A Logic-Based Approach to Data Integration

J. Grant[3], & J. Minker[1,2]
jgrant@towson.edu, minker@cs.umd.edu
[1]*Department of Computer Science*
[2]*and Institute for Advanced Computer Studies*
University of Maryland at College Park
College Park, Maryland, U.S.A.
[3]*Department of Computer and Information Sciences and*
Department of Mathematics
Towson University
Towson, Maryland, U.S.A.

Abstract

An important aspect of data integration involves answering queries using various resources rather than by accessing database relations. The process of transforming a query from the database relations to the resources is often referred to as query folding or answering queries using views, where the views are the resources. We present a uniform approach that includes as special cases much of the previous work on this subject. Our approach is logic-based using resolution. We deal with integrity constraints, negation, and recursion also within this framework.

1 Introduction

An important part of data integration involves answering queries using various resources rather than by accessing database relations. The process of transforming a query from the database relations to the resources is often referred to as query folding or as answering queries using views, where the views are the resources. For instance, a database of interest to a user may be distributed over a network. It is necessary to bring data distributed over a network to a user's machine so that the data may be manipulated to answer user queries. In a distributed environment it is likely that one will want to save answers to queries in the local user's machine so that if the same or a related query is posed to the distributed database, one can look in the local machine's cached database for the answer, rather than have to go out over the network to answer the query. In this situation the resources are the cached relations and the use of these resources is an important aspect of query optimization. In some data integration systems the database relations are themselves virtual and the data must be obtained from the resources. Resources may also be materialized views.

Several researchers have considered various aspects of this problem. In this paper we present a logic-based approach to the query folding problem using the method of resolution. As a consequence,

1. We obtain a uniform approach that includes as special cases much of the previous work on this topic.
2. If the algorithm finds a rewriting of the query, then we are guaranteed that the answers are sound, that is, the answers are correct.
3. The approach also allows us to determine under certain conditions if the folded query contains all answers to the original query, that is, it is complete.

We consider a database that consists of an extensional database (**EDB**), an intensional database (**IDB**), a set of integrity constraints (**ICs**), and a set resources ($\mathbf{Res_{DB}}$), where the resources have been obtained by using resource rules. These resources are referred to as materialized views. That is, they have been made explicit in a local computer as a result, for example, of an answer to a conjunctive query. The **EDB**, **IDB**, **ICs** are part of a conventional **Datalog** database.

Section 2 describes related work, complexity results, systems and algorithms that have been developed with respect to the folding problem. Section 3 provides background for the definitions and notations used in the paper. Section 4 contains several examples and our query folding algorithm. The algorithm is logic-based and deals uniformly with integrity constraints, extensional and intensional predicates and extends the work in (Qia96). Functional and inclusion dependencies are considered in Section 5. We show that our query folding algorithm handles all integrity constraints in a uniform way without the need for specialized techniques as in (DGQ96, Gry98) and (DPT99, PDST00), where the latter also consider physical access structures, which we do not treat. Section 6 deals with the case where resources are obtained by the use of several definitions or queries. Our work on multiple rules for the same resource relates to the work in (Dus97b, AGK98, FG01). Whereas they are concerned primarily with maximal containment, we are concerned with a uniform method that checks both for soundness and completeness of answers. Section 7 discusses negation. We show that the logic-based framework handles stratified negation in both rules and intensional predicates. Our approach is different from the rewriting used in (FG01). Recursion is considered in Section 8. We differ from the work in (DG97a) and (DGL00) by handling a special case of recursive views as well as recursive queries. We compare the contributions made in this paper with other efforts in Section 8. The paper is summarized in Section 9.

2 Related Work

There has been a substantial amount of work done in connection with data integration and query folding. Apparently the first papers to propose algorithms for query folding were in (LY85, YL87), where they developed an algebraic method. Other early work was by (TSI94) and by (CKPS95). An algorithm for rewriting conjunctive queries over non-recursive databases was provided in (Qia96). In (DGQ96, Gry98) it is shown how to use materialized views in the presence of functional and inclusion dependencies. Additional work on conjunctive query optimization and on information integration appears in (Ull97, LRO96a, LRO96b, DG97a). Levy et al. (LMSS95) showed that the question of determining whether a conjunctive query can be rewritten to an equivalent conjunctive query that only uses views is NP-complete. This work was extended in (RSU95) to include binding patterns in view definitions. Duschka (Dus97a), discusses the concept of *local completeness*, where it is known that some subset of the data an information source stores is complete, although the entire data stored by the information source might not be complete. (DG97a) were the first to extend the work to general recursive queries. They show that the problem of whether a Datalog program can be rewritten into an equivalent program that only uses views, is undecidable. (DGL00) also discuss answering queries using views to recursive queries for Datalog programs. Duschka and Levy (DL97) introduce the new class of *recursive* query plans for information gathering. Plans are extended to be recursive sets of function-free Horn clauses. In his thesis, Duschka (Dus97b) deals with multiple definitions of the same resource and shows how to obtain a maximally contained query using these definitions. Duschka and Genesereth (DG98) were the first to publish results concerning how to handle multiple definitions of the same resource predicate. Afrati et al. (AGK98) extend this work to disjunctive queries and related results. Popa and his co-authors, (DPT99, PDST00), showed how to do query folding with some forms of integrity constraints using the so-called 'chase method'. Flesca and Greco (FG01) deal with disjunction and negation and formulate their answers in terms of classical and default negation. The complexity of answering queries using materialized views for conjunctive queries with inequality, positive queries, Datalog and first-order logic is addressed in (AD98). The paper (Lev01) surveys the methods proposed for answering queries using views. See Ullman, (Ull97), for a survey of work concerning information-integration tools to answer queries using views that

represent the capabilities of information resources. The formal basis of techniques related to containment algorithms for conjunctive queries and/or Datalog programs is discussed there. Approaches taken by AT&T Labs's *Information Manifold* and the Stanford *Tsimmis*, (GMPQ$^+$95) project are compared. Levy in (Lev00) describes several algorithms proposed for data integration: the bucket and inverse-rules algorithms.

Several papers address complexity problems associated with folding, Chandra and Merlin, (CM77) showed that the query containment problem is NP-complete. Several subclasses of conjunctive queries were identified that have polynomial-time containment algorithms (ASU79a, ASU79b, JK83).

The query folding problem is thus at least NP-hard. It has been shown to be NP-complete for conjunctive queries and resources in (LMSS95). Many variants of the problem of answering queries using views are discussed in (Lev01). The problem was shown to be NP-complete even when queries describing the sources and the user query are conjunctive and do not contain interpreted predicates ((LMSS95)). (LMSS95) further show that in the case of conjunctive queries, the candidate rewritings can be limited to those that have at most the number of subgoals in the query. The complexity of the problem is polynomial in the number of views (i.e., the number of data sources in the context of data integration). Since query containment is a special case of query folding, Qian's algorithm, (Qia96) degenerates to a polynomial-time containment algorithm for the class of acyclic conjunctive queries.

Abiteboul and Duschka (AD98) show that recursion and negation in the view definition lead to undecidability. They show that the *Closed World Assumption (CWA)* complicates the problem. Under the *Open World Assumption (OWA)*, the certain answers in the conjunctive view definitions/Datalog queries case can be computed in polynomial time. On the other hand, the conjunctive view definitions/conjunctive queries case is co-NP-complete under the CWA. They prove that inequalities (a weak form of negation) lead to intractability. Even under the OWA, adding inequalities to the queries, or disjunction to the view definitions make the problem co-NP hard. In his thesis, Duschka, (Dus97b) provides a summary of results in complexity associated with these results.

Chekuri and Rajaraman, (CR97), present polynomial-time algorithms to test the containment of an arbitrary conjunctive query in an acyclic query and to minimize an acyclic query. They generalize the query containment and minimization algorithms to arbitrary queries. They consider the problem of finding an equivalent rewriting of a conjunctive query Q using a set of views $\mathcal{V}$ defined by conjunctive queries, when Q does not have repeated predicates, and show how their algorithms for query containment can be modified for this problem. A restricted variant of this problem, where neither Q nor the views in $\mathcal{V}$ use repeated predicates, is known to be $NP - complete$ (LMSS95).

Duschka and Genesereth, (DG98), treat views that may be defined by disjunction. Their focus is on maximal query containment. They show a duality between a query plan being maximally contained in a query and this plan computing exactly the certain answers. They show that the plan they generate is maximally contained in the query and that the disjunctive plan can be evaluated in co-NP time. The complexity results described above also apply to the problems that we discuss in this paper.

Several systems, and a number of algorithms have been implemented for the folding problem. Levy, Rajaraman, and Ordille (LRO96b), developed the *Information Manifold System* at AT&T Labs. The system incorporates the *bucket algorithm*, which controls search by first considering each subgoal in a query in isolation, and creating a bucket that contains only the views relevant to that subgoal. The algorithm then creates rewritings by combining one view from each bucket.

Qian and Duschka and Genesereth (Qia96, DG97a, DG97b, Dus97b), are responsible for the *InfoMaster System*. They use the *inverse-rules algorithm*, and consider rewritings for each database relation independent of any particular query. Given a user query, these rewritings are combined appropriately. They show that rewritings produced by the inverse-rules algorithm need to be further processed in order to be appropriate

for query evaluation. They show this additional processing step duplicates much of the work done in the second phase of the bucket algorithm. The bucket algorithm is also shown to have several deficiencies and does not scale up. Details of these algorithms are presented by Levy in (Lev00).

Pottinger and Levy (PL00) have developed a scalable algorithm for answering queries with views. They describe and analyze the bucket and the inverse-rule algorithms. They then describe the MiniCon algorithm, for finding the maximally contained rewriting of a conjunctive query using conjunctive views. They provide the first experimental study of algorithms for answering queries. They show that the MiniCon algorithm both scales up and significantly outperforms the previous algorithms. They further develop an extension of the MiniCon algorithm to handle comparison predicates, and show its performance experimentally.

Afrati, Li, and Ullman (ALU01), discuss generating efficient, equivalent rewritings using views to compute the answer to a query. Each rewriting of a query is passed as a logical plan to an *optimizer*, which translates the rewriting to a *physical plan*. Each physical plan accesses the stored ("materialized") views, and applies a sequence of relational operators to compute the answer to the original query. They consider three cost models for evaluating the efficacy of a plan. They develop and experimentally evaluate an efficient algorithm, *CoreCover*, based on a simple cost model M_1 that counts only the number of subgoals in a physical plan.

3 Background

This section contains a summary of the background and notation used in this paper. We use the language and terminology of logic databases (also known as deductive databases) ((Das92), (Llo87), and (LMR92)). Logic databases express data, rules (views), integrity constraints and queries in first-order logic ((BJ89) and (Llo87)). [1]

We use standard syntax for first-order logic, with the usual symbols for variables, connectives, quantifiers, punctuation, equality, constants, and predicates. The notions of term, formula, and sentence (a formula with no free variables) are defined in the usual way. We do not necessarily restrict formulas to be function-free. We shall make it clear when we utilize function symbols. Any formula may be considered as a *query* (although, below, we restrict the form of standard queries). A formula is called *ground* if it contains no variables.

A *substitution* is a set of *substitution pairs*, for example, $\{X/a, Y/b\}$, such that every element of a substitution pair is a variable or constant (or, more generally, a term), and such that the collection of left-hand sides of the substitution pairs—X and Y in our example—are unique variables. A substitution *applied* to a formula is a rewrite of the formula by replacing any occurrence in the formula of a left-hand element from the substitution by its right-hand counterpart, in parallel. Let the formula $\mathcal{F}$ be $p(X, Y)$ and the substitution θ be again $\{X/a, Y/b\}$. The substitution θ applied to formula $\mathcal{F}$, written as $\mathcal{F}\theta$, is the formula $p(a,b)$.

We assume that the reader is familiar with the unification algorithm (Rob65). The unification algorithm takes a set of relations with the same relation name and attempts to find a substitution for the variables that will make the relations all identical.

An important class of sentence is the *clause*. A clause has the general form:

$$\forall \cdot A_1 \vee \ldots \vee A_k \vee \neg A_{k+1} \vee \ldots \vee \neg A_n \tag{1}$$

in which each A_i is an atomic formula, for $i \in \{1, \ldots, n\}$, and in which the variables are understood to

[1] Some proposals for integrity constraints express them in a higher-order logic, while keeping the database—the facts, rules, and, sometimes, queries—in first-order.

be universally quantified (denoted by '$\forall$'). Any clause can be written in a logically equivalent form as an implication.

$$\forall \cdot A_1 \vee \ldots \vee A_k \leftarrow A_{k+1} \wedge \ldots \wedge A_n \cdot \tag{2}$$

This is often written in further shorthand as

$$A_1, \ldots, A_k \leftarrow A_{k+1}, \ldots, A_n \cdot \tag{3}$$

in which disjunction is assumed on the left-hand side of the implication arrow, conjunction on the right-hand side, and the universal quantification is understood. A clause in this form is also called a *rule*. The collection of atoms on the left-hand side $(A_1, \ldots, A_k)$ is called the *head* of the rule, and the collection of atoms on the right-hand side $(A_{k+1}, \ldots, A_n)$ the *body*. When $k = n$, the body is empty, and when $k = 0$, the head is empty. A *Horn* rule has at most one atom in the head: $k \leq 1$. A *definite* clause has exactly one atom in the head: $k = 1$. A *ground* rule contains no variables. In a rule a variable is called *limited* if it appears in the body of an ordinary (non built-in) atom or in an equality with a constant or a variable that is limited. A rule is called *safe* if all its variables are limited.

A rule with an empty head is generally considered to be a *query* or an integrity constraint. An *answer* to a query is a ground substitution of the query formula such that the resulting ground formula is *true* with respect to the database; that is, the grounded query formula is *logically entailed* by the database. A ground rule with an empty body is called a *fact*. Definite rules have a clear procedural interpretation. Consider

$$A \leftarrow B_1, \ldots, B_n \cdot \tag{4}$$

We call this clause a rule *for* A. The above rule can be interpreted to say that A is shown (or *proven*) whenever all the B_i's are shown (*proven*).[2] Rules are essentially *views*, in the parlance of relational databases. Logically, rules are more expressive than views in relational databases because recursion is permitted.[3] A fact then, having no conditions in the body of its "rule", is simply interpreted as *true*. We define an *expanded rule* ((Cha85, CGM88)) to be one in which all predicates have been expanded. An *expanded predicate* is one in which all constants and repeated variables have been replaced by unique new variables, and the appropriate equalities have been added to the body of the rule in which the predicate appears ((CGM90)). This is related to the term *rectified* set of rules where the head of each rule in the set is identical with each argument a distinct variable ((Ull89)).

A database may then be defined as a collection of rules and facts. When all the rules and facts are definite (that is, the rules and facts have at most a single atom in the heads of the clauses that define them), the database is called *definite*. It is called *disjunctive* (or *indefinite*) otherwise. We call the language in which the database is written with definite clauses as defined above Datalog (Ull89). Recall that terms in clauses are function-free, as noted above, hence all Datalog terms are function-free. When disjunctive clauses are permitted for rules or facts (and whose terms are function-free), we call the language Disjunctive Datalog (EGM97, LMR92). A database **DB** often is defined as consisting of two parts:

- the extensional database, **EDB**, and
- the intensional database, **IDB**.

The **EDB** is the database's collection of *facts*. The **IDB** is the database's collection of rules. (We soon

[2] The interpretation for disjunctive rules is less apparent. Essentially, a disjunctive rule states that at least *one* of the atoms in the rule's head is *proven* whenever all the atoms in the body are.

[3] The SQL-3 standard extends SQL to support recursion (MS93), however. So once SQL-3 becomes the standard, this difference in expressiveness will go away, since any relational database that supports SQL-3 will, in fact, be a deductive database system.

redefine databases to have two additional components, the set of the database's integrity constraints (**ICs**) and the set of resource rules ($\mathbf{Res_{DB}}$)).

Conventionally, negative data is not represented explicitly in a logic database. There are several standard approaches to allow negative data to be inferred. The *closed world assumption* (*CWA*) is a default rule for the inference of negative facts (Rei78b). For any ground atom A, the negation of A is accepted as *true* if A is not provable from the database. The set of all negated atoms inferable in this way is written as **CWA [DB]**. Another approach to negation is the *Clark completion* of a database (Cla78). This formalizes the concept that the set of tuples *true* for a predicate is precisely the set that can be proven to be *true* via the facts and rules. In brief, this is accomplished by adding a formula to the database for each predicate (to correspond with the collection of rules for that predicate), to supply the logical *only if* half of the definition of the predicate. Certain negated facts are then *deducible* from the *completed* database, the database with these "only if" formulas added. We refer to (Cla78) for the precise definition. In our application we will typically have a situation where a resource predicate is defined by some extensional predicates, such as

$$r(X, Y) \leftarrow h(X, Z), k(Z, Y) \tag{5}$$

where r is the resource predicate and h and k are extensional predicates, saying essentially that if $< a, b >$ is in the join of h and k, it is in r. The Clark completion changes the implication to an equivalence to say that $< a, b >$ is in the join of h and k if and only if $< a, b >$ is in r. When we compute the Clark completion of a rule such as in clause 5 we obtain two rules, one for h and one for k. The variable Z in clause 5 represents an existensionally quantified variable whose value depends on the variables X and Y. When we obtain the only-if part of the Clark completion, namely, the rules with the implication arrow reversed, the predicates with the variable Z appear in the heads of the clauses and are replaced by the Skolem function $f(X, Y)$. Thus, the only-if portion of the Clark completion become:

$$h(X, f(X, Y)) \leftarrow r(X, Y) \tag{6}$$

$$k(f(X, Y), Y) \leftarrow r(X, Y) \tag{7}$$

In the text, when there is no loss of information, we omit the variable portion of the Skolem functions. For example, we replace $f(X, Y)$ by f.

In our formulas we allow built-in predicates, such as $=, <, >$. When built-in predicates occur in formulas the appropriate axioms need to be added to the theory. The axioms for equality, for example, are given below.

Equality Axioms:

$\forall X(X = X)$
$\forall X \forall Y((X = Y) \rightarrow (Y = X))$
$\forall X \forall Y \forall Z((X = Y) \wedge (Y = Z) \rightarrow (X = Z))$
$\forall X_1 \cdots \forall X_n(P(X_1, \cdots, X_n) \wedge (X_1 = Y_1) \wedge \cdots \wedge (X_n = Y_n) \rightarrow P(Y_1, \cdots Y_n))$

We will discuss the use of equality axioms when they are needed in proofs.

So far, we have assumed that the body of a database rule (clause) contains only positive atoms. However, it is useful sometimes to define database rules that allow negated atoms in the body of a rule. We need default negation in logic databases if we want to subsume the relational algebra, which includes set difference. We can extend deductive databases with default negation. A rule which has a negated atom, i.e. an atom preceded by *not* in its body is called a *normal* rule, and deductive databases that have normal rules are called *normal* databases. We call Datalog that has been extended with default negation Datalog$^\neg$. For example, the normal rule

$$p(X) \leftarrow not\, q(X) \cdot \tag{8}$$

is interpreted, in general, to mean that, for any constant a, if $q(a)$ is *not true* (or cannot be *proven* to be *true*), then $p(a)$ is *true*.

We write this negation with *not* rather than with the symbol for logical negation, '$\neg$', and refer to it as *default negation*. This is because most semantics that have been defined for normal databases, interpret the use of default negation differently from one another and from logical negation. There are a number of semantics that have been defined for normal databases, and no one semantics is universally accepted. Also, since the notion of default negation is generally based on provability, not logical truth, such default negation is beyond first-order logic.[4]

Thus it is not equivalent to exchange the rules in the **IDB** that use default negations with seemingly equivalent disjunctive rules, in which the negated atoms in the body have been moved to the head. Consider the following example.
DB_1 contains two clauses: (1) $p \leftarrow q, not\, r$, and (2) q.
DB_2 contains two clauses: (1) $p \vee r \leftarrow q$, and (2) q.
That is, let $\mathbf{DB}_1$ consist of the single rule for the predicate p and the single fact for the predicate q, and $\mathbf{DB}_2$ consist of the single disjunctive rule and the single fact for the predicate q. If the rule in $\mathbf{DB}_1$ were written with logical negation instead of default negation, $\mathbf{DB}_1$ and $\mathbf{DB}_2$ would be logically equivalent. However, in $\mathbf{DB}_1$, we should be able to infer p, because the fact q can be inferred (it is a fact), and the fact r *cannot* be inferred, (thus, $not\, r$ can be assumed *true* by default). In $\mathbf{DB}_2$, p cannot be inferred. Only the weaker, disjunctive fact $p \vee r$ can be inferred. Note that default negation results in non-monotonicity. If we were to *add* the fact r to $\mathbf{DB}_1$ above, we would no longer be able to infer p.

The intuition behind the use of default negation becomes confused when it is combined with recursion. One solution to this confusion is simply not to allow recursive definitions through negation. A canonical example of recursion through negation is

$$p(X) \leftarrow not\, q(X) \cdot \tag{9}$$

$$q(X) \leftarrow not\, p(X) \cdot \tag{10}$$

The restriction not to allow recursion through negation leads to what are called *stratified* databases, and such databases have a unique standard model called the *perfect model* of the database. (Stratified databases are defined in (ABW88, LMR92)). In some cases, a non-stratified database may also have a unique standard model. Some of these cases may be captured by the concept of *stable* database. Two important model semantics for normal databases, and normal logic programs, are the *well-founded semantics* (GRS91) and the *stable model semantics* or the semantically equivalent *well-supported model semantics* (Fag91, GL88). (Min96) provides a retrospective on work in semantics for logic programs and deductive databases.

4 Query Folding

This section contains the basic material on query folding. We assume that the resource rules define the predicates of the data sources that are conveniently available while the **EDB** and **IDB** predicates may take longer to use or may be unavailable. Consequently, a query is optimized in the sense that it has been rewritten using the data sources, which presumably are readily available. The folded query can then be optimized by other well-known techniques. In this section we provide an algorithm for such query rewriting in a special case. In later sections we extend this algorithm to more complex databases. We start by giving

[4] We still speak in terms of first-order logic even for normal databases, as most of the first-order framework of deductive databases remains applicable.

the restrictions on the type of database we consider in Sections 4 and 5. Our query folding algorithm is illustrated on an example before it is described. We end this section by giving several additional examples.

4.1 Database Restrictions

From now on a database will consist of four parts: the **EDB**, **IDB**, **IC**, and $\mathbf{Res_{DB}}$. We assume that the resource rules define the predicates of the data sources that are conveniently available while the **EDB** and **IDB** predicates may take longer to use. In this section we provide an algorithm for such query rewriting in a special case. In later sections we will extend this algorithm to more complex databases.

We place the following conditions on the database in this and the following section.

1. No formula contains negation.
2. Each **IDB** predicate may be defined by multiple safe, conjunctive, non-recursive function-free Horn rules.
3. Each distinct $\mathbf{Res_{DB}}$ predicate is defined by a single safe conjunctive function-free Horn formula on **EDB** and/or **IDB** predicates.
4. Each **IC** clause is a safe function-free Horn formula of the form $G \leftarrow F$ where G is either empty or has one **EDB** predicate and F is a conjunction of **EDB** predicates.
5. Each query has the form $q :\leftarrow G$ where G is a conjunction of **EDB** and **IDB** predicates.
6. The database includes axioms for built-in predicates as needed.

When the **IDB** is non-recursive, it has been shown (Rei78a) that the rules can be compiled so that every **IDB** predicate can be written as a set of rules, each rule in terms only of **EDB** predicates. We assume in this section that the compiled rules replace the original rules. Hence, deduction using **IDB** predicates is effectively one-step.

In the following we consider the concept of *bounded recursion*. Minker and Nicolas (MN82) were the first to show that there are forms of rules that lead to *bounded recursion*. That is, the deduction process using these rules must terminate in a finite number of steps. This work has been extended by Naughton and Sagiv (NS87). We illustrate here one special case of bounded recursion, namely, singular rules.

A recursive rule is *singular* if it is of the form

$$R \leftarrow F \wedge R_1 \wedge \wedge \ldots \wedge R_n,$$

where F is a conjunction of possibly empty base (i.e. *EDB*) relations and $R, R_1, R_2, \ldots, R_n$ are atoms that have the same relation name iff:

1. each variable that occurs in an atom R_i and does not occur in R only occurs in R_i;
2. each variable in R occurs in the same argument position in any atom R_i where it appears, except perhaps in at most one atom R_1 that contains all of the variables of R.

Thus, the rule

$$R(X, Y, Z) \leftarrow R(X, Y', Z), R(X, Y, Z')$$

representing the multivalued dependency, $R : X \rightarrow\rightarrow Y$ is singular since (a) Y' and Z' appear respectively in

the first and second atoms in the head of the rule (condition 1), and (b) the variables X, Y, Z always appear in the same argument position (condition 2). It is known that all singular rules have bounded recursion.

We now specify three cases for the **IC** since each case has to be handled in a different manner in the folding algorithm:
Cases of Integrity Constraints (ICs)

1. Case 1: The **ICs** have no recursion and no built-in predicate in the head of a clause.
2. Case 2: The **ICs** have bounded recursion and no built-in predicate in the head of a clause. For example this is the case if there is a multivalued dependency.
3. Case 3: Either the **ICs** are recursive, or there is a built-in predicate in the head of a clause. For example, for a functional dependency, an induced recursion may arise since the equality axioms are recursive.

4.2 Illustrative Example

We start by illustrating our algorithm on a simple example. This example has a simple integrity constraint. We deal with functional and inclusion dependencies in the next section. We write the formal description afterwards and show how it subsumes other algorithms used for this type of query rewriting.

Example 1. **EDB**: $p_1(X, Y, Z)$, $p_2(X, U)$, $p_3(X, Y)$
IDB: $\emptyset$
IC: $p_3(X, Y) \leftarrow p_1(X, Y, Z), Z > 0$
Res$_{\mathbf{DB}}$: $r(X, Y, Z) \leftarrow p_1(X, Y, Z), p_2(X, U)$
Query: $q(X, Y) :\leftarrow p_1(X, Y, Z), p_2(X, U), p_3(X, Y), Z > 1$

The first step involves reversing the resource rules to define the **EDB** predicates in terms of the resource predicates. This process supplies the only if half of the definition of the resource predicates; so we call these rules the Clark Completion resource rules. In this example we obtain for the Clark Completion resource rules:

$CCrr1 : p_1(X, Y, Z) \leftarrow r(X, Y, Z)$
$CCrr2 : p_2(X, f(X, Y, Z)) \leftarrow r(X, Y, Z)$

Figure 1 shows the derivation starting with the query as the top clause of a linear resolution tree. Both the **IC** and the Clark Completion resource rules are used. The clause $Z > 0$ is subsumed by the clause $Z > 1$. The final rewritten query at the bottom of the tree
$\leftarrow r(X, Y, Z), Z > 1,$
contains only the r predicate and the evaluable predicate $>$. Using this query we obtain correct answers to the original query.

This example uses an integrity constraint to obtain a clause that contains only a resource predicate and an evaluable predicate. When the theory is Horn, Reiter has shown (Rei78a) that the use of integrity constraints is not necessary to obtain answers. In this case, if the integrity constraint were not used, the clause at the end of the proof tree would have been a partial folding consisting of a resource predicate, an **EDB** predicate and an evaluable predicate. Using the integrity constraint eliminates the **EDB** predicate and provides an optimization step, as in the case of semantic query optimization (CGM90). We can illustrate another aspect of semantic query optimization by changing the integrity constraint in this example to:

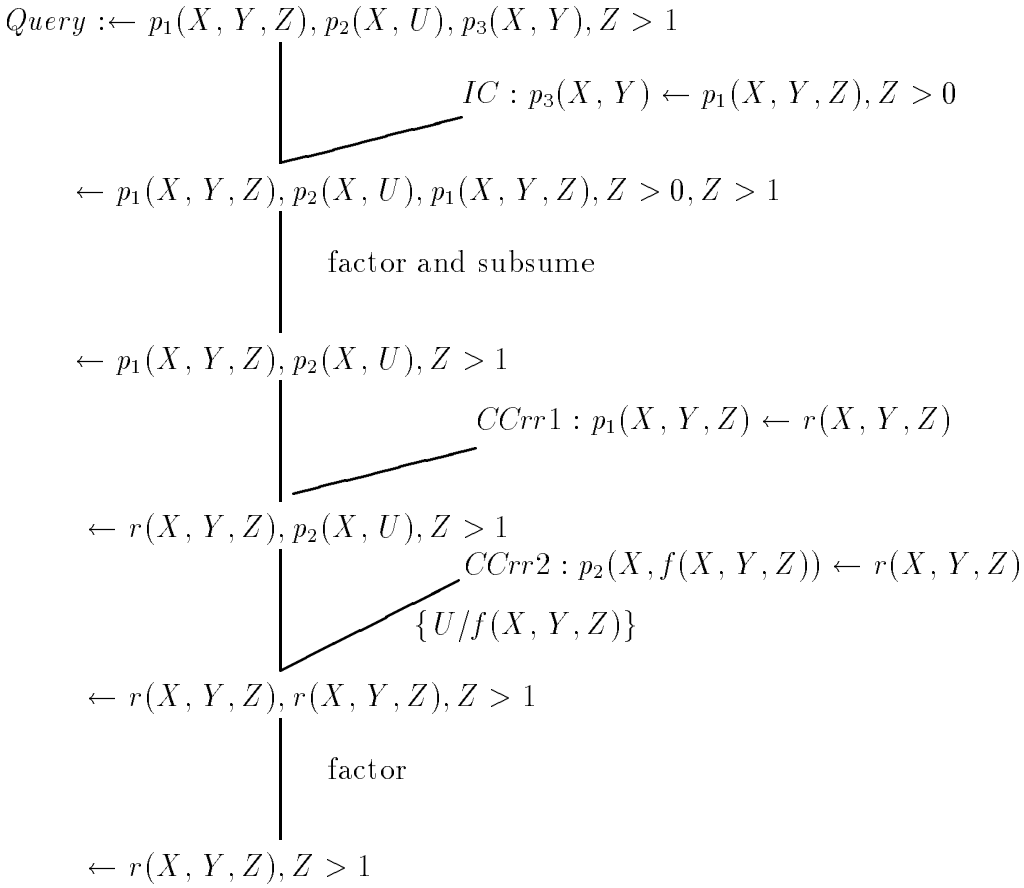

Fig. 1. Literal Elimination Example

IC: $\leftarrow p_2(X, U), p_3(X, Y)$

This integrity constraint subsumes the query; hence the query has no answers and there is no need to try to fold the query.

4.3 Query Folding Algorithm

At this point we describe the first version of the query folding algorithm, where the database satisfies the six conditions given at the beginning of Section 4.1. In particular each resource predicate is defined by a single safe conjunctive formula on **EDB**.

As mentioned in Example 1 the algorithm uses the Clark Completion resource rules for the resource predicates. We obtain these rules by a preprocessing algorithm that needs to be done only once for a database.

Preprocessing Algorithm (Clark Completion)

Input: Res$_{\mathbf{DB}}$. We may assume that each resource predicate r is written in the form
$r(\bar{X}) \leftarrow p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$,
where $\bar{X}$ contains variables, each $\bar{X}_i$ $(1 \leq i \leq n)$ consists of terms (constants or variables) and $\bar{X} \subseteq \bigcup_{i=1}^{n} \bar{X}_i$.

Output: The Clark Completion resource rules **CCrr** in clausal form.

begin

Step 1. Apply the Clark Completion to each resource predicate definition to write it as
$r(\bar{X}) \leftrightarrow \exists \bar{Z}(p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n))$
where $\bar{Z}$ is the set of variables in $(\bigcup_{i=1}^{n} \bar{X}_i) - \bar{X}$.

Step 2. Rewrite the equivalences obtained in Step 1 into rules in clausal form, called the Clark Completion resource rules (**CCrr**) as
$r(\bar{X}) \leftarrow p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$
$p_1(\bar{X}_1') \leftarrow r(\bar{X})$
$\vdots$
$p_n(\bar{X}_n')) \leftarrow r(\bar{X})$
where $\bar{X}_i'$ $(1 \leq i \leq n)$ is obtained from $\bar{X}_i$ by replacing every variable $X_j \in \bar{Z}$ by $f_{r,j}(\bar{X})$. (In our examples we will usually use variables such as X, Y, and Z, and function symbols f, g, and h, and omit subscripts.)

end

We now describe the simplest form of the folding algorithm, the one with the database restrictions of Section 4.1.

The algorithm described below uses linear derivation ((CL73)) which includes a backtracking mechanism. Backtracking occurs when we find a linear derivation that has no resource predicates. Before the algorithm commences we assume that there is a test to determine which of the three cases applies to **ICs**. In Case 1, nothing has to be done. In Case 2, we assume that the linear derivation is modified to include a check to determine if the clause, L, that has been generated, satisfies the bounding condition, and if it does, then backtracking occurs. In Case 3, a depth bound k is specified and if the depth is reached, backtracking occurs. We also assume that if there are built-in predicates such as $=, \neq, \geq$, then the input clauses are placed in expanded form. If there are no built-in predicates, it is unnecessary to do the expansion.

Folding Algorithm 1 (Finding a Single Folding)

Input: $\mathcal{C}$: the set of clauses in the **EDB**, **IDB**, **IC**, and **CCrr**, and
the query, $q(\bar{X})$: $\leftarrow G(\bar{Y})$, where $\bar{X} \subseteq \bar{Y}$, and G is a conjunction of atoms. We call $\bar{X}$ the query variables.

Output: A query $fq(\bar{X})$: $\leftarrow L(\bar{Z})$, where $\bar{X} \subseteq \bar{Z}$.

begin
Starting with $\leftarrow G$ find a linear derivation using $\mathcal{C}$ that results in a clause $\leftarrow L$ that contains the query variables and no function symbols. When L contains at least one resource predicate, and no **EDB** predicates, it constitutes a complete folding; otherwise L may contain some **EDB** predicates and hence it constitutes a partial folding.
end

In our figures, we show illustrative derivations, but not the detailed steps leading to that derivation that may have arisen by backtracking. Starting with $\leftarrow G$, we essentially find a linear derivation using $\mathcal{C}$ that results in a clause $\leftarrow L$ that contains the query variables. If L contains at least one resource predicate, and no **EDB** predicates, then it constitutes a complete folding; otherwise L may contain some **EDB** predicates, then it constitutes a partial folding. Note that when the algorithm terminates with an answer, backtracking may find additional answers.

At any point if an integrity constraint subsumes a clause in the linear derivation, the process backs up because the query cannot have any answers along that path. We have omitted this step since subsumption is time consuming. An algorithm for subsumption may be found in (CL73). If a clause in a derivation contains

a constant, the expansion of a clause may allow us to derive a solution. This modification is omitted from the algorithm.

Next we show that the Folding Algorithm is *sound* or *correct*. By this we mean that every tuple obtained by solving the query $fq(\bar{X})$ is also obtained by solving the query $q(\bar{X})$. That is, every answer to a folded query is an answer to the original query.

Theorem 1. The Folding Algorithm is sound (correct).

Proof. Using the notation $q(\bar{X}) :\leftarrow G(\bar{Y})$ and $fq(\bar{X}) :\leftarrow L(\bar{Z})$, by the soundness of resolution, we obtain
$(\leftarrow G(\bar{Y})) \cup \mathcal{C} \models (\leftarrow L(\bar{Z}))$,
so, by logical equivalence, we have,
$L(\bar{Z}) \cup \mathcal{C} \models \leftarrow G(\bar{Y})$.
Suppose that $\bar{a}$ is a solution to $fq(\bar{X})$ in DB. This means that there is a $\bar{b}$, with $\bar{b}[\bar{X}] = \bar{a}$, such that $DB \models L(\bar{b})$. Also, for every formula $C \in \mathcal{C}$, $DB \models C$. Therefore, there is a $\bar{d}$, where $\bar{d}[\bar{X}] = \bar{a}$, such that $DB \models G(\bar{d})$. But this means that $\bar{a}$ is a solution to $q(\bar{X})$ in DB. □

4.4 Additional Examples

Next we apply our algorithm to various examples considered by researchers and show that our algorithm can be used to obtain the same results. We start by taking two examples from (Qia96). In that paper the **EDB** consists of six relations that represent a patient record database:

Example 2. (Examples 2 and 3, Qian (Qia96))

Patients (patient_id,clinic,dob,insurance)
Physician (physician_id,clinic,pager_no)
Drugs (drug_name,generic?)
Notes (note_id,patient_id,physician_id,note_text)
Allergy (note_id,drug_name,allergy_text)
Prescription(note_id,drug_name,prescription_text)
The **IDB** and **IC** are empty; hence Case 1 of the **IC** Cases (see page 9) applies. The $\mathbf{Res_{DB}}$ consists of two relations, **Drug_Allergy** and **Prescribed_Drug**. For convenience we write **Drug_Allergy** as r_1 and **Prescribed-Drug** as r_2. In **Datalog** they are expressed as:

$$r_1(X_1, X_2, X_3) \leftarrow notes(U_1, X_1, U_2, U_3), allergy(U_1, X_2, X_3)$$

$$r_2(Y_1, Y_2, Y_3, Y_4) \leftarrow notes(V_1, Y_1, Y_2, V_2), prescription(V_1, Y_3, V_3), drugs(Y_3, Y_4).$$

Preprocessing yields two Clark completion resource rules for r_1 and three Clark completion resource rules for r_2. In the following formulae the functions f_i, g_j are abbreviations for the Skolem functions $f_i(X_1, X_2, X_3)$ and $g_j(Y_1, Y_2, Y_3, Y_4)$, respectively.

$$CCrr1 : notes(f_1, X_1, f_2, f_3) \leftarrow r_1(X_1, X_2, X_3) \tag{11}$$

$$CCrr2 : allergy(f_1, X_2, X_3) \leftarrow r_1(X_1, X_2, X_3) \tag{12}$$

$$CCrr3 : notes(g_1, Y_1, Y_2, g_2) \leftarrow r_2(Y_1, Y_2, Y_3, Y_4) \tag{13}$$

$$CCrr4 : prescription(g_1, Y_3, g_3) \leftarrow r_2(Y_1, Y_2, Y_3, Y_4) \tag{14}$$

$$CCrr5 : drugs(Y_3, Y_4) \leftarrow r_2(Y_1, Y_2, Y_3, Y_4) \tag{15}$$

Let's consider first the query of Examples 2 and 3 of (Qia96):

$$q(X, Y) :\leftarrow notes(W_1, X, W_2, W_3), allergy(W_1, Y, W_4), notes(W_5, X, W_6, W_7), prescription(W_5, Y, W_8)$$

Again, as in our first example, we start with the body of the query to find a derivation:

$$\leftarrow notes(W_1, X, W_2, W_3), allergy(W_1, Y, W_4), notes(W_5, X, W_6, W_7), prescription(W_5, Y, W_8)$$

The derivation is shown in Figure 2. Four of the Clark Completion resource rules are used. The rewritten query at the bottom of the tree,
$\leftarrow r_1(X, Y, X_3), r_2(X, Y_2, Y, Y_4)$
consists of only resource predicates.

Now we consider Example 6 of (Qia96).

Example 3. (Example 6 of Qian (Qia96)) The **EDB**, **IDB** and **IC** are the same as before. $\mathbf{Res_{DB}}$ consists of one relation r defined as follows:

$$r(X_1, X_2, X_3) \leftarrow notes(U_1, X_1, X_2, U_2), prescription(U_1, X_3, U_3), drugs(X_3, U_4)$$

.

Preprocessing yields three Clark completion resource rules for r as follows:

$$CCrr1 : notes(f_1, X_1, X_2, f_2) \leftarrow r(X_1, X_2, X_3) \tag{16}$$

$$CCrr2 : prescription(f_1, X_3, f_3) \leftarrow r(X_1, X_2, X_3) \tag{17}$$

$$CCrr3 : drugs(X_3, f_4) \leftarrow r(X_1, X_2, X_3) \tag{18}$$

The query of this example is:

$$q(X, Y) :\leftarrow patients(X, W_1, W_2, medicare), notes(W_3, X, W_4, W_5), prescription(W_3, Y, W_6), drugs(Y, no)$$

For simplicity, we did not place the query in expanded form. If we had, at the end we would have had to change the new variable back to the constant which it replaced. The derivation is shown in Figure 3. Two of the Clark completion resource rules are used. The rewritten query at the bottom of the tree

$$\leftarrow patients(X, W_1, W_2, medicare), r(X, W_4, Y), drugs(Y, no)$$

consists of the resource predicate (replacing two extensional predicates) and two extensional predicates. This is an example of a partial folding.

5 Handling Functional and Inclusion Dependencies

This section illustrates the use of our algorithm in the special cases where the integrity constraints are functional and inclusion dependencies. As explained earlier, the presence of functional dependencies means that Case 3 for **ICs** applies (see page 9). (DGQ96) gives algorithms in the presence of functional dependencies. Their basic idea is to use a functional dependency to decompose a relation into several relations, using a lossless join decomposition, and then apply the standard folding algorithm. We show that such a decomposition is not necessary. Instead of decomposing a relation that contains a functional dependency, we generate additional Clark completion rules and then apply our standard algorithm.

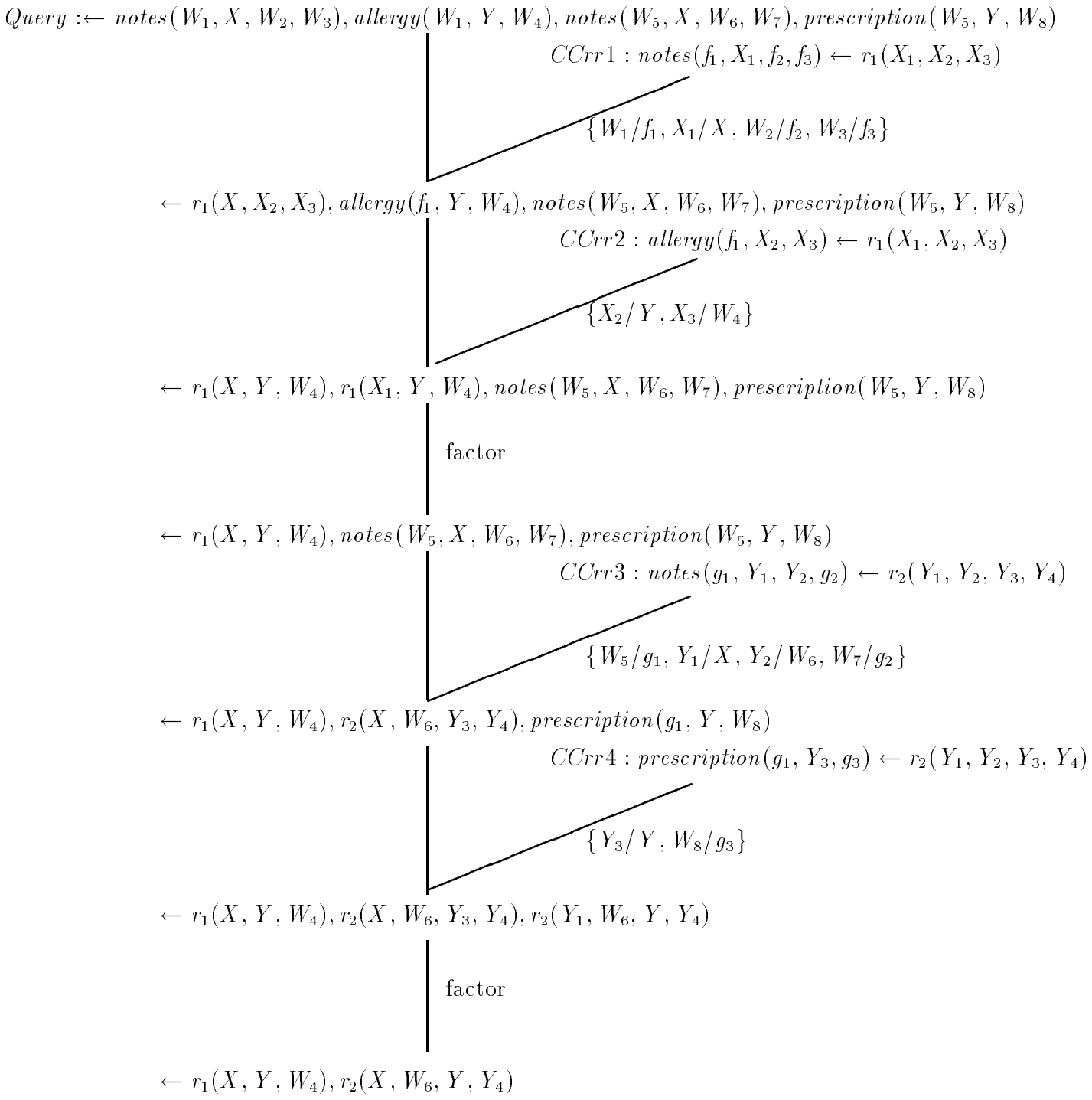

Fig. 2. Example 2 from (Qia96)

5.1 Example with Key Constraint

We consider how our algorithm applies to Example 3 of (DGQ96). The **EDB** contains three relations, again involving a patient record database as follows:

Patients (name,dob,insurance)
Procedure (patient_name,physician_name,procedure_name,time)

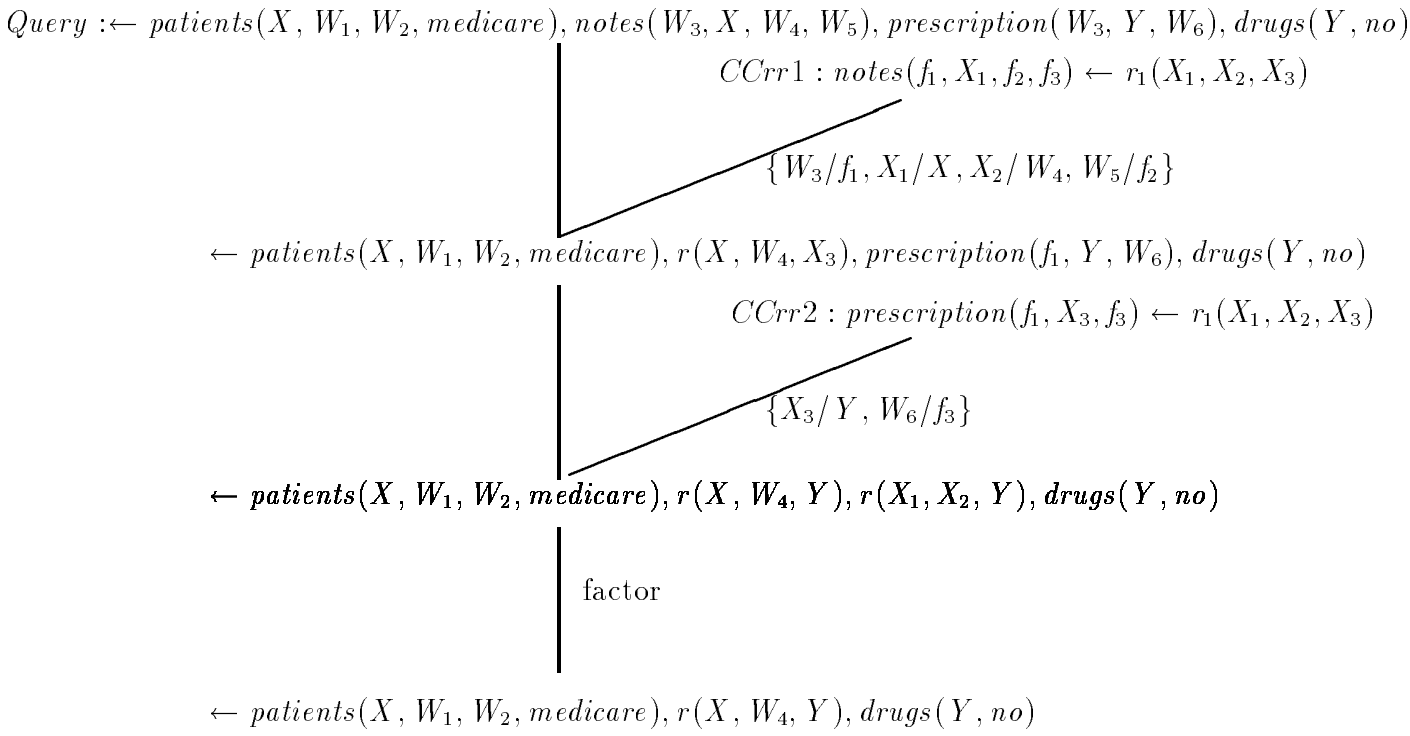

Fig. 3. Example 6 from (Qia96)

Insurer (company,address,phone)

Again, $\mathbf{IDB} = \emptyset$. The $\mathbf{Res_{DB}}$ consists of two relations **Clinical_History** and **Billing** which we write as r_1 and r_2 with the following definitions:

$$r_1(X_1, X_2, X_3, X_4) \leftarrow patients(X_1, X_2, U_1, U_2), procedure(X_1, U_3, X_3, X_4)$$

$$r_2(Y_1, Y_2, Y_3) \leftarrow patients(Y_1, V_1, Y_2, V_2), insurer(V_2, Y_3, V_3)$$

The **IC** contains the key constraint $patients : patient\_id \rightarrow clinic, dob, insurance$, written in Datalog as three clauses:
$X_2 = Y_2 \leftarrow patients(X_1, X_2, X_3, X_4), patients(X_1, Y_2, Y_3, Y_4)$
$X_3 = Y_3 \leftarrow patients(X_1, X_2, X_3, X_4), patients(X_1, Y_2, Y_3, Y_4)$
$X_4 = Y_4 \leftarrow patients(X_1, X_2, X_3, X_4), patients(X_1, Y_2, Y_3, Y_4)$

Preprocessing yields four Clark completion resource rules for r_1 and r_2 as follows:

$$CCrr1 : patients(X_1, X_2, f_1, f_2) \leftarrow r_1(X_1, X_2, X_3, X_4) \tag{19}$$

$$CCrr2 : procedure(X_1, f_2, X_3, X_4) \leftarrow r_1(X_1, X_2, X_3, X_4) \tag{20}$$

$$CCrr3 : patients(Y_1, g_1, Y_2, g_2) \leftarrow r_2(Y_1, Y_2, Y_3) \tag{21}$$

$$CCrr4 : insurer(g_2, Y_3, g_3) \leftarrow r_2(Y_1, Y_2, Y_3) \tag{22}$$

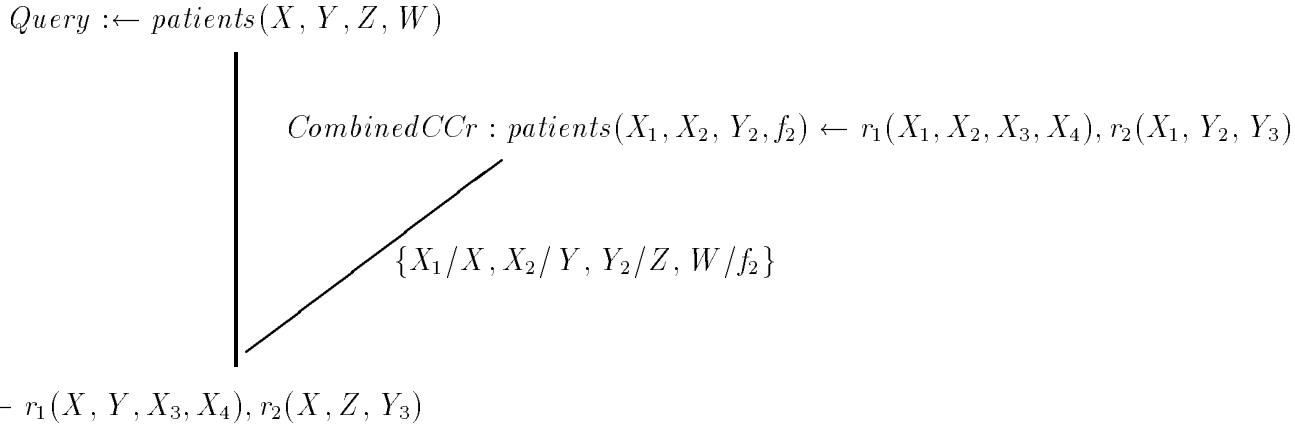

Fig. 4. Key Constraint Example

As we will show below, in the case of a key constraint it is sometimes possible to combine Clark completion resource rules. In this particular case the rules Equation 19 and Equation 21 can be combined to yield

$$CombinedCCrr : patients(X_1, X_2, Y_2, f_2) \leftarrow r_1(X_1, X_2, X_3, X_4), r_2(X_1, Y_2, Y_3) \tag{23}$$

The query of this example is:

$$q(X, Y, Z) :\leftarrow patients(X, Y, Z, W)$$

As shown in Figure 4, starting with the body of this query and using the combined Clark Completion resource rule, the derivation takes one step to obtain

$$\leftarrow r_1(X, Y, X_3, X_4), r_2(X, Z, Y_3)$$

We note, however, that we could not answer the query

$$q(X, Y, Z, W) :\leftarrow patients(X, Y, Z, W)$$

this way because W does not appear in the folded query.

5.2 Combining Clark Completion Resource Rules

In the presence of functional dependencies it is possible under certain conditions to combine Clark completion resource rules for the same predicate in such a way that function symbols are replaced by variables. Using the combined rules may simplify the derivation of the folded query. In this subsection we deal with the special useful case of key constraints.

We start by introducing notation involved in combining two Clark completion resource rules. The same basic method as described below will handle more than two rules. We assume that there exist two such rules of the form

$$p(\bar{X}f) \leftarrow r_1(\bar{X}) \tag{24}$$

$$p(\bar{Y}g) \leftarrow r_2(\bar{Y}) \tag{25}$$

where p is an n-ary predicate, $\bar{X}f$ is an n-tuple all of whose variables are in $\bar{X}$ and may contain functions

symbols f_i, and $\bar{Y}g$ is an n-tuple all of whose variables are in $\bar{Y}$ and may contain functions symbols g_j. For any tuple $\bar{U}$ we write $\bar{U}[i]$ for the i-th component of $\bar{U}$. Define the n-ary tuple $\bar{Z}$

$$\bar{Z}[i] = \begin{cases} \bar{Y}g[i] & \text{if } \bar{X}f[i] \text{ is a function symbol and } \bar{Y}g[i] \text{ is a variable} \\ \bar{X}f[i] & \text{otherwise} \end{cases}$$

Also define the tuple $\bar{Y}/(\bar{X}\,k)$ to have the same number of components as $\bar{Y}$ and defined as

$$\bar{Y}/(\bar{X}\,k)[i] = \begin{cases} \bar{X}[i] & \text{if } 1 \leq i \leq k \\ \bar{Y}[i] & \text{if } k < i \end{cases}$$

Proposition 1. For the two rules given in 24 and 25, if $\bar{X}f[i]$ and $\bar{Y}g[i]$ are variables for $1 \leq i \leq k$ and the first k columns of p form a key for p, then the combined Clark completion resource rule written as

$$p(\bar{Z}) \leftarrow r_1(\bar{X}), r_2(\bar{Y}/(\bar{X}\,k)) \tag{26}$$

is also a valid rule.

Proof. Proof: By 24

$$p(\bar{X}f) \leftarrow r_1(\bar{X}), r_2(\bar{Y}/(\bar{X}\,k)) \tag{27}$$

By 25

$$p(\bar{Y}g/(\bar{X}\,k)) \leftarrow r_1(\bar{X}), r_2(\bar{Y}/(\bar{X}\,k))\cdot \tag{28}$$

By the hypothesis that $\bar{X}f[i]$ and $\bar{Y}g[i]$ are variables for $1 \leq i \leq k$, we obtain $\bar{X}f[i] = \bar{Y}g/(\bar{X}\,k)$ for $1 \leq i \leq k$. As the first k columns of p form a key, the corresponding elements of $\bar{X}f$ and $\bar{Y}g/(\bar{X}\,k)$ must be equal. Since $\bar{Z}$ contains the first k columns of $\bar{X}f$ and the rest of the columns are from $\bar{X}f$ or $\bar{Y}g$,

$$p(\bar{Z}) \leftarrow r_1(\bar{X}), r_2(\bar{Y}/(\bar{X}\,k)) \tag{29}$$

follows. □

Going back to the example of Section 4.1, $CCrr1$ and $CCrr3$ are two Clark completion resource rules for the *patients* predicate. The first attribute is the key and both have a variable for the first attribute: X_1 and Y_1. Now set $Y_1 = X_1$. This forces $g_1 = X_2$, $f_1 = Y_2$, and $g_2 = f_2$, and we obtain the *Combined CCrr*.

5.3 Using the Lossless Join Decomposition Property for Key Constraints

In (DGQ96) the query in 4.1 is solved using a property of key constraints. Namely, the key constraint *patients* : $patient\_id \rightarrow clinic, dob, insurance$ implies that the decomposition of the relation $patients(patient\_id, clinic, dob, insurance)$ to the three relations $pat1(patient\_id, clinic)$, $pat2(patient\_id, dob)$, $pat3(patient\_id, insurance)$ is a lossless join decomposition. Therefore we can deal with the three relations $pat1$, $pat2$, $pat3$ instead of *patients*.

Now, the **Res_DB** relations are defined as follows:

$$r_1(X_1, X_2, X_3, X_4) \leftarrow pat1(X_1, X_2), pat2(X_1, U_1), pat3(X_1, U_2), procedure(X_1, U_3, X_3, X_4) \tag{30}$$

$$r_2(Y_1, Y_2, Y_3) \leftarrow pat1(Y_1, V_1), pat2(Y_1, Y_2), pat3(Y_1, V_2), insurer(V_2, Y_3, V_3) \tag{31}$$

and there are eight Clark Completion resource rules:

$$CCrr1 : pat1(X_1, X_2) \leftarrow r_1(X_1, X_2, X_3, X_4) \tag{32}$$

$$CCrr2 : pat2(X_1, f_1) \leftarrow r_1(X_1, X_2, X_3, X_4) \tag{33}$$

$$CCrr3 : pat3(X_1, f_2) \leftarrow r_1(X_1, X_2, X_3, X_4) \tag{34}$$

$$CCrr4 : procedure(X_1, f_3, X_3, X_4) \leftarrow r_1(X_1, X_2, X_3, X_4) \tag{35}$$

$$CCrr5 : pat1(Y_1, g_1) \leftarrow r_2(Y_1, Y_2, Y_3) \tag{36}$$

$$CCrr6 : pat2(Y_1, Y_2) \leftarrow r_2(Y_1, Y_2, Y_3) \tag{37}$$

$$CCrr7 : pat3(Y_1, g_2) \leftarrow r_2(Y_1, Y_2, Y_3) \tag{38}$$

$$CCrr8 : insurer(g_2, Y_3, g_3) \leftarrow r_2(Y_1, Y_2, Y_3) \tag{39}$$

The query

$$q(X, Y, Z) :\leftarrow patients(X, Y, Z, W) \tag{40}$$

is rewritten as

$$q(X, Y, Z) :\leftarrow pat1(X, Y), pat2(X, Z), pat3(X, W) \tag{41}$$

but $pat3(X, W)$ is superfluous, because the query does not contain W, hence we obtain

$$q(X, Y, Z) :\leftarrow pat1(X, Y), pat2(X, Z) \tag{42}$$

The derivation is shown in Figure 5. Two of the eight Clark Completion resource rules are used. The final query is the same as the one we obtained using the Combined Clark Completion rule. Again, if the query were

$$q(X, Y, Z, W) :\leftarrow patients(X, Y, Z, W) \tag{43}$$

then we could not obtain a complete folding because after applying $CCrr1$ and $CCrr6$ we would be left with

$$\leftarrow r_1(X, Y, X_3, X_4), r_2(X, Z, Y_3), pat3(X, Y) \tag{44}$$

and now applying applying either $CCrr3$ or $CCrr7$ would lead to a function symbol for one of the variables in r_1 or r_2.

5.4 Decomposition Cannot Always Handle Functional Dependencies

Our next example also contains a functional dependency, but in this case the functional dependency cannot be handled by a decomposition. However, our algorithm can be used to obtain a folding.
The **EDB** consists of two relations $\mathbf{p_1}(X, Y)$ and $\mathbf{p_2}(X, Y)$, the **IDB** is empty and the **IC** contains the key constraint $p_2 : X \rightarrow Y$, written as:
$Y = Y' \leftarrow p_2(X', Y), p_2(X', Y')$.
Note that neither **EDB** relation can be decomposed.
The **Res_DB** consists of two relations:

$$r_1(X, Z) \leftarrow p_1(X, W), p_2(Z, W) \tag{45}$$

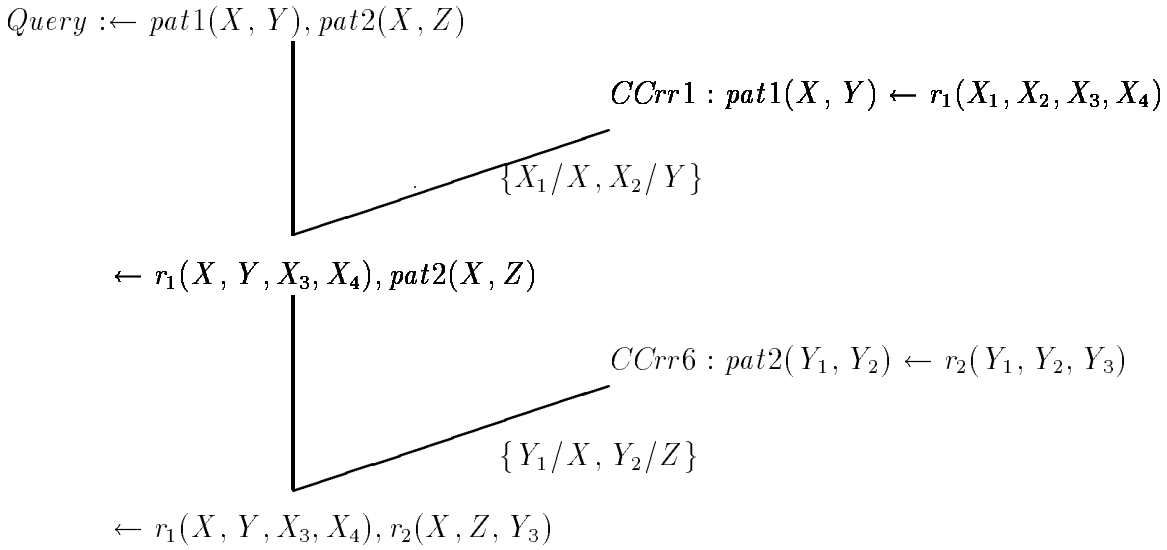

Fig. 5. Key Constraint Example Using Relation Decomposition

$$r_2(X, Y) \leftarrow p_2(X, Y) \tag{46}$$

Preprocessing yields two Clark completion resource rules for r_1 and one Clark completion resource rule for r_2 as follows:

$$p_1(X, f) \leftarrow r_1(X, Z) \tag{47}$$

$$p_2(Z, f) \leftarrow r_1(X, Z) \tag{48}$$

$$p_2(X, Y) \leftarrow r_2(X, Y) \tag{49}$$

Consider the query

$$q(X) :\leftarrow p_1(X, c) \tag{50}$$

The solution is given in Figure 6. We start by expanding the query, that is, rewriting the query to

$$q(X) :\leftarrow p_1(X, W), W = c \tag{51}$$

in order to take the constant out of the predicate allowing for a substitution later. All three Clark completion resource rules are used in the derivation to obtain

$$\leftarrow r_1(X, Z), r_2(Z, c) \tag{52}$$

which is a complete folding.

We note a generalization of the above result:
Suppose the **EDB** consists of n relations
$p_1(X, Y_2, \ldots, Y_n), p_2(X, Y_2), \ldots, p_n(X, Y_n)$ with the **IC** containing the $n-1$ key constraints $p_i : X \rightarrow Y_i$ for i = 2, ..., n and the **Res_DB** consisting of the n relations r_1, r_2, ..., r_n, defined as

$$r_1(X, Z) \leftarrow p_1(X, W_2, \ldots, W_n), p_2(Z, W_2), \ldots, p_n(Z, W_n) \tag{53}$$

$$r_2(X, Y) \leftarrow p_2(X, Y) \ldots \tag{54}$$

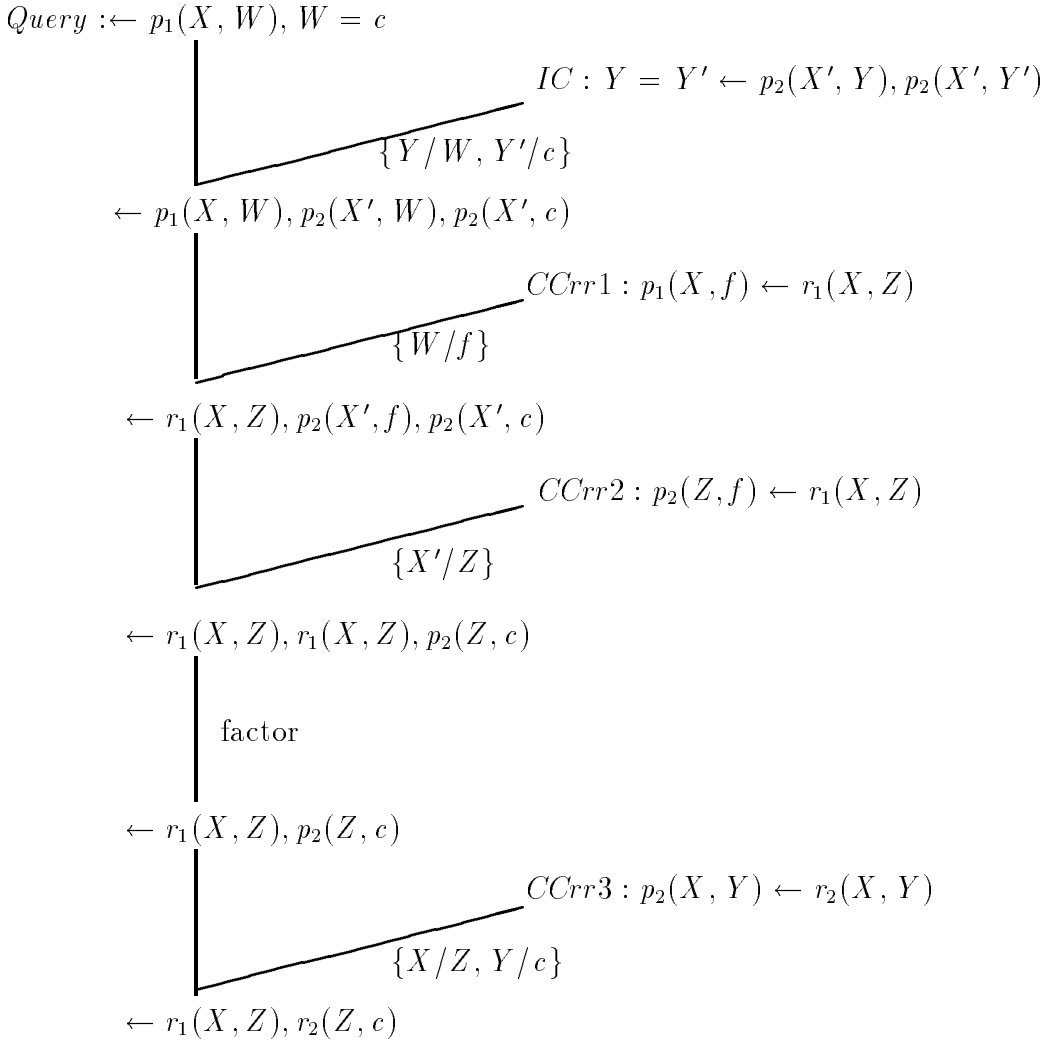

Fig. 6. Functional Dependency That Cannot Be Handled by Decomposition

$$\vdots$$

$$r_n(X, Y) \leftarrow p_n(X, Y) \ldots \tag{55}$$

where the ... indicate the possibility of additional predicates.
Then the query

$$q(X) :\leftarrow p_1(X, c_2, \ldots, c_n) \tag{56}$$

can be folded as

$$\leftarrow r_1(X, Z), r_2(Z, c_2), \ldots, r_n(Z, c_n) \tag{57}$$

5.5 Functional Dependencies and Recursion

The following example, discussed in (DGL00) illustrates how functional dependencies may introduce recursion. We now consider how this is handled in our approach and show that, although the non built-in predicates are not recursive, recursion is introduced by the recursive transitivity rule of equality.

Example 4. **EDB**: $schedule(Airline, Flight-No, Date, Pilot, AirCraft)$

IDB: $\emptyset$

IC:

$A_1 = A_2 \leftarrow s(A_1, N_1, D_1, P_1, C_1), s(A_2, N_2, D_2, P_1, C_2)$

(i.e., the functional dependency $Pilot \rightarrow Airline$)

$A_1 = A_2 \leftarrow s(A_1, N_1, D_1, P_1, C_1), s(A_2, N_2, D_2, P_2, C_1)$

(i.e., the functional dependency $Aircraft \rightarrow Airline$)

Res$_{\mathbf{DB}}$: $r(D, P, C) \leftarrow s(A, N, D, P, C)$

Query: $q(P) :\leftarrow s(A, N, D, mike, C), s(A, N', D', P, C')$

$CCrr : s(f(D, P, C), g(D, P, C), D, P, C) \leftarrow r(D, P, C)$

In (DGL00), they discuss how they obtain an infinite set of folded queries, one for each n of the form:

$$q_n(P) \leftarrow r(D_1, mike, C_1), r(D_2, P_2, C_1), r(D_3, P_2, C_2), r(D_4, P_3, C_2), \ldots,$$
$$r(D_{2n-2}, P_n, C_{n-1}), r(D_{2n-1}, P_n, C_n), r(D_{2n}, P, C_n)$$

Using our approach we start with the expanded clause:

$$\leftarrow s(A, N, D, P', C), s(A', N', D', P, C'), A = A', P' = mike$$

By applying the functional dependency $C \rightarrow A$, and factoring the resolvent clause twice, and applying the $CCrr$ twice, we obtain the clause

$$\leftarrow r(D, P', C), r(D', P, C), P' = mike$$

which is equivalent to $q_1(P), i > 1$. Although we do not provide the details here, in order to obtain the other $q_i(P)$, we need to use the recursive transitivity rule of equality as well as the two integrity constraints several times and factoring several times..

5.6 Inclusion Dependency Example

The last example of this section illustrates the use of an inclusion dependency. This example is taken from Example 4.3 of (Gry98) and contains an example given earlier with some modifications:
The **EDB** contains four relations:
Patient (name,dob,address,insurer)
Procedure (patient_name,physician_name,procedure_name,time)
Insurer (company,address,phone)
Event (event_name,description,patient_name,location)
The **IDB** is empty.
The **IC** contains a single inclusion dependency:
Procedure(procedure_name,patient_name) $\subseteq$ *Event(event_name,patient_name)*

written in **Datalog** as
$event(X_3', f_1, X_1', g) \leftarrow procedure(X_1', X_2', X_3', X_4')$.
As before, the **Res_DB** consists of two relations **Clinical_History** and **Billing**, written as r_1 and r_2 with the following definitions:

$$r_1(X_1, X_2, X_3, X_4) \leftarrow patient(X_1, X_2, U_1, U_2), procedure(X_1, U_3, X_3, X_4) \tag{58}$$

$$r_2(Y_1, Y_2, Y_3) \leftarrow patient(Y_1, V_1, Y_2, V_2), insurer(V_2, Y_3, V_3) \tag{59}$$

Preprocessing yields the same Clark completion resource rules as before:

$$CCrr1 : patient(X_1, X_2, f_1, f_2) \leftarrow r_1(X_1, X_2, X_3, X_4) \tag{60}$$

$$CCrr2 : procedure(X_1, f_3, X_3, X_4) \leftarrow r_1(X_1, X_2, X_3, X_4) \tag{61}$$

$$CCrr3 : patient(Y_1, g_1, Y_2, g_2) \leftarrow r_2(Y_1, Y_2, Y_3) \tag{62}$$

$$CCrr4 : insurer(g_2, Y_3, g_3) \leftarrow r_2(Y_1, Y_2, Y_3) \tag{63}$$

The query asks for the names of events recorded for patients born before 1930:

$$q(X_3) :\leftarrow patient(X_1, X_2, Y_1, Y_2), event(X_3, Y_3, X_1, Y_4), X_2 \leq 1930 \tag{64}$$

The derivation is shown in Figure 7. Using the inclusion dependency and two Clark completion resource rules we obtain the answer as

$$\leftarrow r_1(X_1, X_2, X_3, X_4), X_2 \leq 1930 \tag{65}$$

6 Multiple Definitions for Resources

In this section we consider the case where there are resources that were developed from several definitions or queries. This is in contrast to work in the previous section where it was assumed that a resource was constructed from a single conjunctive view. We also allow queries that are in disjunctive normal form, that is, the queries involve disjunctions of conjunctions. Our assumptions for the database are different from the ones given at the beginning of Section 4. We list our assumptions here.

1. No formula contains negation.
2. Each **IDB** predicate may be defined by multiple safe, conjunctive, non-recursive function-free Horn rules.
3. Each $\mathbf{Res_{DB}}$ predicate is defined by a set of safe conjunctive function-free Horn formulas on **EDB** and/or **IDB** predicates.
4. Each **IC** clause is a safe function-free formula of the form $G \leftarrow F$ where G is a disjunction of zero or more **EDB** predicates and F is a conjunction of **EDB** predicates.
5. Each query has the form $q :\leftarrow G$ where G is in disjunctive normal form on **EDB** and **IDB** predicates.
6. The database includes axioms for built-in predicates as needed.

Resource definitions may contain constants. We assume that the resource rules are expanded, and rectified (see page 5) so that a single resource that has multiple definitions is defined by the same variables in each definition.

We first provide a simple example, that illustrates what has to be done to handle such cases.

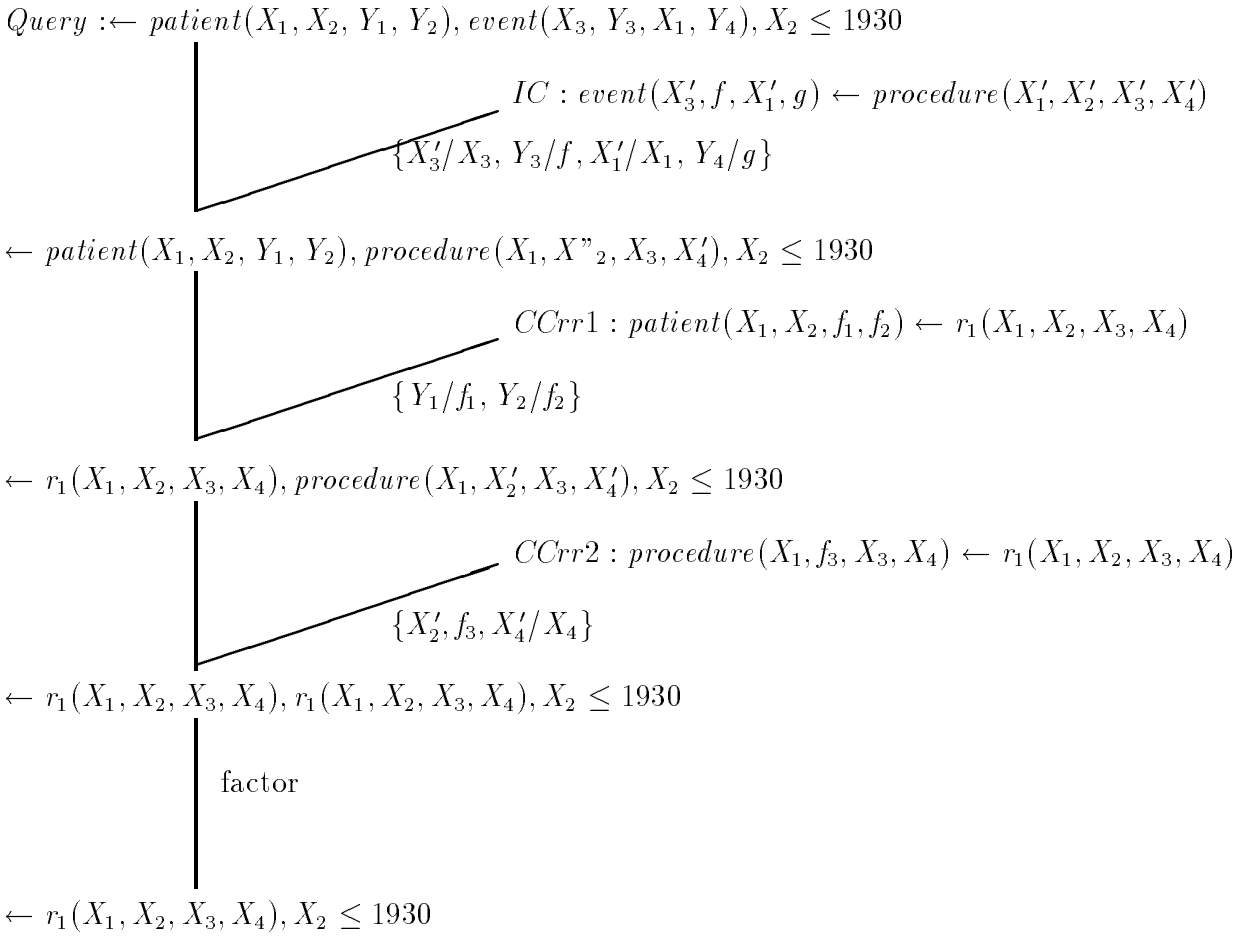

Fig. 7. Inclusion Dependency Example

Example 5. (Multi-Resource Example) **EDB**: $p_1(X)$, $p_2(X)$, $p_3(X)$. **IDB**: $\emptyset$.

$$\textbf{IC} : p_2(X) \vee p_3(X) \leftarrow p_1(X) \cdot \tag{66}$$

The integrity constraint is a non-Horn clause and states that whenever $p_1(a)$ is in the database, for some constant a, then either $p_2(a)$ or $p_3(a)$ or both are in the database. We also have the following resource definitions:

$\textbf{Res}_{\textbf{DB}}$:

$$r(X) \leftarrow p_1(X), p_2(X) \cdot \tag{67}$$

$$r(X) \leftarrow p_3(X) \cdot \tag{68}$$

The Clark completion of the resource rules (ignoring equality) is:

$$r(X) \leftrightarrow (p_1(X) \wedge p_2(X)) \vee p_3(X) \tag{69}$$

After the use of De Morgan's rules we obtain two Clark completion resource rules:

$$CCrr1 : p_1(X) \vee p_3(X) \leftarrow r(X) \cdot, \tag{70}$$

and

$$CCrr2 : p_2(X) \vee p_3(X) \leftarrow r(X) \cdot \tag{71}$$

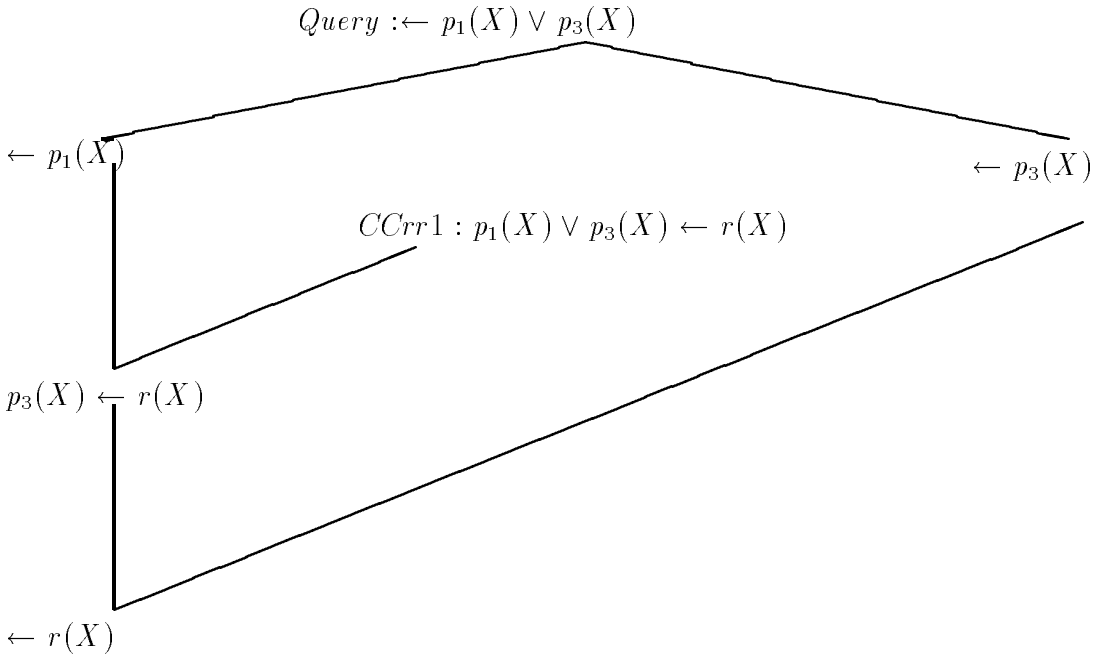

Fig. 8. Deriving Resource Queries

Note that the use of more than one conjunctive definition of a resource leads to non-Horn clauses. Such clauses are outside of **Datalog**. Hence to work with such clauses, we need, in general **Disjunctive Datalog**, that is $Datalog^{\vee}$.

Next, consider the query:

$$q(X) :\leftarrow p_1(X) \vee p_3(X). \tag{72}$$

Now we have to determine if we can derive answers to the query from the **EDB**, the **IDB**, and the **CCrr** (that is, rule 70 and rule 71). Figure 8 shows the steps required to achieve this result. Because the query is a disjunction, $\leftarrow p_1(X) \vee p_3(X)$, the derivation can be split into two clauses: $\leftarrow p_1(X)$ and $\leftarrow p_3(X)$. The derivation terminates with a clause that contains only the resource predicate. We will show that all answers obtained by querying the resource will yield correct answers to the query $q(X)$.

Example 6. (Example 5 Continued) Consider the same example, where the query was:

$$q(X) :\leftarrow p_1(X). \tag{73}$$

Figure 8 applies without the right branch $\leftarrow p_3(X)$. We obtain as bottom clause,
$p_3(X) \leftarrow r(X)$. If written with an empty head, we obtain $\leftarrow r(X), \neg p_3(X)$. This is a query that contains logical negation. We cannot replace this by default negation since default negation does not imply logical negation. We can see the problem by assuming that we have $r(c), p_3(a), p_3(b)$. $p_3(c)$ may either be *true* or *false*, but default negation assumes its falsity. We might have $r(c), \neg p_3(c)$ is not *true*, while $r(c), not\, p_3(c)$ is *true*. Hence we could obtain a non-sound answer $q(c)$ by using default negation. However, if one knew that the p_3 predicate were complete (by the Closed World Assumption) one could write an axiom

$$\neg p_3 \leftarrow not\, p_3 \tag{74}$$

in which case, another resolution step would be applied to obtain

$$\leftarrow r(X), not\, p_3(X) \tag{75}$$

The above query is a partial folding. The negation of the atom $p_3(X)$ does not lead to complications since the resource $r(X)$ makes the formula safe.

The folding algorithm we now present uses set-of-support as its inference method. In this inference, there are two types of clauses: one set, referred to as T, is given support. This set consists of the conjunctions that are part of the query. The second set consists of a satisfiable set of clauses, the remaining clauses in the set $\mathcal{C}$, which we may refer to as set S. A set of support resolution is a resolution of two clauses that are not both from $S - T$ ((CL73)).

Before the algorithm commences we assume that there is a test to determine which of the three cases (see page 10) applies to **ICs**. In Case 1, nothing has to be done. In Case 2, we assume that the set-of-support derivation is modified to include a check to determine if the clause, L, that has been generated, satisfies the bounding condition, and if it does, then backtracking occurs. In Case 3, a depth bound k is specified and if the depth is reached, backtracking occurs. We also assume that if there are built-in predicates such as $=, \neq, \geq$, then the input clauses are placed in expanded form. If there are no built-in predicates, it is unnecessary to do the expansion.

Folding Algorithm 2 (Finding Multiple Foldings)
Input: $\mathcal{C}$: the set of clauses in the **EDB**, **IDB**, **IC**, and **CCrr**, and
the query, $q(\bar{X}) \leftarrow G(\bar{Y})$, where $\bar{X} \subseteq \bar{Y}$, and G is in disjunctive normal form.
Output: A proof tree starting with the query.
begin
Split the query, $\leftarrow G$, into a set of clauses, (each of which has support). Find a proof tree using set-of-support resolution that results in leaf nodes of the form $\leftarrow L$ that contains the query variables and no function symbols.
end

We consider four cases for the clauses that are leaf nodes in the proof tree.

1. The clause has an empty head and a conjunction of resource and built-in predicates in the body.
2. The clause has an empty head and a conjunction of resource, **EDB** and built-in predicates in the body.
3. The clause has no resource predicates.
4. The clause has a non-empty head.

Theorem 2. Consider a clause that is a leaf node of the proof tree constructed by Folding Algorithm 2. In Cases 1 and 2, the query can be answered using the clause and every answer obtained that way is sound. These cases are instances of complete and partial folding respectively. In Case 3, the clause is not a folding. In Case 4, the clause is not a folding, but if the *CWA* can be applied to all the predicates in the head, then the atoms in the head can be moved to the body using default negation and Case 2 or Case 3 becomes applicable.

Proof. For a clause of Case 1 or 2, the proof of Theorem 1 applies. In Case 3, there is no folding. In Case 4, without the *CWA*, the clause is not a query (since it does not have an empty head). The disjunction of atoms in the head of such a clause can be moved to the body and negated (for example, by $\neg p$). Since the *CWA* applies, the axioms $\neg p \leftarrow not\, p$ can be added to the set of clauses $\mathcal{C}$. These axioms may be used to eliminate the logically negated atoms and replaced by default negated atoms. When this is done, this case reduces to Case 2 or Case 3. □

We illustrate the theorem with an example:

Example 7. Consider the **EDB** with the relations $p_1(X_1, Y_1), \ldots, p_7(X_7, Y_7)$ Assume there are no **IDB** predicates and no **ICs**. Let there be the following resource rules:

$r_1(X) \leftarrow p_1(X, Z), p_2(X, Z), Z \neq a$
$r_2(X, Y) \leftarrow p_5(X, Z), p_6(Z, Y)$
$r_2(X, Y) \leftarrow p_7(X, Y)$

The Clark Completion resource rules are:

$$CCrr1 : p_1(X, f) \leftarrow r_1(X) \tag{76}$$

$$CCrr2 : p_2(X, f) \leftarrow r_1(X) \tag{77}$$

$$CCrr3 : f \neq a \leftarrow r_1(X) \tag{78}$$

$$CCrr4 : p_5(X, f) \vee p_7(X, Y) \leftarrow r_2(X, Y) \tag{79}$$

$$CCrr5 : p_6(X, f) \vee p_7(X, Y) \leftarrow r_2(X, Y) \tag{80}$$

Let the query be given by:
$q(X) \leftarrow p_1(X, Z) \wedge ((p_2(X, Z) \wedge Z \neq a) \vee p_3(X, V)) \vee (p_3(X, Z) \wedge p_4(X, Z)) \vee (p_5(X, V) \wedge p_6(V, Y))$
We construct the proof tree in Figure 9 after converting the query to disjunctive normal form. There are four leaf nodes in Figure 9. The leftmost leaf node has only resource predicates, and hence, it can be used to obtain correct answers. The second leftmost leaf node has both a resource predicate and an extensional predicate. Hence, it is a partial folding and if the database and the resource predicate were used, correct answers would be found. The third leftmost leaf provides nothing with respect to resources. The final leaf node has something in the head of the clause and provides no useful information (without the Closed World Assumption on p_7). If the CWA applies to p_7 then we obtain a partial folding:
$\leftarrow r_2(X, Y), not\, p_7(X, Y)$.

In the case of the query $q(X, Y, Z, W)$ given at the end of Section 4.1, no resolution steps are possible, hence Case 3 applies.

Answering the query with resource rules as described in the above theorem does not mean that all answers to the original query have been found. We want to determine when using resource predicates (or partially using resource predicates) will provide all the answers, that is, if the method is complete. This assumes that the database from which the resources were constructed has not been updated.

When we do not have completeness using resource predicates we may still wish to find all answers. This may be done as follows. Let the answers be given by the formula defining $Q_{resource}$, and let the query be Q. Then the remaining answers may be found by using the query: $Q - Q_{resource}$. In some cases this may be simpler than trying to find all answers to Q. For example, if the query Q is given by $Q : (p_4 \wedge p_5) \vee (p_1 \wedge p_2) \vee (p_2 \wedge p_3)$, and $r_1 : p_1 \wedge p_2$ and $r_2 : p_2 \wedge p_3$. then the remaining answers can be expressed by $p_4 \wedge p_5$, and hence it is easier in this case to find the complete set of answers to the query by using this subformula together with the answers found by $Q_{resource}$ than having to answer the original query, Q.

In the following algorithm, where we test for completeness, we have to consider two cases. In the first case there are no built-in predicates. In the second case, we allow built-in predicates. In this case we have a bound on the depth of the proof tree. This algorithm is based on the subsumption algorithm. In the subsumption algorithm two clauses, C and D are given and the algorithm checks to determine if C subsumes D, which means that there is a substitution θ such that $C\theta$ is a subset of D, where the clauses C and D are considered as sets of literals. In our completeness test, D corresponds to the original query and the C is the union of the leaf nodes. What we are trying to show is that the original query is implied by the union of the leaf nodes using the **IDB**, **IC**, and $\mathbf{Res_{DB}}$.

Completeness Test Algorithm

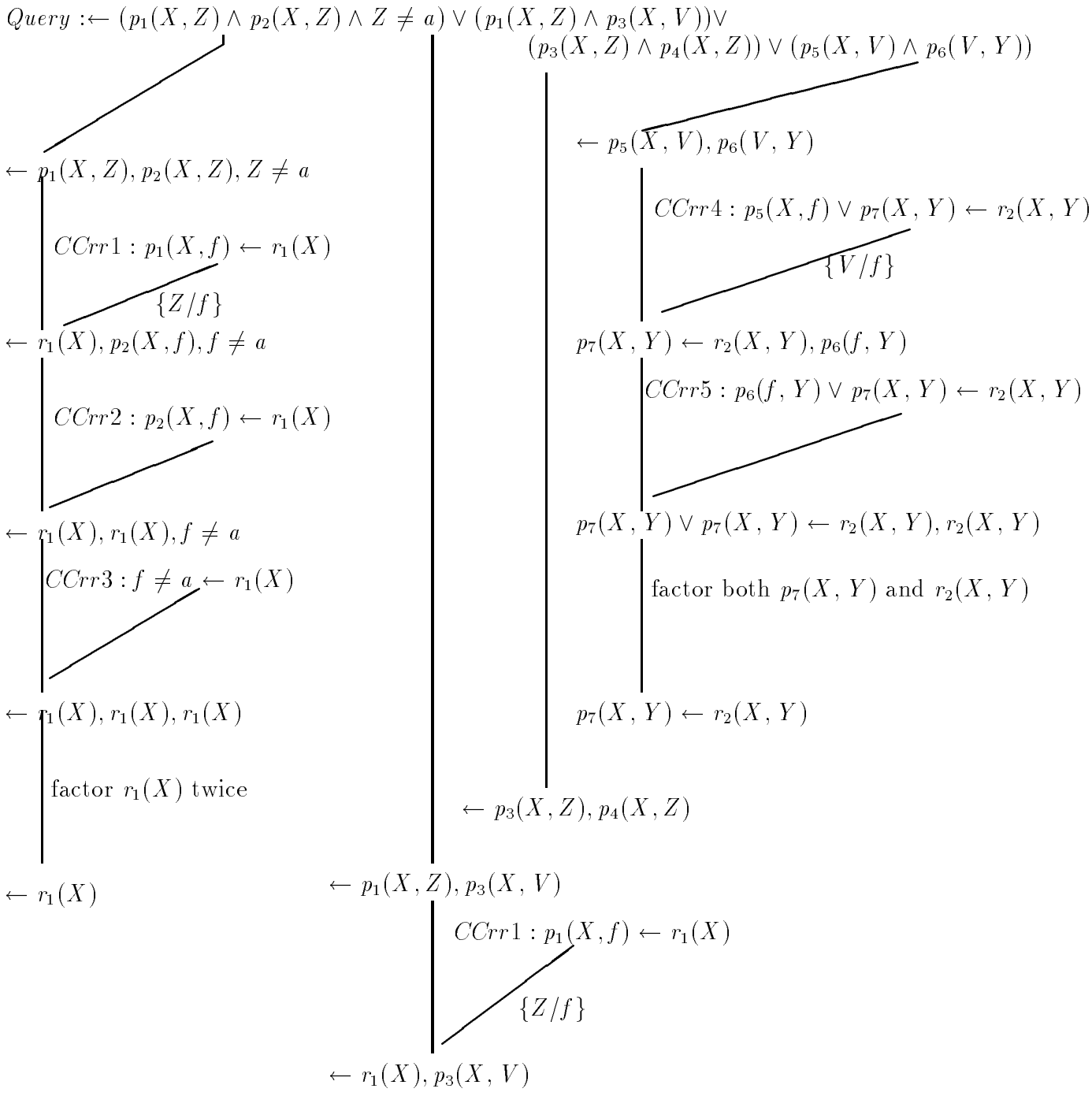

Fig. 9. Four Cases of Theorem 1 Illustrated

Input: The **IDB**, **IC**, $\mathbf{Res_{DB}}$, the non-negated form of the query, where each variable is replaced by a unique new constant, and the leaf nodes of Cases 1 and 2 of Folding Algorithm 2. If there are built-in predicates, the appropriate axioms, such as equality, are also included as well as a bound on the depth of the proof tree.

begin

We give support to the clauses of the leaf nodes obtained in Algorithm 6, and apply set-of support resolution on the input clauses. If we obtain the null clause we terminate. If we allow built-in predicates, when we reach the depth bound, we terminate.

end

In the following theorem we show when completeness is obtained.

Theorem 3. If the Completeness Test Algorithm yields the null clause, then the set of answers obtained by solving the folded queries is complete. If the Completeness Test Algorithm ends without reaching the null clause, then the set of answers obtained by solving the folded queries may not be complete.

Proof. Suppose that the null clause has been found. Let $\bar{a}$ be a solution to $q(\bar{X})$. Writing $q(\bar{X}) :\leftarrow G(\bar{Y})$, we obtain $DB \models G(\bar{b})$, where $\bar{b}[\bar{X}] = \bar{a}$. That is, the constants obtained by answering the query only return that part of the tuple required by the variables in $\bar{X}$. The projection of $\bar{b}$ on the variables in $\bar{X}$ yields $\bar{a}$. Suppose that the leaf nodes used in the completeness test are $\leftarrow L_1(\bar{Z}_1), \ldots, \leftarrow L_n(\bar{Z}_n)$, where $\bar{X} \subset \bar{Z}_i$ for $1 \leq i \leq n$. Thus,

$$\mathbf{IDB} \cup \mathbf{IC} \cup \mathbf{Res_{DB}} \cup \{G(\bar{b})\} \cup \leftarrow L_1(\bar{Z}_1) \cup \ldots \cup \leftarrow L_n(\bar{Z}_n)$$

form a contradiction. Hence,

$$DB \models (\mathbf{IDB} \cup \mathbf{IC} \cup \mathbf{Res_{DB}} \cup \{G(\bar{b})\}) \rightarrow L_1(\bar{Z}_1) \vee \ldots \vee L_n(\bar{Z}_n).$$

Since $DB \models \mathbf{IDB} \cup \mathbf{IC} \cup \mathbf{Res_{DB}} \cup \{G(\bar{b})\}$,
$DB \models L_1(\bar{d}_1) \vee \ldots \vee L_n(\bar{d}_n)$, for some $\bar{d}_i$, $1 \leq i \leq n$, where $\bar{d}_i[\bar{X}] = \bar{a}$. Hence, the solution $\bar{a}$ to $q(\bar{X})$ is also obtained by one of the $L_i, 1 \leq i \leq n$.

Suppose the null clause is not obtained. Proceeding as in the previous case, given a solution $\bar{a}$ to $q(\bar{X})$ we cannot prove that for some $\bar{d}_i$, where $\bar{d}_i[\bar{X}] = \bar{a}$, $DB \models L_1(\bar{d}_1) \vee \ldots \vee L_n(\bar{d}_n)$. Hence, it is possible that the solution $\bar{a}$ to $q(\bar{X})$ may not be obtained by solving the folded query. □

Note that if the null clause is found in Theorem 3, and all of the leaf nodes are of case 1, we have query completeness for a complete folding, otherwise we have query completeness for a partial folding.

We illustrate this theorem by reconsidering the example at the beginning of this section. The derivation is shown in Figure 10. The original query, $\leftarrow p_1(X) \vee p_3(X)$ is changed to its non-negated form with the new constant k substituted for the variable X to become $p_1(k) \vee p_3(k) \leftarrow$. We start with the rewritten query, the leaf node, $\leftarrow r(X)$, and eventually obtain the null clause. This shows that the query using the resource predicate obtains all the answers to the original query. Both factoring and ancestry-resolution are used.

7 Negation

In this section we deal with stratified databases. We allow default negation in the **IDB**, **IC**, $\mathbf{Res_{DB}}$, and the query. Otherwise our assumptions on the database are the same as in Section 5. Problems arise when we have just one resource rule that contains a conjunction of atoms, such as,

$$r(X) \leftarrow p_1(X), p_2(X) \tag{81}$$

If we have the negation of $p_1(X)$ or $p_2(X)$ in the query with another atom, there is no way to resolve either of these atoms with the *CCrr* since the negated atoms do not appear on the left hand side of any inverse rule. To handle this, we represent a rule for the negation of the resource as,

$$not\, r(X) \leftarrow not\, (p_1(X), p_2(X)) \cdot \tag{82}$$

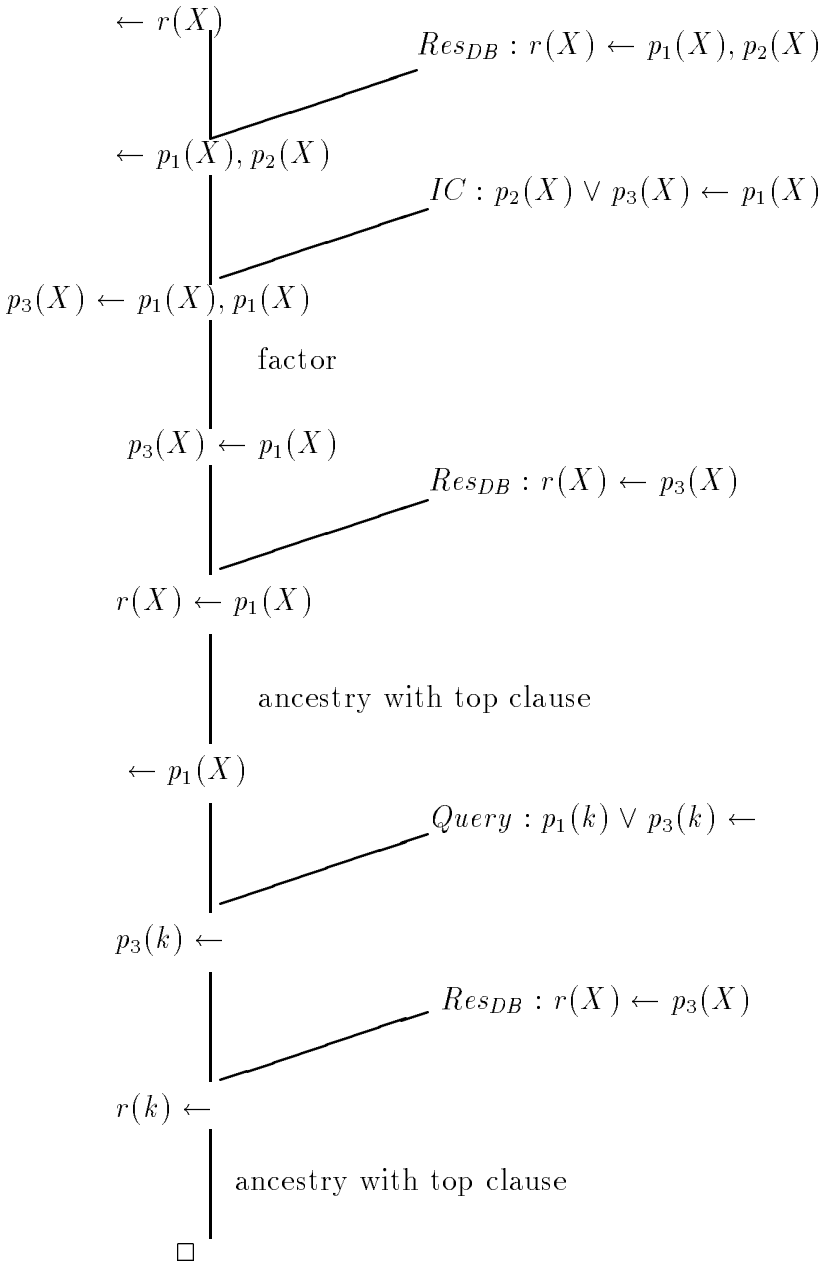

Fig. 10. Use of Ancestry Resolution

This would lead to two clauses

$$not\, r(X) \leftarrow not\, p_1(X), \tag{83}$$

and

$$not\, r(X) \leftarrow not\, p_2(X) \tag{84}$$

If we rename default negated atoms uniformly to new names with primes ('), we obtain:

$$r'(X) \leftarrow p_1'(X), \tag{85}$$

and

$$r'(X) \leftarrow p'_2(X) \tag{86}$$

Now, we have two rules that define $r'(X)$. When we do the Clark completion of this atom, we obtain a disjunctive inverse rule,

$$p'_1(X) \vee p'_2(X) \leftarrow r'(X) \tag{87}$$

We must rename the negated atom in the query in the same manner. Since we have a disjunctive rule, as in Theorem 1, we may be in case (4) and not be able to compute an answer.

We have to first deal with how to handle stratified databases. The underlying idea as developed by (ABW88, VG88) is that given a stratum, and a negation of a predicate in the body of a rule in that stratum, the predicate that is negated must have been defined in an earlier stratum, and hence, the predicate may be calculated. Thus its negation can be obtained. An algorithm to make this explicit may be found in (Ull89) (Algorithm 3.6, Vol. 1). We adapt this algorithm by compiling the **IDB** predicates to rules that involve **EDB** predicates. We give two versions of the compilation process. In the first version, which requires a restriction on the types of formulae allowed, we do a complete compilation, so that the final rules involve **EDB** predicates only. In the second version we do not compile all **IDB** predicates.

First, in order to do a complete compilation, we must restrict the type of formula that we allow for defining **IDB** and $\mathbf{Res_{DB}}$ predicates. Namely, in addition to safety, we require that for all definitions the set of variables in the body be identical to the set of variables in the head. That is, there are no existential quantifiers in the right-hand side of the formula. We call such a formula *extra safe*. By the definition of safe formula, no variable may appear in the head that does not appear in the body. Now we show why there should not be any variable in the body that is not in the head. Consider the case where the **IDB** predicates p and t are defined in terms of the **EDB** predicates h, k, and s as follows:

$$p(X\,,\,Y) \leftarrow h(X\,,Z), k(Z\,,\,Y) \tag{88}$$

$$t(X\,,\,Y) \leftarrow s(X\,,\,Y), not\, p(X\,,\,Y) \tag{89}$$

Note that the definition of p is safe but contains a variable Z in the body that is not in the head, and hence is not extra safe. In our compilation process, to be described below, we replace $not\, p(X\,,\,Y)$ by the negation of the body of the definition of p, so we obtain the following two formulas as the compiled definition of t:

$$t(X\,,\,Y) \leftarrow s(X\,,\,Y), not\, h(X\,,Z) \tag{90}$$

$$t(X\,,\,Y) \leftarrow s(X\,,\,Y), not\, k(Z\,,\,Y) \tag{91}$$

obtaining two unsafe formulas as well as losing the connection between h and k in t, namely, that t is the join of h and k.

Now we describe the compilation algorithm. In a stratified database, each stratum is numbered in an increasing fashion. Since there is no recursion in the **IDB**, we can modify the strata so that each stratum contains definitions for one predicate. For example, if s and t originally had definitions in the same stratum and s is the head of a rule that contains t in the body of the rule, we move the definitions for s to the next higher stratum and adjust other stratum values accordingly. Call the predicate with definitions in stratum i, p_i.

Compilation Algorithm

Input: EDB (all predicates have stratum 0) and **IDB** (predicates with strata $1, \ldots n$).

Output: The compiled **IDB** (each **IDB** predicate defined in terms of **EDB** predicates).

begin For stratum i = 1 To n Do (Note: **EDB** predicates are not compiled)

1. If p_i has multiple definitions, i.e., there are multiple rules with p_i as their head, replace these rules with a single definition by taking the disjunction of the bodies of the multiple definitions.
2. Substitute for each predicate p_j, in the definition of p_i, its compiled form.
3. Simplify the definition of p_i by using De Morgan's rules and put in disjunctive normal form.

end

The following result shows that the Compilation Algorithm does not change the extra safeness of predicates.

Theorem 4. Suppose that every predicate in **IDB** is defined by extra safe rules. Then the compiled definitions are also extra safe rules.

Proof. We proceed by induction on the strata. A predicate at stratum 0 is extensional, and the statement is vacuously true. Let p_i be the IDB predicate at strata i. By the inductive hypothesis, all predicates, $p_j, j < i$, in the body of a rule for p_i have been compiled to extra safe formulas. Replacing the multiple extra safe rules in step 1 by a single rule via a disjunction preserves the extra safe property. Since the compiled definition of each p_j has exactly the same variables as p_j, the substitution of step 2 also preserves the extra safe property. Finally, in step 3 using De Morgan's rules and converting to disjunctive normal form preserves the extra safe property as well. □

We now are able to handle stratified databases with extra safe rules as given by the following theorem.

Theorem 5. Let **DB** be a stratified database where each **IDB** rule is extra safe. Compile the **IDB** predicates using the Compilation Algorithm to $\mathbf{IDB^C}$. Apply $\mathbf{IDB^C}$ to the $\mathbf{Res_{DB}}$ rules to rewrite them in terms of compiled **IDB** predicates as $\mathbf{Res^C_{DB}}$. Rename every negated atom $(not\, p)$ in the query and in $\mathbf{Res^C_{DB}}$ to a new predicate (p') in a consistent manner and add the integrity constraints $\leftarrow p, p'$. Then, the results of Theorem 6.1 and Theorem 3, that deal with the leaves of the proof tree, and the completeness test algorithm apply.

Proof. By Theorem 4 the Compilation Algorithm preserves extra safety for the predicates of $\mathbf{IDB^C}$. Hence, the variables in $\mathbf{Res_{DB}}$ and $\mathbf{Res^C_{DB}}$ are the same. By renaming $not\, p$ to p' we eliminate negation and the assumptions of Section 6 (see page 26) are satisfied. The result follows because now, Theorem 6.1, and Theorem 3 apply, and the additional integrity constraints assure that it is not possible to have both p and $not\, p$ at the same time. □

This theorem allows us to obtain proof trees and test for completeness. In actually computing a folded query, we need to change back the primed atoms p' to $not\, p$. The following example illustrates the theorem. (We omit the transformation of $not\, p$ to p'.)

Example 8. Let the **EDB** consist of: $e_{01}(X, Y)$, $e_{02}(X, Y, Z)$, $e_{03}(X, Y)$, and $e_{04}(X, Y)$. Let the **IDB** consist of the following clauses:

$$e_{11}(X, Y, Z) \leftarrow e_{01}(X, Y), e_{02}(X, Y, Z) \tag{92}$$

$$e_{11}(X, Y, Z) \leftarrow e_{02}(X, Y, Z), not\, e_{03}(X, Y) \tag{93}$$

$$e_{12}(X, Y) \leftarrow e_{04}(X, Y), not\, e_{01}(X, Y) \tag{94}$$

$$e_{21}(X, Y) \leftarrow e_{03}(X, Y), not\, e_{12}(X, Y) \tag{95}$$

Now, let $\mathbf{Res_{DB}}$ consist of:

$$r(X, Y, Z) \leftarrow e_{11}(X, Y, Z), not\, e_{12}(X, Y) \tag{96}$$

The above database is not compiled. Using the algorithm given above, the following database is found, where all rules are written in terms of **EDB** predicates. The compiled definitions, $\mathbf{IDB^c}$ become:

$$e_{11}(X\,,\,Y\,,Z) \leftarrow e_{02}(X\,,\,Y\,,\,Z), (e_{01}(X\,,\,Y) \vee not\, e_{03}(X\,,\,Y)) \tag{97}$$

$$e_{21}(X\,,\,Y) \leftarrow e_{03}(X\,,\,Y), (not\, e_{04}(X\,,\,Y) \vee e_{01}(X\,,\,Y)) \tag{98}$$

The compiled resource, $\mathbf{Res^C_{DB}}$ becomes:

$$r(X\,,\,Y\,,\,Z) \leftarrow e_{02}(X\,,\,Y\,,\,Z), (e_{01}(X\,,\,Y) \vee not\, e_{03}(X\,,\,Y)), (e_{01}(X\,,\,Y), \vee not\, e_{04}(X\,,\,Y)) \tag{99}$$

The Clark Completion resource rules become:

$$CCrr1 : e_{02}(X\,,\,Y\,,\,Z) \leftarrow r(X\,,\,Y\,,\,Z) \tag{100}$$

$$CCrr2 : e_{01}(X\,,\,Y) \vee not\, e_{03}(X\,,\,Y) \leftarrow r(X\,,\,Y\,,\,Z) \tag{101}$$

$$CCrr3 : e_{01}(X\,,\,Y) \vee not\, e_{04}(X\,,\,Y\,,\,Z) \leftarrow r(X\,,\,Y\,,\,Z) \tag{102}$$

Let the negation of the query be: $Q :\leftarrow e_{21}(X\,,\,Y)$

In Figure 11, we show that using the query and the Clark Completion resource rules yields a partial folding on the leftmost branch:

$$\leftarrow r(X\,,\,Y\,,\,Z), e_{03}(X\,,\,Y) \tag{103}$$

We now consider the second case where the rules are safe but not necessarily extra safe. In this case modify the Compilation Algorithm in step 2 so that predicates with a definition that is not extra safe are not compiled. In this case the end result of the compilation may contain **IDB** predicates in some definitions. We use the same notation for the compiled versions, i.e. $\mathbf{IDB^C}$ and $\mathbf{Res^C_{DB}}$ as before. Theorem 5 extends to this case as follows.

Theorem 6. Let **DB** be a stratified database. Compile the **IDB** predicates using the Compilation Algorithm modified as explained above to $\mathbf{IDB^C}$. Proceed as in the statement of Theorem 5, compiling $\mathbf{Res_{DB}}$ to $\mathbf{Res^C_{DB}}$ and renaming the negated predicates. Then, the results of Theorem 5.1 and Theorem 3 apply.

Proof. Similar to the proof of Theorem 5 except that the modified Compilation Algorithm is used and so the compilation process stops earlier for certain predicates. □

The following example illustrates the Theorem. (Again we omit the transformation of $not\, p$ to p'.)

Example 9. Let the **EDB** consist of: $h(X\,,\,Y)$, $k(X\,,\,Y)$, $s(X\,,\,Y)$, and $\ell(X\,,\,Y\,,\,Z)$. Let the **IDB** consist of the following clauses:

$$p(X\,,\,Y) \leftarrow h(X\,,\,Z), k(Z\,,\,Y) \tag{104}$$

$$t(X\,,\,Y) \leftarrow s(X\,,\,Y), not\, p(X\,,\,Y) \tag{105}$$

Now, let $\mathbf{Res_{DB}}$ consist of:

$$r(X\,,\,Y\,,\,Z) \leftarrow t(X\,,\,Y), \ell(\,Y\,,\,Z\,,\,U\,) \tag{106}$$

Note that the definition for p is not extra safe. So in the definition of $t(X\,,\,Y)$, $not\, p(X\,,\,Y)$ is not changed. Thus, $\mathbf{IDB^C = IDB}$ and the compiled resource, $\mathbf{Res^c_{DB}}$ becomes:

$$r(X\,,\,Y\,,\,Z) \leftarrow s(X\,,\,Y), not\, p(X\,,\,Y), \ell(\,Y\,,\,Z\,,\,U\,) \tag{107}$$

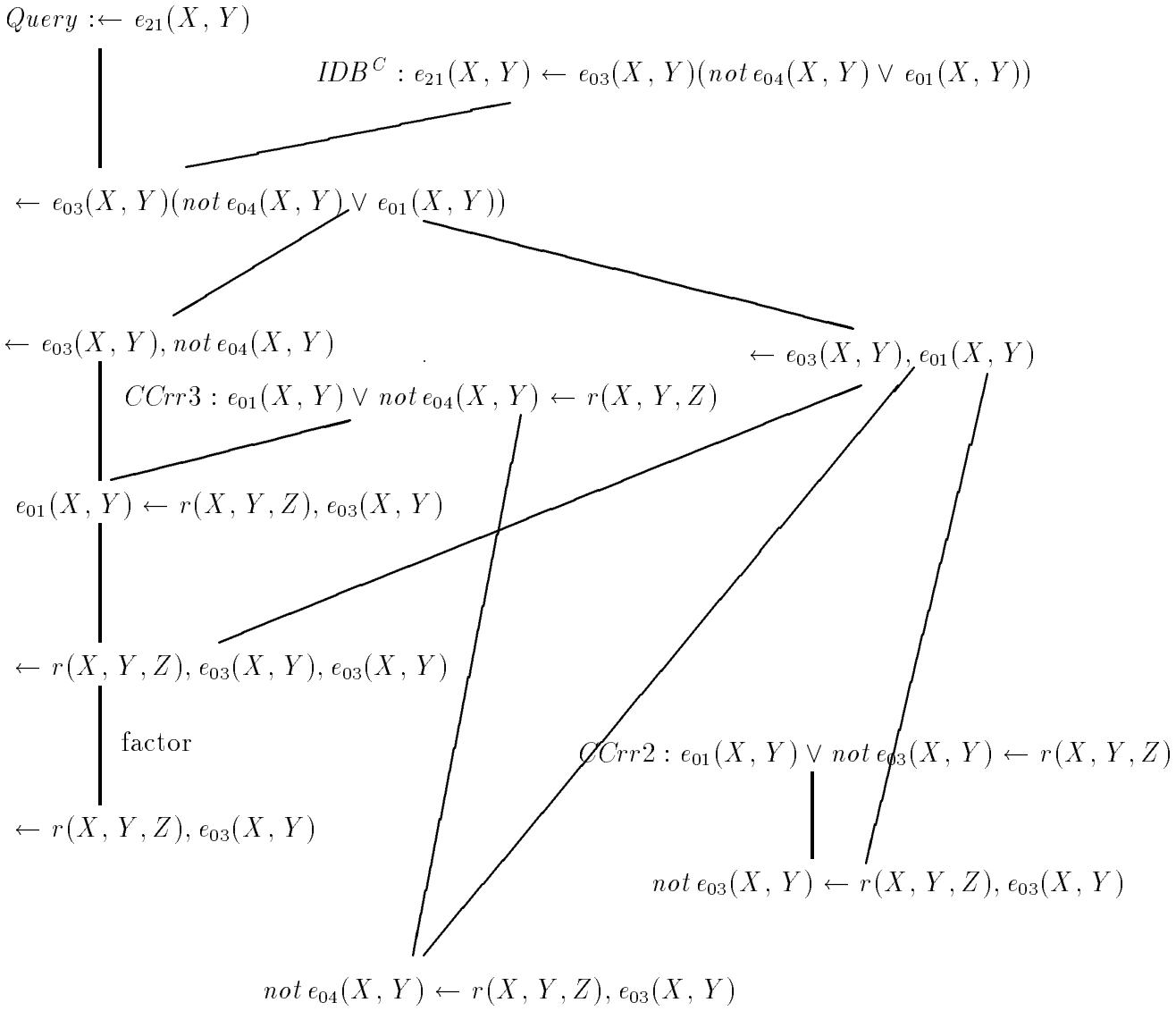

Fig. 11. Stratification Example

The Clark Completion resource rules become:

$$CCrr1 : s(X, Y) \leftarrow r(X, Y, Z) \tag{108}$$

$$CCrr2 : not\, p(X, Y) \leftarrow r(X, Y, Z) \tag{109}$$

$$CCrr3 : \ell(Y, Z, f) \leftarrow r(X, Y, Z) \tag{110}$$

Let the negation of the query be: $Q :\leftarrow s(X, Y), k(Y, Z)$

In Figure 12 we show that using the query and the Clark Completion resource rules yields a partial folding:

$$\leftarrow r(X, Y, Z), k(Y, Z) \tag{111}$$

We end this section by showing that any folded query resulting from Theorems 6.2 or 6.3 is safe. We start by proving a general result.

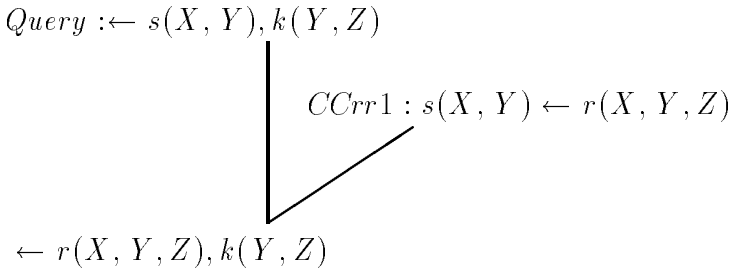

Fig. 12. Safe Negation Example

Proposition 2. The resolution of two safe formulas (in Datalog with negation and disjunction) is a safe formula.

Proof. Let the two safe formulas have the form:
$A_1, \ldots, A_k \leftarrow A_{k+1}, \ldots, A_m$ and
$B_1, \ldots, B_\ell \leftarrow B_{\ell+1}, \ldots, B_n$,
where each A_i, B_j, $1 \leq i \leq m$, $1 \leq j \leq n$ is an atom. We may assume without loss of generality that A_1 is resolved with B_n to yield
$A_2, \ldots, A_k, B_1, \ldots, B_\ell \leftarrow A_{k+1}, \ldots, A_m, B_{\ell+1}, \ldots, B_{n-1}$.
Actually, the resolution involves a substitution θ, so here each A_i, B_j, $2 \leq i \leq m$, $1 \leq j \leq n-1$ is really $A_i\theta$, $B_j\theta$, but this can be ignored for the purpose of this proof because any changed variable in the head of a clause must be changed the same way in the body.

We need to show that every variable in the resulting formula is limited. If X is a variable in any A_i, $2 \leq i \leq m$, it must already be limited in $A_{k+1}, \ldots, A_m$ because the A formula is safe. If X is a variable in any B_j, $1 \leq j \leq n-1$, that did not appear in B_n, it must already be limited in $B_{\ell+1}, \ldots, B_{n-1}$ because the B formula is safe. Finally, if X is a variable in some B_j, $1 \leq j \leq n-1$, that was limited in the B formula in B_n, by the resolution X must now be limited in $A_{k+1}, \ldots, A_m$. □

Corollary 1. Every folded query resulting from Theorem 6.3 is safe.

Proof. The query is safe and so are the rules in **IDB** and **IC**. Consider the way the Clark Completion resource rules are obtained. In each such rule the body contains only the resource predicate and every additional variable in the body of an original resource rule becomes a function symbol. Since these function symbols cannot be iterated, they can be treated as constants. Hence the Clark Completion resource rules are also safe and the result follows from the Proposition. □

8 Recursion

Query folding becomes problematic in the presence of recursion, since one does not know when to terminate recursion in a top-down approach. However, if it is known that the recursion is bounded (MN82, NS87), that is, the recursion is known to terminate after a number of stages using only intensional rules, then one can use the methods we describe in the previous sections to handle this case. In this section we assume that recursion is not bounded.

Unlike the previous sections, the computations we present here follow the bottom-up approach rather than the top-down method we used earlier. The reason is that in the presence of recursion it is difficult to tell when all the solutions have been obtained in the infinite proof tree. In this case, we are not doing query folding, but we are doing query answering. That is, we do not find a rewriting of the query in terms of the resources. Actually, there are strong connections between the top-down and bottom-up approaches: Bry

(Bry90) describes a combined top-down, bottom-up interpreter that incorporates the magic set technique used for recursion.

We start with the case where the recursion occurs only in the query, so the **IDB**, **IC**, and $\mathbf{Res_{DB}}$ contain no recursion. This is equivalent to the case where recursion appears in the **IDB**, a query is asked in terms of the **IDB** and the $\mathbf{Res_{DB}}$ definitions include only **EDB** predicates (and **IDB** predicates defined in such a way that they can be compiled to **EDB** predicates without recursion).

Within the case where recursion only occurs in the query we start with the subcase where everything is positive, that is, there is no negation. The database restrictions are as in Section 4.1 except that the query is recursive. This case was solved in (DGL00). We start by considering their Example 3.1.

Example 10. **EDB**: $edge(X, Y)$ **IDB**: $\emptyset$
IC: $\emptyset$
$\mathbf{Res_{DB}}$: $r(X, Y) \leftarrow edge(X, Z), edge(Z, Y)$
Query: $q(X, Y) \leftarrow edge(X, Y)$
$q(X, Y) \leftarrow edge(X, Z), q(Z, Y)$

In this example the recursive query determines the transitive closure of the relation $edge$, while the resource predicate stores endpoints of paths of length 2. The Clark Completion resource rules here are

$CCrr1 : edge(X, f(X, Y)) \leftarrow r(X, Y).$

$CCrr2 : edge(f(X, Y), Y) \leftarrow r(X, Y).$

In this type of situation the solution is a two-step process. In the first step the extensional predicates, $edge$ in this case, are evaluated from the resource predicates using the Clark Completion resource rules, and in the second step a bottom-up Datalog evaluation is done for the recursive query from the extensional predicates. It is shown there why this process always terminates in a finite amount of time. Specifically, they note that the key observation is that function symbols are only introduced in inverse rules. Because inverse rules are not recursive, no terms with nested function symbols can be generated. As we mentioned earlier the problem with the top-down approach is that it is difficult to find out when to stop processing. We draw one branch of the top-down tree in Figure 13.

The second subcase is where the recursive query is stratified, so negation is allowed. A similar two-step process works here also. The first step is the same as before. However, in the second step the query is computed by computing the stratified database (Ull89).

The final subcase we consider is where the resources are defined by multiple definitions (but still without recursion) as in Section 6. Recall that now the Clark Completion resource rules will contain disjunctions in the head. We illustrate the idea by reconsidering the simple example from the beginning of Section 6 where one Clark completion resource rule (written as a single rule) is:

$$(p_1(X) \wedge p_2(X)) \vee p_3(X) \leftarrow r(X) \tag{112}$$

Suppose we have $r(a)$. Then we separately do three subcomputations:

- generate $p_1(a)$ and $p_2(a)$,
- generate $p_3(a)$,
- generate $p_1(a)$, $p_2(a)$, and $p_3(a)$.

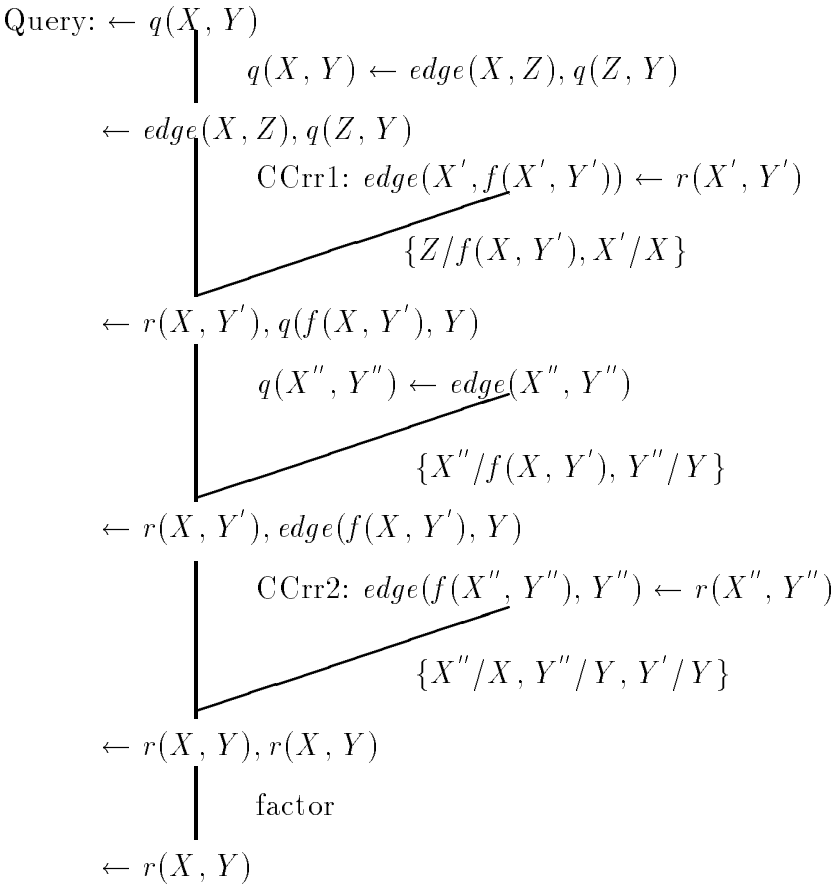

Fig. 13. One branch of the recursive proof tree

In Step 1 apply this process to all the Clark Completion resource rules to obtain all the subcomputations. For every subcomputation apply Step 2, the bottom-up Datalog evaluation of the query. Place a tuple in the answer only if it is an answer in every subcomputation. This is essentially the minimal models approach for disjunctive logic programming (LMR92).

We end our exploration of recursion by considering the case where recursion is allowed in the $\mathbf{Res_{DB}}$. In general, there is no known method to handle this. In the following, we discuss a specific form of recursion for which we obtain sound answers. The problem is how to get the Clark Completion resource rules in a usable form to give us definitions of extensional predicates in terms of resource predicates. Going back to the previous *edge* example, suppose that instead of the resource predicate storing endpoints of paths of length 2, the resource predicate stores the endpoints of all paths. This would be written as

$\mathbf{Res_{DB}}$: $r(X, Y) \leftarrow edge(X, Y)$

$r(X, Y) \leftarrow edge(X, Z), r(Z, Y)$

Now we write a modified Clark Completion resource rule as:

$$MCCrr : edge(X, Y) \leftarrow r(X, Y), not \exists Z(r(X, Z), r(Z, Y)) \tag{113}$$

obtaining all those paths in r that cannot be broken up into two paths. The two steps of the computation process are as before: evaluate *edge* first from r (using the modified Clark Completion resource rule) and then evaluate q from *edge* whether or not q is recursive. The general result is as follows:

Proposition 3. Suppose that a $\mathbf{Res_{DB}}$ predicate r is recursive and has the form

$r(\bar{X}) \leftarrow e(\bar{X})$

$r(\bar{X}) \leftarrow t(\bar{Y})$
where t is a conjunction that may include e and r and $\bar{X} \subseteq \bar{Y}$
Then the modified Clark Completion resource rule is:
$MCCrr : e(\bar{X}) \leftarrow r(\bar{X}), not \exists \bar{Z}\, t(\bar{Y})$
where $\bar{Z} = \bar{Y} - \bar{X}$.
$MCCrr$ can be used to evaluate e from r in a sound manner.

Proof. We proceed by contraposition. Suppose that for some tuple $\bar{a}$ $e(\bar{a})$ is false and $r(\bar{a})$ is true. This means that $r(\bar{a})$ must have been obtained by the second, (recursive) definition. But then $\exists \bar{Z}\, t(\bar{Y})$ must be true. □

9 Comparison

In this section we discuss the contributions made in this paper and compare the work with other efforts.

Qian ((Qia96)) was the first to consider the problem of folding. In her paper she introduced the concept of inverse rules to permit one to use resources to compute answers to queries. Inverse rules basically state that the only way in which the information in the resource can be computed is through the resource. As noted in this paper, the concept of inverse rules was introduced first in the context of logic programming by Clark (Qia96), and is the basis of the *closed world assumption*. It represents an if-and-only-if condition for rules. Qian showed that if, for each resource predicate, defined by a conjunctive rule, there is at most one such rule, that it is possible to compute the answer to queries using resources in many instances. We have extended Qian's result slightly to include databases that contain arbitrary integrity constraints. Duschka and his associates (Dus97a, DG97a, Dus97b) show how to extend the work to classes of integrity constraints. (DG97a, Dus97b) were the first to extend the work to handle general recursive queries.

Dawson, Gryz and Qian (DGQ96) show how to compute answers in query folding when there are functional dependencies. (DL97, Dus97b) introduced the new class of *recursive* query plans for information gathering. Instead of plans being only sets of conjunctive queries, they can now be recursive sets of function-free Horn clauses. Using recursive plans, they settle two open problems. First, they describe an algorithm for finding the maximally contained rewriting in the presence of functional dependencies. Second, they describe an algorithm for finding the maximally-contained rewriting in the presence of binding-pattern restrictions, which was not possible without recursive plans. We have shown how integrity constraints containing equality, such as functional dependencies, generate a possibly infinite set of folded queries.

Duschka and Genesereth (DG98, Dus97b) developed the first algorithm to solve the problem of answering queries using views when view definitions are allowed to contain disjunction. They use the Clark completion to obtain the inverse rules. Disjunctive definitions for rules implies that *Datalog* is not sufficiently powerful to handle such situations and it is necessary to use $Datalog^{\vee}$. Their focus is on maximal query containment. They show a duality in between a query plan being maximally contained in a query and this plan computing exactly the certain answers. They show that the disjunctive plan can be evaluated in co-NP time. Afrati et al. (AGK98) also treats the problem of disjunctive materialized views. The relationship with our approach is that we show how, using theorem proving concepts we can handle such theories, including integrity constraints. We also show how one can determine if the query plan that has been developed is complete, that is, if all answers to the original query have been obtained. We do this in the context of theorem proving.

We know of no work that covers negation in the folding problem. We show how to handle stratified negation in views which may be defined by disjunctive rules, and may contain integrity constraints. Duschka (Dus97b) discusses complexity results with respect to negation in his thesis.

With respect to recursion, as noted above, (DG97a) handle recursion. We show how to handle recursion

in a similar manner to their work, and extend the results to stratified databases. In general, there is no known method to handle recursion in resource rules. We suggest one limited case where recursion appears in resource rules and sound answers may be found.

10 Summary

We have shown that a logic-based approach using resolution unifies techniques used earlier for the aspect of data integration also known as query folding. We considered a deductive database where a query written on the database needs to be rewritten in terms of given resources. We showed how to handle integrity constraints and the case where a resource has multiple definitions. We also showed when the folded query yields all or only some of the answers of the original query. We extended our results to some cases involving negation and recursion.

Acknowledgements

We would like to express our appreciation to Parke Godfrey and Jarek Gryz for many discussions we have had with them and for their suggestions. We also wish to thank Alon Levy who made several valuable suggestions which helped to improve the paper. We are also grateful to the referees for their many helpful comments and suggestions.

References

K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Pub., Los Altos, California, 1988.

S. Abiteboul and O.M. Duschka. Complexity of answering queries using materialized views. In *Proceedings Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 98)*, pages 254–263, Seattle, Washington, June 1-3 1998. Association for Computing Machinery.

F. N. Afrati, M. Gergatsoulis, and T. Kavalieros. Answering queries using materialized views with disjunctions. In C. Beeri and P. Buneman, editors, *ICDT'99, LNCS 1540*, pages 435–452. Springer-Verlag, 1998.

F.N. Afrati, C. Li, and J.D. Ullman. Generating efficient plans for queries using views. In *Proc. of the ACM SIGMOD 2001*, pages 319–330, Santa Barbara, California, May 21-24 2001. ACM.

A.V. Aho, Y. Sagiv, and J. Ullman. Efficient optimization of a class of relational expressions. *TODS*, 4(3):434–454, 1979.

A.V. Aho, Y. Sagiv, and J. Ullman. Equivalence of relational expressions. *SIAM Journal of COmputing*, 8(2):218–246, 1979.

George S. Boolos and Richard C. Jeffrey. *Computability and Logic*. Open University Set Book. Cambridge University Press, third edition, 1989.

F. Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data & Knowledge Engineering*, pages 289–312, 1990.

U. S. Chakravarthy, J. Grant, and J. Minker. Foundations of semantic query optimization for deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 243–273. Morgan Kaufmann Pub., Washington, D.C., 1988.

U. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems*, 15(2):162–207, June 1990.

U. S. Chakravarthy. *Semantic Query Optimization in Deductive Databases*. PhD thesis, University of Maryland, College Park, Maryland 20742, 1985.

S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th ICDE*, pages 190–200, 1995.

C. L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. Ninth ACM Symposium on the Theory of Computing*, pages 77–90, 1977.

C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. pages 56–70. Springer Verlag, 1997.

Subrata Kumar Das. *Deductive Databases and Logic Programming*. Addison-Wesley, Wokingham, England, 1992.

O.M. Duschka and M.R. Genesereth. Answering recursive queries using views. In *Proceedings Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 97)*, pages 109–116, Tucson, Arizona, May 12-14 1997. Association for Computing Machinery.

O.M. Duschka and M.R. Genesereth. Query planning in Infomaster. In *Proceedings of the 1997 Symposium on Applied Computing*, pages 109–111, San Jose, California, 1997. Association for Computing Machinery.

O.M. Duschka and M.R. Genesereth. Query planning with disjunctive sources. In *Proc. of the AAAI-98 Workshop on AI and Information Integration*, Menlo Park, California, 1998. AAAI, AAAI Press.

O.M. Duschka, M. Genesereth, and A.Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, 43(1):49–73, April 2000.

S. Dawson, J. Gryz, and X. Qian. Query folding with functional dependencies. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1996.

O. Duschka and A. Levy. Recursive plans for information gathering. In *Proc. of 15th IJCAI*, pages 778–784, Nagoya, Japan, Aug 1997.

A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *Proc. VLDB'99*, pages 459–470, 1999.

O.M. Duschka. Query optimization using local completeness. In L.M. Pereira and E. Orlowska, editors, *Proc. of the 14th National Conf. on AI and 9th Innovative Applications of AI Conf.*, Menlo Park, California, 1997. American Association for Artificial Intelligence Press.

O.M. Duschka. *Query Planning and Optimization in Information Integration*. PhD thesis, Stanford University, Stanford, California, December 1997.

T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

F. Fages. A new fixpoint semantics for general logic programs compared with the well-founded and the stable model semantics. *New Generation Computing*, 9:425–443, 1991.

S. Flesca and S. Greco. Rewriting queries using views. *IEEE TKDE*, 2001. To appear.

M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5^{th} International Conference and Symposium on Logic Programming*, pages 1070–1080, Seattle, Washington, August 15-19 1988.

H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS approach to mediation: data models and languages. In *Second Workshop on Next-Generation Information Technologies and Systems*, Neharia, Israel, June 1995.

A. Van Gelder, K.A. Ross, and J. Schlipf. Unfounded sets and well-founded semantics for general logic programs. *J. of the ACM*, 38(3), 1991.

J. Gryz. Query folding with inclusion dependencies. In *Proc. of 14th ICDE*, Orlando, Florida, 1998.

D. S. Johnson and A. Klug. Optimizing conjunctive queries that contain untyped variables. *SIAM Journal of Computing*, 12(4):616–640, 1983.

Alon Y. Levy. Logic-based techniques in data integration. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 575–595. Kluwer Academic Publishers, Dordrecht, 2000.

A. Y. Levy. Answering queries using views: A survey. *VLDB Journal*, 2001. To appear.

John W. Lloyd. *Foundations of Logic Programming*. Symbolic Computation—Artificial Intelligence. Springer-Verlag, Berlin, second edition, 1987.

Jorge Lobo, Jack Minker, and Arcot Rajasekar. *Foundations of Disjunctive Logic Programming*. M.I.T. Press, Cambridge, Massachusetts, 1992.

A.Y. Levy, A.O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS-95*, pages 95–104, May 1995.

A. Y. Levy, A. Rajaraman, and J. J. Ordille. Query-answering algorithms for information agents. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI)*, pages 40–47, 1996.

A.Y. Levy, A. Rajaraman, and J.J. Ordille. Querying heterogeneous information sources using source-descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases*, pages 251–262, Bombay, India, 1996.

P.-Å. Larson and H.Z. Yang. Computing queries from derived relations. In *Proc. VLDB'85*, pages 259–269, 1985.

J. Minker. Logic and databases:a 20 year retrospective. In D. Pedreschi and C. Zaniolo, editors, *Logic in Databases*, pages 3–57. Springer, July 1996. Proc. Int. Workshop LID'96, San Miniato, Italy.

J. Minker and J-M. Nicolas. On recursive axioms in deductive databases. *Information Systems*, 7(4):1–15, 1982.

J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Mateo, California, 1993.

J.F. Naughton and Y. Sagiv. A decidable class of bounded recursions. *Proc. of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 227–236, March 1987.

L. Popa, A. Deutsch, A. Sahuguet, and V. Tannen. A chase too far? In *Proc. ACM SIGMOD 2000*, pages 273–284, 2000.

R. Pottinger and A. Levy. A scalable algorithm for answering queries using views. In *Proc. of the 26th VLDB Conference*, pages 484–495, Cairo, Egypt, 2000.

X. Qian. Query folding. In *Proceedings of the* 12^{th} *International Conference on Data Engineering*, pages 48–55, 1996.

R. Reiter. Deductive question-answering on relational data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 149–177. Plenum Press, New York, 1978.

R. Reiter. On Closed World Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

J.A. Robinson. A machine-oriented logic based on the resolution principle. *J.ACM*, 12(1), January 1965.

A. Rajaraman, Y. Sagiv, and J.D. Ullman. Answering queries using templates with binding patterns. In *Proceedings Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 95)*, pages 105–112. Association for Computing Machinery, 1995.

O.G. Tsatalos, M.H. Solomon, and Y.E. Ioannidis. The GMAP: a versatile tool for physical data independence. In *Proc. VLDB'94*, pages 367–378, 1994.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I & II.* Principles of Computer Science Series. Computer Science Press, Incorporated, Rockville, Maryland, 1988/1989.

J.D. Ullman. Information integration using logical views. *Proceedings of ICDT*, pages 19–40, January 1997.

A. Van Gelder. Negation as failure using tight derivations for general logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 149–176. Morgan Kaufmann, 1988.

H. Z. Yang and P.-Å. Larson. Query transformation for PSJ-queries. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pages 245–254, 1987.