ENFORCEMENT OF INTEGRITY CONSTRAINTS IN RECURSIVE DATABASES

by

Lifang Zhu

B.CS Huazhong University of Science and Technology, Wuhan, China, 1983
MS.CS Peking University, Beijing, China, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the School
of
Computing Science

SIMON FRASER UNIVERSITY
December 1992

APPROVAL

Name: Lifang Zhu

Degree: Master of Applied Science

Title of thesis: **Enforcement of Integrity Constraints in Recursive Databases**

Examining Committee: Dr. Warren Burton, Chairman

Dr. Nick Cercone, Senior Supervisor

Dr. Jiawei Han, Senior Supervisor

Dr. Fred Popowich, Examiner

Date Approved: December 16, 1992

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

The Enforcement of Integrity Constraints in Recursive Databases

Author: _____
(signature)

Lifang Zhu
(name)

Dec. 16, 1992
(date)

ABSTRACT

Integrity constraint(ic) enforcement forms an essential component in deductive database processing. Some interesting methods which enforce integrity constraints have been proposed by Topor, Lloyd, Decker, Kowalski, Sadri, Soper, Martens, Bruynooghe, Yum and Henschen. In this thesis we further analyze and develop efficient simplification algorithms and methods for the enforcement of integrity constraints in recursive deductive databases. We combine theorem-proving methods with compilation techniques in our approach. Theorem-proving methods are used to prune the size of the integrity constraint checking space and compilation techniques are also used to derive necessary implicit modifications and evaluate the simplified integrity constraint set against the actual database. Synchronous and asynchronous chain recursions are discussed. By exploiting the hierarchical structure of a deductive database, we can precompile or partially precompile integrity constraints and ic-relevant rules to simplify integrity constraint checking and validate some modifications by static qualitative analysis. By analyzing predicate connection and variable binding, and compiling recursive rules independently, we can simplify ic-relevant queries and generate efficient checking plans. Some asynchronous and synchronous chain recursive integrity checking relevant queries can be simplified to non-recursive or simpler queries. Efficient processing algorithms are developed for integrity checking and derivation of implicit modification. To perform integrity checking against the actual database we utilize the 'affected graph' of a modification. We achieve by focusing our attention only on the part of the database which is affected by the update and relevant to integrity constraints.

To my parents.

ACKNOWLEDGMENTS

My greatest thanks to my Senior Supervisor Dr. Nick Cercone for his supporting and helping me throughout my work on this research. Without his encouragement this work may never be finished.

Thanks to my Supervisor Dr. Jiawei Han for his informative instruction at the early stage of this research.

Also thanks to Dr. Fred Popowich for examining the thesis and helping with the revision of the thesis.

CONTENTS

CHAPTER 1

Introduction

Integrity constraint[IC] enforcement is an essential part of deductive database processing. An integrity constraint specifies a property that the database must satisfy to remain in a valid state. Any modification to the database may affect the consistency defined by the constraints, so that integrity constraints must be enforced after modifications to ensure the consistency of the database.

Integrity checking has been an interesting database research topic since the 1970's. The enforcement of integrity constraints is time-consuming, which largely affects the overall performance of database processing. Extensive research had been done on integrity constraint enforcement in relational databases in the 1970's and at the beginning of the 1980's. Thereafter, some researchers extended their interests to deductive databases. Some interesting methods for the enforcement of ICs in deductive databases have been proposed by Topor and Lloyd [14], Decker [4], Kowalski, Sadri

and Soper [6], Martens and Bruynooghe[15], in non-recursive databases. Yum and Henschen [21, 22] proposed a method for maintaining state-constraints in deductive databases with transitive and linear recursions. In this thesis I explore this problem extensively and develop efficient methods for the enforcement of integrity constraints in non-recursive and recursive deductive databases. From this point on, *database* is used to refer to deductive database, for simplicity.

Recently, query independent recursion compilation techniques have provided a very good recursive query evaluation method for some practical recursions[3, 7, 8, 10, 11]. On one hand, the theorem-proving technique provides a very good method for static analysis of integrity constraints and the intensional database[IDB] structure. On the other hand, the compilation technique provides an efficient processing strategy for the evaluation of integrity constraints and the derivation of implicit modification. We combine these two techniques in our approach to simplify integrity constraints, derive implicit modification and check integrity constraints. We partially or fully compile and simplify integrity constraints and integrity constraint relevant (ic-relevant) rules. By static qualitative analysis of the relationships among compiled integrity constraints and rules we can validate most of the updates. The static analysis result can direct the necessary integrity checking procedure against the actual database. Query independent compilation techniques are adopted to compile linear recursions and some other complex recursions into synchronous chains and asynchronous chains. By predicate connection analysis and variable binding analysis, recursive ic-relevant queries can be simplified and evaluated by an efficient evaluation strategy extended from the multi-counting method[11].

1.1 General approach

Integrity constraints are used to express meta-knowledge about the facts and rules present in the database. For the information represented by facts and rules to be 'correct', the database has to 'satisfy' its integrity constraints. As with many researchers, we adopt the consistency view that consistency among facts, rules and constraints is demanded [15, 6, 21]. We say an integrity constraint is violated if it is inconsistent with the the facts, rules and constraints in the database. Thus, we can take use of the well-developed query techniques to check integrity constraints efficiently.

Simply evaluating integrity constraints after any modification is time-consuming and unnecessary. It is reasonable to assume that the database satisfies its constraints before the transaction, so that any violation afterwards must involve at least one of the modifications in the transaction. In the 70's and early 80's, many methods had been proposed for the enforcement of integrity constraints in relational databases based on this assumption; a survey can be found in [17]. Following Nicolas, Decker[4], Toper and Lloyd[14], others have extended this idea into deductive databases.

Kowalksky[6], Martens[15], Yum and Henschen[21, 22], etc., improved integrity checking by reasoning forward and focusing only on the part of the database affected by the update. Based on the derivation systems they used, we can classify their approaches into three classes.

- *Theorem-proving approach.* Decker[4], Toper and Lloyd[14], Kowalski, Sadri and Soper [6], and others make use of a Prolog-like tuple-at-a-time derivation

system to check integrity constraints within the theorem-proving framework, and achieve the simplification effect proposed by Nicolas [17].

- *Logical query planning approach.* Martens and Bruynooghe [15] make use of an implementation of a logical query language approach proposed by Ullman [19] as supporting evaluation mechanism, and incorporate it with integrity checking simplification methods.

- *Compilation approach.* Yum and Henschen [21, 22] based their integrity maintenance method on query compilation techniques to derive implicit modifications, simplify and check integrity constraints as in relational databases.

These different approaches are developed along with the development of the reasoning and query processing techniques of deductive databases. We will illustrate and analyze each approach with an example in the second chapter.

1.2 Syntax of a logical database

We adopt the Prolog convention of denoting logical datalog programs. We use strings of characters starting with an upper case letter to denote *variables* and strings beginning with a lower case letter (a, b, c,...) to denote *constants.* Integers are constants. We use identifiers starting with lower case letters (p, q, r, ...) for *predicate names* and the same identifiers starting with capital letters to represent the corresponding *relation* defined by the predicates.

Relations and predicates refer to the same object from database terminology and logic terminology respectively. Tuple and fact refer to the same object also.

A *literal* is of the form $p(t_1, t_2, ..., t_n)$ where p is a predicate name of arity n and each $t_i (1 <= i <= n)$ is a constant or a variable. A literal can be negated by a prefix $\sim$. We call $(t_1, t_2, ..., t_n)$ a tuple.

We consider a deductive database consisting of the following three exclusive components:

- the *extensional database* (EDB), which is composed of base predicates.
- the *intensional database* (IDB), which is composed of derived predicates which are defined by deductive rules of the following form:

```
r :-  p1, p2, ..., pn, ~ q1, ~ q2, ..., ~ qm.
```

- *integrity constraints* (IC), which are defined by the following form:

```
:- p1, p2, ..., pn, ~ q1, ~ q2, ..., ~ qm.
```

A query is represented in relational algebra or a Horn clause. For example,

```
? A(_, ?Z), B(Z, Y)
```

where, _ in the position of predicate attributes means we have no interest in the corresponding attributes.

and ? before a variable indicates the corresponding variable is inquired.

The following notations are used for the representation of relational algebra formulas:

$\cup$ represents *union*

$\Pi_{1,\$}B$ represents the relation projection of B on the first and last attributes.

$\bigcup_{i=1..n} \theta_i = \theta_1 \cup \theta_2 \cup ... \cup \theta_n$.

$chn^* = chn(X_0, X_1), chn(X_1, X_2), ...$

$chn^i(X_0, X_i) = chn(X_0, X_1), chn(X_1, X_2), ..., chn(X_{i-1}, X_i)$.

Predicate-ic connections among the predicates and integrity constraints is very important in the compilation of recursions and the enforcement of integrity constraints. The simplest variable pattern of a rule is the *linear variable pattern* (just like the above *chn* chain), where 1) all the predicates are binary and contain neither constants nor repeated variables, 2) two consecutive predicates share variables at their neighboring argument positions and there is no other shared variable among predicates, and 3) the two variables of the head predicate correspond to the starting and ending variables of the body [10]. Sometimes, variables are omitted in some formulas in this thesis if they are in a linear variable pattern.

1.3 Knowledge modification

Knowledge modification in a deductive database falls into three classes:

- data modification, modify the extensional database. We can further classify data modifications into three kinds of operations: addition, deletion, and change.
- rule modification, modify the intensional database.
- integrity constraint modification.

Actually we can classify rule modification and integrity constraint modification as for data modification, but we do not want to because we can not gain anything from it in the enforcement of integrity constraints.

Any of these three kinds of knowledge modification may affect the integrity and consistency of the knowledge database. Thus, integrity constraints must be enforced whenever these kinds of modification occur.

1.4 Assumption

We discuss and study the enforcement of integrity constraints in deductive databases based on the following assumptions.

Assumption 1 We assume that the database is function-free, range-restricted and stratified.

Assumption 2 We assume the recursions in the database are compilable to asynchronous or synchronous chains.

1.5 Thesis organization

This thesis is organized as follows. In Chapter 2, we survey recent research in the area of integrity constraint enforcement and recursive query processing. In chapter 3, we outline an integrity constraint transformation (compilation) algorithm, static qualitative analysis algorithm, predicate-ic connection analysis method, and integrity checking algorithm, for the enforcement of integrity constraints in non-recursive deductive databases. In chapter 4, we discuss simplification of integrity constraints, evaluation of primitive transitive recursive ic-queries, evaluation of complex recursive ic-queries, and integrity checking in recursive databases. In chapter 5, we discuss our methods, and compare the performance of our methods with previous methods. In the last chapter, we summarize our work and discuss future research directions.

CHAPTER 2

Background

2.1 Recursion in deductive databases

A predicate p is said to *imply* a predicate r (p => r) if there is a Horn clause in an IDB with predicate r as the head and p in the body, or there is a predicate q where p => q and q=> r [8]. A predicate p is said to be *recursive* if $p => p$. Two predicates p and q are *mutually recursive* if $p => q$ and $q => p$. A recursive rule is linear if there is no mutual recursion and any rule with r as head contains no more than one occurrence of r in the body. A transitive recursion, i.e., single chain recursion, which contains only a one-sided join with the recursive predicate is a special case of linear recursion. Linear recursion is a very important kind of recursion and has been studied extensively because it is believed that practical recursions are linear or transferable to linear ones.

A *rule cluster* of a predicate R is the maximum subset of rules in an IDB in which

all of the head predicates of the rules are either R or P where P => R.

As above, an **integrity constraint cluster** can be defined as a set of rules of the union of clusters of all of the rules which appeared in the ic.

Recursion is a very important feature of Knowledge-Base Management Systems. Much research has been done on the evaluation of recursion. Many good methods have been proposed like Magic sets[3], Counting[3], Henschen-Naqvi [2, 7], and query-independent compilation[8, 9, 10]. Bancilhon and Ramakrishnan have written a very good survey on the problem of evaluation of recursive queries against deductive databases [2].

Recursions can be classified based on definitions or compilation results[9]. We are interested at the classification based on compilation results from the evaluation point of view. It is possible and desirable to treat different recursions differently in the enforcement of ICs.

1. A *bounded recursion* is a recursion whose compiled formula consists of finite relational expressions, which is equivalent to a set of nonrecursive rules.

2. An *asynchronous chain recursion (AC)* (especially, *single-chain recursions*) is recursion whose compiled formula consists of a finite number of asynchronous chains and possibly a small number of other predicates. *Asynchronous chains* means that the length of one chain is independent to the lengths of the other chains in the formula.

3. A *Synchronous chain recursion* is a recursion which is compilable to one or a set of synchronous chains. *Synchronous chains* means that all chains have the same length.

4. *Hyper-string recursions* an infinite set of strings with irregular patterns.

2.2 Recursive query processing

Theorem [Han1] A recursive cluster consisting of one linear recursive rule and one or more nonrecursive rules is compilable to either a bounded recursion or an n-chain recursion[8].

By analyzing predicate connections in a deductive database, a function-free linear recursion can be compiled to[8]:

1. a bounded recursion, in which recursion can be eliminated from the program,

2. an n-chain recursion, whose compiled formula consists of one chain, or n synchronized chains.

Example 2.3 Following is the definition of a well-known same generation recursion.

```
same_generation(X, Y) :- parent(X, X1), parent(Y, Y1),
                         same_generation(X1, Y1).
same_generation(X, Y) :- sibling(X, Y).
```

The above single linear recursion can be compiled to a two-chain recursion as shown below:

$$same\_generation(X_0, Y_0) = \bigcup_{i=0..\infty} (parent^i(X_0, X_i), parent^i(Y_0, Y_i), sibling(X_i, Y_i)).$$

Theorem [HL1] Many interesting function-free recursions such as, multiple linear recursions in linear variable patterns, some mutual recursions, canonical multi-linear recursions, can be compiled to asynchronous chain recursions and processed by transitive closure query processing strategies[10].

The processing of primitive n-chain recursions has been studied extensively. In case of n = 1 (Single chain recursion) or asynchronous chain recursion, they can be processed by well-studied transitive closure algorithms. Other multi-chain recursion ($n > 1$) can be handled by synchronized processing of n chains. Most recognized processing methods include: Henschen-Naqvi algorithm[7], the counting methods and the magic sets method [3] and their extensions [2].

2.3 Theorem-proving approach

Application domain: function-free, range-restricted non-recursive database.

Derivation system based: tuple-at-a-time Prolog-like derivation system.

Methods for avoiding redundantly rechecking constraints which are unaffected by the transaction were proposed by Topor and Lloyd [14], Decker [4]. Their simplification algorithms extend Nicolas' algorithm[17] for relational databases and primarily

consist of the following two steps:

- Generate a simplified checking set from the transaction, which is possibly simpler and more highly instantiated than the original set, and whose satisfaction ensures the consistency of the updated database.
- Evaluate the derived constraint set against the actual database by top-down reasoning.

The reasoning system they used is a purely top-down, Prolog-like tuple-at-a-time derivation system. Backward reasoning does not fit for checking integrity constraints because it fails to focus only on the part of the database which is affected by the updates. To overcome the drawback of top-down reasoning of Prolog for integrity checking, Kowalski, Sadri and Soper [6] extended the Prolog-like derivation system by:

- allowing forward reasoning as well as backward reasoning.
- incorporating additional inference rules for reasoning about implicit deletions caused by changes to the database, and
- incorporating a generalized resolution step, which is needed for reasoning forwards from negation as failure.

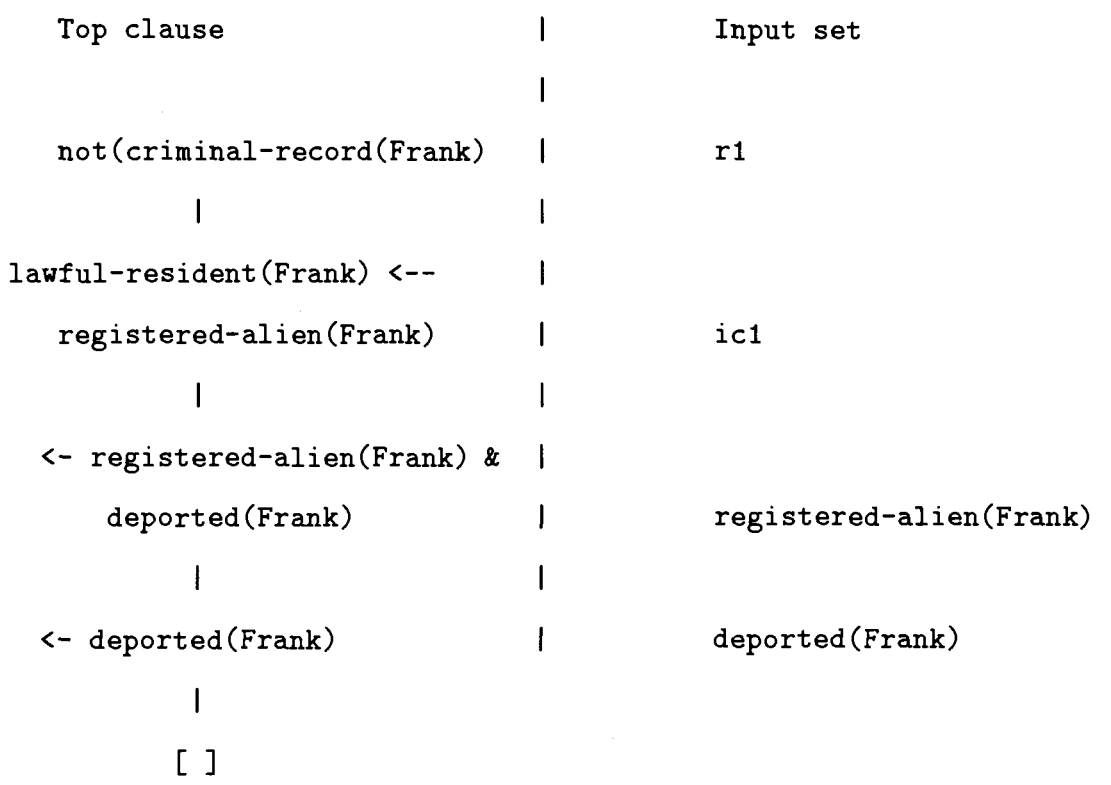
In the Prolog-like derivation system we can only reason starting from a denial. To reason forward, the proof procedure underlying the consistency method allows us reason from any deductive rules, denial or negated atom. In short, the consistency

method works as follows: Take the added or deleted fact, added or deleted rule, inserted constraint as top clause and the database (EDB, IDB and ICs) as the input set to reason forward. If a refutation is found, the update violates the constraints. Otherwise, if no refutation can be found, the update is valid.

Example 2.1 In the following database, *citizen* and *registered-alien without criminal-record* are defined to be lawful residents. To be in a valid state the database is required that no lawful residents are *deported.* Suppose we have a transaction to delete Frank's criminal record. The database is assumed to be consistent before the transaction.

```
IDB:   lawful-resident(x) :- registered-alien(x),
                             not(criminal-record(X)).   (r1)

       lawful-resident(x) :- citizen(x).                (r2)

ICs:       :- lawful-resident(x), deported(x).          (ic1)

EDB:   deported(John),
       criminal-record(Frank),
       registered-alien(Frank),
       deported(Frank),
       citizen(Tom)

Transaction(T): delete  criminal-record(Frank)
```

By taking *not(Criminal-record(Frank))* as the top clause we can show the integrity checking procedure as shown in the diagram below.



The search space consists of one refutation illustrating that the update causes the database to violate the integrity constraints.

It is difficult to implement forward reasoning in a backward reasoning system. Forward reasoning makes the consistency method more efficient than other previous methods. Kowalski, Sadri and Soper found the importance of focusing only on the affected part of the database by the update.

2.4 Query planning approach, Martens' integrity checking method

Application domain : function-free, stratified and non-recursive deductive database. All rules and constraints in the database must be range-restricted.

Derivation system based : relation-at-a-time logical query language implementation.

Martens and Bruynooghe[15] tried to integrate an approach to implementing a logical query language presented by Ullman [19], and efficient techniques for integrity checking as advocated by Nicolas[15], Decker[4], Topor and Lloyd[12], Kowalski, Sadri and Soper[6], etc. Their method was based on Ullman's query planning method using rule/goal graphs and capture rules[19]. The purpose of rule/goal graphs is to allow the planning of evaluation strategies. Martens and Bruynooghe extended it for the enforcement of integrity constraints. One advantage is that using it, the system can fix the best order in which to evaluate implicit modification and check integrity constraints.

The rule/goal graph Martens and Bruynooghe used is slightly different from Ullman's original rule/goal graph. They draw arcs in opposite directions [19, 15]. In their algorithms they use posocc-ic and negocc-ic to represent a positive occurrence or a negative occurrence in an ic. They use posocc-r and negocc-r to represent a positive occurrence or a negative occurrence in a rule. For integrity checking, Martens and Bruynooghe extended the rule/goal graph as follows:

1. If ic is a constraint with n (ordered) variables, then there is a node ic^p for each adornment, p, where p is a string of b (bound) or f (free) of length n, representing the state of arguments or variables.

2. We add arcs $ic^p -> B_i^q$ (arc from ic^p to B_i^q), where q is such that those arguments of B_i are bound which contain variables bound by p or no variables at all. The other arguments of B_i are free.

3. For every predicate A occurring in the database, add a transaction node A_t.

4. For every node A_t, we add arcs:

   - a posocc-ic (negocc-ic) arc $A_t -> ic^p$ for every constraint ic where A occurs in a positive (negative) literal and p is such that all variables appearing as arguments to (one occurrence of) A are bound and no other variables are bound.
   - a posocc-r (negocc-r) arc $A_t -> r^p$ for every rule r where A occurs in a positive (negative) literal and p is such that all variables appearing as arguments to (one occurrence of) A are bound and no other variables are bound.

Martens' method treats different kinds of modifications separately, adding a fact, deleting a fact, adding a rule, deleting a rule, adding an integrity constraint, deleting an integrity constraint. Their main algorithm is for the addition and deletion of facts. Compared with the case of addition or deletion of facts, the integrity checking for adding or deleting rules is easier. They merely treat the receipe of the added or

deleted rule as added or deleted facts, then go through the same procedure as adding or deleting facts. When adding an integrity constraint, they just treat it as a query; if it is not false it causes integrity problem. The easiest case to consider is for the deletion of constraints; no checking needs to be done, only integrity checking related information needs to be updated accordingly. The basic procedure for checking adding or deleting a fact is as follows:

- First check whether the update directly violates integrity constraints
- Then derive all implicit additions and deletions which are an immediate consequence of the update.
- Let the implicit addition or deletion undergo the same procedure.

Let us take Martens' algorithm for the addition of a fact as an example to illustrate and analyze his method.

Adding a fact $A(a_1, a_2, ..., a_n)$

1. If the addition of this fact has already been treated, continue with another change.
2. If $A(a_1, a_2, ..., a_n)$ is provable in the initial database, that is, $A(a_1, a_2, ..., a_n)$ is redundant; then, continue with another change.
3. For every posocc-ic arc $A_t - - > ic^p$ where $A(a_1, a_2, ..., a_n)$ can be unified with the A-occurrence in ic, compute the relation for ic^p. If a non-empty relation

is found, stop the processing of this transaction, undo it and inform the user about the integrity constraint violation that occurred.

4. For every posocc-r arc $A_t - - > r^p$ where $A(a_1, a_2, ..., a_n)$ can be unified with the A-occurrence in r, compute the relation for r^p. Translate to facts concerning the predicate occurring in the head of r. Treat every resulting tuple as an added fact.

5. For every negocc-r arc $A_t - - > r^p$ where $A(a_1, a_2, ..., a_n)$ can be unified with the A-occurrence in r, compute the relation for r^p. Translate it to facts concerning the predicate occurring in the head of r. Treat every resulting tuple as a deleted fact.

Martens and Bruynoogheh's method reasons relation-at-a-time instead of Prolog-like tuple-at-a-time, which is preferable and more efficient in deductive databases. It is possible to take advantage of precompilation to optimize integrity checking if modification to rules and constraints is not frequent. The reasoning method is bottom-up, so Martens' method has advantages of a bottom-up evaluation system, which can make the method focusing only on the part of the database which is affected by the update.

One potential problem is caused by their pure bottom-up reasoning system, which fails to focus only on the parts of the database which may affect constraints. All of the implicit modifications are derived and without further consideration, some of them may not affect any constraints.

This method is recursive. For the integrity checking of an addition of a fact, the first step of *adding a fact* is trivial. The second step is not so simple, to find if A(a1, a2, ..., an) is provable in the old database, we need to reason against the old database. They spend almost the same amount time on checking boundary conditions as deriving implicit modifications. The third step is unvoidable but not very expensive in the entire checking procedure. The last two steps equal several queries. We need to evaluate all of the rules in which A occurs. These steps will generate more implicit modifications which need to undergo the same procedure as A(a1, a2,...). This is the most expensive part of the method.

Martens and Bruynooghe's method can only handle the situation in which a predicate may occur only once in the body of a rule or integrity constraint. In this case, simply unifying the added or deleted fact with its occurrences in a rule's body is not enough to derive the immediate effect on the rule in the last two steps. Nevertheless, it is easy to extend their method to handle more complex situations.

Example 2.2: The following uses abbreviations l-r for lawful-residents and e-v for eligible-visitor, etc. for the database in *Example 2.1.* One rule defining that *eligible visitor* are *dependents* of lawful residents is added.

```
IDB:   l-r(X) :- r-a(X), not(c-r(X)).          (r1)
       l-r(X) :- cit(X).                       (r2)
       e-v(X) :- depen(X, Y), l-r(Y).          (r3)

ICs:       :- l-r(X), deported(X).             (ic1)
```

```
EDB:   depen(John, Tom),
       deported(John),
       c-r(Frank),
       r-a(Frank),
       deported(Frank),
       cit(Johnson),
       cit(Tom).

T: Add r-a(George)
```

In rule r3 of the database above, e-v has one variable, which has two binding situations, free and bound, so that there are two nodes, $e - v^b$ and $e - v^f$, for e-v in the extended rule/goal graph as shown in Fig. 2.2 (a). Rule r3 has two variables X and Y. There is an arc from $e - v^b$ to r_3^{bf} because X of r3 is bound and Y of r3 is free due to X of e-v is bound. For the same reason, there is an arc from $e - v^f$ to r_3^{ff}. There is an arc from r_3^{bf} to $depen^{bf}$ because *depen* occurred in the body of r3 and X and Y of *depen* are bound and free accordingly when X and Y in r3 are bound and free. In the same way we can complete the extended rule/goal graph for rule r3 and other rules as shown in Fig. 2.2 (a) and (b). There is an transaction node for each predicate. When variables in *depen* are bound, the binding can be passed to variables in rule r3, so that there is an arc from $depen_t$ to r_3^{bb}. In this way, the entire extended rule/goal graph for the database of *Example 2.2* can be built and is shown in Fig. 2.1.

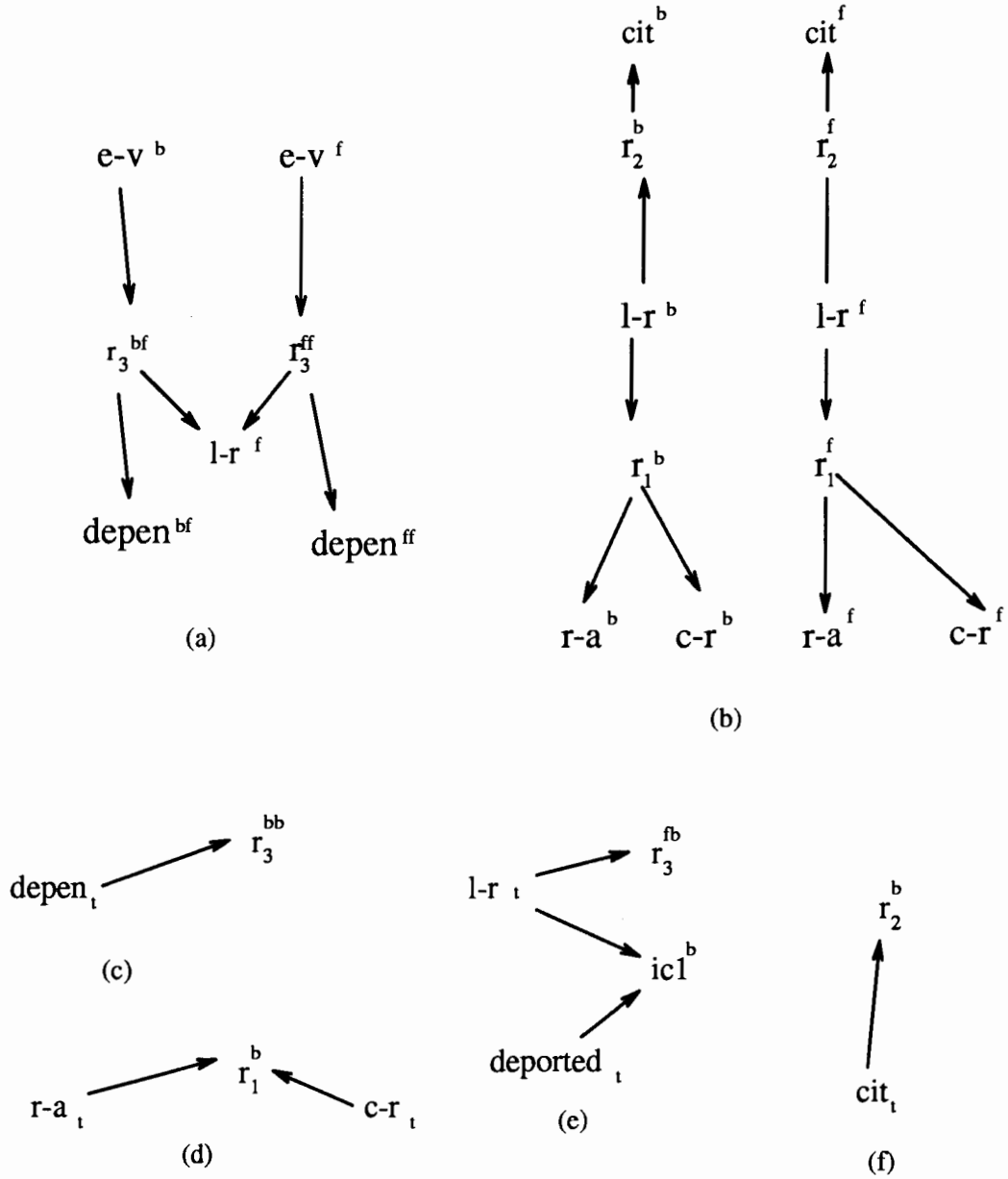


Fig. 2.1 The extended rule/goal graph for the database in Example 2.2

If we start from the transaction node of r-a_t in Fig. 2.1 (d), we find r-a_t does not occur in any integrity constraint, so it will not affect any integrity constraint

directly. From r-a_t to r1, there is a posocc-r arc, so we need to unify r-a(George) with r-a's occurrence in r1's body and treat the resulting relation as added facts for l-r, l-r(George). l-r(George) needs to undergo the same procedure as adding r-a(George). l-r(George) is verified by the second step as not derivable in the old database. l-r occurs in ic1 as shown in Fig. 2.1 (e), so ic1 needs to be checked with the unification of l-r(X) with l-r(George); fortunately the result is empty. l-r occurs in r3 as shown in Fig. 2.1 (e), we have to evaluate its influence on e-v and the result need to undergo the same procedure as before according to Martens' method. But we find any modification of e-v has no chance to violate any integrity constraint in the database, so our last effort is in vain.

2.5 Compilation approach, maintaining state constraints

Application domain: function-free, single linear recursive Horn database. Integrity constraints are required to be range restricted.

Derivation system based: compilation approach.

McCune and Henschen [16] proposed a method for maintaining state constraints in a relational database. From a theoretic point of view, they attempt to represent the relationship between the database before and after an update by transition axioms, so that integrity constraints on the new database can be represented and simplified on the initial database state. For a transaction, they try to generate a complete test set to be applied before the transaction is performed. They emphasize that checking integrity

constraint should be done before updating to avoid unnecessary undoing of illegal transactions. Unlike Nicolas' direct substitution and simplification method, they use a more cumbersome computation extensive method to simplify integrity constraints. Generic constants are used during constraint compilation at their specification time. When the database is in use and a user requests an update, the appropriate compiled formulas are retrieved, the generic constants are replaced with the update values, the formulas are simplified, and (if need be) tested against the database.

Yum and Henschen [21, 22] generalized the state-constraint maintaining method in relational databases above to deductive databases involving Horn clauses. They try to deduce all of the relevant implicit and explicit updates upon arrival of a transaction, then the same integrity constraint enforcement strategy as in relational databases can be exploited to maintain state constraints in deductive databases. Their method has three stages:

- Simplification of integrity constraints at integrity constraint specification time.
- Derivation of induced updates upon arrival of a transaction through a compilation approach.
- Evaluation of the relevant simplified integrity constraints.

Yum and Henschen use the same method of simplifying integrity constraints proposed by McCune and Henschen [16]. The simplification method is outlined below.

Algorithm 2.3.1 Simplification of integrity constraints

1. Form a transition axiom (TAX) for the update.
2. Replace all of the instances of the new relation in the integrity constraint axiom with the right-hand-side of TAX, which is equal to the new relation.
3. Convert the above integrity constraint clause into a conjunctive form.
4. Delete all of the conjuncts subsumed by the old integrity constraint clause, which is true because we assume the database satisfies its constraints before the update.
5. Convert the remaining clause into a disjunctive form.

Induced updates are called **redundant** when they are already derivable before the update for insertion or they are still derivable after the update for deletion. Redundant facts are removed by querying them on the initial database state or the new database state for redundant induced insertion or deletion separately.

To deduce induced updates, they compile all the IDB rules, whose head relation may violate some integrity constraints(which appears in some integrity constraints). When an insertion update arrives, evaluate the increment of each compiled rule with the update predicate in its body. Thus, we can deduce a set of non-redundant facts that are implicitly or explicitly inserted to the database due to an update. The set of deleted facts can be derived in the same way for deletion updates. The same procedure can be applied to the update of an IDB.

Let us take insertion of a fact P' as an example to illustrate the derivation of

induced updates.

A forward chaining rule cluster for a base predicate, F_cluster(P) is a set of compiled rules, with a predicate P in its body, whose head relation may violate some integrity constraint. $F\_cluster_\theta(P')$ is $\bigcup_{i=1..n} R\theta_i$ where R is a unified rule in the F_cluster(P), $\theta_1, ..., \theta_n$ are most general unifiers(mgus) of each occurrence of P in the $F\_cluster(P)$ and an update P', and n is the number of mgus. $F\_cluster_{rel}(P')$ is a set of relational database expressions for evaluating $F\_cluster_\theta$ (P').

The set of induced updates due to an update can be obtained by evaluating $F\_cluster_{rel}(P')$ and removing redundant facts.

Yum and Henschen proposed two algorithms to deduce induced updates for transitive closure and primitive linear recursive rules. The transitive closure rules actually are single chain recursions. Primitive linear recursive rules are single linear recursions [22, 8]. The single linear recursive rule set consists of

```
r(X, W) :- b(X, Y), r(Y, Z), c(Z, W)
r(X, Y) :- a(X, Y).
```

The compilation result formula for r is:

$$R = \bigcup_{k=0..\infty} B^k A C^k \qquad (2.1)$$

Thus, the relational expression for $F\_cluster_{rel}(B')$ is

$$\bigcup_{j=0..\infty} (\bigcup_{k=0..\infty} \Pi_{1,\$}(B^j B' B^k)) AC^m \qquad (2.2)$$

where m = j + k +1

After careful examination, the above equation (2.2) can be transformed into equation (2.3) because B' is a tuple and $B' \bowtie B'$ is still B'.

$$\bigcup_{j=0..\infty} \bigcup_{k=0..\infty} ((\Pi_1(B^j B') \times \Pi_{\$}(B' B^k)) AC^m) \qquad (2.3)$$

where m = j + k +1

The above compiled formula (2.3) can be processed efficiently using counting method, the B chain and C chain should be synchronized.

The last stage of their method is the evaluation of simplified integrity constraints. For each induced inserted and deleted fact, the related simplified integrity constraints are retrieved, the generic constants are replaced with the fact, and the formula is evaluated against the updated database.

We should notice the difference between the evaluation method here with McCune and Henschen's corresponding method for relational databases. They evaluate the simplified constraint set against the initial database state before. Here they evaluate constraints against the updated database for simplicity and avoiding the uncontrollable complexity of simplification of integrity constraints in deductive databases in

the same way as in relational databases.

This method can focus its attention on the affected part of the database which is related to integrity constraints fairly well. The compilation approach is adopted to deduce induced updates, which makes it more efficient than other approaches. That integrity constraints can be compiled at their specification time is also one of the processing advantages.

Yum and Henschen derive induced updates and check integrity constraints separately. The overall optimization of derivation of induced updates and evaluation of integrity constraints is impossible. All the relevant induced updates are deduced before IC evaluation; unnecessary derivation of induced updates may occur when a violation of integrity does happen.

Redundant facts are removed by querying them on the initial database, which is quite expensive, especially for those facts of recursive predicates.

2.6 Summary

We have already known that reasoning forward (bottom-up) from updates can achieve the effect of focusing only on the part of the database which is affected by updates, and that reasoning backward (top-down) from integrity constraints can focus only on the constraint closures, i.e., the part of the database relevant to constraints. It

is desirable to reach these two goals at the same time while checking integrity constraints. We propose a new integrity checking method which will combine these two reasoning strategies. We will use a top-down theorem-proving approach to prune the forward reasoning space first, and use the result to guide integrity constraint checking.

Recursion makes integrity constraint checking much more challenging. Generally speaking reasoning forward from facts is not a best fit for dealing with integrity constraint checking in the presence of functions and recursion[15]. Yum & Henschen presented a processing method for a very simple kind of recursion. Recently, many interesting methods have been proposed for recursive query processing. We will make use of recent research results on the query independent compilation of special recursions to deal with the recursion problem in integrity constraint checking.

CHAPTER 3

IC checking in non-recursive databases

3.1 Preliminary remarks

In deductive databases, any update to base predicate space(defined in an EDB) or IDB predicate definitions may cause changes to relations defined by other derived predicates(defined in an IDB). This kind of modifications is called an implicit modification. A modification may affect an integrity constraint directly if the modified predicate occurs in the integrity constraint, and/or indirectly if the modification causes implicit modification on an derived predicate which occurs in a constraint.

Induced updates are the set of facts that are implicitly or explicitly inserted to or deleted from a database due to the updates of a transaction. An induced update

is called **redundant** if it already derivable before the update for insert or they are still derivable after the update for delete [21].

An **ic-query** is defined as a query for integrity checking or derivation of induced updates. In section 3.6 we will explain how to compose ic-queries.

Generally speaking, the database is consistent before an transaction. Any modification may affect only a part of the database and some of the integrity constraints. Most of the evaluations will be redundant if we simply evaluate all of the integrity constraints after any modification. To simplify integrity constraints and to reduce redundant checking has become a key issue in the development of integrity checking methods.

In summary, we face the following challenges to enforce integrity constraints:

- Simplification of integrity constraints. Redundant checking should be deduced as much as possible.
- Derivation of implicit modification and redundancy exclusion.
- Integrity checking against the real databases.

3.2 Basic simplification principles

We all know that the relational join operation has monotonic behavior. So, the result of a query of an integrity constraint must be monotonic. Consider a relation

$$R = A \bowtie B.$$

Deletion on A or addition on B can never increase R. Two basic principles can be deduced from this monotonic behavior[17].

Principle 1: The deletion of tuples from a positive literal in an integrity constraint will not cause integrity problems to this integrity constraint directly.

Principle 2: The addition of tuples to a negative literal in an integrity constraint will not cause integrity problems to this integrity constraint directly.

By analysis of the structure of the IDB we can determine that some EDB's modifications will not violate any ICs or simplify integrity checking at least. This is one kind of modification which will not cause an integrity violation. Actually, there is another group of EDB modifications which will never cause integrity problems. By carefully examining the structure of all of the Horn clauses in a deductive database, we find two facts: generally a predicate has quite a few attributes and integrity constraints impose constraints on only a few attributes. Thus, we can predict that most *change* modifications on EDB predicates may not affect the constrained attributes, thus no integrity violation.

Consider the integrity constraint,

the fare of each flight must be greater than 0,

:- flight(Fno, ... , Fare), $Fare <= 0$.

We can soon find that modification on attributes of *flight* other than *Fare* will never cause a violation to the integrity constraint mentioned above.

3.3 Simplification of integrity constraints

Yum and Henschen [21, 22] used a compilation method to derive induced updates for pure Horn databases. In Horn databases with no negation, non-recursive rules can be compiled into formulas including only EDB predicates by unification. In Horn databases augmented with negation, the compilation is not so straightforward, and the output cannot be so regular. To derive induced updates, we cannot simply unify the added/deleted fact with their occurrences in the compiled formula to derive induced updates if there are some negative literals or subformulas in the compiled formula.

Example 3.1: In the following database, citizen and registered-alien without criminal-record are defined to be lawful residents. A person is said to have criminal record if he is a criminal in Canada or has a criminal record in his/her home country. To be in a valid state the database is required that all lawful residents are not deported.

```
IDB:   lawful-resident(X) :- registered-alien(X),
                             not(criminal-record(X)).   (r1)

       lawful-resident(X) :- citizen(X).                (r2)

       criminal-record(X) :- criminal(X, Canada).       (r3)
       criminal-record(X) :- citizen-of(X, Country),
                             criminal(X, Country).      (r4)

IC:      :- lawful-resident(X), deported(X).            (ic1)
```

The compilation form of r1 could be:

```
lawful-resident(X) :- registered-alien(X),
                      ~ (criminal(X, Canada) V citizen-of(X, Y) &
                      criminal(X, Y) ).
```

It is wrong to derive induced updates by unifying the update with criminal or citizen-of. We have to treat negated predicates differently in the compilation process.

We already know that induced updates, which can be gotten by unifying updates with their occurrence in a rule, are probably redundant. It is wrong to unify redundant deletion with its negative occurrence, which may result false conclusion. *It is impossible to avoid redundancy checking of implicit modification if the predicate occurs negatively in another predicate's definition or in an integrity constraint.*

For example, an integrity constraint below:

```
:- p, ~q
```

If an implicit deletion Q' on q is redundant, an integrity violation is concluded because P & Q' happens to be true. Actually, deletion of Q' is redundant (Q' is still in Q), P & Q is still false, that is, no integrity violation occurred.

There are two ways to handle redundant implicit modification, Martens checks every derived implicit update[15], and Yum & Henschen checked only for predicates occurring in integrity constraints. We will adopt the compilation approach, and not check redundancy unnecessarily. But due to possible negative literals in our databases, we do some redundancy checking for implicit modifications. As a preliminary result, a method checking less implicit modification redundancy will have better performance in most situations than one with more checking, because the compilation approach can have better optimization.

An **expansion** of a rule or integrity constraint is the unification of a derived predicate in the body of the rule or constraint with a definition of the predicate.

Algorithm 3.1. Transformation of integrity constraints and rules by expansion.

Input Integrity constraints and an IDB.

Output Partially compiled integrity constraints and ic-relevant rules which have no derived predicates occurring in their bodies.

1. Expand an integrity constraint. Unify a derived predicate p, which occurred positively in an integrity constraint, with its definitions respectively, which may result in several new integrity constraints.

2. Perform the above step recursively until no derived predicate occurred positively in any integrity constraint.

3. Transform ic-relevant rules (defining derived predicates occurring negatively in some integrity constraints) similarly.

    - Expand an ic-relevant rule. Unify a derived predicate p, which occurs positively in the body of an ic-relevant rule, with its definitions, respectively. Unification may result in several new ic-relevant rules.
    - Perform the above step recursively until no derived predicate occurring positively in any ic-relevant rule.
    - All of the rules defining those predicates which occur negatively in ic-relevant rules are ic-relevant rules also; so that they need to be compiled and go through step 3.

Example 3.2: Using algorithm 3.1 we can transform the integrity constraints and relevant rules in the database in Example 3.1. Because *criminal-record* is defined by base predicates alone, no rules need to be compiled. The integrity constraint ic1 is transformed into two integrity constraints:

```
:- registered-alien(X), not(criminal-record(X)), deported(X)   (ic1-1)
:- citizen(X), deported(X).                                    (ic1-2)
```

3.4 Static qualitative analysis simplification method

After transformation of integrity constraints and relevant rules, it is easier to use the static qualitative analysis method [23] to analyze the database to find some integrity checking related information. By static analysis of the transformed integrity constraints and relevant rules, we can determine:

- when a modification may affect which integrity constraints,
- when a modification may affect which ic-relevant derived predicates,
- an optimum evaluation plan for integrity constraint checking when a modification occurs.

An implicit modification is said to be *redundant* if it has no opportunity to affect any integrity constraint directly and/or indirectly. This kind of implicit modification should be avoided in the integrity constraint checking process.

A predicate is *ic-relevant* if it belongs to a cluster of an integrity constraint of the database. Because it *implies* some derived predicates occurred in some integrity constraints, modification of the predicate provides the opportunity to cause violations to those integrity constraints.

An extended dependency graph is a dependency graph[20] extended with representation of the relationships between predicates and integrity constraints, and arcs marked with *posocc* and *negocc* explicitly showing a predicate's occurrence in a rule or an integrity constraint is positive or negative. *Posocc-p* and *negocc-p* mean a positive occurrence and a negative occurrence of a predicate in the definition of a predicate respectively. *Posocc-ic* and *negocc-ic* mean a positive occurrence and a negative occurrence of a predicate in the definition of an integrity constraint respectively.

An affected graph is a part of the extended dependency graph relevant to one EDB predicate.

The static qualitative analysis simplification method is straightforward[23]. First, an expanded dependency graph[23] is built for the database as explained with Fig. 3.1 below, which represents the hierarchical structure of the database.

Second, an affected graph[23] is built for each base predicate's addition and deletion operation separately to determine which derived predicates and integrity constraints are affected by the modification.

Third, the affected graph is pruned to eliminate irrelevant integrity constraints and derived predicates. An integrity constraint should be discarded from the affected graph if it was found to be not affected by the modification by qualitative analysis. A derived predicate, an intermediate node, should be discarded from the integrity constraint checking process if it has no way to affect any integrity constraint.

Fourth, record the pruned affected graph for further use in real integrity checking.

Finally, we may find that many modifications on an EDB may never affect any integrity constraint if its pruned affected graph is empty.

Normally, after transformation of integrity constraints and relevant rules, the integrity constraint relevant part of the database has a very simple structure. Most of the base predicates only occur in integrity constraints, not in definitions of ic-relevant derived predicates. For those base predicates which do not occur in any compiled ic-relevant rules, they do not need to go through the above procedure thoroughly. As in relational databases, we can determine the necessary information simply by using the two basic principles 1 & 2.

Example 3.3: Let us analyze the transformed database shown in *Example 3.1 & 3.2* by the static qualitative analysis method. The extended dependency graph for the database is shown in Fig. 3.1. There is one node for each predicate and integrity

constraint. There is an arc for each occurrence of a predicate in a rule or an integrity constraint with a mark showing if it occurs positively or negatively. For example, the arc from *citizen* to *lawful-resident* indicates that *citizen* occurs positively in *lawful-resident's* definition (r2), and the arc from *citizen* to *ic1-2* marked *posocc-ic* indicates that *citizen* occurs positively in *ic1-2*.

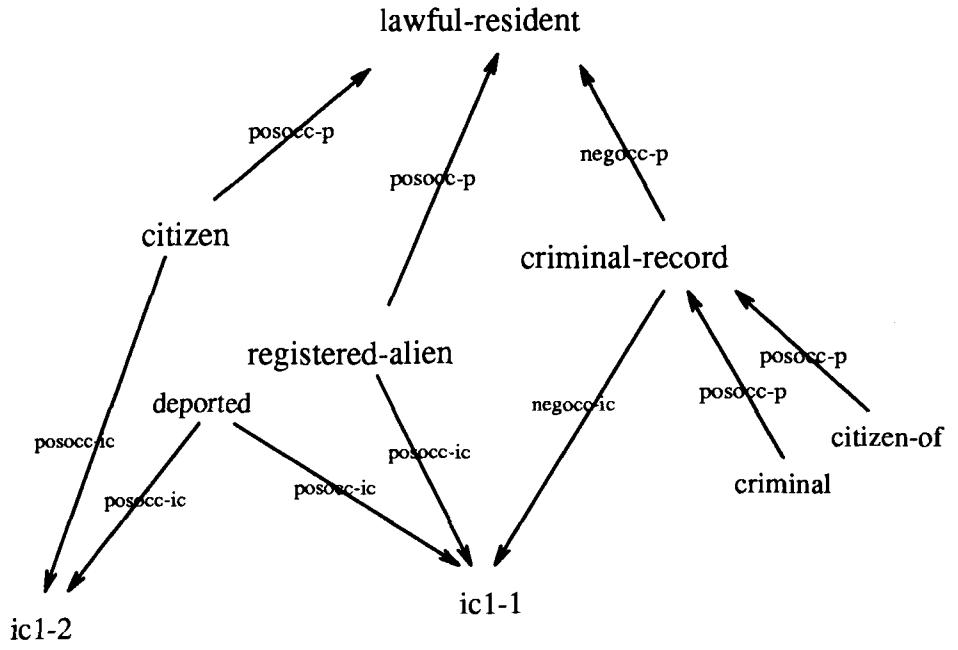


Fig. 3.1 -- An extended dependency graph

We can pre-analyze the relationships among integrity constraints and predicates. From Fig. 3.1 we find that *lawful-resident* has no influence on ic1-1 and ic1-2 because there is no path from *lawful-resident* to reach the integrity constraint nodes ic1-1 or ic1-2. In other words, *lawful-resident* does not imply any integrity constraint. Thus, *lawful-resident* can be discarded from the process of enforcement of integrity constraints. After general pruning of nodes which represent ic-irrelevant predicates and

which cannot reach integrity constraint nodes, and pruning of dangling arcs in the extended dependency graph, we obtain a common affected graph for the database. The general affected graph (for all ic-relevant base predicates) for our example database is shown in Fig. 3.2 after that node *lawful-resident* and relevant arcs are discarded from Fig. 3.1.

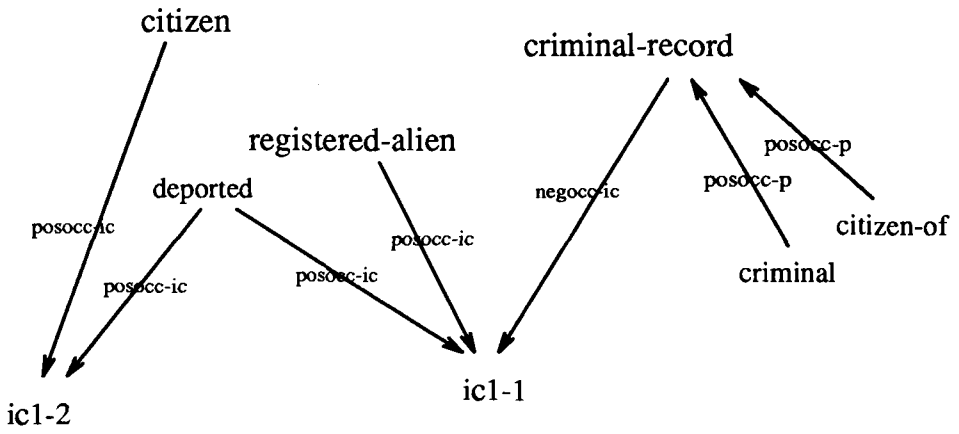


Fig. 3.2 -- A general affected graph

According to *Principle 1 & 2* we can determine that deletion on *citizen, registered-alien*, and *deported* will not cause integrity constraint violations. By qualitative analysis, we can determine that addition on *criminal* and *citizen-of* will not cause any integrity constraint violation because they can only cause implicit addition to *criminal-record*, which will not affect ic1-1. The following information in Table 3.1 can be obtained by qualitatively analyzing the potential effect on the IDB and ICs of the database shown in *Example 3.1* that is caused by various EDB modifications. All other EDB modifications not listed will not affect any integrity constraint directly or indirectly.

Predicate	Operation	Derived-predicates affected	ics affected
citizen	addition		ic1-2
deported	addition		ic1-1 & ic1-2
reg.-alien	addition		ic1-1 & ic1-2
criminal	deletion	criminal-record	ic1-1
citizen-of	deletion	criminal-record	ic1-1

Table 3.1 Affected information of the EDB modifications

3.5 Variable-ic connection analysis

Let us define a variable to be *ic-relevant* to an integrity constraint if a change modification which occurs on the variable may cause the integrity constraint to be violated. A predicate is ic-relevant to an integrity constraint if its modification may cause a violation to the integrity constraint. A variable in a formula is call shared if it occurs in more than one predicates in the formula.

Algorithm 3.2. Find ic-relevant variables for an integrity constraint:

- Put each predicate occurring in the ic with its shared variables in the list OPEN,
- Fetch a derived predicate from the OPEN queue, and put it in the list, CLOSE. For each of the rules defining this predicate, analyze the rule body and determine ic-relevant variables for each predicate occurring in the rule body. Other

than shared variables, those variables corresponding to the head predicate's ic-relevant variables are ic-relevant. Then record the result in the OPEN list if the predicate is not in CLOSE or with different ic-relevant variables as before.

- If there are no derived predicates left in the OPEN list, return the OPEN list. If there are extensional predicates occurring more than once in the result, we merge their ic-relevant variables.

Algorithm 3.2 is complete; it can determine all of the ic-relevant variables because: (1) if there is no derived predicate in the integrity constraint, then algorithm 3.2 can determine all of the shared variables; and only shared variables are ic-relevant, so the algorithm can find all ic-relevant variables. (2) if there is some derived predicates in the constraint, then the recursive step can determine all of the ic-relevant variables. The algorithm determines those ic-relevant distinguished variables (appearing in the head predicate) and all of the shared variables, which are the only variables which may change the corresponding ic-query result. Thus, we are sure that the algorithm is complete in this situation.

The use of *Example 3.2* is illustrated in *Example 3.4* below.

Method 2: Integrity checking by predicate-ic connection analysis.

- For a change modification on an extensional predicate, retrieve the predicate's related information about variable ic-relevance.
- For each potentially affected integrity constraint, if the modification does not affect any ic-relevant variables, then we conclude that this modification will not

cause any problem to the integrity constraint defined by this ic.

Generally, a *change* operation modifies only a few attributes; we can easily identify whether ic-relevant attributes are affected. If some ic-relevant attributes are affected, we must check these constraints against the database.

We can use a simple situation to reason the correctness of the second method. For example, for an integrity constraint

```
:- a(X, Z), b(Z, Y)
```

the corresponding integrity checking query is

$$?A(X,Z), B(Z,Y)$$

which by transformation into relational algebra, is equal to:

$$\Pi_Z A(X,Z) \bowtie \Pi_Z B(Z,Y)$$

The modification on X in A will not affect $\Pi_Z A(X,Z)$ and the modification on Y in B will not affect $\Pi_Z B(Z,Y)$. Thus, a *change* modification on X of A and/or Y of B will not change the result of the above query.

Example 3.4: Let us make the database in Example 3.1 more complex, and add a few attributes to each predicate.

```
IDB:
   lawful-resident(Name) :- registered-alien(Name, Nationality,
     Married-status)
                not(criminal-record(Name, CriminalCase)).          (r1)

   lawful-resident(Name) :- citizen-of(Name, Canada).              (r2)

   criminal-record(Name, CriminalCase) :-
                  criminal(Name, CriminalCase, Canada).            (r3)

   criminal-record(Name, CriminalCase) :- citizen-of(Name, Country),
                  criminal(Name, CriminalCase, Country).           (r4)

ICs:       :-  lawful-resident(Name),
               deported(Name, Date, Reason)                        (ic1)

EDB:
   registered-alien(John, Greek, Single)

T1:  modify John's Marriage status from Single to Married in
     registered-alien.
```

First *Algorithm 3.1* can be used to transform this more complex database as shown in *Example 3.2.* The integrity constraint ic1 is transformed into two new constraints

1c1-1 and ic1-2 as illustrated below.

```
IC:   :- registered-alien(Name, Nationality, Married-status),
         not(criminal-record(Name, CriminalCase)),
         deported(Name, Date, Reason).                    (ic1-1)
      :- citizen-of(Name, Canada),
         deported(Name, Date, Reason).                    (ic1-2)
```

Let us analyze predicate-ic connections in ic1-1 and ic1-2. For ic1-1, *Name* is ic-relevant because it is shared by registered-alien, criminal-record and deported. By analyzing rule r3, we find that *Name* of criminal is ic-relevant. By analyzing r4, we find that *Name* of citizen-of and *Name* of criminal are also ic-relevant. We can analyze ic1-2 in the same way. We obtained the following predicate-ic connection information for integrity constraint ic1-1 and ic1-2.

ic1-1	registered-alien	Name
ic1-1	criminal	Name
ic1-1	citizen-of	Name
ic1-1	deported	Name
ic1-2	citizen-of	Name
ic1-2	deported	Name

We find the only ic-relevant attribute of registered-alien is *Name* and T1 only modifies John's Marriage-status, so that we can conclude that T1 will not violate any integrity constraints.

3.6 Integrity constraint checking

There are many good methods for integrity constraint checking [16, 17]. We propose an integrity constraint checking method based on the database query mechanism. Similar to the extended rule/goal graph which Martens and Bruynooghe used in their algorithm, we use the affected graph to direct the derivation of implicit modification and integrity checking against actual databases. The affected graph for database modifications are obtained by qualitative analysis of the effect on integrity constraints imposed by the modifications. Only the direct or indirect effect on integrity constraints and derived predicates which are relevant to integrity constraints are retained in the affected graph. Thus we can focus on only the part of the database which is affected by the updates and has the potential to affect integrity constraints.

Algorithm 3.3: Integrity constraint checking for the addition or deletion on an EDB predicate p using an affected graph.

1. *For every posocc-ic arc from p to an ic where the addition on p can be unified with the positive occurrences of p in the ic.* If there is more than one p-occurrence in the ic, only one occurrence can be unified at a time. Evaluate the unified formulas, if a non-false result is reached, an integrity constraint violation is caused, and the transaction should be refused and undone. Otherwise, continue the checking process.

2. *For every negocc-ic arc from p to an ic where the deletion on p can be unified with the negative occurrences of p in the ic.* If there is more than one p-occurrence in the ic, only one occurrence can be unified at a time. Evaluate the unified

formulas, if a non-false result is reached, an integrity constraint violation is caused, and the transaction should be refused and undone. Otherwise, continue the checking process.

3. *For every posocc-r arc from p to a derived predicate r where the addition on p can be unified with the positive occurrences of p in the bodies of rules which defines r.* If there is more than one p-occurrence in the r, only one occurrence can be unified at a time, and the all unified formula should be 'unioned' together. Evaluate the unified formula as a query inquiring into the ic-relevant variables to derive induced updates (addition) on r. Query the result on the database before the transaction to exclude redundant addition updates.

4. *For every posocc-r arc from p to a derived predicate r where the deletion on p can be unified with the positive occurrences of p in the bodies of rules which defines r.* If there is more than one positive occurrence in the r, only one occurrence can be unified at a time, and the all unified formula should be 'unioned' together. Evaluate the unified formula as a query inquiring into the ic-relevant variables to derive induced updates (deletion) on r. Query the result on the modified database to exclude redundant deletion updates.

5. *For every negocc-r arc from p to r*, we can treat addition or deletion on p as we will treat induced updates on other predicates in step 6.

6. *For induced updates*

   - *For every posocc-ic or negocc-ic arc from r to ics*, treat induced updates (addition) as in step 1 and deletion as in step 2.

- *For every negocc-r arc from r to a derived predicate s where the implicit addition on p can be unified with the negative occurrences of p in the bodies of rules which defines s.* If there is more than one negative occurrence in s, only one occurrence can be unified each time, and the all unified formula should be 'unioned' together. Evaluate the unified formula as a query inquiring into the ic-relevant variables to derive induced updates (deletion) on s. Query the result on the modified database to exclude redundant induced deletion updates.
- *For every negocc-r arc from r to a derived predicate s where the implicit deletion on p can be unified with the negative occurrences of p in the bodies of rules which defines s.* If there is more than one negative occurrence in the s, only one occurrence can be unified each time, and the all unified formula should be 'unioned' together. Evaluate the unified formula as a query inquiring into the ic-relevant variables to derive induced updates(addition) on s. Query the result on the database before update to exclude redundant induced addition updates.
- Repeat step 6 for induced updates.

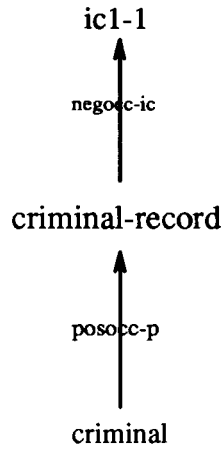


Fig. 3.3 -- The affected graph for deletion on criminal

Example 3.5: After static qualitative analysis of a database, we can obtain an affected graph for each EDB modification which requires non-trivial integrity constraint checking. Starting from *criminal* in Fig. 3.2, prune those nodes unreachable from criminal and discard useless arcs. Deletion from *criminal* may cause deletion from *criminal-record* because there is a *posocc-p* arc from *criminal* to *criminal-record*, which represent that *criminal* occurs positively in the definition of *criminal-record*. Deletion from *criminal-record* may cause an integrity constraint violation to ic1-1 because *criminal-record* occurs negatively in ic1-1. We obtained the affected graph shown in Fig. 3.3 for the deletion on *Criminal* relation in the database shown in *Example 3.1*. Interested readers can refer to [23] for details about how to qualitatively analyze the effect of a modification on the database and how to build affected graphs. If there is a update, delete criminal(John, Canada) from the Criminal relation; integrity constraints should be checked by the following procedure:

- derive the implicit modification on criminal-record, because in the affected graph there is an arc from criminal to criminal-record. Unify *criminal(John, Canada)* with criminal's occurrence in r3's body, which defines *criminal-record*, and obtain an induced update, deletion of *criminal-record(John)*. It is impossible to unify *criminal(John, Canada)* with criminal's occurrence in r4's body, which defines *criminal-record*, that is, the deletion of *criminal(John, Canada)* will not cause any implicit modification through r4.

- query *criminal-record(John)* in the new database after the update to determine if it is redundant. If it is redundant, the update is valid.

- unify criminal-record(John) with *criminal-record's* negative occurrence in ic1-1 because there is a negocc-ic arc from criminal-record to ic1-1 in the affected graph, and obtain the following integrity checking query:

```
registered-alien(John), criminal-record(John), deported(John)
```

- evaluate the above query. If the result is false, the deletion *criminal(John, Canada)* is valid, otherwise, an integrity violation has occurred and the update should be refused and undone. In fact, the result is false, which means that the update is valid.

3.7 Enforcement of ICs in non-recursive databases

Integrity constraint simplification and compilation of relevant rules, predicate-ic connection analysis, and static analysis should be performed when the database is established and revised whenever any IDB or IC modification occurs. By making use of the

above integrity checking knowledge, integrity constraints in non-recursive, function-free databases can be enforced as follows:

- Upon arrival of a modification, retrieve qualitative analysis information if it already exists or qualitatively analyze the potential effect on the database to validate the modification and obtain the affected graph for integrity checking against the actual database.
- An EDB *change* modification may be validated by predicate-ic connection analysis. If the modified attributes are not ic-relevant to any integrity constraint, the modification will not cause an integrity constraint violation.
- If a modification cannot be validated in the two steps above, the simplified integrity constraint set should be evaluated against the actual database using algorithm 3.3.

3.7.1 EDB modification

The most frequent modification to deductive database is EDB modification, which consists of addition of facts to a base relation, deletion facts from a relation, and change of facts in a relation.

Method 3: Modify a base predicate

- an EDB modification can be validated by checking the static analysis information. By qualitatively and statically analyzing the direct and indirect affect

of a modification on the intensional database and integrity constraints, we can determine which modifications will not cause integrity constraint violations.

- an EDB *change* modification may be validated by predicate-ic connection analysis. If the modified attributes are not ic-relevant to any integrity constraint, the modification will not cause integrity constraint violations. If some modified attributes are ic-relevant to some integrity constraints, the irrelevant ics should be discarded from the integrity constraint checking process and we check the affected ones at the next step.

- Use algorithm 3.3 to take care of the remaining modifications and integrity constraint pairs.

3.7.2 IDB modification

IDB modification can be validated using the EDB modification integrity constraint checking method. To simplify our method, we limit an IDB modification to be deletion or addition of one single rule; it is straightforward to extend the method to accommodate multi-rule modification. The general idea is that we treat rule addition as adding facts to head predicates of the rule and treat rule deletion as deleting facts from head predicates of the rule. Thus, we can enforce integrity constraint checking for rule modification as we do for data modification. In addition, qualitative analysis and predicate-ic connection analysis information must be revised to incorporate the modification of the IDB.

3.7.3 Integrity constraint modification

For convenience, we limit ourselves to adding or deleting one integrity constraint at a time. The deletion of an integrity constraint is always legal. Generally speaking, when an integrity constraint is added, we need to check its conformance with the current IDB. Sometimes the added ic may be trivial, that is, the current database does not have any possibility of violating the new constraint, or the new ic is contradictory to the current database. Contradictory constraints are beyond the scope of this thesis, we will not go any further with them. We only check if the new ic is satisfied in the current database by evaluating the integrity constraint's body as a query. If the result is not empty, a violation occurs and this modification should be refused. We undo the transaction and inform the user.

For the convenience of subsequent integrity constraint checking, we need to update the static analysis and predicate-ic connection analysis information about the deleted ic or the added ic. In the case of deletion of an ic, simply deleting all of the relevant information is enough. When an ic is added, we need to do qualitative analysis and variable analysis for the new integrity constraint.

3.7.4 Transactions

For a transaction consisting of several EDB modifications, IDB modifications and/or IC modifications, we can analyze each modification separately, and evaluate all of the implicit modifications and integrity constraint queries together after the transaction.

Let us use a simple example to illustrate that the integrity checking is complete and correct. For example

```
ic:        :- p, q

T:       add P' and Q'
```

For addition P', we will check

$$P' \& Q_{new}$$

For addition Q', we will check

$$P_{new} \& Q'$$

where P, Q are relations corresponding to predicate p and q, Pnew and Qnew represent the relations of p and q after modification.

$$P_{new} = P \cup P'$$
$$Q_{new} = Q \cup Q'$$

The integrity constraint after the transaction should be,

$$\begin{aligned}
&Pnew \& Qnew \\
<==> &\ Pnew \& (Q \cup Q') \\
<==> &\ (Pnew \& Q) \cup (Pnew \& Q') \\
<==> &\ (P \& Q) \cup (P' \& Q) \cup (P \& Q') \cup (P' \& Q') \\
<==> &\ (P' \& Q) \cup (P \& Q') \cup (P' \& Q') \qquad\qquad P \& Q = Empty
\end{aligned}$$

$$<==> (P'\&Q) \cup (P'\&Q') \cup (P'\&Q) \cup (P\&Q')$$

$$<==> (P'\&(Q \cup Q')) \cup ((P' \cup P)\&Q')$$

$$<==> P'\&Qnew \cup Pnew\&Q'$$

This last step illustrates that $P_{new}\&Q_{new}$ can be verified be checking $P'\&Q_{new}$ and $P_{new}\&Q'$ separately.

CHAPTER 4

IC checking in recursive databases

Recursion makes integrity constraint checking more challenging. Developing more efficient enforcement algorithms is demanding for recursive database processing. Checking of integrity constraints requires evaluation of a very special query, which amounts to existence checking (no variable is inquired). This existence property is very important in the processing of queries involving chains; efficient evaluation strategies can be developed for the evaluation of this kind of query.

We discuss only recursive integrity constraints (their clusters include recursive predicates) in this chapter because non-recursive integrity constraints can be enforced as in non-recursive databases.

4.1 Recursive query evaluation

Query independent compilation is a powerful tool in the enforcement of integrity constraints in recursive databases. First, the compilation can aid the evaluation of recursive integrity constraints and derivation of implicit modification. Second, the ic-queries can be simplified by analyzing the query forms. Combined with predicate-ic connection analysis discussed in chapter 3, a quad-state variable analysis method [11] can be used to simplify the ic-queries; and some chains can be excluded from the evaluation procedure and appropriate evaluation strategies can be planned.

Multi-way counting method[11] is the generalization of the counting method for the processing of versatile queries on complex chain recursions. It is preferable and more efficient to choose different processing strategies for different queries. A variable of a queried recursive predicate has four different states; **p**, *instantiated and inquired*; **c**, *instantiated and not inquired*; **u**, *not instantiated and inquired*; and **i**, *not instantiated and not inquired.* The query analysis method based on the four possible states of variables is called the quad-state variable binding analysis method [10, 11]. Four possible processing strategies for a recursive query exist:

- a *nonrecursive strategy*, which uses only traditional nonrecursive processing algorithms.
- a *total closure strategy*, which derives the entire recursive relation.
- a *query closure strategy*, which derives a query-related closure.
- an *existence checking strategy*, which checks the existence of answers in the

database.

The efficiency of these four different processing strategies differs largely[11]. The last processing strategy is the most efficient strategy which requires the least overhead. Using the quad-state analysis method, it is possible to evaluate ic-queries efficiently, and we can simplify an integrity constraint with chains before evaluation by analyzing the relevance of variables of a recursive predicate to integrity constraints.

Lemma [Han] If only the exit vectors (consists of all the exit variables) of an n-chain recursion are relevant (inquired or instantiated or both) to a query, the query can be processed by examining the exit rule only [11].

In an integrity checking query, only ic-relevant variables are possibly instantiated because they occurred in other base predicates. Also there is no variable inquired. Our analysis leads to the following corollary of *Lemma*[Han]for integrity checking queries.

Corollary 4.1 If only exit variables are ic-relevant in an ic-query, the n-chain can be dropped from the query form.

A chain of an AC is *irrelevant* to a query if and only if its chain variables are in the state of i, uninstantiated and not inquired[10].

Lemma [HanLu] (Discard irrelevant chains) If the query is irrelevant to chains of

the query form, the irrelevant chains can be discarded from the processing and the answer to the query remains the same[10].

A chain of an AC is *irrelevant* to a ic-query if and only if its chain variables are ic-irrelevant.

Corollary 4.2 All irrelevant chains in a AC ic-query can be discarded without affecting the integrity checking.

Example 4.1 The compiled formula of a recursion consists of three asynchronous chains as below:

$$R(X_0, Y_0, Z_0, W_0) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\sum_{k=0}^{\infty}(A^i(X_0, X_k), B^j(Y_0, Y_k), C^k(Z_0, Z_k), E(X_i, Y_j, Z_k, W_0)).$$

Consider the query:

$$R(\_, Y, Z, abc).$$

where, _ in the position of predicate attributes means we have no interest in the corresponding attributes; and $A^i(X_0, X_i) = A(X_0, X_1), A(X_1, X_2), ...A(X_{i-1}, X_i)$.

Because X is irrelevant to the query, chain A can be discarded from the query processing, thus

$$R(\_, Y_0, Z_0, abc) = \sum_{j=0}^{\infty}\sum_{k=0}^{\infty}(B^j(Y_0, Y_k), C^k(Z_0, Z_k), E(X_0, Y_j, Z_k, abc)).$$

4.2 Compile and simplify ICs in recursive databases

It is impossible to transform a recursive rule into a non-recursive rule by finite expansion except for bounded recursions. We adopt the query independent compilation method to compile a recursion before simplification of integrity constraints. Integrity constraints and rules in recursive databases can be transformed by treating recursive predicates as base predicates using a similar transformation method like *Algorithm 3.1.*

Algorithm 4.1 Simplification of integrity constraints in recursive databases.

Input Integrity constraints.

Output Ic-queries for integrity constraint evaluation and derivation of necessary induced updates.

1. Transform the integrity constraints and relevant rules using *Algorithm 3.1* by treating recursive predicates as base predicates.
2. Find ic-relevant variables of each EDB predicate by predicate-ic connection analysis using *Algorithm 3.2*, as above treating recursive predicates as base predicates.
3. Compile ic-relevant recursive predicates using query-independent compilation methods to n-chain forms or asynchronous chains.

4. Unify the compiled formula of the recursive predicate with its occurrences in transformed integrity constraints and rules defining ic-relevant derived predicates.

5. Analyze the bodies of unified integrity constraints and rules, discard irrelevant chains according to *Corollary* 4.1 & 4.2.

Example 4.2 The database shown in *Example 3.1* is extended as below. We define a citizen's descendants as lawful residents. *Descendant* is defined recursively, X's descendants are defined to be his children and their descendants.

```
IDB:  lawful-resident(X) :- registered-alien(X),
                            not(criminal-record(X)).        (r1)

      lawful-resident(X) :- citizen(X).                     (r2)

      lawful-resident(X) :- citizen(Y), descendant(Y, X).    (r3)

      descendant(X, Y) :- descendant(X, Z), children(Z, Y).  (r4)

      descendant(X, Y) :- children(X, Y).                   (r5)

IC:     :- lawful-resident(X), deported(X).                 (ic1)
```

First, the integrity constraints and IDB can be transformed using *Algorithm 3.1* by treating the recursive predicate, descendant, as base predicate. Thus, the integrity

constraint ic1 can be transformed into three integrity constraints as below.

```
IC:
  :- registered-alien(X), not(criminal-record(X)), deported(X)     (ic1-1)
  :- citizen(X), deported(X)                                       (ic1-2)
  :- citizen(Y), descendant(Y, X), deported(X)                     (ic1-3)
```

We will omit ic1-1 and ic1-2 in hte later discussion, because they are not recursive and can be enforced as in non-recursive databases.

Using *Algorithm 3.2* we can obtain all of the ic-relevant information related with ic1-3 as below.

ic1-3	citizen	X
ic1-3	deported	X
ic1-3	descendant	X, Y

Descendant can be compiled into a single chain by query-independent compilation methods:

$$descendant(X_0, X_i) = \bigcup_{i=1..\infty} children^i(X_0, X_i)$$

Because the chain variable X of *descendant* is ic-relevant, the *children* chain can not be discarded from the integrity constraint checking processing. After unifying *descendant's* compiled formula with its occurrence in ic1-3, ic1-3 becomes:

$$citizen(Y_0), \bigcup_{i=1..\infty} children^i(Y_0, Y_i), deported(Y_i) \qquad (ic1 - 3c)$$

4.3 Primitive transitive recursion

Primitive transitive recursion is a very common recursion in deductive databases. Sometimes it is called single chain recursion because it is compilable to a single chain. Well developed transitive closure processing strategies can be adopted to evaluations of this kind of recursion[2].

For simplicity, we assume the primitive transitive closure rule is defined by a recursive rule and an exit rule as illustrated by the following:

```
r(X, Z) :- chn(X, Y), r(Y, Z)
r(X, Z) :- exit(X, Z).
```

The compiled form of the above recursive rule is: [8]

$$r(X_0, Y_0) = \sum_{i=0}^{\infty} chn^i(X_0, X_i), exit(X_i, Y_0)$$

or we simply write as (in linear variable pattern)

$$r = chn^*, exit.$$

where $chn^* = chn(X_0, X_1), chn(X_1, X_2), ...$; X is a chain variable, and Y is an exit variable.

First let us consider the implicit modification caused by the addition of a tuple chn'. The implicit modification can be obtained by unifying chn' with each *chn* in the chain separately as:

$$\Delta R = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} chn^i, chn', chn^j, exit. \qquad (4.1)$$

Notice the induced updates ΔR may have redundancy.

The above formula (4.1) can be transformed to two highly instantiated asynchronous chain formula:

$$(chn^*, chn') \times (chn' chn^*), exit \qquad (4.2)$$

After compilation and simplification of integrity constraints and ic-relevant rules by *Algorithm 4.1*, if there is any primitive transitive recursion in integrity constraints or ic-relevant rules, the chain variables must be ic-relevant, otherwise the chain can be discarded from our integrity constraint checking if they are irrelevant.

When a primitive transitive recursion occurs in an integrity constraint, there are two typical situations. The integrity constraint may have the following forms:

```
(icf1)   p(X, Y), r(Y, Z)          The chain variable is ic-relevant
```

(icf2) `p(X, Y), r(Y, Z), q(Z, S)` **Both the chain variable and the exit variable are ic-relevant**

Where p and q can be replaced with a complex formula.

The above integrity constraint forms icf1 and icf2 can be transformed into two query forms after unified with the implicit modification on r:

(icq1) `p, chn*, chn' X chn', chn*, exit`

(icq2) `p, chn*, chn' X chn', chn*, exit, q`

There is a special case for icq1, when the exit predicate is the same as chain predicate, the query form icq1 is equal to:

$$p, chn^* chn'$$

because *chn', chn** is always non-empty.

The above query forms icq1 and icq2 have the following characteristics:

- no variables are inquired,
- one end of the chains is highly instantiated

Thus, the existence checking processing strategy can be adopted to checking the integrity constraints. This processing strategy is very efficient in terms of chain processing[12]. Let us sketch the general integrity checking procedure for primitive transitive recursive integrity constraints. First evaluate p, chn^*, chn' to check if the query

is false. If it is false, the update chn' is valid. Second, check $chn', chn^*, exit$ or $chn', chn^*, exit, q$, if the result is false the update is valid, otherwise an integrity constraint violation occurred, and the update should be rejected and undone.

In the same way, we can derive induced updates for a primitive transitive closure recursion. The only difference from evaluation of integrity constraints is that some variables are inquired, so that an appropriate processing strategy should be selected by quad-state binding analysis. The obtained induced updates need to be queried on the *initial* database to check if they are redundant.

For deletion on *chn*, we can derive induced deletions in the same way as for addition. Notice we have no situation to check the direct effect of a deletion on chain predicates. The obtained induced updates need to be queried on the *updated* database to check if they are redundant.

For modification on the exit predicate, there is no specific algorithm available. Nevertheless, after unification of the update with its occurrence in the compiled formula, the primitive transitive recursive query is highly instantiated at the exit end, so that efficient processing is expected.

Our discussion is based on that the chain predicates and exit predicate are base predicates. We believe that our method fits the situation that the chain predicates and exit predicate are not base predicates with slight adjustment.

Example 4.3 Consider add a tuple, children(John, Pete), to *children* relation in the database shown in *Example 3.2.* To check the simplified integrity constraint ic1-3, formula ic-13c can be transformed into a two highly instantiated asynchronous chain recursive query as below.

$$(citizen(Y_0), \bigcup_{i=1..\infty} children^i(Y_0, John), children(John, Pete)) \times$$
$$(children(John, Pete), \bigcup_{i=1..\infty} children^i(Pete, Y_i), deported(Y_i)) \qquad (ic1-3d)$$

The above query ic1-3d can be evaluated efficiently by existence checking processing strategy. For details, please refer to [12].

4.4 Complex recursion

The evaluation method for integrity constraints and implicit modification involving primitive transitive recursions discussed in section 4.3 is applicable to asynchronous chain recursion with more than one asynchronous chain. We can simplify and evaluate the affected chain as above and evaluate the remaining chains by the general transitive closure processing strategy as discussed in [10].

Counting and generalized counting methods are recognized as one of the best performing algorithms among many interesting linear recursive query processing methods. The methods achieve their efficiency by focusing on the facts relevant to a query and reducing the interaction of different chain components in the recursion by registering the relative distances(levels) from query constants. Such an isolation makes

many optimization techniques available to multi-chain recursions.

For integrity constraint evaluation of complex recursions, we need to extend the multi-way counting method[11]. As general counting methods, we need to register extra level information in the evaluation of ic-queries. Let us use a 2-chain recursion to illustrate how to evaluate integrity constraints and derive induced updates. We adopt a standard compiled formula as follows with two synchronous chains A and C:

$$R = \bigcup_{k=0..\infty} A^k E C^k \qquad (4.3)$$

Suppose an addition of tuple A' occurs, which may cause the following implicit addition to R:

$$\Delta R = \bigcup_{j=0..\infty} (\bigcup_{k=0..\infty} \Pi_{1,\$}(A^j A' A^k)) E C^m \qquad (4.4)$$

where m = j + k +1

Equation 4.4 is equivalent to

$$\Delta R = (\bigcup_{j=0..\infty} \Pi_1(A^j A') \times \bigcup_{k=0..\infty} \Pi_{\$}(A' A^k)) E C^m \qquad (4.5)$$

where m = j + k +1

Thus, we can evaluate the *A chain* with the following algorithm 4.2.

Algorithm 4.2 Evaluation of updated chains in a multi-chain recursion.

- Evaluate Closure_A1, the projection of A^*A' on the first variable and record the level information as in any counting method.
- Evaluate Closure_A2, the projection of $A'A^*$ on the last variable and record the level information.
- Join the two intermediate closures, Closure_A1 and Closure_A2, where the level number is the sum of the two level numbers in Closure_A1 and Closure_A2.

The *B chain* can be evaluated as in general chain processing. We suggest the use of the quad-state binding analysis method to select a proper processing strategy for non-updated chains.

Generally, we can evaluate the updated chain using *Algorithm 4.2* and treat other chains as in general chain processing. While evaluating an ic-query for constraints, the existence checking strategy can be adopted. While evaluating ic-queries deriving induced updates, other processing strategies may be required. Furthermore, we can always start at the updated chain because it is always highly instantiated.

4.5 Integrity checking in recursive databases

When an update requires integrity constraint checking against the actual database, we can do so in a quite similar manner as in non-recursive databases using *Algorithm 3.3.* We still need the affected graph of the modification to direct our integrity constraint checking. The only difference is that when an ic-query is required for evaluation, we should evaluate the recursive ic-queries as discussed in the last two sections. We can

use transitive closure techniques to treat single chain recursions and asynchronous recursions and extended multi-way counting method for multi-chain recursions.

4.6 Enforcement of integrity constraints in recursive databases

The integrity constraint enforcement in recursive databases is not as difficult as that in non-recursive databases except the evaluation of recursive ic-queries is more time-consuming. We have discussed the simplification of integrity constraints in section 4.2, evaluation of primitive transitive recursive ic-queries in section 4.3, evaluation of complex recursive ic-queries in section 4.4, and integrity checking in recursive databases in section 4.5. Now, we can sketch our method for the enforcement of integrity constraints in recursive databases.

Method 3: The enforcement of integrity constraints in recursive deductive databases.

- Transform, qualitatively analyze, and simplify integrity constraints and ic-relevant rules using *Algorithm 4.1*
- Validate the modification by qualitative analysis.
- Validate *change* modification by predicate-ic connection analysis.
- Derive necessary induced updates and check affected integrity constraints against the actual database.

In summary, we can simplify the enforcement of integrity constraints whose cluster involve recursive predicates in the following aspects:

- As in non-recursive databases, we can validate a large group of modifications by static qualitative analysis.
- By analyzing ic-relevant variables in a recursive predicate, some recursive integrity constraints can be simplified into nonrecursive or simpler integrity constraints.
- Using the predicate-ic connection analysis method we can further validate another group of modifications.

Example 4.4 Consider the database shown in *Example 4.2* and a transaction of the addition of *children(John, Pete)*. First the database is transformed and simplified as described in *Example 3.2* using *Algorithm 4.1*. Then, by qualitative analysis, we can determine that deletion on *children* will not affect ic1-3. The affected graph for the addition on *children* is built as illustrated in *Example 3.5* and is shown in Fig. 4.1. This affected graph shows the part of the database affected by the addition modification on *children* and relevant to integrity constraints.

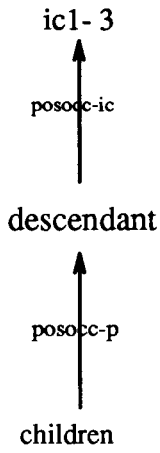


Fig. 4.1 -- The affected graph for addition on children

For addition of *children(John, Pete)*, the integrity constraint ic1-3 can be transformed into formula ic1-3d as shown in section 4.3. According to the affected graph, Fig. 4.1, we can generate an integrity constraint checking plan for the addition of a tuple, *children(John, Pete)*, to *children* relation as below.

- Evaluate the first part of query ic1-3d.

$$\bigcup_{i=1..\infty} citizen(Y_0), children^i(Y_0, John), children(John, Pete)$$

  if the query is false, then the addition is valid. Otherwise,

- evaluate

$$\bigcup_{i=1..\infty} children(John, Pete), children^i(Pete, X_i), deported(X_i)$$

  if the query is false, then the addition is valid. Otherwise, the addition caused an integrity constraint violation, and has to be redone.

CHAPTER 5

Discussion

As shown in Chapter 2, we can generally approach the enforcement of integrity constraints in three different ways, theorem-proving, query planning and compilation. There are various advantages and disadvantages associated with each approach. We combined them in our approach to achieve most of their advantages and overcome some disadvantages.

The theorem proving approach [4, 14, 6] suffers a general efficiency problem for large databases, because of the reasoning system it used is the Prolog-like tuple-at-a-time top-down derivation system. Martens' query planning approach[15] failed to concentrate their effort only on the part of the database which is relevant to integrity constraints because their reasoning system is purely bottom-up.

Yum & Henschen's compilation approach[21] can only treat Horn databases, that

is, no negation is allowed in the IDB. It is based on the speciality of Horn clause databases, whose IDB predicates can be compiled straightforwardly. This method needs non-trivial extension to be be applicable to general deductive databases with negation.

We use a compilation approach to transform the ic-relevant part of the databases like Yum & Henschen[21]. By treating negative literals and recursive predicates differently from other derived predicates and using an affected graph to direct integrity constraint checking against the actual databases as Martens and Bruynooghe[15] use rule/goal graph, we successfully extended the application domain to Horn databases augmented with negation.

To achieve the effect of focusing our effort only on the part of the database relevant to integrity constraints as backward reasoning from constraints could, we use a static qualitative analysis method to prune the set of integrity constraints. Thus, we check only those explicit or implicit modifications against the actual database which have the potential to affect integrity constraints.

The transformation method 3.1 of integrity constraints and relevant rules is the basis of our integrity enforcement method. Integrity constraints and relevant rules are expanded as deeply as possible, so that we achieved good global optimization over the derivation of implicit modification and evaluation of integrity constraints. On the other hand, predicates occurring negatively need special attention. There is no way to avoid implicit modification redundancy checking over these predicates. Redundant

deletion on A is actually false addition to $\sim$ A, which may cause us to make a wrong conclusion. This is one of the reasons why Yum & Henschen chose to deduce relevant induced updates separating from the evaluation of integrity constraints.

Using the compilation method to derive induced updates can reduce intermediate results and increase global optimization [19]. This method is quite fit for transactions which modify much data at a time due to the validation by relation-at-a-time. The predicate-ic connection analysis method is simple, but works effectively for the *change* operations.

The extended dependency graph can represent the dependency relationships among predicates. Also it can represents *imply* relationships for predicates (an *imply* relation is transitive). The extended dependency graph is designed to capture the relationships among integrity constraints and predicates. From an extended dependency graph we can find that an integrity constraint is relevant to which predicates. Thus, we can qualitatively analyze the effect that a modification may cause on integrity constraints and derived predicates based on the extended dependency graph.

The predicate-ic connection analysis method is simple and effective to validate *change* modifications. This method is found greater usage in recursive databases. It forms the basis for simplification and evaluation of recursive ic-queries. By analyzing predicate-ic connection, binding information of variables of recursive predicates can be obtained, some chains can be discarded or simplified and efficient evaluation strategies can be planned.

Qualitative analysis and overall analysis of predicate-ic connection can be done initially statically and/or incrementally. The only adaptation of this information needed is when rule and/or integrity constraint modifications happen. Even the integrity constraint checking against databases can be precompiled for each EDB's various modifications. As Yum and Henschen [21, 22], ic-queries can be precompiled using generic constants to represent updates and replaced with real updates at evaluation time.

For each modification, we build an affected graph, if it is possible to affect some integrity constraints directly or indirectly. This graph depicts which ic-relevant predicates and which integrity constraints are affected by the update. Only the ic-relevant part of the database affected by the update is present in the affected graph. Thus, the affected graph can direct integrity checking effectively with less redundant checking.

Our integrity constraint enforcement methods for recursive databases is based on the query independent compilation technique. Further improvement is possible along with the development of recursive query processing techniques. Our integrity evaluation algorithms are efficient because of the following facts:

- We collected all potential constraints for our ic-queries. The integrity constraints are expanded as deeply as possible, thus any recursive ic-queries we need to evaluate obtained maximum constraints. [12]
- Duplication of evaluation is decreased. We can always evaluate necessary chains starting from the updates, thus only the update-relevant part is evaluated.

Generally, *duplication* and *size of immediate result* may greatly affect the performance of recursive query evaluation[2].

Compared with other methods proposed by Topor and Lloyd [12], by Decker [4] and by Kowalski et al. [6]. Martens [13], Yum and Henschen [19, 20], the efficiency of our algorithm derives from the following aspects:

- Decreased intermediate results. We need only derive implicit modifications for ic-relevant negated predicates.
- Increased global optimization. Compilation techniques are used in the derivation of necessary implicit modifications and evaluation of integrity constraints.
- In addition to exploiting the assumption that the database satisfies the integrity constraints before a transaction like other methods, our methods exploit the overall hierarchical structure of the database. In a deductive database, if we modify EDB predicates, such modification may cause implicit modification on some IDB predicates which may or may not cause an integrity violation. We achieved focusing our attention only on the updated part of the database which is relevant to integrity constraints.
- Using predicate-ic connection analysis, we can validate most of the *change* modifications with no integrity checking against the large database.
- If a modification does need real integrity checking, query processing can be done in a very efficient way.

- Many integrity constraints involving chain recursions are degenerated to non-recursive ones by analyzing the relationship between chain variables and integrity constraints or are simplified to simpler chain recursions.
- We extended the application domain to a reasonable scope. Solved wholely or partially the problems related to complex transactions and recursions in integrity constraint checking.

CHAPTER 6

Summary

We further studied and developed efficient simplification algorithms and methods for the enforcement of integrity constraints in recursive deductive databases. We combined the theorem-proving method with compilation techniques in our methods. The theorem-proving method is used to prune integrity constraint checking space and compilation techniques are used to derive necessary implicit modifications and evaluate the simplified integrity constraint set against the actual database. We achieved the effect that focussed our effort only on the part of the database which is affected by the transaction as Martens[15], Yum and Henschen[21, 22], etc., and focussed only on part of the affected part which is relevant to integrity constraints. By exploiting the hierarchical structure of a deductive database we precompiled or partially precompiled integrity constraints and ic-relevant rules to simplify integrity constraint checking and validated some modification by static qualitative analysis. By analyzing predicate-ic connection and variable binding, and compiling recursive rules independently, we simplified ic-relevant queries and generated efficient checking plans. Some

asynchronous and synchronous chain recursive integrity checking relevant queries can be simplified to non-recursive or simpler chain recursions. Efficient processing algorithms were developed for integrity checking and derivation of implicit modification for asynchronous and synchronous chain recursive ic-queries.

Our integrity enforcement methods in recursive databases were based on the query-independent compilation of recursions. The recursions discussed in this thesis are confined to the function-free synchronous and asynchronous chain recursions. It is still an open research problem to compile complex recursions into regular chains and/or irregular chains. We believe that static analysis method and simplification method of the intensional database and integrity constraints are applicable to function deductive databases and complex recursive databases. It is an interesting and challenging research issue to extend our methods to the enforcement of integrity constraints in function databases and complex databases.

REFERENCES

[1] Balbin, J. and Ramamohanarnao, K., A differential Approach to Query Optimization in Recursive Deductive Databases., Technical report, Dept. of CS, Univ. of Melbourne, Australia, 1986

[2] F. Bancilhon, and R. Ramakrishnan, An amateur's Introduction to Recursive Query Processing Strategies. Proceedings of the ACM-SIGMOD '86 Conference, Washington, USA, C. Zaniolo (ed.),SIGMOD Record, Volume 15, Number 2, 1986, pp. 16-51

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman, Magic sets and other strange ways to implement logic programs. Proceedings of the 5th ACM Symposium on Principles of Database Systems, Cambrige, MA, 1986, ppl-15.

[4] H. Decker, Integrity Enforcement on Deductive Databases. Proceedings of the first International Conference on Expert Database Systems, Charleston, USA, L. Kerschberg (ed.), Univ. of South Carolina, Columbia, USA, 1986.

[5] H. Gallaire, J. Minker and J. M. Nicolas, Logic and Databases: A deductive Approach. ACM Computing Surveys, Volume 16, Number 2, 1984, pp 153-185.

[6] R. Kowalski, F. Sadri and P. Soper, Integrity Checking in Deductive Databases. Proceedings of the 13th VLDB Conference, Brighton, Great Britain, P.M. Stocker and W. Kent (eds.), Morgan Kauffmann, Los Altos, USA, 1987, pp61-69.

[7] L. J. Henschen, and S. Naqvi, On compiling queries in recursive first-order databases. Journal of ACM, 31(1), 1984, pp47-85.

[8] J. Han, Compiling General Linear Recursions by variable Connection Graph analysis, Computational Intelligence, 5(1), 1989, pp12-31.

[9] J. Han and L. Liu, Processing Multiple Linear Recursions. Proceedings of the North American Conference on logic programming, 1989, pp816-830.

[10] J. Han and W. Lu, Asynchronous Chain Recursions. IEEE transactions on Knowledge and Data Engineering, Vol. 1, No. 2, June 1989.

[11] J. Han, Multi-way Counting Method. Information Systems, Vol. 14, No. 3, 1989, pp219-229.

[12] J. Han, Constraint-Based Reasoning in deductive Databases. Proc. 7th Int'l Conf. on Data Engineering, Kobe, Japan, April 1991, pp257-265.

[13] J.L. Lassez, T. Huynh and K. MaaAloon, Simplification and Elimination of Redundant Linear Arithmetic Constraints. Logical Programming, Proceedings of the North American Conference 1989, Lusk and Overbeek(ed.), pp37-60.

[14] W. Lloyd , E.A. Sonenberg and R.W. Topor, Integrity Constraint Checking in Stratified Databases. Technical Report 86/5, Dept. of Computer Science, Univ. of Melbourne, Australia, 1986.

[15] B. Martens and M. Bruynooghe, Integrity Constraint Checking in Deductive Databases Using a Rule/Goal Graph. Proceedings of the Second International Conference on Expert Database Systems, 1988, pp567-601.

[16] William W. McCune and L.J. Henschen, Maintaining State Constraints in Relational Databases: A proof theoretic Basis, Journal of ACM, Vol.36, No.1, January 1989, pp 46-48.

[17] LJ.-M. Nicolas, Logic for Improving Integrity Checking in Relational Database. Acta Informatica, Volume 18, Number 3, 1982, pp. 227-253.

[18] W. Weber, W. Stueby and J. Karszt, Integrity Checking in Data Base Systems. Information Systems, Vol. 8, No. 2, pp 125-136, 1983.

[19] J.D. Ullman, Implementation of Logical Query Languages for Databases. ACM Transactions on Database Systems, Volume 10, Number 3, 1985, pp. 289-321.

[20] J. D. Ullman, Principles of Database and Knowledge-Base systems, Vols. 1 & 2, Computer Science Press, Rockville, MD, 1989.

[21] H. W. Yum and L. J. Henschen, Integrity checking in deductive databases: a compiled approach. Northwestern Univ. EECS Technical Report 90-12-DBM-01.

[22] H.W. Yum and L. J. Henschen, Maintaining State Constraints in Deductive Databases. Proc. of Software Eng. and Knowledge Eng., 3rd international conf., Skokie, Illinois, USA, June 1991, 187-192.

[23] L. Zhu, Enforcement of Integrity Constraints in Recursive Databases. 2nd Annual Proc. of CSS Graduate Student Paper awards competition, 1992, pp58-77.