# Further Improvement on Integrity Constraint Checking for Stratifiable Deductive Databases

Sin Yeung LEE     Tok Wang LING

Department of Information Systems and Computer Science
Lower Kent Ridge, Singapore 119260, Singapore.
email : {jlee, lingtw}@iscs.nus.sg

**Abstract**

Integrity constraint checking for stratifiable deductive databases has been studied by many authors. However, most of these methods may perform unnecessary checking if the update is irrelevant to the constraints. [Lee94] proposed a set called relevant set which can be incorporated in these works to reduce unnecessary checking. [Lee94] adopts a top-down approach and makes use of constants and evaluable functions in the constraints and deductive rules to reduce the search space. In this paper, we further extend this idea to make use of relational predicates, instead of only constants and evaluable functions in [Lee94]. We first show that this extension is not a trivial one as extra database retrieval cost is incurred. We then present a new method to construct a pre-test which can be incorporated in most existing methods to reduce the average checking costs in terms of database accesses by a significant factor. Our method also differs from other partial checking methods as we can handle multiple updates.

## 1 Introduction

Integrity constraint checking for stratifiable deductive databases has been studied by many authors. Many of these researches [Lloy87, Bry88, Celm95] are on strict integrity checking — a test that succeeds if and only if the constraint remains satisfied. However, as shown in [Lee94], all these methods are sometimes expensive and may perform a lot of redundant checkings which can be eliminated easily. Thus this motivates another area of study — to evaluate a cheaper pre-test such that if the pre-test succeeds, then the constraint is not violated. However, if the pre-test fails, nothing can be said about the constraint, and a strict checking is still needed. [Koya87, Gupt94] are examples of this area of research. This class of method usually makes use of evaluable functions, known constants and some local information to devise the pre-test. However, most of these methods either focus on single insertion or only work for some special forms of constraints or updates. In some cases, the incorporation of the pre-test can be more expansive. Our method, however, takes both cost and probability into consideration, hence, it is likely to reduce the cost of checking significantly, while keeping the cost at a small amount. And our method can handle transactions for general constraints. It makes use of some relational predicates and evaluable functions in the deductive rules and constraints to compute a set called relevant set. This relevant set can be incorporated in most of the existing methods in two ways:

1. It detects those updates that are irrelevant to a given constraint, and thus can be ignored by the validation process.
2. It reduces the search space by early abortion of computation on irrelevant predicates during refutation processes. We will show that it improves the performance of methods such as [Kowa87, Das89] by a significant factor.

The paper is organized as follows: Section 2 first give some basic terminologies used in this paper. Section 3 briefly describes some of the existing integrity checking methods for stratifiable databases and their inadequacies. We then give a brief description on how our method uses relational predicates as well as evaluable functions to improve the efficiencies of those existing methods. Section 4 discusses two problems of using relational predicates in a relevancy pre-test for a general transaction. Section 5 first gives some required definitions and then introduces a heuristic algorithm called O(1)-heuristic to solve the problems described in Section 4. Finally, a complete algorithm on how to generate a relevant set is given. Section 6 continues to give examples on how our relevant set can be used in existing methods [Lloy87,

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

**Proceedings of the 22nd VLDB Conference**
**Mumbai (Bombay), India, 1996**

Das89] to improve their performances in at least two different ways. Section 7 concludes the paper.

## 2 Some basic terminologies

The databases we are using in this work are stratifiable deductive databases with recursive rules. There are many different methods to specify deductive rules. In our work, the rules are specified by general program clauses [Lloy87b] with evaluable functions. A general program clause with evaluable functions is of the form

$$A \leftarrow L_1 \wedge \cdots \wedge L_n$$

where $A$ is a predicate, and $L_i$'s are either predicates/evaluable functions or negation of predicates/evaluable functions.

For example, the followings is a general program clause:

$$sponsor(x,y) \leftarrow guardian(x,y) \wedge employed(x) \wedge \neg stipend(y)$$

Since we allow recursive rules with negations, we shall focus ourselves on the stratifiable database model. A database is stratified if all clauses can be partitioned into several group $G_1, \ldots, G_k$ such that

1. If an atom $A$ occurs positively in the body of a clause in $G_i$, then the definition of its predicate symbol is contained within $G_j$ with $j \leq i$.
2. If an atom $A$ occurs negatively in the body of a clause in $G_i$, then the definition of its predicate symbol is contained with $G_k$ with $k < i$.

For example, the following is not stratifiable.

$$p(X,Y) \leftarrow \neg p(X,Y)$$

A full description can be found in [Ull89].

Beside different database models, as shown in [Gupt94], the information level available results in different integrity verification methods. For example, knowing only the set of constraints, [Gupt94] discussed how constraint subsumption can be used to eliminate some unnecessary checking. In this paper, we assume that we need only the information of the integrity constraints and the deductive rules to precompute a set called relevant set, which will be described in Section 5. However, during the verification step, our method assumes that the transactions as well as all the database facts are known.

The form in which integrity constraints are written determines what type of constraints that can be expressed. For example, conjunctive queries [Chan77] cannot express constraints such as "An employee age must be between 18 to 60." In this work, constraints are expressed in the form

$$L_1 \wedge \cdots L_n \rightarrow A_1 \vee \cdots \vee A_m$$

where $L_i$'s are conjunctions of predicates, evaluable functions and/or negative formulas. $A_j$'s are either predicates or evaluable functions. Variables that appear in any $L_i$ is assumed universal quantified, while local variables which appear only in $A_j$ but not in any $L_i$ is assumed existential quantified before $A_j$. For example, the constraint

$$worker(X) \rightarrow manager(Y,X)$$

is interpreted as

$$\forall X\ worker(X) \rightarrow \exists Y\ manager(Y,X)$$

In this paper, we only deal with static constraints without aggregate functions.

Lastly, some authors such as [Gupt93] addressed integrity checking for single update only. This is quite restrictive. Our technique can handle transactions, where a transaction is a set of tuple insertions and tuple deletions. Modification can be handled if we express it as a deletion followed by an insertion.

## 3 Problem of existing methods

For integrity checking for stratifiable deductive databases, [Lloy87] proposed first based on the given update and deductive rules to compute a set of possibly inserted tuples, which are the relational/deductive tuples that may be newly generated after the update; and a set of possibly deleted tuples, which are the relational/deductive tuples that may be removed after the update. The method then evaluates the given constraint after restricting to only those possibly inserted and possibly deleted tuples. On the other hand, [Deck86] proposed to evaluate the exact inserted and deleted tuples, and applied each of them to a simplified version of the integrity constraint. [Bry88] improves on [Deck86] by pre-computing the set of relation tables, updates of which may falsify the constraint. Bry's technique then applied [Deck86] to check for only those updates if they violate the constraint. [Kowa87] used an extension of the proof procedure to construct a refutation tree using each of the updates as a top clause. If all refutations fail, then the constraint is preserved. Similarly, [Das89] proposed to construct a path from each update literal to the head of each integrity constraint. If such a path can be constructed, then the constraint is violated. [Celm93, Celm95] improved [Lloy87] on some optimizations on the constraint evaluation process.

However, since all these methods work from the update literal, they all suffer one important inadequacy: they may redundantly compute some facts, refutation paths or partially instantiated predicates, which are irrelevant to the constraint checking. Consider the following example,

**Example 3.1** In a university database, three deductive predicates $greater$ to capture if a student $S\#$ in the course $C\#$ has scored at least $Sc$ in one of year; $maxScore$ to capture the maximum score of a student $S\#$ in the course $C\#$; and $failCore$ to capture if a student $S\#$ has failed any core module in all his/her attempts, are defined as follows,

$$greater(S\#,C\#,Sc) \leftarrow exam(S\#,C\#,Yr_2,Sc_2) \wedge (Sc_2 > Sc)$$

$$maxScore(S\#,C\#,Sc) \leftarrow exam(S\#,C\#,Yr,Sc) \wedge \neg greater(S\#,C\#,Sc)$$

$$failCore(S\#) \leftarrow course(C\#,core) \wedge maxScore(S\#,C\#,Sc) \wedge (Sc < 50)$$

We have another constraint which states that any first year student who does not fail any core module must also be attached to an industrial project.

$Stud(S\#,first) \wedge \neg failCore(S\#) \rightarrow proj(S\#,J\#)$

Intuitively, a deletion of the tuple $exam(s_1,c_1,1995,60)$ should not induce any deletion on $failCore$. For even if the student $s_1$ fails some core module, it cannot be $c_1$ as the student at least scores 60 in it. This deletion also cannot make him pass the other failed module. Hence, $failCore(s_1)$ cannot be changed from true to false by the above deletion. Thus the deletion cannot violate the given constraint. On the other hand, if the deleted tuple is $exam(s_1,c_1,1995,30)$, and if this is the only exam he takes, then $s_1$ no longer fails any core module. In this case, $failCore(s_1)$ may be deleted, and thus the constraint can be violated.

However, all the existing methods [Lloy97, Celm95] fail to detect that there is a difference between the two above mentioned updates. In both cases, they need to evaluate the predicates $maxScore$ and $failCore$ only to find that the first update cannot falsify the constraint at all. □

Other researchers have observed the inefficiency of strict integrity checking and proposed that some cheaper pre-tests should be applied first. Only when the test fails will the strict checking be applied. However, most of these researches work for only some particular cases. For example, when the single tuple $proj(s_1,j_1)$ is deleted, [Koba87] and [Gupt93] proposed to check if student $s_1$ is a first year student before a full checking. However, in the case of insertion of the single tuple $stud(s_1,first)$, [Gupt93, Gupt94] proposed to check if the student $s_1$ is in the database before the insertion. It is, however, not a very useful pre-test, especially when the student number is the key of the student relation. Furthermore, in the case where the relation appears in both sides of an integrity constraint (for example, multi-value dependency), the method may fail to work. [Blak86] has a similar irrelevancy idea as in this paper to update derived relations. However, it only works for evaluable functions and relational databases without multiple-updates. It is also unclear of how these methods can be extended to handle transactions as well as deductive databases with recursive rules.

In view of this, [Lee94] has discussed how to use constants and evaluable functions to perform pre-test for stratifiable deductive databases before each integrity checking. It basically pre-computes in a top-down manner a set of partially-instantiated predicates, each with an associated condition which is conjunction of only evaluable functions. In the above example, [Lee94] makes use of the evaluable function $(Sc<50)$ and deduces that if $m_1$ is not less than 50, then the deletion of $exam(s_1,c_1,y_1,m_1)$ will not cause any deletion to the predicate $failCore$ and therefore the given constraint will not be violated. In this way, it reduces all the unnecessary computations of the changes of $greater$, $maxScore$ and $failCore$ as required by all other existing methods.

To further extend the idea in [Lee94], we use not only evaluable functions and constants in the constraint and the deductive rules, but also make use of the relational predicates as well. Consider the following example,

**Example 3.2** With reference to the same rules and constraint in Example 3.1, if we delete the tuple $exam(s_1,c_1,y_1,33)$ from the database, the constraint may be violated, and according to [Lee94], the evaluable function $(Sc<50)\{Sc/33\}$ correctly concludes that the deletion may violate the constraint in general. However, there are still some unnecessary checkings that can be detected easily. Consider if the course $c_1$ is not a core module, then intuitively we know that the above deletion cannot change the predicate $failCore$, and thus the checking of the given constraint is redundant. In other words, this redundancy can be easily detected if we perform a relevancy pre-test to check if course $c_1$ is a core, i.e., we test

$Course(c1,core)$

If it is evaluated to be false, no checking is needed. Only when it is satisfiable should we proceed a normal integrity checking procedure by making use of any of the existing methods such as [Lloy87, Kowa87, Celm95].

Similarly, if student $s_1$ is not a first year student, then the deletion shouldn't affect the constraint. In other words, we can have another relevancy pre-test:

$Stud(s_1,first)$

If it is unsatisfiable, again no checking is required. Only when it is satisfiable should we then make use of other existing methods [Celm95] to validate the constraint. □

The above example illustrates how [Lee94] can be extended. Instead of only making use of the evaluable predicate $(Sc<50)$, we can extend the method to make use of the two predicates: $Course(C\#,core)$ and $Stud(S\#,first)$ in the second rule and the constraint respectively. In other words, to verify if delete $exam(s_1,c_1,y_1,m_1)$ may violate the constraint, we can test the following pre-test:

$Course(c_1,core) \wedge Stud(s_1,first) \wedge (m_1<50)$

If it is unsatisfiable, then no further checking is required. Otherwise, we then apply other existing methods to prove the database integrity after the deletion.

Similar to [Lee94], the expected reduction of database accesses are usually quite significant. Consider if 30% of the available courses are core, then 70% of the time, we can correctly eliminate all the fruitless database accesses to recompute the predicate $maxScore$ and $failCore$ with just one database read. Furthermore, if one third of the students are first year students, then two third of the remaining time we can eliminate all the redundant database accesses. Only 10% (i.e. 30% × 1/3) of the time the deletion concerns both core module and first year students. If we further assume that the passing rate is about 80%, then only 2% (i.e. 10% × 20%) of the time when we delete an $exam$ tuple, a normal constraint evaluation is needed. 98% of the time we do not need to compute the constraint at all. This gives an average about 50 times improvement of performance. Yet the overhead cost is only two extra database accesses, which is quite insignificant as compared to the database accesses cost during constraint evaluation.

## 4 Problems in using updatable predicates

The previous example shows that the extension of [Lee94] to include relational predicates can eliminate more redundant constraint checkings algorithmatically. In summary, given a transaction *TR* and an integrity constraint *IC*, our proposed method will check for each update operation in *TR* if they are relevant to *IC*. If they are not, the update is removed from *TR* before performing the checking on *IC*. Before we discuss the algorithm to generate pre-test to test the relevancy of an update with respect to a given transaction and a given constraint, there are two new problems that is different from [Lee94]. We will discuss each of them in detail now.

### 4.1 Evaluation of the pretest

The first problem is on the evaluation of the pretest. Since the pretest may contain relational predicates which may be updated by the same transaction, hence, the evaluation can generate different results depending on whether the evaluation is done before or after the update. In certain cases, the evaluation of the pre-test should be based on the database before the update, but in some other cases, it should be done after. The following shows both situations:

1. Based on the same constraint and rules in Example 3.1, the transaction {modify $stud(s_1, second)$ to $stud(s_1, first)$, delete $exam(s_1, c_1, 1995, 63)$} can violate the constraint that "each first year student who does not fail any core module should continue in the industrial attachment." Indeed, each operation in this transaction is relevant to the constraint. To test if *delete* $exam(s_1, c_1, 1995, 63)$ is relevant, a possible pre-test is,

   $stud(s_1, first)$

   If we evaluate this pre-test using the database before the update, we get the wrong conclusion that the deletion of $exam(s_1, c_1, 1995, 63)$ is irrelevant. In this case, we must evaluate $stud(s_1, first)$ based on the database AFTER the update.
2. Consider the case when we modify the course $c_1$ from core to elective. Obviously, the constraint may be violated only when there was some students who fail $c_1$ before the update. This condition should not be evaluated after the update. It is because after $c_1$ is updated to be an elective module, the constraint will not be affected by whether there are still some students who fail $c_1$ in the updated database. In this case, the evaluation of the pre-test must be done based on the database BEFORE the update in order to draw the correct conclusion.

Hence, it is important for us to decide which database our evaluation should be based on. We shall denote $pred_{OLD}$ to indicate that *pred* is evaluated using the database before the update, and $pred_{NEW}$ to indicate that *pred* is evaluated using the database after the update throughout the paper.

For example, in Example 3.1, to decide if deletion of $exam(s_1, c_1, y_1, m_1)$ may falsify the constraint, we need to check if course $c_1$ is a core before the update and the student $s_1$ is a first year student after the update, and $m_1$ is less than 50. This can be written as,

$$stud_{NEW}(s_1, first) \wedge course_{OLD}(c_1, core) \wedge (m_1 < 50)$$

Note that the evaluable function is unaffected by the database updates.

As we have introduced the OLD and NEW subscripts to the predicates in the pre-test, we therefore need to be able to

i) compute an expression mixed with NEW and OLD subscripts, and

ii) decide for each predicate *pred* in the pre-test, which version: $pred_{OLD}$ or $pred_{NEW}$, is to be used.

The first problem on computing expression mixed with NEW and OLD subscripts can be easily solved by using differential calculus used in [Hens84, Hsu85, Ling87, Levy93]. This can be illustrated as follows,

**Example 4.1** Given a transaction {modify $stud(s_1, first)$ to $stud(s_1, second)$, delete $exam(s_1, c_1, y_1, 30)$} and the following pre-test,

$$stud_{NEW}(s_1, first) \wedge course_{OLD}(c_1, core) \wedge (m_1 < 50)$$

To evaluate it prior update, according to [Ling87], the formula can be modified to,

$$FALSE \wedge course(c_1, core) \wedge (m_1 < 50)$$ □

The second problem on deciding which denotation of each predicate with subscripts OLD and NEW can be solved by the following theorem,

**Theorem 4.1** Given a deductive rule of the form,

$$p \leftarrow a \wedge b \wedge \neg c$$

where *a*, *b* and *c* represent a relational/deductive predicates. Variables are ignored for simplicity.

Now, the relation *p* may acquire some new tuples after a transaction *TR* only when

i) there are new inserted tuples which can be unified with *a* (or *b*) such that *b* (or *a* respectively) is *true* AFTER the database is updated by *TR*, or

ii) there are deletions of tuples which can be unified with *c* such that $a \wedge b$ is *true* AFTER the database is updated by *TR*.

Similarly, the relation *p* has some old tuples being removed after a transaction *TR* only when

i) there are deleted tuples which can be unified with *a* (or *b*) such that *b* (or *a* respectively) is *true* BEFORE the database is updated by *TR*, or

ii) there are inserted tuples which can be unified with *c* such that $a \wedge b$ is *true* BEFORE the database is updated by *TR*. □

**Example 4.2** Consider Example 3.1, insertion of $exam(s_1, c_1, y_1, m_1)$ will insert new tuple into the predicate *maxScore* only when $course(c_1, core)$ is true at the updated database. On the other hand, deletion of $exam(s_1, c_1, y_1, m_1)$ may remove some tuples from *failCore* only when $course(c_1, core)$ is true at the original database before the update. □

### 4.2 Significant extra database access cost

To incorporate relational predicates into [Lee94], we have yet another problem. This is about the improvement of the performance. Contrast with only using evaluable functions, the evaluation of a relational predicate needs to access database and thus incurs an extra cost. Hence, in order to have a significant database access reduction, we are required to have

1. a significant probability that the pre-test can eliminate irrelevant updates, and
2. the cost of pre-test is not too high, as compared to the integrity checking.

Consider the following two cases where the above conditions fail,

1. Assume the pre-test needs to evaluate the predicate $course(c_1, T)$, but it does not really help to remove any irrelevant update as $course(c_1, T)$ is always satisfiable. Hence we pay off an extra database access, but cannot eliminate any unnecessary checking. In this case, our first requirement fails.
2. Consider we insert the tuple $stud(s_1, first)$ in the database, we can construct a pre-test to verify if $\neg failCore(s_1)$ is satisfiable. However, the cost of such pre-test is significantly high as compared to the entire constraint evaluation. In fact, the evaluation of $\neg failCore(s_1)$ constitutes the major costs of the constraint evaluation. In this case, the pre-test evaluation is as bad as the integrity evaluation itself, and the performance is getting worse. This is the consequence if our second requirement is not fulfilled.

From the above discussion, a relevancy pre-test may not necessarily reduce database accesses during constraint validations. Hence, we need to have a method to select the right predicates to be present in the relevancy pre-test. Before we continue to discuss our algorithm to construct such a relevancy pre-test, which is not costly to compute, but has a significant chance of eliminating irrelevant updates, we shall modify some of the basic definitions used in [Lee94] now.

## 5 Possible Falsifier and Relevant set

**Definition 5.1** An **extended literal** is a tuple either of the form $[\![p:C]\!]$ or $[\![\neg p:C]\!]$ where $p$ is an atom and is called the **associated atom**. $C$ is a conjunction of evaluable functions and partially instantiated relational/deductive predicates subscripted with OLD and NEW. It is called the **associated condition**. If the number of conjunctives in $C$ is zero, then it is replaced by TRUE. □

**Example 5.1** The followings are extended literals: $[\![course(C\#, elective): exam_{\text{NEW}}(S\#, C\#, Yr, M)]\!]$, and $[\![\neg exam(S\#, C\#, Yr, M):(Yr \neq 1995) \wedge proj_{\text{OLD}}(S\#, J\#)]\!]$ □

**Definition 5.2** A positive atom $t$ is **extended unified** with an extended literal $[\![p:C]\!]$ with respect to two given databases $DB_{\text{OLD}}$ and $DB_{\text{NEW}}$ where $DB_{\text{NEW}}$ is the database after a given transaction $TR$ on $DB_{\text{OLD}}$, if $t$ is unifiable with $p$ with a most general unifier (mgu) $\sigma$, and $C\sigma$ (subscripted with OLD and NEW) is evaluated to be *true* under the given databases $DB_{\text{OLD}}$ and $DB_{\text{NEW}}$. A negative atom $\neg t$ is extended unified with an extended literal $[\![\neg p:C]\!]$ if $t$ is unifiable with $p$ with a mgu $\sigma$, and $C\sigma$ is evaluated to be *true* under the given databases. □

**Example 5.2** The positive atom $a(1,2)$ can be extended unified with $[\![a(X,Y):(X \neq Y)]\!]$ under any database. Furthermore, it can be extended unified with $[\![a(X,Y):(X \neq Y) \wedge b_{\text{NEW}}(X,Y)]\!]$ wrt databases $DB_{\text{OLD}}$ and $DB_{\text{NEW}}$ only when the predicate $b(1,2)$ is *true* under $DB_{\text{NEW}}$. However, the positive atoms $a(1,1)$ and $b(1,2)$ cannot be extended unified with both of the above extended literals under any database. □

**Definition 5.3** An extended literal $P$ is called a **possible falsifier** with respect to a given integrity constraint $IC$ if there is a database $DB_{\text{OLD}}$ and there is a tuple $t$ such that either

1. $DB_{\text{NEW}}$ is the database after $t$ is inserted into $DB_{\text{OLD}}$, and
2. $t$ can be extended unifiable with $P$ wrt the two databases $DB_{\text{OLD}}$ and $DB_{\text{NEW}}$, and
3. $IC$ is satisfiable under $DB_{\text{OLD}}$, but is violated after insertion, i.e. under $DB_{\text{NEW}}$,

or

1. $DB_{\text{NEW}}$ is the database after $t$ is deleted from $DB_{\text{OLD}}$, and
2. $\neg t$ can be extended unifiable with $P$ wrt the two databases $DB_{\text{OLD}}$ and $DB_{\text{NEW}}$, and
3. $IC$ is satisfiable under $DB_{\text{OLD}}$, but is violated after the deletion, i.e. under $DB_{\text{NEW}}$. □

Intuitively, a possible falsifier wrt constraint $IC$ captures a set of insertions and deletions which can violate a satisfied constraint $IC$ in some database. Since modification can be viewed as deletion followed by insertion, the above definitions can be easily extended to handle modification.

**Example 5.3** Refer to the constraint in Example 3.1,

$stud(S\#, first) \wedge \neg failCore(S\#) \rightarrow proj(S\#, J\#)$

As there exists some database state which obeys the constraint, but is violated when $stud(s1, first)$ is inserted, hence $[\![stud_{\text{NEW}}(s_1, first):\text{TRUE}]\!]$ is a possible falsifier. $[\![\neg proj(s_1, j_1):\text{TRUE}]\!]$ is also a possible falsifier as deleting the project information may violate the constraint. Similarly, $[\![\neg failCore(s_1):\text{TRUE}]\!]$, $[\![\neg course(c_1, core):\text{TRUE}]\!]$ and $[\![\neg proj(s_1, j_1): stud(s_1, first)]\!]$ are all possible falsifiers. On the other hand, $[\![\neg course(c_1, elective):\text{TRUE}]\!]$ is not a possible falsifier as deleting an elective course cannot falsify the constraint. Similarly, modification of the course $c_1$ from elective to other type cannot falsify the constraint. $[\![\neg proj(s_1, j_1): stud(s_1, second)]\!]$ is also not a possible falsifier as deleting a second year student's project should not affect the constraint. □

**Definition 5.4** An extended literal $[\![P_1:C_1]\!]$ (or $[\![\neg P_1:C_1]\!]$) is said to **subsume** another extended literal $[\![P_2:C_2]\!]$ (or $[\![\neg P_2:C_2]\!]$ respectively) if there is a mgu $\sigma$ such that

i) $P_2 = P_1\sigma$,

ii) $C_2 \rightarrow C_1\sigma$ under any database. □

**Example 5.4** $[\![ Course(C\#,T):\text{TRUE} ]\!]$ subsumes both $[\![ Course(C\#,core):\text{TRUE} ]\!]$ and $[\![ Course(C\#,T):(T \neq core) ]\!]$. However, the latter two do not subsume each other. □

**Theorem 5.1** Given any integrity constraint *IC*, if the possible falsifier *P* wrt *IC* subsumes another possible falsifier *Q* wrt *IC*, then for any update *U* which violates *IC*, *U* is extended unifiable with *Q* implies that it is also extended unifiable with *P*. □

**Definition 5.5** A **relevant set** with respect to an integrity constraint is a collection of possible falsifiers such that any possible update which violates the constraint can be extended unified with some possible falsifiers in the set. □

Note that, however, not all updates which extended unify with some of the possible falsifiers in the relevant set will definitely violate the constraint. Hence, since the relevant set does not describe exactly only all the updates which will violate the constraint, but just updates which may violate the constraint, the relevant set, by definition, is not an unique set for a given constraint *IC*. In particular, we can always replace a possible falsifier $P1$ in a relevant set by another possible falsifier $P2$ if $P2$ subsumes $P1$.

Following the discussion in section 4.2, we know that if the database access cost to verify if $P1$ can extended unified with a tuple $t$ (or $\neg t$) is high, we can remove some conjunctive predicates from $P1$ to form $P2$, which clearly subsumes $P1$, and has a lower database access cost. But in return, removing some of the conjunctives in the associated condition also means that the condition is less restrictive and the chances is increased to be satisfiable. According to the discussion in section 4.2, if the possible falsifier is almost always satisfiable, then it is less useful to eliminate any unnecessary informations and checkings. To balance these two conflicting objectives, we propose a heuristic called **O(1)-heuristic**, which is to drop any conjunctive predicates except those predicates that can be verified within one database read operation and has a significant chance to be unsatisfiable.

The first type of predicates to be kept are those evaluable functions which used only variables found in the associated atom of a possible falsifier as shown by [Lee94]. For example, given a possible falsifier of $[\![ exam(S\#,C\#,Yr,Sc):(Yr \neq Yr_2) \wedge (Sc < 50) ]\!]$, we will not keep the first evaluable function $(Yr \neq Yr_2)$ as it uses a variable $Yr_2$ which is not found in the associated atom $exam(S\#,C\#,Yr,Sc)$. On the other hand, we keep the second evaluable function $(Sc < 50)$ as it satisfies our requirement. For this class of evaluable functions, their evaluations take no database read. Furthermore, after the associated atom is unified with a tuple, the evaluable function will be fully instantiated, and it is quite unlikely that the instantiated function is always satisfiable.

The second type of predicates are those relational predicates which obey the following two criteria:

1. The first criterion is that all those variables which appear in the primary key attributes' positions must also be found in the associated atom of the possible falsifier. Satisfying this criterion, all the primary key attributes in the predicate will be bounded by constants during the unifying step. Since after the value of the key of a relational predicate is known, it only takes one database read to retrieve the entire tuple. Therefore, the satisfiability of the relational predicate can be evaluated within one database read. For example, given the possible falsifier *P*: $[\![ failCore(S\#):stud(S\#,first) \wedge exam(S\#,C\#,Yr,Sc) ]\!]$ the predicate $stud(S\#,first)$ satisfies this criterion as the variable $S\#$, which appears in the primary key position in relation *stud*, also appears in the atom of *P*. However, for predicate $exam(S\#,C\#,Yr,Sc)$, the variables $C\#$ and $Yr$, which appear in the primary key position of relation *exam*, do not appear in the atom of *P*, hence this predicate does not satisfy the first criterion.
2. The second criterion requires that there is at least some conditions to bind the value of some of the non primary attributes in the predicate. The binding can be just some constants or instantiation of variables used in the associated atoms. Having some restrictions on some of the non primary attributes, it will provide a significant chance that the associated condition of the possible falsifier to be unsatisfiable. Thus the chance to eliminate some irrelevant updates is better. For example, given the possible falsifier: $[\![ failCore(S\#):stud(S\#,first) ]\!]$ the predicate $stud(S\#,first)$ satisfies the second criterion since the constant *first* binds at least one of the non-primary key attributes. On the other hand, the predicate $stud(S\#,Yr)$ does not satisfy this criterion as none of the non-primary key attribute is bounded with any constant.

Finally, we will not use any deductive predicate. The evaluation of deductive predicates usually requires more than one database read operation, especially if the predicate is a recursive one.

We can summarize the above discussion by the following algorithm. The algorithm modifies a given possible falsifier *P* by reducing the condition part of *P* so that each conjunct satisfies the O(1)-heuristic.

**Algorithm 5.1**

```
O1_heuristic(Possible_Falsifier& P)
 begin
 for each conjunct C in the condition of P
   begin
   case 1: if IC is an evaluable function and
      if (C uses some variables not found in the associated
      atom of P)
         remove C from the condition part of P;
   case 2: if (C is a relation predicate OR
      C is a negation of a relation predicate)
```

if some of the variables which appear in some primary key positions in C, but does not appear in the associated atom of P,
OR
if none of the variables of non-primary key attributes is binded with constants, then
remove C from the condition part of P;

case 3: if C is a deductive predicate then
remove C from the condition part of P;
end;
end;

**Example 5.5** Consider the following possible falsifier $P$,

$[\![ exam(S\#,C\#,Yr,Sc): stud_{\text{NEW}}(S\#,first) \wedge proj_{\text{OLD}}(S\#,J\#) \wedge course_{\text{OLD}}(C\#,core) \wedge exam_{\text{NEW}}(S\#,C\#,Yr,Sc_2) ]\!]$

This possible falsifier can be reduced by Algorithm 5.1 as followed:

1. The conjunct $stud_{\text{NEW}}(S\#,first)$ corresponds to the second case in the Algorithm 5.1. Since all of its primary key attributes ($S\#$) appear in the associated atom of $P$, and some of its non-primary key attributes (i.e. year-level) bind with a constant (i.e. *first*), hence, we will not remove it from $P$, and keep this conjunct.
2. For the conjunct $proj_{\text{OLD}}(S\#,J\#)$, the primary key's attributes are $S\#$ and $J\#$. However, $J\#$ does not appear in the associated atom of $P$, hence, the conjunct is removed.
3. The conjunct $course_{\text{OLD}}(C\#,core)$ is not removed. The reason is the same as the first conjunct $stud_{\text{NEW}}(S\#,first)$.
4. Finally, the conjunct $exam_{\text{NEW}}(S\#,C\#,Yr,Sc_2)$ is removed as its non-primary key attribute $Sc_2$ is not bounded by a constant.

Now, the modified $P$ by Algorithm 5.1 is

$[\![ exam(S\#,C\#,Yr,Sc): stud_{\text{NEW}}(S\#,first) \wedge course_{\text{OLD}}(C\#,core) ]\!]$ □

### 5.1 Computation of relevant set

The computation of a relevant set with respect to a given constraint is essentially top-down. As we do not require the set to be fully instantiated, we do not need to access the database to generate the set and the process can always terminate. Moreover, this computation needs only to be done once for each constraint and is independent of any transaction until some database rules or constraints are changed. Hence, it can be classified as compiled approach. The following algorithm describes how we construct a relevant set with respect to a given constraint,

**Algorithm 5.2** Given a stratifiable deductive database *DB* and a constraint *IC*, we construct a relevant set as follows:

1. Temporary add the deductive rule
   $violated \leftarrow \neg IC$
   into *DB* where the predicate *violated* is not an existing predicate in *DB*. Convert $\neg IC$ to a closed first-order formula if necessary.
2. Initialize *S* to contain only a single possible falsifier $[\![ violated: \text{TRUE} ]\!]$
3. If $[\![ p(y_1,\ldots,y_n):C ]\!]$ is in *S*, and if there is a deductive rule in *DB*,
   $p(x_1,\ldots,x_n) \leftarrow W(x_1,\ldots,x_n)$
   such that $p(y_1,\ldots,y_n)$ can unify with $p(x_1,\ldots,x_n)$ with a mgu $\theta$, then
   i) rename the local variables (i.e. not $x_1,\ldots,x_n$) in $W$ if necessary so that they do not share the same name with variables in $C$ and $p(y_1,\ldots,y_n)$
   ii) for each positive literal $q(z_1,\ldots,z_m)$ (or negative literal $\neg q(z_1,\ldots,z_m)$) in $W$, construct the possible falsifier $[\![ q(z_1,\ldots,z_m)\theta: C \wedge (W'\theta) ]\!]$ (or $[\![ \neg q(z_1,\ldots,z_m)\theta: C \wedge (W'\theta) ]\!]$ respectively) where $W'$ is the same as $W$ except that the predicate $q(z_1,\ldots,z_m)$ (or $\neg q(z_1,\ldots,z_m)$ respectively) is removed, and every relational predicate in $W'$ is labeled as NEW. Apply Algorithm 5.1 on this possible falsifier and generate a new possible falsifier $[\![ q(z_1,\ldots,z_m)\theta: C' ]\!]$. (or $[\![ \neg q(z_1,\ldots,z_m)\theta: C' ]\!]$ respectively) Include it into *S* if it is not just a renaming of any existing element in *S*.
4. If $[\![ \neg p(y_1,\ldots,y_n):C ]\!]$ is in *S*, and if there is a deductive rule in *DB*,
   $p(x_1,\ldots,x_n) \leftarrow W(x_1,\ldots,x_n)$
   such that $p(y_1,\ldots,y_n)$ can unify with $p(x_1,\ldots,x_n)$ with a mgu $\theta$, then
   i) Rename the variable in $W$ if necessary similar to previous case.
   ii) for each positive literal $q(z_1,\ldots,z_m)$ (or $\neg q(z_1,\ldots,{}_m)$) in the body of $W$, construct the possible falsifier $[\![ \neg q(z_1,\ldots,z_m)\theta: C \wedge (W'\theta) ]\!]$ (or $[\![ q(z_1,\ldots,z_m)\theta: C \wedge (W'\theta) ]\!]$ respectively) where $W'$ is the same as $W$ except that the predicate $\neg q(z_1,\ldots,z_m)$ (or $\neg q(z_1,\ldots,z_m)$ respectively) is removed, and every relational predicate in $W'$ is labeled as OLD. Apply Algorithm 5.1 on this possible falsifier and generate a new possible falsifier $[\![ \neg q(z_1,\ldots,z_m)\theta: C' ]\!]$ (or $[\![ \neg q(z_1,\ldots,z_m)\theta: C' ]\!]$ respectively). Include it into *S* if it is not just a renaming of any existing element in *S*.
5. [Optional simplification step] For any pair of possible falsifiers $P1$ and $P2$ in *S*, if $P1$ subsumes $P2$, then remove $P2$ from *S*.
6. Repeat step 3 until no more new possible falsifier is included in *S*.
7. Remove the possible falsifier $[\![ violated: \text{TRUE} ]\!]$ from S and remove the temporary deductive rule,
   $violated \leftarrow \neg IC$
   from the deductive database.
8. Return *S* as a relevant set of *IC*. □

**Theorem 5.2** Algorithm 5.2 will terminate and correctly generate a relevant set. In other words, any update which is not extended unifiable with any of the elements of the set generated from Algorithm 5.2 cannot falsify the constraint. □

**Example 5.6** With reference to Example 3.1, we have three deductive rules R1, R2 and R3:

$$\mathit{greater}(S\#,C\#,Sc) \leftarrow \mathit{exam}(S\#,C\#,Yr_2,Sc_2) \wedge (Sc_2 > Sc) \quad \text{(R1)}$$

$$\mathit{maxScore}(S\#,C\#,Sc) \leftarrow \mathit{exam}(S\#,C\#,Yr,Sc) \wedge \neg \mathit{greater}(S\#,C\#,Sc) \quad \text{(R2)}$$

$$\mathit{failCore}(S\#) \leftarrow \mathit{course}(C\#,core) \wedge \mathit{maxScore}(S\#,C\#,Sc) \wedge (Sc < 50) \quad \text{(R3)}$$

and the constraint that a first year student who does not fail any core module must do some project:

$$\mathit{stud}(S\#,first) \wedge \neg \mathit{failCore}(S\#) \rightarrow \mathit{proj}(S\#,J\#)$$

the relevant set wrt to the given constraint can be computed as follows,

1. According to the first step of Algorithm 5.1, we add the following deductive rule,
   $$\mathit{violated} \leftarrow \mathit{stud}(S\#,first) \wedge \neg \mathit{failCore}(S\#) \wedge \neg \mathit{proj}(S\#,J\#) \quad \text{(R0)}$$
2. According to the second step of Algorithm 5.1, we initialize the relevant set $S$ to contain only $[\![ \mathit{violated} : \text{TRUE} ]\!]$.
3. On applying Algorithm 5.1 in R0, we instantiated *violated* with $p(x_1, \ldots, x_n)$ in step 3, and the first predicate of $W(x_1, \ldots, x_n)$ is $\mathit{stud}(S\#,first)$. However, every predicate in $W'$ is a deductive predicate and is removed by O(1)-heuristic, We generate $[\![ \mathit{stud}(S\#,first) : \text{TRUE} ]\!]$. Similarly, two other possible falsifiers are generated in this step: $[\![ \neg \mathit{failCore}(S\#) : \mathit{stud}_{\text{NEW}}(S\#,first) ]\!]$ and $[\![ \neg \mathit{proj}(S\#,J\#) : \mathit{stud}_{\text{NEW}}(S\#,first) ]\!]$.
4. For the first possible falsifier $[\![ \mathit{stud}(S\#,first) : \text{TRUE} ]\!]$, $\mathit{stud}(S\#,first)$ cannot be unified with any head predicate in any deductive rule. For the second possible falsifier, $[\![ \neg \mathit{failCore}(S\#) : \mathit{stud}_{\text{NEW}}(S\#,first) ]\!]$, we can make use of R3 to expand *failCore* and generate two more possible falsifiers: $[\![ \neg \mathit{course}(C\#,core) : \text{TRUE} ]\!]$, and $[\![ \neg \mathit{maxScore}(S\#,C\#,Sc) : \mathit{stud}_{\text{NEW}}(S\#,first) \wedge \mathit{course}_{\text{OLD}}(C\#,core) \wedge (Sc < 50) ]\!]$.
   Note that the condition "$\mathit{stud}_{\text{NEW}}(S\#,first)$" is dropped in the first possible falsifier $[\![ \neg \mathit{course}(C\#,core) : \text{TRUE} ]\!]$ by our algorithm as $S\#$ no longer appears in the associated atom of the new possible falsifier.
5. We expand further on *maxScore* by R2, and we generate $[\![ \neg \mathit{exam}(S\#,C\#,Yr,Sc) : \mathit{stud}_{\text{NEW}}(S\#,first) \wedge \mathit{course}_{\text{OLD}}(C\#,core) \wedge (Sc < 50) ]\!]$, and $[\![ \mathit{greater}(S\#,C\#,Sc) : \mathit{stud}_{\text{NEW}}(S\#,first) \wedge \mathit{course}_{\text{OLD}}(C\#,core) \wedge (Sc < 50) ]\!]$.
6. Finally, using R1, we generate the followings, $[\![ \mathit{exam}(S\#,C\#,Yr_2,Sc_2) : \mathit{stud}_{\text{NEW}}(S\#,first) \wedge \mathit{course}_{\text{OLD}}(C\#,core) ]\!]$.
   Note that the evaluable function $(Sc < 50)$ is removed in the last possible falsifier as variable $Sc$ is not in the associated atom.

Finally, Algorithm 5.2 computes the relevant set to have the following elements:

1. $[\![ \mathit{stud}(S\#,first) : \text{TRUE} ]\!]$,
2. $[\![ \neg \mathit{course}(C\#,core) : \text{TRUE} ]\!]$,
3. $[\![ \neg \mathit{proj}(S\#,J\#) : \mathit{stud}_{\text{NEW}}(S\#,first) ]\!]$,
4. $[\![ \neg \mathit{exam}(S\#,C\#,Yr,Sc) : \mathit{stud}_{\text{NEW}}(S\#,first) \wedge \mathit{course}_{\text{OLD}}(C\#,core) \wedge (Sc < 50) ]\!]$,
5. $[\![ \mathit{exam}(S\#,C\#,Yr,Sc_2) : \mathit{stud}_{\text{NEW}}(S\#,first) \wedge \mathit{course}_{\text{OLD}}(C\#,core) ]\!]$,
6. $[\![ \neg \mathit{failCore}(S\#) : \mathit{stud}_{\text{NEW}}(S\#,first) ]\!]$,
7. $[\![ \neg \mathit{maxScore}(S\#,C\#,Sc) : \mathit{stud}_{\text{NEW}}(S\#,first) \wedge \mathit{course}_{\text{OLD}}(C\#,core) \wedge (Sc < 50) ]\!]$.
8. $[\![ \mathit{greater}(S\#,C\#,Sc) : \mathit{stud}_{\text{NEW}}(S\#,first) \wedge \mathit{course}_{\text{OLD}}(C\#,core) \wedge (Sc < 50) ]\!]$.

Hence, with the first five elements in this relevant set, the only updates that may violate the constraint are

1. an insertion of a first year student, or
2. a deletion of a core module, or
3. a deletion of a project tuple of a first year student in the updated database, or
4. a deletion of an exam tuple which the student is a first year students and the course was a core module and the mark is less than 50, or
5. an insertion of an exam tuple which the student is a first year students and the course was a core module.

Note that the last three items in the relevant set do not concern updates on relation table. They, however, will still be useful in reducing search space during refutation. We will discuss it in the next section. □

## 6 Application of the relevant set

In this section, we shall discuss how relevant sets can be used in various steps of most existing constraint validation processes [Lloy87, Das89, Celm95]. Furthermore, we will show that the additional overhead cost is negligible as compared with the expected saving we can achieve.

### 6.1 Eliminate irrelevant updates

A relevant set provides the information on whether a partially instantiated update can violate a given constraint. Hence, a direct application of this method is to eliminate irrelevant updates in a transaction, and so the integrity method needs only check a transaction of smaller size. In some methods such as [Lloy87], a reduction of the transaction size means a reduction of the checking cost. We shall now show how our method can improve [Lloy87].

[Lloy87] computes the possibly-inserted and possibly-deleted partially instantiated instances iteratively without first consulting the database. To use the relevant set in [Lloy87], we can first check if the update literal can be extended unified with any of the possible falsifier of the

relevant set. If none is found, then no further computation is necessary. Otherwise, we shall apply [Lloy87] to verify the constraint.

**Example 6.1** Given the rules,

$p(X,Y) \leftarrow a(X,Z) \wedge b(Z,Y)$
$q(X,Y) \leftarrow p(X,Z) \wedge c(Z,Y)$

and a constraint

$p(X,X) \rightarrow q(1,X)$

A relevant set for this constraint is

$\{[\![p(X,X):\text{TRUE}]\!], [\![\neg q(1,X):\text{TRUE}]\!],$
$[\![a(X,Z):b_{\text{NEW}}(Z,X)]\!], [\![b(Z,X):\text{TRUE}]\!],$
$[\![\neg p(1,X):\text{TRUE}]\!], [\![\neg c(Z,Y):\text{TRUE}]\!],$
$[\![\neg a(1,Z):\text{TRUE}]\!], [\![\neg b(Z,Y):a_{\text{OLD}}(1,Z)]\!]\}$

Since $\neg a(2,2)$ cannot extended unify with any element in the relevant set, hence no checking is required for this constraint for deleting $a(2,2)$. Furthermore, if $a(1,5)$ is not in the database before the update, then the deletion of $b(5,Y)$ for any $Y$ will also be impossible to falsify the constraint. It is because that $\neg b(5,Y)$ fails to extended unify with $[\![\neg b(Z,Y):a_{\text{OLD}}(1,Z)]\!]$ as the condition $a_{\text{OLD}}(1,Z)\{Z/5\}$ fails. Note that without the relevant set, [Lloy87] needs to redundantly compute both the positive and negative sets only to discover that no checking is necessary. The same problem appears in later works [Bry88, Celm93, Celm95]. For example, concerning the deletion of $b(5,Y)$ when $a(1,5)$ is not in the database before the update, [Celm95] will lose the constant 5 during the computation of the changes to the database. Hence, lots of unnecessary database accesses are needed in this case to discover that the constraint is not falsified. For our method, however, such unnecessary checking is detected just by a few database reads.

When more evaluable functions and constants appear in the constraint, our method has even better performance. For example, given the same deductive rules in Example 6.1 and the constraint

$p(X,Y) \wedge (X>5) \rightarrow c(X,Y)$

our method can conclude that inserting $a(1,2)$ is irrelevant without any database access due to the only possible falsifier using relation $a$ is $[\![a(X,Z):(X>5)]\!]$. However, [Lloy87] still needs to evaluate the entire constraint only to find that the constraint can never be violated. This is very costly. Again this problem also exists in [Deck86, Bry88, Celm95]. □

### 6.2 Use of relevant sets during refutation

While our method can be applied to eliminate irrelevant updates in a transaction, it can also be used during the constraint evaluation process. In particular, given a bottom-up constraint checking method, we can inspect if the intermediate computed instances are unifiable with any possible falsifier in the relevant set. If it is not, then further computation from that instance is irrelevant to the validation. For example, during the process when [Deck86] computes exact induced updates, we can check if each computed update is relevant to the constraint. If it is not, then further computations based on it can be eliminated. This idea can be applied to other bottom-up constraint evaluation process such as in [Das89]. We shall illustrate how the relevant set can improve [Das89].

[Das89] uses a refutation procedure to construct a refutation path to reach to the given constraint. If such path exists, then the constraint is violated. Now, given the following deductive database with the rules,

$p(X,Y) \leftarrow q(X,Y)$ (R1)
$p(X,Y) \leftarrow q(X,U) \wedge p(U,V) \wedge q(V,Y)$ (R2)
$q(X,Y) \leftarrow a(X,U) \wedge b(U,Y)$ (R3)
$r(X,Y) \leftarrow c(X,Y) \wedge \neg q(X,Y)$ (R4)

and the facts

$a(2,0), a(3,0), \ldots, a(99,0),$
$c(2,1), c(3,1), \ldots, c(99,1), d(1)$

To verify the constraint,

$d(X) \rightarrow r(1,X)$

after the transaction {insert $b(0,1)$}, [Das89] will try to construct a path which leads to the constraint as shown in *Figure 1*.

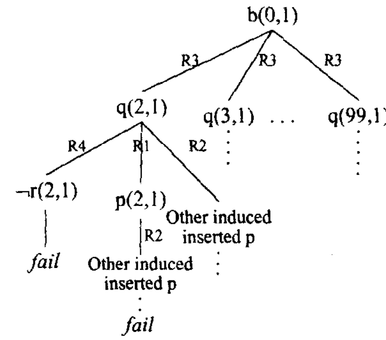


*Figure 1*

The search space involves many unnecessary computations, such as $\neg r(2,1)$ and all the induced insertions of predicate $p$. However, knowing that the relevant set with respect to this constraint is

$\{[\![d(X):\text{TRUE}]\!], [\![\neg r(1,X):\text{TRUE}]\!],$
$[\![\neg c(1,X):\text{TRUE}]\!], [\![q(1,X):c_{\text{OLD}}(1,X)]\!],$
$[\![a(1,X):\text{TRUE}]\!], [\![b(X,Y):a_{\text{NEW}}(1,X)]\!]\}$

we can deduce that insertion of $b(0,1)$ alone cannot violate the constraint. The inserted tuple fails to extended unify with the possible falsifier $[\![b(X,Y):a_{\text{NEW}}(1,X)]\!]$. The condition $a_{\text{NEW}}(1,X)\{X/0\}$ fails in the new database as the tuple $a(1,0)$ does not exist in the new database. This gives a huge reduction of search space as shown in *Figure 2*.

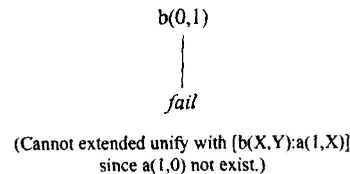


*Figure 2*

Even if $a(1,0)$ is true in the updated database, there is still a considerable saving according to our method. Consider the transaction {insert $b(0,1)$, insert $a(1,0)$}, [Das89] will verify $b(0,1)$ according to *Figure 3*.

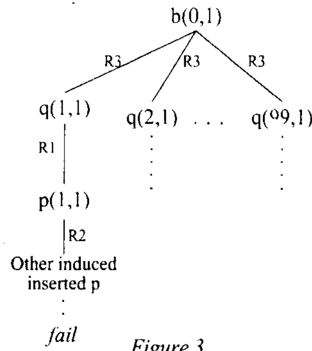


*Figure 3*

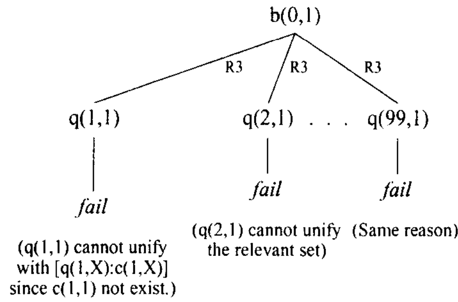


*Figure 4*

However, by incorporating the relevant set, our method aborts any further computation from the branch of $q(X,1)$ whenever $X$ is not 1. As shown in *Figure 4*, the search space is now much reduced. This alone already gives us at least a 99 times reduction as compared to [Das89]. Furthermore, as $c(1,1)$ does not exist in the database, we can stop any further computation from $q(1,1)$. This gives us another large reduction of the search space. Without using a relevant set, [Das89] needs to explore much larger search space to validate the constraint. Hence, using the relevant set in this case proves to gain a great saving, especially when the relations $p$ and $q$ are considerably large.

Note that methods which clearly separate constraint simplification from constraint evaluation such as [Deck86, Bry88] cannot eliminate this type of irrelevant evaluation as no database facts are known to these methods during the simplification step.

### 6.3 Overall performance of relevant set method

We now further discuss the extra costs incurred by our method and show that these extra costs are insignificant. There are three extra costs incurred:

1. Computation of a relevant set,
2. extra cost of extended unification as compared with conventional unification,
3. extra cost to access the relevant set before each step of the refutation process.

The computation of the relevant set needs only to be done once for each constraint until some deductive rules are changed. Furthermore, this one-time computation does not need to access the database and is done in the main memory. Hence, the cost is insignificant for normal database applications.

As compared with the conventional unification, our extended unification requires some database accesses to check for its associated condition. However, as the condition is only conjunctions of only those predicates which can be computed in at most one database read access, the overhead cost remains a small constant. and is insignificant compare to the possible reduction of the number of database accesses by our method as shown in the previous two examples.

Lastly, as the size of a relevant set is usually small, it can be stored in the main memory and hence, searching possible falsifiers in the relevant set to test for extended unification does not require any extra database accesses.

The savings gained from the relevant set can vary a lot. While the relevant set can eliminate the entire integrity checking process which other methods fail to do, or reduce the search space during refutation by a significant proportion, it is also possible that extra overhead worsen the performance without eliminating any informations. Recall that unnecessary checkings are detected based on those constants, evaluable functions, relational predicate evaluation as well as those relational symbol symbols which are relevant to the constraints. Hence, if there is no constant and evaluable function, nor any useful relational predicate in the constraint, and all the deductive rules, as well as all the deductive predicate symbols need to be re-calculated in order to evaluate the constraint (i.e. no irrelevant predicate), then there will be no saving gained from using the relevant set. However, when this situation happens, it can be easily identified by the following,

i) The associated atom for each extended literal is not instantiated with any constant, and
ii) the associated condition for each extended literal is TRUE, and
iii) every deductive relation is included in the set.

In this case, such relevant set cannot reduce the search space. However, we can implement an extra flag to disable the relevancy checking if necessary. So even in this worst case, our method, as compared as the existing methods, only requires an additional pre-computation of the relevant set once for each integrity constraint, and to check the flag in O(1) time for each transaction. The extra cost is insignificant as no database access is involved. With this flexibility, our method can give much better overall performance than other existing methods most of the time.

## 7 Conclusion

We have presented the extension of relevant set method in [Lee94] and how it can be used to incorporated into other existing methods to reduce their average database accesses during constraint validation. This is achieved by making use of existing evaluable functions, constants and certain relational predicates in the deductive rules and constraints to detect irrelevant updates and prune off computations of many deductive tuples which cannot falsify the constraint. However, as relational predicates

are run-time updatable and their evaluation requires database accesses, this brings up the timing of validation as well as selection of predicates problem. We therefore presented a heuristic called O(1)-heuristic to solve the selection problem. For further research, we can revise the O(1)-heuristic so that more relational and deductive predicates can be used to eliminate more unnecessary checkings.

## REFERENCES


[Blak86] J.A.Blakeley, M.Coburn and P.A.Larson, "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates", *Proceedings of the 12th VLDB*, Kyoto, 1986, 457-466.

[Bry88] F. Bry, H. Decker and R. Manthey, "A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases", *Proceedings of Extending Database Technology*, Venice, 1988, 488-505.

[Celm93] M. Celma, J.C. Casamayor and H. Decker, "Improving Integrity Checking by Compiling Derivation Paths", *Proceedings of the Fourth Australian Database Conference*, 15-160, Australia, 1993.

[Celm95] M. Celma, H.Decker, "Integrity Checking in Deductive Databases. The Ultimate Method?", *Proceedings of the 5th Australiasian Database Conference*, 136-146, Australia, 1995.

[Chan77] A.K. Chandra and P.M.Merlin, "Optimal implementation of conjunctive queries in relational databases", *Proc Ninth Annual ACM Symposium on the Theory of Computing*, pp 77-90, 1977.

[Das89] S.K. Das and M.H. Williams, "A path finding method for constraint checking in deductive databases", *Data and Knowledge Engineering 3* (1989) 223-244.

[Deck86] H. Decker, "Integrity enforcements on deductive databases", in L.Kerschberg (ed.) *Proceedings of the 1st International Conference on Expert Database Systems,* Charleston, South Carolina (Apr 1986) 271-285.

[Gupt93] A.Gupta and J.Widom, "Local verification of global integrity constraints in distributed databases", *ACM SIGMOD* (1993) 49-58.

[Gupt94] A.Gupta, Y.Sagiv, J.D.Ullman and J.Widom, "Constraint Checking with Partial Information", *PODS* 1994, Minneapolis, 45-55.

[Hens84] L.J. Henschen, W.W. McCune and S.A. Naqui, "Compiling constraint-checking programs from first-order formulas", *Advances in Data Base Theory*, Vol. 2, Plenum Press, New York, 1984, 145-169.

[Koba87] Isamu Kobayashi, "Database Consistency and Update Validation", *Sanno College Bulletin* Vol 7 No 2 Mark 1987, 105-129.

[Kowa87] R.A. Kowalski, F. Sadri and P. Soper, "Integrity constraint in deductive databases", *Proceedings of the 13th VLDB Conference*, Brighton (1987) 61-69.

[Lee94] S.Y. Lee, T.W. Ling, "Improving Integrity Constraint Checking for Stratified Deductive Databases", *Proceedings of Database and Expert Systems Applications*, 591-600, Athens, 1994.

[Levy93] A.Y.Levy and Y.Sagiv, "Queries independent of update", *Proceedings of the 19th VLDB Conference*, 1993, 171-181.

[Ling87] T.W. Ling, "Integrity constraint checking in deductive database using Prolog not-predicate", *Data and Knowledge Engineering*, Vol 2, 1987, 145-168.

[Lloy87] J.W. Lloyd, E.A. Sonenberg and R.W. Topor, "Integrity Constraint Checking in Stratified Databases", Journal of Logic Programming, Vol 4, No 4, 1987.

[Lloy87b] J.W. Lloyd, *Foundation of Logic Programming*, 2nd edition, Springer-Verlag, 1987.

[Ull89] J.D. Ullman, *Principles of Database and Knowledge-Base Systems*, Computer Science Press, 1989.