iteration of the semi-naive marking algorithm, we derive the atom $p(1,1)$ using rule $N1$. The mark is finite and the $\mathsf{deriving}$ set is empty. In the second iteration, the algorithm derives $p(1,1)$ again, using rule $N2$ and atoms $e(1,1)$ and $p(1,1)$. The new $\mathsf{deriving}^{\nu}$ set for $p(1,1)$ now includes the atom $p(1,1)$. By Case 2a, the atom $p(1,1)$ is given an infinite mark. □

Theorem 4.1 *The semi-naive marking algorithm marks an atom a as infinite iff there are infinitely many distinct derivations of a in E.* □

Theorem 4.2 *The semi-naive marking algorithm terminates in time polynomial in the size of the database.* □

Theorem 4.3 *The finiteness problem for safe stratified DDatalog is decidable in polynomial time.* □

5 Universal Finiteness

Definition 5.1 Universal Finiteness Problem for a Language L: Given a program P in language L, with a distinguished predicate q, does every derivable tuple of q have a finite number of derivation trees for every extensional database E ? □

EXAMPLE 5.1 The `path` program N of Example 1.1 does not have a finite number of derivation trees for the edb $E = \{(1,2),(2,1)\}$. Program N is written in language DDatalog (or recursive SQL).

If there were an algorithm to check whether any given DDatalog program has a finite number of derivation trees, the universal finiteness property would be decidable for DDatalog. $\square$

In this section we prove that

1. The universal finiteness problem is *undecidable* for Duplicate Datalog with stratified negation (Section 5.1).
2. The universal finiteness problem is *decidable* for Duplicate Datalog with constants and disequalities ($\neq$) (Section 5.2).
3. The universal finiteness problem is *decidable* for Duplicate Datalog with base negation on unary edbs (Section 5.3).

5.1 *Duplicate Datalog with Safe Stratified Negation*

Definition 5.2 Languages $DD^{\neg}$ and $D^{\neg}$: Consider range-restricted DDatalog with safe stratified negation. All idb predicates are interpreted as multisets. Call the above language $DD^{\neg}$.

Consider range-restricted Datalog with safe stratified negation. All idb predicates are interpreted as sets. Call the above language $D^{\neg}$. $\square$

Theorem 5.1 *The universal finiteness problem for language $DD^{\neg}$ is undecidable.* $\square$

Proof: The equivalence problem for language $D^{\neg}$ can be reduced to the universal finiteness problem for language $DD^{\neg}$. The equivalence problem for language $D^{\neg}$ is known to be undecidable[13], so the undecidability for the universal finiteness problem follows.

Given the equivalence problem T: Is $(P_1, q_1) \equiv (P_2, q_2)$, derive program P by taking a union of programs P_1 and P_2 (assume all idb's are distinct) and adding the following three rules:

$q(X,Y) :- q_1(X,Y) \ \& \ \neg q_2(X,Y).$
$q(X,Y) :- q_2(X,Y) \ \& \ \neg q_1(X,Y).$
$q(X,Y) :- q(X,Y).$

It is easy to see that predicate q evaluates to an empty set iff $(P_1, q_1) \equiv (P_2, q_2)$. Further, if q_1 and q_2 differ in tuple (a,b), then $q(a,b)$ has an infinite number of derivations.

Thus (P,q) has a finite number of derivation trees iff $(P_1, q_1) \equiv (P_2, q_2)$. $\square$

5.2 *Duplicate Datalog with Constants and Disequalities*

Definition 5.3 Language $DD^{c\neq}$**:** Consider range-restricted Duplicate Datalog with constants and disequalities. A disequality is of the form $X \neq Y$ or $X \neq c$, where X and Y are variables that must also appear in a positive body literal of the rule, and c is a constant. Call the above language $DD^{c\neq}$. □

Theorem 5.2 *The universal finiteness problem for language* $DD^{c\neq}$ *is decidable.* □

Proof: Given a program P with distinguished predicate q, with k_1 constants $(b_1, \ldots, b_{k_1})$, and with k_2 being the maximum number of variables in a rule, let $E_{k_1+k_2}$ be a database consisting of all edb atoms that can be constructed from the $k_1 + k_2$ distinct symbols $\{b_1, \ldots, b_{k_1}\} \cup \{f_1, \ldots, f_{k_2}\}$.

We will prove that if there exists a database E for which some atom a of predicate q has a derivation tree of height h, then in the particular database $E_{k_1+k_2}$ also there is a derivation tree of height h for a related (in a way to be explained) atom a' of predicate q. Then, if atom a has an infinite number of derivations in database E, it must have derivation trees of an infinite number of heights in database E, and a related atom a' must also have derivation trees of an infinite number of heights in database $E_{k_1+k_2}$. We can then check finiteness of program P on the database $E_{k_1+k_2}$ using the techniques of Section 4 and obtain an answer to the universal finiteness problem.

Inductive Hypothesis We prove, by induction, that if there exists a database E, say with the n constants $(a_1, \ldots, a_n)$, that derives a derivation tree of height h for an atom $p(d_1, \ldots, d_m)$, where $d_i \in (\{a_1, \ldots, a_n\} \cup \{b_1, \ldots, b_{k_1}\})$, and $[d_{i_1} = d_{j_1}, \ldots, d_{i_q} = d_{j_q}]$ are the only q equality relationships amongst the arguments, and $[d_{u_1} = b_{v_1}, \ldots, d_{u_g} = b_{v_g}]$ are the only g instances of the constants from $\{b_1, \ldots, b_{k_1}\}$ in $(d_1, \ldots, d_m)$, then

1. Every atom $p(e_1, \ldots, e_m)$ of predicate p that can be constructed from the $k_1 + k_2$ constants $\{f_1, \ldots, f_{k_1}\} \cup \{b_1, \ldots, b_{k_2}\}$ under the following constraints:

   (a) $[e_{i_1} = e_{j_1}, \ldots, e_{i_q} = e_{j_q}]$ are the only q equality relationships amongst the arguments.

   (b) $[e_{u_1} = b_{v_1}, \ldots, e_{u_g} = b_{v_g}]$ are the only g instances of the constants from $\{b_1, \ldots, b_{k_1}\}$ in $(e_1, \ldots, e_m)$.

   has a derivation tree of height h from database $E_{k_1+k_2}$.

2. There is at least one p atom that satisfies the constraints of Case 1, and therefore has a derivation tree of height h from database $E_{k_1+k_2}$.

The induction is over the height h of the derivation tree for an atom of predicate p in program P.

Details of the inductive proof are omitted. □

EXAMPLE 5.2 We again consider the transitive closure `path` program of Example 1.1. There are no constants or disequalities in program N. The maximum number of variables in a rule $= k_2 = 3$. Thus we can check universal finiteness by checking finiteness against a complete database E_3 of 3 constants 1, 2, and 3. (In this case, since there are no disequalities, only one constant suffices). The atom $\texttt{edge}(1,1)$ is in the edb E_3, and the semi-naive marking algorithm will mark atom $\texttt{path}(1,1)$ as having an infinite number of derivations. □

5.3 *Duplicate Datalog with Restricted Forms of Negation*

In this section we consider two types of negation for which universal finiteness is decidable. First we allow negation on unary edb predicates, and second we allow stratified negation in programs where all edb predicates are unary.

Definition 5.4 Language $DD^{\neg(edb,1)\neq}$**:** Consider range-restricted Duplicate Datalog with safe negation on unary edb predicates, and with disequalities ($\neq$). A disequality is of the form $X \neq Y$, where X and Y are variables that must also appear in a positive body literal of the rule. Call the above language $DD^{\neg(edb,1)\neq}$. □

Theorem 5.3 *The universal finiteness problem for language* $DD^{\neg(edb,1)\neq}$ *is decidable.* □

Proof: The proof for Theorem 5.3 uses the idea that unary edb's basically define sets, and a collection of unary edb's identify constants to be of certain *types*, depending on which sets they are a member of. The proof is very similar to the proof for Theorem 5.2. A particular database E_k is constructed with the property that the finiteness test using edb E_k gives the answer to the universal finiteness question for a given program P. The database E_k is populated so as to have sufficiently many constants of enough types that all derivation trees on an arbitrary database E can be mapped onto derivations in database E_k. □

If constants are added to the above languages $DD^{\neg(edb,1)\neq}$ and $D^{edb(1)\neg\neq}$, the universal finiteness problem is still decidable.

6 Conclusions

We defined two properties related to checking finiteness of derivation trees.

The first property is *finiteness*. A query and an edb are finite if the query has a finite number of derivation trees. We have shown that it is possible to determine, in polynomial time, whether a recursive SQL query without aggregation, or alternatively a Datalog program with safe stratified negation, has a finite number of derivation trees. The decidability results would extend, in a straightforward manner, if stratified aggregation were included.

It would be ideal to be able to determine for a query whether it has a finite number of derivation trees for all edbs. This is captured by the concept of *universal finiteness*. We show that universal finiteness is undecidable for Datalog (or recursive SQL) extended with safe stratified negation. However, for queries that do not use negation, or where negation is limited to unary edbs, universal finiteness can be checked.

The results are encouraging, and should lead to work in developing more efficient algorithms to test infinities while evaluating queries. Applications are available in evaluating recursive SQL queries, influencing the ISO definition of recursion in SQL, determining whether the optimization of delaying duplicate elimination during query evaluation is applicable, and checking if incremental view maintenance through counts can be used. It should also be possible to evaluate queries that have infinite answers and give a finite representation of the infinite answers.

Mumick[9] considers a language *Datalog with tables* that permits edbs and idbs to be sets or multisets. The results presented in this paper extend to such a generalization.

There are several related issues that need more work. Can we check for universal finiteness for a language that allows base negation on all edb's? It is of interest to determine if a query has a finite number of derivation trees when all (or a few) edb relations are restricted to be acyclic. We define an edb relation $R(\bar{X}, \bar{Y})$ of even arity to be acyclic if the transitive closure of the relation does not have a path from $\bar{X}$ to $\bar{X}$. Is the universal finiteness problem for DDatalog decidable under the acyclicity constraint? Is the universal finiteness problem for DDatalog extended with safe negation on edb predicates decidable under the acyclicity constraint? For example, the transitive closure program has the universal finiteness property under an acyclicity constraint on the `edge` relation.

The universal finiteness problem is related to satisfiability, and we believe that some of the techniques used to check satisfiability[7, 6] may be useful in solving the above finiteness problems. We need more efficient algorithms for checking finiteness and universal finiteness. This paper presents initial work showing the problems to be decidable, and should lead to more efficient algorithms. An interesting related problem is to assign meaningful counts to the number of derivations when there are an infinite number of derivations. There is potential application for this in incremental view maintenance.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[3] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[4] A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases*, Washington, D. C., November 14 1992.

[5] ISO_ANSI. Iso-ansi working draft: Database language sql2 and sql3; x3h2; iso/iec jtc1/sc21/wg3, 1990.

[6] A. Levy, I. S. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, query-reachability, and satisfiability in datalog. In *Workshop on Deductive Databases*, Washington, D. C., November 14 1992.

[7] A. Levy and Y. 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. ACM SIGACT-SIGMOD-SIGART.

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

[9] I. S. Mumick. *Query Optimization in Deductive and Relational Databases.* PhD thesis, Stanford University, Stanford, CA 94305, USA, Dec. 1991. Technical Report No. STAN-CS-91-1400. Also available from University Microfilms International, 300 N. Zeeb Road, Ann Arbor, MI 48106. (313)761-4700.

[10] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. 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.

[11] I. S. Mumick, H. Pirahesh, and R. 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.

[12] I. S. Mumick and O. Shmueli. Aggregation, computability, and complete query languages. In *Workshop on Structural Complexity and Recursion-theoretic Methods in Logic Programming*, Washington, D. C., November 13

1992. Also Available as AT&T Technical Memorandum 11256-090820-14TMS, AT&T Bell Labs, September 1992.

[13] O. 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, Mar. 1987. ACM SIGACT-SIGMOD-SIGART.

[14] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2.* Computer Science Press, 1989.