Semantic Query Optimization in Datalog Programs

(Extended Abstract)

Alon Y. Levy
AT&T Bell Laboratories
levy@research.att.com

Yehoshua Sagiv*
Hebrew University, Jerusalem
sagiv@cs.huji.ac.il

Abstract

Semantic query optimization refers to the process of using integrity constraints (ic's) in order to optimize the evaluation of queries. The process is well understood in the case of unions of select-project-join queries (i.e., nonrecursive datalog). For arbitrary datalog programs, however, the issue has largely remained an unsolved problem. This paper studies this problem and shows when semantic query optimization can be completely done in recursive rules provided that order constraints and negated EDB subgoals appear only in the recursive rules, but not in the ic's. If either order constraints or negated EDB subgoals are introduced in ic's, then the problem of semantic query optimization becomes undecidable. Since semantic query optimization is closely related to the containment problem of a datalog program in a union of conjunctive queries, our results also imply new decidability and undecidability results for that problem when order constraints and negated EDB subgoals are used.

1 Introduction

An *integrity constraint* (ic) is a rule with an empty head, describing some semantic properties of data. Using ic's it is possible to express a variety of constraints, such as data dependencies (functional dependencies, multivalued dependencies and inclusion dependencies) as well as constraints involving comparisons (e.g., an employee's salary is less than his manager's salary). *Semantic query optimization* refers to the process of using ic's in order to optimize the evaluation of a query. For example, we can use integrity constraints to push selections in the query to the earliest point where they can be applied, or to remove redundant joins. Semantic query optimization is especially important in applications that require integrating multiple heterogeneous sources of data (e.g., [CGMH+94, LSK95]). The topic of semantic query optimization has been investigated in many papers (e.g., [Kin81, CGM88]) and is well understood for queries that can be represented as a union of conjunctive queries. For queries involving recursion, or which cannot otherwise be translated to a union of conjunctive queries (because of aggregation or duplicates), semantic query optimization has largely remained an unsolved problem.

Recent results have shed new light on this subject. First, in [LS92, LMSS93] it was shown how to push order constraints (i.e., selections) in recursive rules. These techniques can be used directly for complete semantic query optimization if, in each ic, only one subgoal is not an order constraint. Secondly, in [CV92] it was shown how to test whether a union of conjunctive queries contains a recursive program (assuming that there are neither order constraints nor negated subgoals). Since ic's are a special case of conjunctive queries, the containment algorithm of [CV92] can be used to determine satisfiability of recursive rules in the presence of ic's. Obviously, determining satisfiability of a set of rules is an important part of semantic query optimization.

In [CGM88], Chakravarthy et al. have shown that the core of semantic query optimization is computing residues. Intuitively, a residue is some part of an integrity constraint that cannot be mapped into the body of a rule, and therefore, its negation can be added to the rule. In particular, if the residue is empty, then the rule is unsatisfiable and can be ignored. In the case of recursive rules, however, it is not enough to look at residues involving each rule in isolation. Instead, we must look at residues w.r.t. derivation trees produced by the datalog program. The problem is to find a finite procedure to compute the residues, while the number of derivations trees can be infinite. We describe an algorithm for computing precisely the residues in a recursive program. We also show how to transform the program so that it does not have any sequence of rule applications that is guaranteed (by the ic's) to produce an empty result. The algorithm uses the

*Work supported in part by BSF grant 92-00360.

PODS '95 San Jose CA USA
© 1995 ACM 0-89791-730-8/95/0005..$3.50

query-tree techniques first developed in [LS92] and later refined in [LMSS93]. The key idea behind the query tree algorithm is to attach a *label* to every node in the tree, describing the residues applicable to that node when it is part of a derivation of the query. The same idea is the key for extending semantic query optimization to other cases in which queries cannot be represented as unions of conjunctive queries, such as SQL queries involving aggregation and duplicates. A detailed discussion of these extensions is beyond the scope of this paper.

Specifically, we show the following results. Semantic query optimization can be completely done in recursive rules with order constraints and negated EDB subgoals, provided that ic's have neither order constraints nor negated EDB subgoals. We also show that if either order constraints or negated EDB subgoals are introduced in ic's, then in general the problem of semantic query optimization is undecidable while in some special cases it is still decidable; in particular, one undecidability result is for the case of ic's that express functional dependencies. Our work also implies new decidability and undecidability results for the containment problem discussed in [CV92], since we consider rules and ic's that may have order constraints and negated EDB subgoals.

2 Preliminaries

A *datalog program* $\mathcal{P}$ consists of function-free Horn rules; its *EDB predicates* are those appearing only in bodies of rules and its *IDB predicates* are those appearing in heads and, possibly, in bodies of rules. An *EDB* (extensional database) is a set of ground facts for the EDB predicates; we usually refer to the EDB just as the *database*. The *IDB* (intensional database) is a set of ground facts that are computed for the IDB predicates by applying rules bottom-up starting with the EDB. We usually distinguish one IDB predicate of $\mathcal{P}$ as the *query* (or goal) predicate.

An *order atom* is an atom of the form $\gamma\theta\delta$, where γ and δ are either variables or constants and θ is one of the usual comparison predicates (i.e., $<$, $>$, $\leq$, $\geq$, $=$ and $\neq$). Order atoms represent a dense order defined on the domain and they may appear in bodies of rules provided that rules are *safe* as defined in [Ull89]. We also consider a limited form of negation by allowing negated EDB atoms in bodies of rules provided that negation is used safely. Note that the term "atom" refers to an atom that appears positively and the term "literal" refers to an atom that appears positively or negatively.

An *integrity constraint* (abbr. ic) is a rule with an empty head; we may think of an ic as a rule that derives **false** if its body can be satisfied. In this paper, we assume that bodies of ic's do not have IDB predicates (but may have EDB predicates as well as dense-order predicates). A database is *consistent* with respect to (or *satisfies*) a set $\mathcal{C}$ of ic's if **false** cannot be derived when applying the ic's to the database (i.e., none of the bodies in $\mathcal{C}$ can be satisfied by the EDB facts and the given order on the domain).

A *derivation tree* for a ground atom b consists of *goal nodes* and *rule nodes*. An EDB goal node is a ground literal of an EDB predicate, and an IDB goal node is a ground atom of an IDB predicate. A rule node is a ground instantiation of a rule, such that all the order atoms in the body of that rule are satisfied. The root of the tree is the goal node b. The only child of an IDB goal node g is a rule with head g. A rule node has a child for every (EDB or IDB) literal in its body. The leaves are goal nodes of EDB predicates; that is, they are ground EDB literals, and we assume that those literals are satisfied by the given database.

A *symbolic derivation tree* t for a predicate p is similar to a derivation tree, except that it has variables instead of constants. The root of t is an atom of p that initially[1] has a distinct variable in every argument position. Every rule node of t is the result of unifying a rule of the program with a goal node. Note that every derivation tree d can be represented by a pair (d', ψ), where d' is a symbolic derivation tree and ψ is an assignment of constants to the variables of d'. We say that a symbolic derivation tree d' is *consistent* with respect to a set of ic's $\mathcal{C}$ if there is an assignment ψ, such that the set of EDB atoms in the derivation tree (d', ψ) is consistent with $\mathcal{C}$ (and, of course, no atom appears in (d', ψ) both positively and negatively).

Satisfiability and Query Reachability Consider a datalog program $\mathcal{P}$ with a query predicate q, and let $\mathcal{C}$ be a set of ic's. We say that a predicate $p \in \mathcal{P}$ is *satisfiable* w.r.t. $\mathcal{C}$ if there is some database D that satisfies $\mathcal{C}$, such that a bottom-up evaluation of $\mathcal{P}$ produces a nonempty relation for p.

An atom $p(\alpha_1, \ldots, \alpha_n)$ (where α_i is either a constant or a variable) is *query reachable* w.r.t. $\mathcal{C}$ if there is some database D that is consistent with $\mathcal{C}$, such that an instantiation of $p(\alpha_1, \ldots, \alpha_n)$ is part of a derivation of some answer to the query predicate q of $\mathcal{P}$. Note that in general, query reachability of $p(\alpha_1, \ldots, \alpha_n)$ implies satisfiability of p, but the converse is not necessarily true. In [LMSS93], it is shown that query reachability and satisfiability are LOGSPACE-reducible to each other.

Denoting Appearance of Dense Order and Negation The presence of order atoms and negated EDB atoms increases the complexity of the problems we consider. Interestingly, order atoms and negated EDB atoms are easier to deal with when they are present in the program but not in the ic's. In our discussion,

[1] During the construction of t some variables of the root may be equated.

we will explicitly state where order atoms and negated EDB atoms are allowed to appear. We use θ to denote the presence of order atoms and $\neg$ to denote the presence of negated EDB atoms. Thus, a $\{\theta\}$-program is a program that may only have order atoms, a $\{\theta, \neg\}$-ic is an ic that may have either order atoms or negated EDB atoms (or both), etc. When we simply say "program" or "ic" it means that there are neither order constraints nor negated EDB atoms.

Local Order and Negated Atoms We say that an order atom (or a negated EDB atom) A, appearing in the body of an ic, is *local* if there is (at least) one positive EDB atom in the body that contains all the variables of A. For example, the order atom in the following ic is local.

$$:- \quad e(X,Y),\ e(Y,Z),\ X < Y.$$

The order atom $X < Z$, however, would not be local in the above ic.

The importance of locality lies in the following. The problems we investigate are decidable as long as ic's have only local order and negated atoms, but become undecidable when ic's have either order or negated atoms that are not local.

3 Illustrative Examples

Semantic query optimization refers to the process of using integrity constraints in order to optimize the evaluation of a query. The key idea in this process is to exploit the possible interactions between a rule of the program and the integrity constraints. Intuitively, given a rule r and an integrity constraint c, it may be the case that every instantiation of r satisfies some of the conjuncts of the integrity constraint. This can be shown by mapping subgoals of c to subgoals or r. Of course, if the mapping is complete, i.e., maps all the subgoals of c into the body of r, it means that the rule can never be satisfied and, hence, can be removed from the program. The more likely case is that there are only partial mappings of subgoals of c into the body of r. Given a partial mapping τ from c to r, the *residue* consists of those conjuncts of c that are not mapped to r. The negation of the residue must be satisfied by instantiations of the rule. Therefore, if we compute the residues $d_1, \ldots, d_n$ for c and r (each residue is obtained by a different mapping from c to r), then $\neg d_1 \wedge \ldots \wedge \neg d_n$ is an additional constraint that can be added to the body of r (note that each d_i is a conjunction of literals). Clearly, for databases that satisfy the integrity constraints, adding $\neg d_1 \wedge \ldots \wedge \neg d_n$ (or any subset of these conjuncts) to r will not change the output of the program. Of course, the goal is to add conjuncts that will facilitate a more efficient evaluation of r.

Example 3.1 The IDB predicate *goodPath* computes paths that are constructed from single steps and connects points given by the EDB predicates *startPoint* and *endPoint*.

$$\begin{aligned} r_1 : path(X,Y) &:- step(X,Y). \\ r_2 : path(X,Y) &:- step(X,Z),\ path(Z,Y). \\ r_3 : goodPath(X,Y) &:- startPoint(X),\ path(X,Y), \\ &\quad\ endPoint(Y). \end{aligned}$$

The following ic states that end points must have a greater value than all start points.

$$:- \quad startPoint(X),\ endPoint(Y),\ Y \leq X.$$

Any instantiation of r_3 satisfies the first two conjuncts of the above ic. Therefore, we get the residue $Y \leq X$ and, consequently, $Y > X$ can be added to the body of r_3 to obtain

$$\begin{aligned} goodPath(X,Y) &:- startPoint(X),\ path(X,Y), \\ &\quad\ endPoint(Y),\ Y > X. \end{aligned}$$

Note that by applying the selection $Y > X$ to $path(X,Y)$, we can reduce the cost of evaluating rule r_3. □

The above optimization could be done by existing methods for semantic query optimization, since the residue is obtained by mapping the ic into a single rule. In recursive programs, however, it is not always possible to find all the interactions between an integrity constraint and the program by considering each rule in isolation. For example, consider the following ic's that state that a step emanates from a start point only if that point is greater than or equal to 100, and that no step goes from a point to another with lesser or equal value:

$$:- \quad startPoint(X),\ step(X,Y),\ X < 100. \quad (1)$$

$$:- \quad step(X,Y),\ X \geq Y. \quad (2)$$

Together these ic's imply that there is no need to consider paths that emanate from points less than 100. This is because paths follow monotonically increasing points, and the starting point needs to be at least 100. However, to see this we must observe the structure of *every* derivation tree produced for *goodPath*. Using the ic's we can rewrite the program for *goodPath* as follows:

$$\begin{aligned} r_1' : path(X,Y) &:- step(X,Y),\ X \geq 100. \\ r_2' : path(X,Y) &:- step(X,Z),\ path(Z,Y), \\ &\quad\ X \geq 100. \\ r_3' : goodPath(X,Y) &:- startPoint(X),\ path(X,Y), \\ &\quad\ endPoint(Y). \end{aligned}$$

Note that when evaluating the rewritten program, we will not produce tuples for *path* for which the first argument is less than 100. Furthermore, note that rewriting the program involved not only identifying the residues w.r.t. derivation trees, but also placing them in the earliest possible point in the evaluation of the program. More generally, the program has the property that it completely incorporates the integrity constraints, defined as follows.

Definition 3.1 *Let $\mathcal{P}$ be a datalog program and $\mathcal{C}$ be a set of integrity constraints. The program $\mathcal{P}$ completely incorporates $\mathcal{C}$ if for all symbolic derivation trees t (for all IDB predicates), every goal node in t is query reachable w.r.t. $\mathcal{C}$.*

The problem we consider in this paper is to find a rewriting of a datalog program that completely incorporates the given integrity constraints.

4 An Algorithm for Semantic Query Optimization

In this section, we describe an algorithm for rewriting a datalog program $\mathcal{P}$ into one that fully incorporates a given set of ic's, $\mathcal{C}$. The algorithm is based on the general paradigm of constructing a query tree [LS92, LMSS93]. A query tree for a given datalog program $\mathcal{P}$ is a finite AND/OR tree that encodes *precisely* all derivations of the query predicate q from databases that satisfy the integrity constraints of $\mathcal{C}$. The query tree actually encodes symbolic derivations (recall that a symbolic derivation is like a derivation, except that constants are replaced by variables, and it represents the set of derivations that can be obtained by assigning constants to those variables). The root of the query tree is a goal node consisting of an atom of q. The children of a goal node are rule nodes for all the rules that can be unified with that goal node. The children of a rule node are the goal nodes for the subgoals in the body of that rule. Intuitively, the query tree can be viewed as a tree automaton that accepts only the possible derivations of the query [Var89]. Once a query tree is constructed for a given program, a rewritten program can be obtained by forming a rule for every rule node in the tree.

The key challenge in building a query tree is to encode *precisely* a possibly infinite set of symbolic derivations using a finite structure. The key idea for achieving finiteness is to attach *labels* to nodes in the tree, define an appropriate equivalence relation on labels (that relation must have a finite number of equivalence classes), and expand only one goal node from each equivalence class. Note that "expanding a goal node" means creating children for that goal node (one child for each rule that unifies with that goal node). Given a query tree T that has been expanded as described above, a symbolic derivation tree t is encoded in T if t can be obtained as follows. Start from the root and choose one rule-node child and its subgoal nodes. Inductively, let t be the tree created so far. If n is a goal node of t for some IDB predicate and n is currently a leaf of t, then let n' be the expanded goal node in T that is equivalent to n (n' may be n itself). Expand n with one of the children of n' (some variable renaming may be needed when doing this expansion).

Clearly, the crux of the query-tree construction is devising a labeling scheme (with an appropriate equivalence relation), such that the resulting query tree will encode precisely the desired set of derivations. For semantic query optimization, we will use labels that represent partial mappings from ic's into symbolic derivation trees (alternatively, the labels represent residues, since the unmapped portion of a partial mapping is a residue). Specifically, the label of a node n is a set of residues that are applicable at node n whenever n appears in a symbolic derivation encoded in the query tree. Formally, the label of a node n having a predicate p is a set of triplets of the form (I, σ, s), where

- I is one of the integrity constraints,
- s is the set of the unmapped EDB atoms of I, and
- σ is a (possibly partial) mapping from variables of s to argument positions of p.

The triplet (I, σ, s) means that if d is a symbolic derivation tree, such that d includes n and is encoded by the query tree, then there is a partial homomorphism that maps every EDB atom of I that is not in s to some leaf of d. Note that the homomorphism maps to leaves of d and not necessarily to leaves of the subtree rooted at n. In the algorithm, we will also use *adornments* that are similar to labels, except that they map EDB atoms of I to leaves of the subtree rooted at n.

Note that σ specifies only the variables of s that are mapped to (the variables in) the argument positions of the predicate p of n, but does not specify how other variables of I are mapped to variables of the symbolic derivation d. It is sufficient to require that the triplet (I, σ, s) be *consistent* in the following sense. If two EDB atoms of I, one in s and the other is not, share a variable Z, then σ maps Z to an argument position of the predicate p of n.

Two goal nodes n_1 and n_2 are defined to be equivalent if the atoms in these nodes are isomorphic and the labels of these nodes are identical (i.e., both labels are the same set of triplets). The key observation is that the number of equivalence classes is finite and, therefore, the construction of the query tree will terminate.

To explain the algorithm, we need to describe *how* the labels are computed. Informally, the query tree is computed in two phases: a bottom-up computation of adornments followed by a top-down creation of the

query tree. In the bottom-up phase, we compute a set of adornments for each predicate in the given program $\mathcal{P}$, and construct a set of adorned rules that use the adorned predicates. In the top-down phase, we create the query tree, including labels, by using the rules constructed in the bottom-up phase. The labels of nodes in the query tree are determined from the adornments of those nodes and the adornments of their ancestors.

We use the following example to illustrate the algorithm. The predicate p describes the transitive closure of paths created by edges of a and of b. The only integrity constraint, I, specifies that an edge of a cannot be followed by an edge of b.

$$\begin{array}{lll} r_1 : p(X,Y) & :- & a(X,Y). \\ r_2 : p(X,Y) & :- & b(X,Y). \\ r_3 : p(X,Y) & :- & a(X,Z),\ p(Z,Y). \\ r_4 : p(X,Y) & :- & b(X,Z),\ p(Z,Y). \\ I : & :- & a(X,Z),\ b(Z,Y). \end{array}$$

In the example, a triplet of the form (I, σ, s) will be denoted by describing s as a set of atoms; the mapping σ will be described implicitly by using an additional atom that represents the goal node with which (I, σ, s) is associated, and some of the variables in s are from that atom.

4.1 The Algorithm

We will now describe the algorithm for a $\{\theta, \neg\}$-program and ic's (i.e., the program, but not the ic's, may have order atoms and negated EDB atoms). Later, we will generalize the algorithm also to some cases of $\{\theta, \neg\}$-ic's. We assume that the given program $\mathcal{P}$ has already been processed by the algorithm of [LMSS93] for completely incorporating the constraints implied by the order atoms and negated EDB atoms that appear in the rules. Moreover, we assume that in each rule of $\mathcal{P}$, we have substituted X for Y (or vice-versa) whenever the order atoms of the rule imply that $X = Y$.

Bottom-Up Phase In the bottom-up phase, we compute a set of adornments for each predicate and a set of adorned rules $\mathcal{P}_1$ as follows. Initially, each IDB predicate has an empty set of adornments. An EDB predicate e has to be considered as a collection of several predicates, where each predicate in the collection represents one possible pattern of equalities among the argument positions of e. Formally, we use atoms (e.g., $e(X,X,Y)$, $e(X,Y,Y)$, etc.) to represent different patterns of equalities. Each one of these atoms, say $e(X,X,Y)$, has a single adornment that includes a triplet for every ic I and every possible way of mapping a subset (including the empty subset) of atoms of I to $e(X,X,Y)$. In the rules of the program, we associate each occurrence of e with one of these atoms.

The sets of adornments for IDB predicates are computed by bottom-up iterations as follows. Consider a rule r

$$p :- q_1, \ldots, q_m, \neg q_{m+1}, \ldots, \neg q_n, c.$$

where $q_1, \ldots, q_m$ are the positive subgoals, $q_{m+1}, \ldots, q_n$ are the negative EDB subgoals and c denotes the conjunction of the comparison literals in the body. For each q_i $(i = 1, \ldots, m)$, we choose an adornment A_i from the current set of adornments, Q_i, of q_i, and we construct an adornment for rule r, denoted A_r, as follows. A_r will include all the triplets of the form $(I, \sigma_1 \cup \ldots \cup \sigma_n, s_1 \cap \ldots \cap s_n)$, where

- for $1 \leq i \leq m$, $(I, \sigma_i, s_i) \in A_i$, and
- $\sigma_1, \ldots, \sigma_n$ are compatible, i.e., if σ_i and σ_j map the same variable of I to argument positions a_1 and a_2 of q_i and q_j, respectively, then these two argument positions have the same variable.

If, in one of the triplets generated for A_r, the intersection $s_1 \cap \ldots \cap s_n$ is empty, then A_r is the *inconsistent* adornment. If A_r is consistent, then we create from A_r a new adornment A_p for the head predicate p as follows. Suppose that the following holds.

- $(I, \sigma, s) \in A_r$.
- σ satisfies the following. Whenever two EDB atoms of I, one in s and the other is not, share a variable Z, then σ maps this variable to argument position(s) (in the body of r) having a variable X, such that X also appears in the head of rule r.

Then A_p will include (I, τ, s), where τ maps a variable Z of I to argument position a of the head p if σ maps Z to an argument position in the body of r that has the same variable as argument position a of the head p. If we have not already computed an adornment for p that is identical to A_p, then we add A_p to the set of adornments for predicate p. Finally, we add the rule

$$p^{A_p} :- q_1^{A_1}, \ldots, q_m^{A_m}, \neg q_{m+1}, \ldots, \neg q_n, c.$$

to $\mathcal{P}_1$, and associate the adornment A_r with this rule. Note that since there are "only" a doubly-exponential number of adornments, the bottom-up phase will terminate.

We will now show how adornments are computed in our example. First, it should be noted that there are several ways of partially mapping an ic I, and some are redundant in the following sense. Let (I, σ_1, s_1) and (I, σ_2, s_2) be two triplets for the same goal node, such that $s_1 \subseteq s_2$ and σ_1 is a restriction of σ_2 to the variables that appear in s_1. Clearly, (I, σ_2, s_2)

is redundant with respect to (I, σ_1, s_1). In general, however, redundant triplets can be removed only at the end of the construction of the query tree. In our example, the triplet for the empty mapping is going to appear in all the adornments. However, since it is redundant, we will not show it.

In our example, we begin with an adornment $a(X,Y)\{\{b(Y,Z)\}\}$ for a and $b(X,Y)\{\{a(Z,X)\}\}$ for the predicate b. With the rules r_1 and r_2, we create the following adornments for p, respectively:

p_1: $p(X,Y)\{\{b(Y,Z)\}\}$
p_2: $p(X,Y)\{\{a(Z,X)\}\}$

Using p_1 in r_3 does not produce a new adornment for p, and using p_2 in r_3 results in an inconsistent adornment (i.e., empty residue). Using p_2 in r_4 does not produce a new adornment for p, but using p_1 in r_4 produces the following adornment for p:

p_3: $p(X,Y)\{\{b(Y,Z)\},\{a(Z,X)\}\}$

Using p_3 in r_4 does not produce a new adornment and, so, the bottom-up phase is done. Intuitively, the adorned predicates p_1 and p_2 correspond to the transitive closures of a and b respectively, while p_3 corresponds to paths that are constructed from edges of b followed by edges of a. The set of rules $\mathcal{P}_1$ is the following:

$s_1 : p_1(X,Y) :- a(X,Y).$
$s_2 : p_2(X,Y) :- b(X,Y).$
$s_3 : p_1(X,Y) :- a(X,Z), p_1(Z,Y).$
$s_4 : p_2(X,Y) :- b(X,Z), p_2(Z,Y).$
$s_5 : p_3(X,Y) :- b(X,Z), p_1(Z,Y).$
$s_6 : p_3(X,Y) :- b(X,Z), p_3(Z,Y).$

Top-down Phase In the top-down phase, we actually create the query tree and determine the labels of the nodes in the tree. The labels are determined by the adornments of the predicates and are *pushed down* from parent to child. As stated earlier, the labels of the nodes are used in order to decide when to terminate the construction of the tree. However, the labels are also a refinement of the adornments. An adornment describes partial mappings from ic's to the subtree rooted at node n, while a label describes partial mappings from ic's to a complete symbolic derivation tree in which node n appears. Thus, the mappings described by labels have smaller unmapped portions as compared to the mappings described by adornments and, hence, labels produce smaller residues (which are more effective for the purpose of optimization).

The query tree is actually a forest consisting of one tree, with a root q^A, for each adornment A of the query predicate q. The tree for q^A is constructed top-down from the rules of $\mathcal{P}_1$. Each node of the tree has an adornment (which comes from $\mathcal{P}_1$) and a label that is created during the construction of the tree. Thus, we will denote a goal node as $p^{A,L}$, where p is the predicate of that node, A is the adornment and L is the label. For the root, the adornment and the label are the same and, hence, the root is denoted as $q^{A,A}$. In general, given a node with an adornment A and a label L, the following holds. For every triplet (I, σ, s) in A there is a corresponding triplet (I, σ', s') in L, such that $s' \subseteq s$ and σ' is the restriction of σ to those variables that appear in s'. Note that two goal nodes are equivalent if they have isomorphic atoms (i.e., same variable pattern) and identical labels (rather than identical adornments). The tree is built inductively as follows. Consider a goal node $p^{A,L}$. We create a rule-node child of $p^{A,L}$ for each rule of $\mathcal{P}_1$ that has p^A as its head. Suppose that rule r

$$p^{A_p} :- q_1^{A_1}, \ldots, q_m^{A_m}, \neg q_{m+1}, \ldots, \neg q_n, c.$$

is used to create a child of $p^{A,L}$ (i.e., A is the same as A_p). Let A_r be the adornment that was created for rule r during the bottom-up phase. We will now show how to create labels for rule r and its subgoals. Note that there are certain correspondences between triplets as follows.

- A triplet (I, σ', s') in L has a corresponding triplet (I, τ, s) in A (recall that A is the same as A_p). This is so by the inductive hypothesis associated with the construction of the tree.
- The triplet (I, τ, s) has a corresponding triplet (I, σ, s) in A_r, such that (I, τ, s) was obtained from (I, σ, s) in the construction of A_p (i.e., A) during the bottom-up phase.
- The triplet (I, σ, s) has a corresponding triplet $(I, \sigma_i, s_i) \in A_i$, such that (I, σ, s) was obtained (during the bottom-up phase) when (I, σ_i, s_i) was chosen from A_i.

Thus, the triplet (I, σ', s') of L corresponds to the triplet (I, σ_i, s_i) of A_i and, therefore, we make (I, σ_i', s') a triplet in the label of the subgoal $q_i^{A_i}$, where σ_i' is the restriction of σ_i to the variables of s'. Of course, the triplet (I, σ_i', s') in the label of subgoal $q_i^{A_i}$ corresponds to the triplet (I, σ_i, s_i) in the adornment A_i of $q_i^{A_i}$.

Similarly, the triplet (I, σ', s') of L corresponds to the triplet (I, σ, s) of A_r and, therefore, we make (I, σ'', s') a triplet of the label of rule r, where σ'' is the restriction of σ to the variables of s'. Of course, the triplet (I, σ'', s') in the label of rule r corresponds to the triplet (I, σ, s) in the adornment A_r of rule r.

As stated earlier, we expand a goal node in the tree only if there is no other goal node in the tree that is equivalent to it and was already expanded. After creating the query tree we remove nodes that are not reachable from the EDB leaves and the root (see [LS92]).

The query tree created for our example is shown in Figure 1. In the example, the labels remain identical to the adornments. The rules in the query tree can now be used to obtain a rewritten program that exploits the integrity constraints. In our case, the rewritten program (which is exactly $\mathcal{P}_1$) will be more efficient, because it will not attempt to create paths in which arcs of a are followed by arcs of b (thereby saving the effort involved in performing joins that are guaranteed to be empty).

Our algorithm establishes the following result.

Theorem 4.1 *Suppose that $\mathcal{P}$ is a $\{\theta, \neg\}$-program with a query predicate q and $\mathcal{C}$ is a set of ic's. The above algorithm constructs a query tree T and a new program $\mathcal{P}'$ that consists of the rules in the rule nodes of T, such that the following holds. First, for all databases D that satisfy $\mathcal{C}$, the programs $\mathcal{P}$ and $\mathcal{P}'$ are equivalent (i.e., produce the same relation for the query predicate q). Second, every IDB goal node in every symbolic derivation tree of $\mathcal{P}'$ is query reachable w.r.t. $\mathcal{C}$.*

Proof (sketch): It can be shown that all symbolic derivation trees of the query predicate that are produced by rules of $\mathcal{P}'$ are also produced by rules of the original program $\mathcal{P}$. The symbolic derivation trees of the query predicate that are produced by $\mathcal{P}$, but not by $\mathcal{P}'$, are exactly those that are inconsistent with $\mathcal{C}$. Thus, $\mathcal{P}$ and $\mathcal{P}'$ are equivalent for all databases satisfying $\mathcal{C}$.

The second part of the theorem follows from the following observations. First, recall that we have assumed that the algorithm of [LMSS93] was applied to $\mathcal{P}$ prior to our algorithm. Consequently, every symbolic derivation tree of $\mathcal{P}$ can be instantiated to a derivation tree from some set of EDB facts (which does not necessarily satisfies $\mathcal{C}$) by instantiating each variable to a distinct constant. Our algorithm removes those symbolic derivation trees that do not have any instantiation for which the set of EDB facts satisfies $\mathcal{C}$. Moreover, our algorithm does not equate any two variables in the remaining symbolic derivation trees. Also, instantiating each variable to a distinct constant guarantees that the remaining symbolic derivation trees satisfy $\mathcal{C}$. Thus, the theorem follows. □

4.2 Local Order and Negated Atoms

We can generalize the algorithm to the case of ic's with local order and negated EDB atoms (we will refer to these atoms just as "local atoms"). In order to do that, we need to "transfer" the local atoms from the ic's to the rules of the program. We do it as follows. First, we *associate* each local atom[2] l with one EDB atom a (from the same ic), such that a includes all the variables of l; moreover, we also generate the pair (a, l). We use the pairs to rewrite repeatedly the rules of the program as follows. If (a, l) is one of the pairs and r is a rule of the program with an EDB atom a', such that

- there is a homomorphism h from a to a', and
- neither $h(l)$ nor $\neg h(l)$ are literals in the body of r,

then replace r with two rules that are identical to r, except that one of them also has $h(l)$ and the other has $\neg h(l)$. (Technically, we have not considered negated order atoms; however, a negated order atom is equivalent to a unique positive order atom.)

After the rewriting of the program is completed, we can apply the algorithm with the following modification. Consider the step of creating the adornment A_r for rule r (in the bottom-up phase), and suppose that (I, σ, s) is a triplet in A_r. Then in the modified algorithm, we retain (I, σ, s) in A_r only if (I, σ, s) also satisfies the following condition. If the triplet (I, σ, s) implies that atom a of I is mapped to an EDB atom a' of r, according to a homomorphism h, and if l is a local atom of I that is associated with a (i.e., we created the pair (a, l)), then the triplet (I, σ, s) is in A_r only if either one of the following holds:

- If l is an order atom, then $h(l)$ is in r.
- If l is an EDB atom, then $\neg h(l)$ is in r.

Theorem 4.2 *Suppose that $\mathcal{P}$ is a $\{\theta, \neg\}$-program with a query predicate q and $\mathcal{C}$ is a set of $\{\theta, \neg\}$-ic's in which all order and negated atoms are local. The rewriting of $\mathcal{P}$ followed by the algorithm of Section 4.1 (with the above modification) constructs a query tree T and a new program $\mathcal{P}'$ that consists of the rules in the rule nodes of T, such that the following holds. First, for all databases D that satisfy $\mathcal{C}$, the programs $\mathcal{P}$ and $\mathcal{P}'$ are equivalent (i.e., produce the same relation for the query predicate q). Second, every IDB goal node in every symbolic derivation tree of $\mathcal{P}'$ is query reachable w.r.t. $\mathcal{C}$.*

Proof (sketch): It is easy to show that the rewriting preserves equivalence. Also, the rewriting terminates, since it does not introduce new variables in any rule. Now consider any symbolic derivation tree t of the rewritten program (before applying the modified algorithm). It can be shown that if there is a homomorphism h that maps all the EDB atoms of an ic $I \in \mathcal{C}$ to t, then for all local atoms l of I, either $h(l)$ or $\neg h(l)$ is also in t. Therefore, it follows that t is not consistent with respect to I if and only if there is a homomorphism that maps all the literals of I to t. Moreover, if t is consistent with respect to I, then by substituting a distinct constant for each variable of t, we get a derivation tree that is consistent with I. Next, it can be shown (similarly to the proof of Theorem 4.1) that the modified algorithm creates a query tree T,

[2] Note that l denotes a (positive) atom, which is either an order atom or an EDB atoms that appears negatively in an ic.

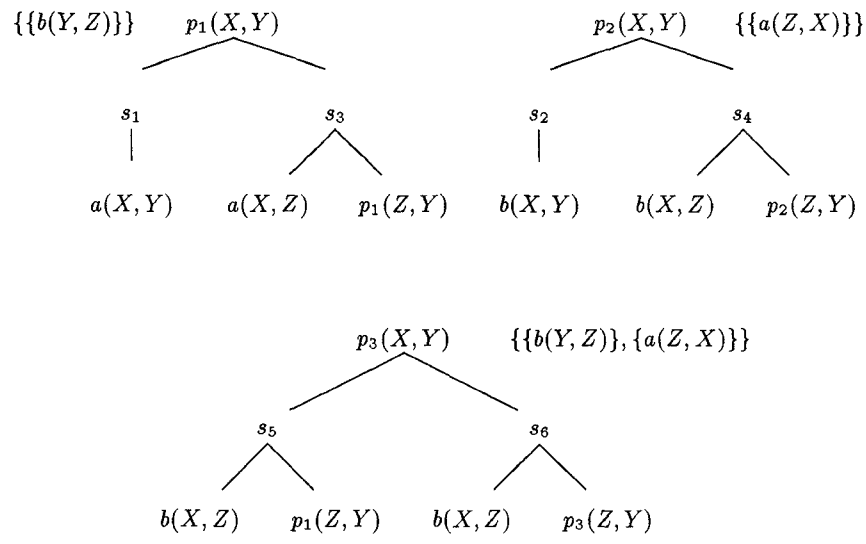

Figure 1: The final query tree. The labels shown in the roots of the trees are common to the entire tree below.

such that T encodes all symbolic derivation trees of the rewritten program that are consistent with $\mathcal{C}$. □

We can further generalize the algorithm as follows. Suppose that we apply the original algorithm to $\mathcal{P}$, while mapping only EDB atoms of ic's and not generating the inconsistent adornment even when all EDB atoms are mapped; let the result be the query tree T. Consider a triplet (I, σ, s) in the label of some IDB goal node of T, such that s is empty (i.e., all EDB atoms of I have been successfully mapped). We say that the triplet (I, σ, s) is *quasi-local* if for each order atom m in I, all the variables of m are mapped (according to the mapping implied by (I, σ, s)) to variables that appear in a single rule node of T. Note that it is easy to test whether (I, σ, s) is quasi-local. We say that that the ic's of $\mathcal{C}$ are *quasi-local* with respect to $\mathcal{P}$ if every triplet (I, σ, s) of T, such that s is empty, is quasi-local. In the full version, we will show that a result similar to Theorem 4.2 holds when $\mathcal{P}$ is a $\{\theta, \neg\}$-program and $\mathcal{C}$ is a set of $\{\theta\}$-ic's that are quasi-local with respect to $\mathcal{P}$.

5 Satisfiability and Containment

The problem of satisfiability in the presence of integrity constraints is closely related to the problem of the containment of a datalog program in a union of conjunctive queries. In fact, the following proposition shows that satisfiability is as hard as containment.[3]

Proposition 5.1 *The following two problems are LOG-SPACE-reducible to each other, even when programs and ic's are allowed to have order atoms and negated EDB atoms.*

- *Is a datalog program not contained in a union of conjunctive queries? (The conjunctive queries have order atoms and negated EDB atoms when the ic's have those atoms.)*
- *Is the query predicate of a datalog program satisfiable with respect to a set of ic's?*

When neither order atoms nor negated EDB atoms are present, containment of a datalog program in a union of conjunctive queries has doubly exponential lower and upper bounds [CV92]. By Proposition 5.1, the same is true for satisfiability of a query predicate with respect to ic's. This result can be generalized as follows.

Theorem 5.1 *Satisfiability of the query predicate of a $\{\theta, \neg\}$-program with respect to a set of ic's has doubly exponential lower and upper bounds.*

Proof As presented, the first step of the algorithm of Section 4.1 is applying the algorithm of [LMSS93], which has an exponential upper bound, and the complexity of the following bottom-up and top-down phases is doubly exponential. In the full version, however, we will show that the algorithm of [LMSS93] can be incorporated into the bottom-up and top-down phases to give an overall doubly-exponential upper bound. The lower bound follows from [CV92]. □

It is worth noting that the complexity of testing unsatisfiability is considerably better than the general case in either one of the following cases (actually, the second case is a generalization of the first).

- The program has a single IDB predicate.
- We want to test whether *all* the IDB predicates of a program $\mathcal{P}$ are unsatisfiable.

[3] Note that in this containment problem there are no ic's.

Formally, we say that a program $\mathcal{P}$ is *empty* if none of its IDB predicates is satisfiable. The following proposition shows that program emptiness is equivalent to containment among unions of conjunctive queries.

Proposition 5.2 *Suppose that $\mathcal{P}$ is a $\{\theta, \neg\}$-program and $\mathcal{C}$ is a set of $\{\theta, \neg\}$-ic's. Let $\mathcal{P}'$ consist of the initialization rules of $\mathcal{P}$ (i.e., rules with no IDB predicates in their bodies). Program $\mathcal{P}$ is empty if and only if $\mathcal{P}'$ is empty.*

Proof Since the rules of $\mathcal{P}'$ appear also in $\mathcal{P}$, it follows that if $\mathcal{P}$ is empty with respect to $\mathcal{C}$, then so must be $\mathcal{P}'$.

Conversely, suppose that $\mathcal{P}'$ is empty. Therefore, all the initialization rules of $\mathcal{P}$ are unsatisfiable and, so, the first iteration of a bottom-up evaluation of $\mathcal{P}$ cannot produce any IDB fact. Consequently, all the IDB relations of $\mathcal{P}$ are empty. □

Theorem 5.2 *The following are complexity results for program emptiness.*

1. *Emptiness of a $\{\neg\}$-program with respect to a set of ic's is NP-complete.*
2. *Emptiness of a $\{\neg\}$-program with respect to a set of $\{\neg\}$-ic's is in EXPSPACE.*
3. *Emptiness of a $\{\theta, \neg\}$-program with respect to a set of $\{\theta\}$-ic's is Π_2^P-complete.*
4. *Emptiness of a $\{\theta, \neg\}$-program with respect to a set of $\{\theta, \neg\}$-ic's is in EXPSPACE.*

Proof The upper bounds follow from [Klu88, LS93]. The lower bounds of Parts 1 and 3 follow from [SY81] and [vdM92a], respectively. □

In the remainder of this section, we give some undecidability results for satisfiability. First, satisfiability is undecidable if ic's have order atoms, even when only monadic recursion and $\neq$ are used [vdM92b, LMSS93]. In fact, the proof of [LMSS93] shows the following.

Theorem 5.3 *Satisfiability of the query predicate of a datalog program $\mathcal{P}$ with respect to a set $\mathcal{C}$ of $\{\neq\}$-ic's is undecidable, even if $\mathcal{P}$ has only unary IDB predicates, one linear recursive rule and two nonrecursive rules.*

When the ic's have negated EDB atoms instead of order atoms, the situation is not any better, as shown by the next theorem; the proof of this theorem is given in the appendix.

Theorem 5.4 *Satisfiability of the query predicate of a datalog program $\mathcal{P}$ with respect to a set $\mathcal{C}$ of $\{\neg\}$-ic's is undecidable, even if $\mathcal{P}$ has only unary IDB predicates, one linear recursive rule and two nonrecursive rules.*

Finally, satisfiability is undecidable even if the integrity constraints are expressing only functional dependencies and the program has only $\neq$.

Theorem 5.5 *Suppose that $\mathcal{P}$ is a datalog program with $\neq$, but without negation or any other comparison predicate. Satisfiability of the query predicate of program $\mathcal{P}$ with respect to a set of integrity constraints $\mathcal{C}$ of the form*

$$:- \quad e(\bar{X}, \bar{Y}_1, Z_1), e(\bar{X}, \bar{Y}_2, Z_2), Z_1 \neq Z_2$$

is undecidable.

Proof Follows from a result of [LMSS93]. □

6 Appendix: Proof of Theorem 5.4

Proof (Sketch) The proof of Theorem 5.3, which is given in [LMSS93], uses $\neq$ in the ic's. We will show how to replace $\neq$ by a binary EDB predicate $neq(X, Y)$.

First, a few words about the proof of [LMSS93]. That proof uses three EDB predicates. The first two, $succ(X, Y)$ and $zero(X)$, represent the successor and the zero relations, respectively. The third EDB predicate, $cnfg(T, C_1, C_2, S)$, represents configurations of a two-counter machine, where T is the time, C_1 and C_2 are the two counters and S is the state. The program computes whether the configurations form a halting computation, and the ic's check that the EDB relations are correct representations of zero, successor and the configurations. Only the ic's use $\neq$.

In our proof, we have three additional EDB predicates. The first, $dom(X)$, represents the domain. We use ic's in order to assure that $dom(X)$ has, at least, all the constants that appear in *succ*, *zero* and *cnfg*. For example, the following two ic's assure that all the constants from the successor relation are in $dom(X)$.

$$:- \quad succ(X, Y), \neg dom(X).$$
$$:- \quad succ(X, Y), \neg dom(Y).$$

There are similar rules for *zero* and *cnfg*.

We also have an EDB predicate, $eq(X, Y)$, for representing equality. The next ic checks that *eq* has a tuple (a, a) for each element that appears in the relation for *dom*.

$$:- \quad dom(X), \neg eq(X, X).$$

Note that *eq* may also have tuples that do not represent equality. However, if a tuple (a, b) appears in the relation for *eq*, then a and b will be considered as equal. Moreover, using the next two ic's, we can force *eq* to be symmetric and transitively closed.

$$:- \quad eq(X, Y), \neg eq(Y, X).$$
$$:- \quad eq(X, Z), eq(Z, Y), \neg eq(X, Y).$$

Using the next two ic's, we guarantee that any two zeros are equal and a zero cannot be equal to an element that is not a zero.

$$:- \quad zero(X),\ zero(Y),\ \neg eq(X,Y).$$
$$:- \quad zero(X),\ eq(X,Y),\ \neg zero(Y).$$

We also need to express the following constraint: If there is a path (of length one or more) from a to b in the successor relation, then a and b are not equal. To do that, we need yet another EDB predicate *neq*. The next two ic's constrain $neq(X,Y)$ to contain $succ(X,Y)$ and to be transitively closed. Note that in the following ic's, we use equality as it is represented by *eq* (rather than enforcing equality by multiple occurrences of the same variable).

$$:- \quad eq(X,X'),\ succ(X',Y'),\ eq(Y',Y),\ \neg neq(X,Y).$$
$$:- \quad eq(X,X'),\ neq(X',Z),\ eq(Z,Z'),\ neq(Z',Y'),\ eq(Y',Y),\ \neg neq(X,Y).$$

The following two ic's guarantee that every two elements from the domain are either equal or not equal, but not both.

$$:- \quad eq(X,Y),\ neq(X,Y).$$
$$:- \quad dom(X),\ dom(Y),\ \neg eq(X,Y),\ \neg neq(X,Y).$$

Note that the first ic above implies that $neq(X,X)$ cannot be satisfied.

The following two ic's say that if two elements are equal, then so are their successors and predecessors.

$$:- \quad succ(X,Y),\ succ(X',Z),\ eq(X,X'),\ neq(Y,Z).$$
$$:- \quad succ(Y,X),\ succ(Z,X'),\ eq(X,X'),\ neq(Y,Z).$$

The next ic says that a zero cannot have a predecessor.

$$:- \quad succ(X,Y),\ zero(Y).$$

The next three ic's check that any configuration at time zero has zeros in the two counters and in the state.

$$:- \quad cnfg(T,C_1,C_2,S),\ zero(T),\ \neg zero(C_1).$$
$$:- \quad cnfg(T,C_1,C_2,S),\ zero(T),\ \neg zero(C_2).$$
$$:- \quad cnfg(T,C_1,C_2,S),\ zero(T),\ \neg zero(S).$$

The following ic enforces *cnfg* to be closed under equality.

$$:- \quad cnfg(T,C_1,C_2,S),\ eq(T,T'),\ eq(C_1,C_1'),\ eq(C_2,C_2'),\ eq(S,S'),\ \neg cnfg(T',C_1',C_2',S').$$

Finally, there are also ic's that check that the transitions are correct. These ic's are the same as in [LMSS93]. In these ic's, we represent $X \neq Y$ as $neq(X,Y)$; we test whether a counter is zero or not zero, using the atoms $zero(C)$ and $\neg zero(C)$, respectively; and we increment and decrement a counter C using the atoms $succ(C,X)$ and $succ(X,C)$, respectively. As an example, the following ic checks the transition $\delta(j,>,=) = (j', pop, push)$.

$$:- \quad cnfg(T,S,C_1,C_2),\ cnfg(T',S',C_1',C_2'),\ succ(T,T'),\ S = j,\ \neg zero(C_1),\ zero(C_2),\ S'' = j',\ neq(S',S'').$$

In the above rule, $S = j$ is a shorthand for the conjunction[4]

$$zero(Z),\ succ(Z,V_1),\ succ(V_1,V_2),\ \ldots,\ succ(V_{j-1},S)$$

provided that j is not zero; otherwise, $S = j$ is just $zero(S)$. The atom $S'' = j'$ represents a similar conjunction.

Note that the above ic is violated if the first configuration satisfies the given transition (i.e., the state is j, the first counter is greater than zero and the second counter is zero) while the second configuration is for a time T', where T' is a successor of T, but the state is not what it should be. There are similar ic's for checking that either counter C_1' is not the result of decrementing C_1 by one or counter C_2' is not the result of incrementing C_2 by one.

Let G be the graph obtained from *succ* and *eq* (i.e., G has an arc for each tuple in the relations of these predicates).

Claim 6.1 *There is no path in G that contains arcs from succ and connects elements that are equal according to eq.*

The claim is true, because *neq* contains the transitive closure of paths of G with at least one arc from *succ*, and the EDB satisfies the constraint:

$$:- \quad eq(X,Y),\ neq(X,Y).$$

Note that, in particular, G does not have cycles containing arcs from *succ*.

Now suppose that the facts $eq(a,a')$ and $eq(b,b')$ are in the EDB and there is a path from a to b that has more successor arcs than some path from a' to b'. Since the ic's imply that successors (and predecessors) of equal elements are also equal (equality is according to *eq*), it follows that there is a portion of the path from a to b that connects equal elements and has at least one successor arc, in contradiction to Claim 6.1.

It thus follows that if all the constraints are satisfied, then the successor relation is a sound (albeit not

[4] We could have used, in this conjunction, equality based on *eq* rather than on multiple occurrences of variables; however, there is no need to do that.

complete) representation of the non-negative integers. Therefore, any sequence of consecutive configurations from the EDB that starts at time zero must correspond to a correct computation of the 2-counter machine.

So far, we have described the ic's. The program includes the following rules that compute the times of those configurations in the EDB that are reachable from the initial one.

$$\begin{aligned} reach(T) &:- cnfg(T, S, C_1, C_2),\ zero(T). \\ reach(T') &:- reach(T),\ succ(T, T'), \\ &\quad cnfg(T', S', C_1', C_2'). \end{aligned}$$

There is a third rule having the (0-arity) query predicate *halt* in its head; this rule computes whether some configuration is reachable from the initial one and its state is the halting state.

$$halt :- reach(T),\ cnfg(T, S, C_1, C_2),\ S = h.$$

It follows from the above discussion that on every consistent EDB, the relation for *reach* consists of times of configurations that form a valid computation. Therefore, the program is satisfiable if and only if the 2-counter machine reaches the halting state. □

References

[CGM88] U. S. Chakravarthy, John Grant, and Jack Minker. Foundations of semantic query optimization for deductive databases. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kaufmann, Los Altos, CA, 1988.

[CGMH+94] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinon, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In proceedings of IPSJ, Tokyo, Japan, October 1994.

[CV92] Surajit Chaudhuri and Moshe Vardi. On the equivalence of recursive and nonrecursive datalog programs. In *The Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, CA.*, pages 55–66, 1992.

[Kin81] J. J. King. *Query Optimization by Semantic Reasoning.* PhD thesis, Stanford University, Stanford, CA, 1981.

[Klu88] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, pages 35(1): 146–160, 1988.

[LMS94] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *Proceedings of the 20th VLDB Conference, Santiago, Chile*, 1994.

[LMSS93] Alon Y. Levy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Equivalence, query-reachability and satisfiability in datalog extensions. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Washington D.C.*, 1993.

[LS92] Alon Y. Levy and Yehoshua Sagiv. Constraints and redundancy in Datalog. In *The Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, CA.*, pages 67–80, 1992.

[LS93] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In *Proceedings of the 19th VLDB Conference, Dublin, Ireland*, pages 171–181, 1993.

[LSK95] Alon Y. Levy, Divesh Srivastava, and Thomas Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems*, 1995. Special Issue on Networked Information Discovery and Retrieval (to appear).

[SY81] Y. Sagiv and M. Yannakakis. Equivalence among relational expressions with the union and difference operators. In *J. ACM 27:4 pp. 633-655*, 1981.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes I, II.* Computer Science Press, Rockville MD, 1989.

[Var89] Moshe Y. Vardi. Automata theory for database theoreticians. In *Proceedings of the Eighth Symposium on Principles of Database Systems (PODS)*, pages 83–92, March 1989.

[vdM92a] Ron van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *The Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, CA.*, pages 331–345, 1992.

[vdM92b] Ron van der Meyden. *The Complexity of Querying Indefinite Information: Defined Relations Recursion and Linear Order.* PhD thesis, Rutgers University, New Brunswick, New Jersey, 1992.