Abstract

Two types of redundancies in datalog programs are considered. Redundancy based on reachability eliminates rules and predicates that do not participate in any derivation tree of a fact for the query predicate. Redundancy based on irrelevance is similar, but considers only minimal derivation trees, that is, derivation trees having no pair of identical atoms, such that one is an ancestor of the other. Algorithms for detecting these redundancies are given, including the case of programs with constraint literals. These algorithms not only detect redundancies in the presence of constraints, but also push constraints from the given query and rules to the EDB predicates. Under certain assumptions discussed in the paper, the constraints are pushed to the EDB as tightly as possible.

1 Introduction

Two major strategies for optimization of recursive rules are removing redundant parts and pushing constants from the query to the database. The algorithm for minimization under uniform equivalence [Sag88] is one example of removing redundant parts from datalog programs. The magic-set transformation (cf. [Ul189]) is a prime example of pushing constants from the query to the database.

More recently, constraints have started to play an important role. Pushing constraints from the query to the database was investigated in [BK*89, MF*90]—essentially these papers proposed generalizations of the magic-set transformation in order to handle constraints that appear in the query and the rules. Constraints as a part of the data were investigated in [KKR90]. Earlier work on constraints was done by Klug [Kl88] who investigated equivalence of conjunctive queries with inequalities.

This paper deals with constraints that are part of the program (but not the data), and the issues of interest are both redundancy and constraint pushing. In fact, it is shown that the two are closely related. Two types of redundancy are considered. The first is the well known concept of unreachability, and the second is a newly introduced concept, which we call irrelevance (and is based on the notion of minimal derivation trees).

Both types of redundancy are investigated when constraints are present and when they are not. One case, namely, the problem of unreachable rules when there are no constraints has an easy solution [Ki88]. The other three cases are not as easy, and we solve them in this paper.

The algorithms we present are based on creating a rule-goal tree that represents all redundancy-free derivations that can be con-
strutted for the query. The construction of the rule-goal tree is guided by the concepts of state-equivalence between nodes of the tree and constraint-labels associated with those nodes. These concepts vary as we move from one type of redundancy to another, or from one type of constraints to another.

The algorithms we present can be used in two ways. First, these algorithms lead to redundancy discovering; that is, the algorithms find redundant rules that can be removed from the program without changing the result. Second, the algorithms can be used to push constraints to the database. In this paper, we emphasize the first aspect of the results. However, we also indicate how the second aspect follows from our work.

Recently, Vardi [Va89] showed that the theory of tree automata is a useful tool for solving problems involving datalog rules. The problems we consider can also be solved as decision problems on certain tree automata. We prefer, however, to present direct algorithms (that do not involve tree automata), and the reasons for that are two: First, it leads to better efficiency (at least in practice if not in theory). Second, a reduction of the redundancy problems, which we consider, into tree automata can only find redundant rules. An explicit construction is needed, however, in order to incorporate constraint pushing into the magic-set transformation. The algorithms presented in this paper construct trees that are used for finding redundant rules, and could also be used for a magic-set evaluation. The approach of tree automata, however, was useful in getting a first insight into these problems.

A recent paper [Co91] characterized a large class of decidable problems involving datalog rules (the theory of tree automata is at the core of that work). However, the problems we investigate do not fall into the framework\(^1\) of [Co91], and therefore, our results are not a corollary of that work.

2 Preliminaries

We discuss datalog programs which are collections of safe Horn rules with no function symbols (i.e., only constants and variables are allowed). We distinguish between two sets of predicates in a given program. The extensional predicates (EDB predicates) which are those that appear only in bodies of rules, and the intensional predicates (IDB predicates) which are the predicates appearing in heads of rules. The EDB predicates refer to the database relations, while the IDB predicates are defined by the program. An extensional database (EDB) consists of a set of ground atomic facts for the EDB predicates. Given a datalog program and an EDB, a bottom-up evaluation is one in which we start with the ground EDB facts and apply the rules to derive facts for the IDB predicates. We continue applying the rules until no more new facts are generated. We distinguish one IDB predicate as the query (or goal) predicate, and the output (or answer) of the program is the set of all ground facts generated for the query predicate in a bottom-up evaluation.

Given a program and an EDB, a derivation tree is a tree consisting of goal-nodes and rule-nodes. A goal node is labeled by a ground atom, and it has a single child, which is an instantiated rule-node. The head of an instantiated rule-node is identical to its parent goal-node. A rule-node has a child goal-node for each one of its subgoals. The leaves of a derivation tree are goal-nodes labeled by ground atoms from the EDB. We say that a derivation tree is for an IDB predicate \(q\), if \(q\) is the predicate at the root of the tree.

In the rest of this section, we define two types of redundancies that may exist in datalog programs. Note that a rule is redundant if its removal does not change the output of the program (for all EDBs).

Definition 2.1: A rule \(r\) of a datalog program is unreachable if there is no EDB, such that \(r\) appears in some derivation tree for the query predicate. \(\blacksquare\)

Finding unreachable rules is easy; it is especially easy if we assume that the query predicate

\(^1\)The reason for that is that our problems involve not only the facts derived by a datalog program, but also the derivation trees.
depends on all other predicates and there are no constants in the program [Ki88]. In this special case, a rule is unreachable if either its body or head has an unreachable IDB predicate. And an IDB predicate \( q \) is unreachable if no fact is generated for \( q \) when the EDB has exactly one fact for each EDB predicate, with 1's as all its arguments.

Example 2.2: Consider the following program:

\[
\begin{align*}
    r_1 &: p(x) :- q(x). \\
    r_2 &: q(x) :- r(x). \\
    r_3 &: q(x) :- e(x). \\
\end{align*}
\]

Note that \( p, q \) and \( r \) are IDB predicates, and \( e \) is an EDB predicate. Rule \( r_2 \) is unreachable regardless of which is the query predicate, and \( r_1 \) is unreachable if \( q \) is the query predicate (note that the latter cannot be discovered by the above algorithm).

However, unreachability accounts only for some redundancies, as shown in the next example.

Example 2.3: Consider the following program:

\[
\begin{align*}
    r_1 &: p(x) :- q(x). \\
    r_2 &: q(x) :- p(x). \\
    r_3 &: q(x) :- e(x). \\
\end{align*}
\]

In this program, \( r_2 \) is reachable, but is redundant (regardless of which is the query predicate).

Generally, finding redundant rules is undecidable [Sar90, Sh87]. A narrower notion is redundancy under uniform equivalence [Sag88], but it cannot show the redundancies in the above examples. In the next definitions describe a new type of redundancy, called irrelevance, which is based on the notion of minimal derivation trees. This notion of redundancy subsumes the one based on unreachability.

Definition 2.4: A derivation tree is minimal (or non-redundant) if there are no two identical goal-nodes in the tree, such that one is an ancestor of the other.

Definition 2.5: A rule \( r \) is irrelevant in a given program if for all EDBs, \( r \) is never used in any minimal derivation for the query predicate.

In example 2.3, rule \( r_2 \) is irrelevant regardless of whether \( q \) or \( p \) is the query predicate, and rule \( r_1 \) is irrelevant if \( q \) is the query predicate.

Example 2.6:

\[
\begin{align*}
    r_1 &: p(x, y) :- p(y, x). \\
    r_2 &: p(x, y) :- e(x, y). \\
    r_3 &: p_1(x) :- p(x, x). \\
\end{align*}
\]

In this example, \( r_1 \) is relevant if the query predicate is \( p \), but is irrelevant if the query predicate is \( p_1 \).

Irrelevance is more general than unreachability, since an unreachable rule does not appear in any derivation tree. However, irrelevance does not capture all redundancies.

3 Finding Irrelevant Rules

This section describes an algorithm for deciding the property of irrelevance. It accepts as input a program \( P \) and a query predicate \( q \), and outputs all rules of \( P \) that are irrelevant to \( q \). If we change the notion of redundancy from irrelevance to unreachability, the algorithm remains almost the same. We discuss this modification later in the section. We begin with a description of the algorithm for determining irrelevance. For clarity, we assume that rules have no constants and heads of rules are rectified, i.e., distinct columns of the head of a rule have distinct variables. In the next section we explain how to relax this constraint (see Remark 4.3).

Informally, the algorithm begins by constructing the rule-goal tree for the query. The root of the rule-goal tree is a goal-node consisting of an atom of the query predicate with a distinct variable in each argument position. The children of a goal-node \( g \) in the tree (also referred to as an OR node) are the subgoals resulting from the unification. The children of a rule-node (also called an AND node) are the subgoals resulting from the unification. The rule-goal tree can be viewed

\[\text{\footnotesize 69}\]
as encoding all the possible derivation trees for facts of $q$. However, when the program has recursive rules, the construction of the rule-goal tree can go on forever. Therefore, the main difficulty in designing the algorithm arises in the decision when to stop expanding the branches of the tree. The following example illustrates this difficulty.

**Example 3.1:**

$$r_1 : q(x, y) :– e(x, t), q(t, t), e(t, y).$$
$$r_2 : q(x, y) :– e(x, y).$$
$$r_3 : q(x, y) :– p(x, y).$$
$$r_4 : p(x, y) :– q(x, y).$$
$$r_5 : q_1(x) :– q(x, x).$$

![Figure 1: A rule-goal tree.](image)

The key to this observation is the concept of a **node-tag** which we introduce below. In the following definitions, we consider a given rule-goal tree for a given program and query predicate. We denote by $V(g)$ the set of variables appearing in a goal-node $g$. Two nodes of the same predicate are said to be **identical** if in each argument position, the two nodes have the same variable.

An equivalence relation is induced on the set of goal-nodes by the following definition.

**Definition 3.2:** Two goal-nodes of the same predicate, $g_1$ and $g_2$, are said to be **equivalent** if there exists a one-to-one mapping, $\phi$, from $V(g_1)$ onto $V(g_2)$, such that $\phi(g_1) = g_2$. The mapping $\phi$ is called an **isomorphism**.

For example, nodes 2, 4, and 10 in Figure 1 are equivalent, but nodes 3 and 8 are not equivalent.

**Definition 3.3:** The **tag** of a goal-node $g$, denoted by $T(g)$, includes itself and all its ancestor goal-nodes that have only variables from $V(g)$. Intuitively, the tag of a node $g$ contains all goal-nodes that should not appear in subtrees of $g$ in order for the rule-goal tree to encode only minimal derivations. Ancestors that contain variables not in $V(g)$ need not be included in the tag of $g$, because these variables will not appear again in any subtree of $g$. The tag of node 2 in Figure 1 includes nodes 1 and 2, while the tag of node 4 includes only itself.

**Definition 3.4:** Two equivalent goal-nodes, $g_1$ and $g_2$, are said to be **tag-equivalent** if $\phi(T(g_1)) = T(g_2)$, where $\phi$ is the isomorphism showing the equivalence of $g_1$ and $g_2$.

In Figure 1, nodes 4 and 10 are tag-equivalent, but neither of them is tag-equivalent to node 2 (even though they are all equivalent). The algorithm uses the condition of **state-equivalence** to determine when to stop expanding a branch in the tree. In Algorithm 3.1, which we describe below, it suffices to define state-equivalence as tag-equivalence. The algorithm works in three steps. Step 1 expands the
rule-goal tree for the query predicate $q$, starting with a goal-node consisting of an atom of $q$ that has a distinct variable in each argument position. Step 1 terminates the expansion of a branch when it reaches a goal-node that is state-equivalent to another goal-node that is already in the tree and has been expanded. Step 1 will not expand a goal-node $g$ with a rule-node $r$ if the unification will produce a subgoal $g'$ which is identical to either $g$ or an ancestor of $g$. Note that since heads of rules are rectified and there are no constants, unification of a goal-node $g$ with a rule $r$ cannot change $g$ or its ancestors. Step 2 marks the nodes of the tree that are accessible from the EDB nodes in a bottom-up fashion. Step 3 marks as relevant, in a top-down fashion, the nodes that are reachable from the root via nodes marked as accessible in Step 2. In both Steps 2 and 3, when a goal-node is marked, all goal-nodes that are state-equivalent to it are marked as well. A rule is relevant if it appears in some rule-node that is marked in Step 3 as relevant. The full algorithm is shown in Figure 2.

The following example shows that terminating the expansion of a branch based on state-equivalence (rather than a less refined notion, such as equivalence (Definition 3.2)) is indeed needed to assure correctness of the algorithm.

Example 3.5:

\[
\begin{align*}
    r_1 &: q(x, y) :- q(x, z), e(z, y). \\
    r_2 &: q(x, y) :- e_1(x, y). \\
    r_3 &: q(x, y) :- p(x, y). \\
    r_4 &: p(x, y) :- e_2(x, y). \\
    r_5 &: p(x, y) :- q(x, y).
\end{align*}
\]

The rule-goal tree created for this program and the query predicate $p$ is shown in Figure 3. In this tree, all the nodes would be marked in Step 2, and therefore, all rules would be deemed relevant in Step 3. Notice that the node $q(x, y)$ is equivalent to the node $q(x, z)$, but they are not state-equivalent. And indeed, had we stopped expanding the tree based only on equivalence of goal-nodes (i.e., not expanded $q(x, z)$), we would have deduced that $r_3$ is irrelevant, since we could

procedure irrelevant-rules($\mathcal{P}, q$)

begin

/* Step 1: Constructing the rule-goal tree */
Let $T_0$ be a tree consisting of a goal-node for $q$
with a distinct variable in each column;
while there is an unexpanded goal-node $g$ in $T_0$, such that $g$ is not state-equivalent to
any expanded goal-node in $T_0$ do
    for each rule $r \in \mathcal{P}$ do
        if rule $r$ unifies with $g$ without
            making any subgoal of $r$ identical
to either $g$ or an ancestor of $g$
then perform the unification and
        make rule $r$ a child of $g$;
end.

/* Step 2: Bottom-up marking */
Mark all EDB nodes in $T_0$ as accessible;
repeat
    if all children of a rule-node $r$
        are accessible
then mark $r$ as accessible;
    if at least one child of a goal-node $g$
        is accessible
then mark $g$ as accessible;
    if a goal-node $g$ is state-equivalent
to an accessible goal-node $h$
then mark $g$ as accessible;
until no new nodes are marked;

/* Step 3: Top-down marking */
if the root of $T_0$ is accessible
then mark it as relevant;
repeat
    if $g$ is a relevant goal-node,
        $r$ is a child rule-node of $g$, and
        all children of $r$ are accessible
then mark $r$ and its children as relevant;
    if a goal-node $g$ is state-equivalent
to a relevant goal-node $h$
then mark $g$ as relevant;
until no new nodes are marked;
end.

The relevant rules are those appearing in
rule-nodes that are marked as relevant;
all other rules are irrelevant;

Figure 2: Algorithm 3.1—Finding irrelevant rules.
means that if the algorithm determines a rule \( r \) to be irrelevant, then \( r \) is indeed irrelevant. Completeness means that if a rule is irrelevant, it will be deemed so by the algorithm.

In this section, \( T_0 \) denotes the rule-goal tree constructed by Algorithm 3.1. In the following proofs we use symbolic derivation trees, i.e., derivation trees with symbolic derivation trees, i.e., derivation trees with variables instead of constants (derivation trees are defined in Section 2).

**Definition 3.7:** Given a goal-node \( g \) of \( T_0 \), we say that a symbolic derivation tree \( t \) is minimal for \( g \) if \( t \) is minimal (according to Definition 2.4), the root of \( t \) is identical to \( g \), and no goal-node of \( t \) is identical to an ancestor of \( g \). □

To prove correctness of Algorithm 3.1, we first establish several properties of the rule-goal tree, \( T_0 \), constructed by the algorithm.

**Lemma 3.8:** Let \( g_1 \) and \( g_2 \) be two goal-nodes in \( T_0 \) that are state-equivalent via the isomorphism \( \phi \). If \( t \) is a minimal symbolic derivation tree for \( g_1 \), then \( \phi(t) \) is a minimal symbolic derivation tree for \( g_2 \).³

The following lemma shows that all the goal-nodes that are marked in Step 2 of the algorithm have minimal symbolic derivation trees. Therefore, those nodes will be part of some minimal symbolic derivation trees for the query predicate if the same holds for their siblings and ancestors (and this is detected in Step 3).

**Lemma 3.9:** A goal-node \( g \) of \( T_0 \) is marked in Step 2 of Algorithm 3.1 if and only if there is a minimal symbolic derivation tree for \( g \).

Finally, the next lemma shows that all minimal derivation trees for the query predicate are encoded in the portion of \( T_0 \) that consists of nodes marked as relevant.

³The isomorphism \( \phi \) is defined only on the variables of \( g_1 \). We extend it to all variables of \( t \) by mapping each variable not in \( g_1 \) to a new distinct variable.
Lemma 3.10: Let $T_0$ be the rule-goal tree created by Algorithm 3.1. Suppose that $d$ is a minimal derivation tree for the query predicate (and program $P$), and is represented by a symbolic derivation tree $T$ and an assignment $\sigma$ to the variables of $T$. Then there is a mapping, $f$, from the nodes of $T$ to the nodes of $T_0$, such that:

1. $f(\text{root}(T)) = \text{root}(T_0)$.
2. For every goal-node $g \in T$, node $g$ is state-equivalent to $f(g)$.
3. For every node $v \in T$ (either a goal-node or a rule-node), $f(v)$ is marked as relevant in $T_0$.
4. For every rule-node $r \in T$, the node $f(r)$ is a rule-node labeled by the same rule as node $r$.
5. If a goal-node $g \in T$ has a child rule-node $r$ with subgoals $n_1, \ldots, n_m$, then $f(n_1), \ldots, f(n_m)$ are children of $f(r)$, and $f(r)$ is a child of either $f(g)$ or a goal-node $h \in T_0$ that is state-equivalent to $f(g)$.

Based on these lemmas we can prove the correctness of the algorithm.

Theorem 3.11: Algorithm 3.1 is sound; that is, if the algorithm determines that a rule $r$ is irrelevant, then it is indeed irrelevant.

Proof: Suppose that there is a minimal derivation tree, $d$, for the query predicate that uses $r$. To prove the theorem it suffices to show that the algorithm will mark some rule-node of $r$ as relevant in Step 3. We show that by applying Lemma 3.10 as follows. Let $d$ be represented by a symbolic derivation tree, $T_d$, and an assignment, $\sigma$, to all variables in $T_d$; and let $f$ be the mapping given by Lemma 3.10. The tree $T_d$ has some rule-node $r_0$ that is an instance of rule $r$, and by Lemma 3.10, $f(r_0)$ is an instance of $r$ in $T_0$ and $f(r_0)$ is marked as relevant.

Theorem 3.12: Algorithm 3.1 is complete; that is, if a rule $r$ is irrelevant, then the algorithm will determine it to be so.

Proof: To prove this, we show that if the algorithm deems $r$ relevant, then there is an EDB and a minimal derivation tree $d$ for the query predicate, such that $d$ uses rule $r$. Recall that $T_0$ is the rule-goal tree constructed by Algorithm 3.1.

Claim 3.13: Let $g$ be a goal-node of $T_0$. Suppose that a rule-node $r$ is a child of $g$ and is marked as relevant (and, hence, $g$ is also marked as relevant). Then there is a minimal symbolic derivation tree for $g$ in which $r$ is a child rule-node of $g$.

The claim is proved as follows. Let $n_1, \ldots, n_l$ be the children of $r$. Since $r$ is marked as relevant, the children $n_1, \ldots, n_l$ must be marked as accessible. By Lemma 3.9, there is a minimal symbolic derivation tree $t_i$ for $n_i$ ($i = 1, \ldots, l$). By a suitable renaming of variables, we can guarantee that every pair of trees $t_i$ and $t_j$ ($1 \leq i < j \leq l$) has a variable $V$ in common only if $V$ appears in both $n_i$ and $n_j$. We create a minimal symbolic derivation tree $t$ for $g$ as follows. The root of $t$ is $g$, the child rule-node of $g$ is $r$, and $t_1, \ldots, t_l$ are the subtrees of $r$. The tree $t$ proves the claim.

In order to complete the proof of the theorem, we will prove the following claim.

Claim 3.14: Suppose that $g$ is a goal-node of $T_0$ and $r$ is a child rule-node of $g$, such that $r$ (and hence also $g$) is marked as relevant. There is a minimal symbolic derivation tree $T$ for the query predicate in which $g$ appears as a goal-node and $r$ is a child rule-node of $g$. Moreover, the occurrences of $g$ in $T_0$ and in $T$ are state-equivalent.

We prove the claim by induction on the order of marking nodes in Step 3. The first node marked in Step 3 is the root of $T_0$. So, let $g$ be the root and let $r$ be a child rule-node of $g$ that is marked as relevant. By Claim 3.13, there is a minimal symbolic derivation tree $T$ for $g$ in which $r$ is a child rule-node of $g$. The tree $T$ proves the basis of the induction.

For the inductive step, consider a goal-node $g$ (other than the root) that has a child rule-node $r$, such that $r$ is marked as relevant. Among all goal-nodes that are state-equivalent to $g$, let $g'$ be
the first one that is marked as relevant ($g'$ could be just $g$). Note that $g'$ could not be the root of $T_0$, because $g$ is not the root, $g$ is state-equivalent to $g'$, and $g$ has been expanded (because it has $r$ as its child rule node). Therefore, $g'$ has a father rule-node $\tilde{r}$ that is marked as relevant, and $\tilde{r}$ has a father goal-node $\hat{g}$.

By the inductive hypothesis, there is a minimal symbolic derivation tree $T'$ for the query predicate, such that $\hat{g}$ is a goal-node of $T'$, rule-node $\tilde{r}$ is a child of $\hat{g}$, and the occurrences of $\hat{g}$ in $T_0$ and $T'$ are state-equivalent. Obviously, $g'$ is a child goal-node of $\tilde{r}$ in $T'$, and the occurrences of $g'$ in $T_0$ and $T'$ are state-equivalent. Let $\phi$ be the isomorphism, such that $\phi(g') = g$. We extend $\phi$ to all variables of $T'$ by mapping each variable not in $g'$ to a new distinct variable. Consider the tree $\phi(T')$. The occurrence of $g$ in $\phi(T')$ (i.e., $\phi(g')$) and the occurrence of $g$ in $T_0$ are state-equivalent (since the occurrences of $g$ and $g'$ in $T_0$ are state-equivalent, and the occurrences of $\hat{g}$ (and hence those of $g'$) in $T_0$ and $T'$ are also state-equivalent).

By Claim 3.13, there is a minimal symbolic derivation tree $t$ for $g$ in which $r$ is a child rule-node of $g$. Let $T$ be obtained from $\phi(T')$ by replacing the subtree rooted at $g$ with $t$. The tree $T$ proves the induction.

The proof of the theorem follows from Claim 3.14 and the following observation. If $T$ is a minimal symbolic derivation tree for the root of $T_0$, then we can replace all occurrences of each variable with a distinct constant, and the result is a minimal derivation tree for the query predicate, where the EDB consists of all the leaves of the tree.

The running time of Algorithm 3.1 is dominated by the time needed to construct the tree in Step 1. For every predicate, there is (at most) an exponential number of nonequivalent goal-nodes. Tags are sets of goal-nodes, and therefore, the number of possible tags for a goal-node is (at most) doubly exponential. Thus, the number of inner nodes in the tree, and hence also the size of the tree, is (at most) doubly exponential (only in the arity of the predicates). If we use the algorithm only to find unreach-plexity is singly exponential (in the arity alone), because the number of nodes in the tree is (at most) the number of nonequivalent goal-nodes. However, as explained in Remark 3.6, unreachability can also be detected in polynomial time. The following theorem gives an exponential time lower bound for detecting irrelevant rules, assuming that rules are not necessarily rectified.

**Theorem 3.15:** Given a program $P$ and a query predicate $q$, the problem of determining irrelevance of any given rule is hard for exponential time.

### 4 Programs with Constraints

Algorithm 3.1 is only guaranteed to be complete for programs without constraints. It treats constraint (i.e., interpreted) literals like any other EDB predicate, i.e., assuming that there is an instance of the EDB in which they are satisfied, independently of the other ground facts. However, considering the semantics of constraint literals can result in deducing additional irrelevance-facts, as in the following example.

**Example 4.1:** Consider the following program:

\[
\begin{align*}
    r_1 : p(x, y) &::= q(x, z), r(z, y), z > 1. \\
    r_2 : r(x, y) &::= q(x, z), e_1(x, y), x < 1. \\
    r_3 : q(x, y) &::= e_2(x, y).
\end{align*}
\]

Algorithm 3.1 would deem all the rules relevant to the query predicate $p$. However, the semantics of the $<$ predicate implies that none of the rules is relevant to $p$, since the first argument of predicate $r$, in the body or rule $r_1$, must be greater than 1, but rule $r_2$ only produces facts for $r$ in which the first argument is less than 1.

This section describes an extension of Algorithm 3.1 that is sound and complete for programs with constraints. This extension can be used with different types of constraint languages provided that the language satisfies some abstract properties, that we define below. Formally, we assume that there is a language $\mathcal{L}$ for expressing constraints. A formula $f$ (in the language $\mathcal{L}$), with free variables
\( X_1, \ldots, X_n \), describes a (possibly infinite) relation \( R_f(X_1, \ldots, X_n) \), which is the set of all tuples satisfying the constraints expressed by \( f \). We assume the following properties:

**Closure:** Given formulas \( f_1 \) and \( f_2 \), it is possible to construct effectively formulas that express:

- The join of \( R_{f_1} \) and \( R_{f_2} \).
- A projection of \( R_{f_1} \).
- A selection \( \sigma_{i=j} R_{f_1} \), where \( i \) and \( j \) are some columns of \( R_{f_1} \).
- A selection \( \sigma_{i=c} R_{f_1} \), where \( i \) is some column of \( R_{f_1} \) and \( c \) is a constant in the language \( L \).

**Equivalence:** Given formulas \( f_1 \) and \( f_2 \), it is decidable whether \( R_{f_1} = R_{f_2} \).

**Satisfiability:** Given a formula \( f \), it is decidable whether \( R_f \) is nonempty.\(^4\)

**Finiteness:** Let \( C \) be a finite set of constants in the language \( L \), and let \( F \) be a finite set of formulas in the language \( L \) that have at most \( n \) free variables (for some fixed \( n \)) and only constants from \( C \). Then applications of the operators (discussed in the Closure Property) to formulas in \( F \) may create only a finite number of nonequivalent formulas over \( n \) (or fewer) free variables.

Moreover, if \( f \) is a formula with a free variable \( X \), then \( f \) can imply \( X = c \), where \( c \) is a constant of the language \( L \), only if \( c \) appears in \( f \).

**Density:** Suppose that \( R_f(X_1, \ldots, X_n) \) is the relation for a formula \( f \), and for all \( 1 \leq i < j \leq n \), formula \( f \) does not imply \( X_i = X_j \) or \( X_i = c \), where \( c \) is a constant. Then any maximal subset of \( R_f(X_1, \ldots, X_n) \) that is equal on a group of \( k \) columns (\( 1 \leq k < n \)) must be infinite.

\(^4\)Note that if we have a formula FALSE in our language, denoting the empty relation, then the Satisfiability Property will follow from the Equivalence Property.

**Remark 4.2:** All of the above five properties are needed in order to have a complete algorithm for finding irrelevant rules in the presence of constraints. A complete algorithm for finding unreachable rules, in the presence of constraints, requires only the first four properties (i.e., the Density Property is not needed in this case).

**Remark 4.3:** Note that any datalog program can be easily rectified by adding constraints to the program. If a constant \( c \) occurs in a rule, we replace it by a variable \( x \), and add the constraint \( x = c \). If a variable \( x \) occurs twice in a subgoal (or in the head), we can replace one occurrence by a new variable \( y \) and add the constraint \( x = y \).

The constraint language of equalities satisfies the properties we require. Therefore the algorithm we describe below can be used directly on a non-rectified datalog program.

Algorithm 4.1, which we describe below, differs from the previous one only in the first step, the creation of the rule-goal tree. A basic justification underlying the correctness of Algorithm 3.1 is that every symbolic derivation tree (i.e., a derivation tree with variables instead of constants) can be instantiated to a derivation tree for some EDB. This is no longer true when constraints are present. To remedy this problem, we should unify a goal-node \( g \) with a rule \( r \) only if this unification could lead to an instantiation that is consistent with the constraint literals. We accomplish this by associating a **constraint label** with every goal-node. The constraint label of a goal-node describes the constraints that must hold on instances of that goal. The constraint label is a formula in the constraint language of the program. For example, a goal-node \( p(x, y) \) with a constraint label \( x < y \) denotes the tuples of the relation for \( p \) in which the first argument is smaller than the second.

Since a predicate \( p \) could be associated with many different constraint labels, \( p \) should really be thought of as representing many different labeled predicates, where each labeled predicate is obtained by associating \( p \) with some constraint label. In the naive approach, a labeled predicate \( p^c \) is created whenever \( c \) is a possible constraint label for predicate \( p \) (note that there are
only finitely many labels for a given predicate, by the Finiteness Property). Rules for the labeled predicates are created by considering all combinations of associating labels with subgoals in the body of each rule. However, the number of possible constraint labels could be very large, and a vast majority of them might not be needed. Therefore, we use a bottom-up procedure to compute the set of constraint labels that could actually be part of some derivation tree. We then create specific rules for the labeled predicates, and use a procedure similar to Algorithm 3.1 in order to create the rule-goal tree. Formally, our procedure is as follows:

Algorithm 4.1

1. Compute the set of constraint labels for the IDB predicates in $\mathcal{P}$. Initially, each IDB predicate has an empty set of labels (note that an empty set is equivalent to FALSE, i.e., it is never satisfied). Each EDB predicate has TRUE as its only label (note that TRUE is satisfied by every tuple). The sets of labels for IDB predicates are computed bottom-up as follows. Consider a rule

$$p : - q_1, \ldots, q_m, c.$$  

where $q_1, \ldots, q_m$ are the subgoals and $c$ denotes the join (i.e., conjunction) of the constraint literals in the body. For each $q_i$ ($i = 1, \ldots, m$), we choose a label $c_i$ from the set of labels, $Q_i$, of $q_i$. The constraint labels $c_1, \ldots, c_m$ and $c$ are joined and projected onto the variables of the head predicate $p$. The resulting label, denoted $c_0$, is added to the set of labels, $P$, for the head predicate $p$, provided that $c_0$ is satisfiable and $P$ does not already have a label that is equivalent to $c_0$. The algorithm is shown in Figure 4. Note that it is guaranteed to terminate, because of the Finiteness Property.

2. Create a new set of rules $\mathcal{P}_1$. The IDB predicates of $\mathcal{P}_1$ are specializations of the predicates of $\mathcal{P}$, according to the constraint labels computed in Step 1. Specifically, if $c$ is a constraint label computed for predicate $p$ in Step 1, then $p^c$ will be a predicate of $\mathcal{P}_1$. The EDB predicates of $\mathcal{P}_1$ are the same as those of $\mathcal{P}$. For ease of notation, however, EDB predicates are also denoted with a constraint label, i.e., if $c$ is an EDB predicate of $\mathcal{P}$, then $c^c$ is an EDB predicate of $\mathcal{P}_1$, where $c$ is the constraint label TRUE.

The rules of $\mathcal{P}_1$ are obtained as follows. Let

$$p : - q_1, \ldots, q_m, c.$$  

be a rule of $\mathcal{P}$, where $c$ denotes the conjunction of the constraint literals in the body. For each combination of constraint labels $c_1, \ldots, c_m$, such that $c_i$ ($i = 1, \ldots, m$) was computed for predicate $q_i$ in Step 1, we create the rule

$$p^{c_0} : - q_1^{c_1}, \ldots, q_m^{c_m}, c, c_1, \ldots, c_m.$$  

where $c_0$ is the result of joining $c$ and $c_1, \ldots, c_m$ and projecting on the variables of the head predicate $p$. The above rule is added to program $\mathcal{P}_1$ provided that the join of $c$ and $c_1, \ldots, c_m$ is satisfiable. Note that the constraint label $c_0$ was computed for $p$ in Step 1, and therefore, the head predicate $p^{c_0}$ is indeed among the IDB predicates defined for $\mathcal{P}_1$.

3. Create a forest of rule-goal trees. The goal-nodes of these trees are atoms of the labeled predicates of $\mathcal{P}_1$. The roots of the forest are determined as follows. If $p$ is the query predicate of the original program $\mathcal{P}$, then for each constraint label $c$ that was computed for $p$ in Step 1, there is a root for predicate $p^c$. A root $g$ for $p^c$ may have variables and constants; and an argument position of $g$ is equal to another argument position or to a constant, exactly as implied by $c$. We construct the rule-goal trees in the forest as in Algorithm 3.1, subject to the following modifications.

A goal-node $g$ for a labeled predicate $p^c$ can be unified only with rules that have $p^c$ as the head predicate.

Each node in the forest has a top-down constraint label (abbr. top-down label) associated with it. The top-down label of a goal node $g$ for a predicate $p^c$ may be stronger
than c, as a result of a top-down constraint propagation through the tree, which we now
describe.

The top-down label of a root for predicate $p^c$ is just c. Immediately after unifying a
goal-node $g$, having a top-down label $td(g)$, with a rule $r$ of the form

$$p^{c_0} : - q_1^{c_1}, ..., q_m^{c_m}, c, c_1, ..., c_m.$$

we do the following. First, top-down labels are created for the subgoals in the body as follows. The labels $td(g)$, $c$, and $c_1, ..., c_m$ are joined and the projection onto the variables of $q_i^{c_i}$ ($i = 1, ..., m$) is the top-down label of $q_i^{c_i}$. Second, variables in the body of $r$ are equated with other variables or with constants whenever such equalities are implied by the join of $td(g)$, $c$, and $c_1, ..., c_m$.

The resulting rule $r$ is added as a child of the goal node $g$ provided that the following two conditions are satisfied. First, the join of $td(g)$, $c$, and $c_1, ..., c_m$ is satisfiable. Second, no subgoal of $r$ is identical to either $g$ or an ancestor of $g$. Note that “identical” has the same meaning as in Section 3; that is, same predicate and same variable in each column, but the constraint label need not be the same.

The tag of a goal-node is defined exactly as in Definition 3.3, and tag-equivalence is defined exactly as in Definition 3.4 (i.e., constraint labels are ignored). Finally, we define two nodes in the forest to be state-equivalent if their top-down constraint labels are equivalent and the nodes are tag-equivalent. Recall that we do not expand a node in the tree if there is another state-equivalent node that is already expanded.

**Example 4.4:** Consider the following program:

$$r_1 : p(x, y) : - q_1(x, z), q_2(x, y).$$
$$r_2 : p(x, y) : - q_2(x, z), q_3(y, z), y \geq z.$$
$$r_3 : q_3(x, y) : - e_3(x, y), y \geq 3.$$
$$r_4 : q_2(x, y) : - e_2(x, y), x \leq 0.$$
$$r_5 : q_1(x, y) : - e_1(x, y), x \geq 1.$$

**procedure bottom-up-labels (P)**

begin
for each IDB predicate $p$ of $P$ do
$$P := \emptyset;$$
for each EDB predicate $e$ of $P$ do
$$E := \{TRUE\};$$
repeat
if $p := q_1, ..., q_m, c$ is a rule of $P$
and $c_i \in Q_i$ ($i = 1, ..., m$)
then begin
$$\bar{c} := \text{join}(c_1, ..., c_m, c);$$
$$c_0 := \text{projection of } \bar{c} \text{ on the variables of } p;$$
if $\bar{c}$ is satisfiable and $P$ has no constraint
label that is equivalent to $c_0$
then add $c_0$ to the set $P$;
end;
until no new constraint label is added to any set;
end.

Figure 4: Computing the constraint labels.

Step 1 of the algorithm will generate the following sets of constraint labels for IDB predicates:

$$P = \{[y \geq 3, x \leq 0]\}$$
$$Q_1 = \{x \geq 1\}$$
$$Q_2 = \{x \leq 0\}$$
$$Q_3 = \{y \geq 3\}$$

Note that in order to avoid ambiguity, a constraint label is surrounded by square brackets if it consists of more than one literal.

Since every predicate has only one constraint label, the set of rules resulting from Step 2 will be the same as the original set, except for rule $r_1$ that will not be included, because it only generates the FALSE constraint label.

The tree resulting from the algorithm is shown in Figure 5. The constraint labels shown next to the nodes are the top-down constraint labels. The algorithm deems rules $r_1$ and $r_5$ irrelevant to the query predicate $p$.

It is important to note another usage of the algorithm. It enables us to push constraints to the EDB. For example, the only constraint on $e_2$ initially was $x \leq 0$. The tree produced by the
Figure 5: A tree with top-down labels.

Example 4.5: Consider the following program:

\[
\begin{align*}
    r_1 : & \ t(x, y) : - p(x, z), t(z, y), x \leq z. \\
    r_2 : & \ p(x, y) : - q(x, y). \\
    r_3 : & \ q(x, y) : - e(x, y), x \geq y. \\
    r_4 : & \ t(x, y) : - p(x, z). 
\end{align*}
\]

The label computed for all three predicates is \( x \geq y \). However, when rule \( r_1 \) is unified with the goal-node \( t(x, y) \) that has a top-down label \( x \geq y \), variables in the body of \( r \) should be equated according to the conjunction \( x \geq y, x \geq z, z \geq y, x \leq z \). Since this conjunction implies \( x = z \), the second subgoal becomes \( t(x, y) \), which is the same as the head. Therefore, rule \( r_1 \) is not included in the tree, and \( r_1 \) is deemed irrelevant to the query predicate \( t \) (note, however, that \( r_1 \) is reachable).

In order to discuss the correctness of Algorithm 4.1, consider a derivation tree \( t \). The constraints in the tree can be propagated bottom-up and top-down. In each step of a propagation, we consider a goal-node \( g \) and its (only) child rule-node \( r \), and join all the constraints in the body of \( r \) and the constraint label of \( g \). In bottom-up propagation, the result of this join is projected onto \( g \) and becomes the new constraint label of the head. In top-down propagation, the result of this join is projected onto each subgoal of \( r \) and becomes the new constraint label of that subgoal.

Initially, the constraint label of each goal-node is just TRUE. Bottom-up propagation starts in the leaves and proceeds to the root. Top-down propagation starts in the root and proceeds to the leaves. We can alternate between phases of bottom-up propagation and top-down propagation until the tree is stable, that is, until no constraint label is changed. Actually, the tree will be stable after one bottom-up phase followed by a top-down phase; note that this is essentially a semijoin reduction, as in Yannakakis’ algorithm (cf. [Ull89]).

In principle, bottom-up and top-down propagation of constraints can also be done in the rule-goal forest constructed by Algorithm 4.1. However, the forest constructed by the algorithm is already stable, since a bottom-up phase is done essentially in Step 1, and a top-down phase is done in Step 3 (via the top-down constraint labels).

Similarly to Lemma 3.10, we can show that the forest constructed by Algorithm 4.1 encodes all minimal and stable derivation trees. Conversely, each symbolic derivation tree encoded in the forest can be instantiated to some minimal and stable derivation tree. The fact that the forest encodes trees that are not just minimal but also stable, indicates that the top-down constraint labels in the forest are as tight as can be. In other words, Algorithm 4.1 pushes constraints from the query and rules to the EDB as tightly as possible (of course, “as tightly as possible” is relative to the specific mode of redundancy which is being detected—in our case, the redundancy is either based on reachability or on relevance). In this respect, Algorithm 4.1 solves not only the irrelevance problem, but also the problem of pushing constraints from a given query to the EDB.

**Theorem 4.6**: Algorithm 4.1 is sound and complete for finding irrelevant rules in the presence of constraints that satisfy the five properties stated earlier.

---

\(^5\)Algorithm 4.1 can also be modified to solve the reachability problem along the lines of Remark 3.6.
4.1 Rational Order Constraints

An important constraint language is that of rational-order constraints. The atomic formulas of this language are of the form $x \theta y$ or $x \theta c$, where $x$ and $y$ are variables, $c$ is a constant, and $\theta \in \{<, \leq, >, \geq, =, \neq\}$. Formulas in the language are either atomic or conjunctions of atomic formulas. There is a complete set of axioms for inferring constraints of this form [Ull89]. However, this language does not satisfy the Closure Property we require and, therefore, the tree with the top-down labels may not be stable. Consequently, the top-down constraint label of a node does not necessarily describe all the constraints that the node has to satisfy. The following example illustrates how the Closure Property can be violated.

Example 4.7: Consider the conjunction:

$$z_1 \leq z, z \leq z_2, z \neq x_3$$

This conjunction implies only one constraint, $z_1 \leq z_2$, among the variables. However, that does not fully describe all the constraints among $z_1, z_2,$ and $x_3$. The constraint $x_3 \neq x_1 \lor x_3 \neq x_2$ is also implied by this conjunction, but since our language does not allow disjunctions, we cannot express this when trying to project the above conjunction onto $z_1, z_2,$ and $x_3$.

Fortunately, we can show that our algorithm will be sound and complete for rational-order constraints as well. The key observation underlying this fact is that although the top-down labels of the nodes in the forest do not completely describe the exact constraints that hold in the goal-nodes, they are related to the exact constraints in the following way. Suppose that $td(g)$ is the top-down label computed for a goal-node $g$, and let $c$ denote the exact constraints that hold for $g$. First, if $td(g)$ is satisfiable, then $c$ is satisfiable. Therefore, the forest does not contain any node that should have been deleted. Second, the only constraints that might be missing from $td(g)$ are of the form $x_i \neq x_j, x_i \neq c,$ or disjunction of those. However, in our algorithm we assume that two variables are distinct unless we can prove otherwise. Therefore, if some constraints are missing from $tg(g)$, they will be implicitly satisfied anyways when we instantiate a symbolic derivation tree encoded in the forest.

Theorem 4.8: Given a datalog program with rational-order constraints, Algorithm 4.1 is sound and complete, i.e., it will find all the irrelevant rules and only them.

5 Conclusion and Related Work

Algorithm 4.1 finds redundant rules. However, it also pushes constraints to the database. Essentially, each EDB predicate can be restricted by the disjunction of top-down labels of leaves having that predicate. Moreover, we can augment the information in the tree with bound-free adornments. Consequently, the definition of state equivalence also requires equality of adornments, and the resulting tree is the basis for a magic-set evaluation (cf. [Ull89]); the details, however, are beyond the scope of this paper. This approach incorporates constraints in the magic-set evaluation more effectively than [BK*89, MF*90].

An interesting issue is the types of constraint languages that satisfy the properties mentioned earlier. For example, we can also view a small EDB relation as a constraint, and in this case the first four properties are satisfied, which is sufficient for completeness when redundancy is based on unreachability. Note that our approach of handling a small EDB relation as a constraint is different from that of [SS90], and it appears to be more effective, since it pushes the exact constraint and not just adornments. However, it may result in a larger magic program.

Linear inequalities do not satisfy the Finiteness Property, and consequently, Algorithm 4.1 may not terminate in this case. However, one may think of ad hoc heuristics that provide termination and soundness without guaranteeing completeness. The approaches of [BS91, VG90] may be useful in finding more effective heuristics. Another direction is to find subcases in which Al-
Algorithm 4.1 terminates when the constraints are linear inequalities.

Acknowledgements

We would like to thank Pandu Nayak and Hiroshi Motoda for many discussions on the issues discussed in this paper, and comments on earlier drafts of this paper. Part of this work was done while the first author was visiting the Hitachi Ltd., Advanced Research Laboratory, and he thanks Hitachi and especially Hiroshi Motoda for their generous support.

References


