Efficient Integrity Checking for Databases with Recursive Views

Davide Martinenghi and Henning Christiansen

Roskilde University, Computer Science Dept.
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: {dm,henning}@ruc.dk

Abstract. Efficient and incremental maintenance of integrity constraints involving recursive views is a difficult issue that has received some attention in the past years, but for which no widely accepted solution exists yet. In this paper a technique is proposed for compiling such integrity constraints into incremental and optimized tests specialized for given update patterns. These tests may involve the introduction of new views, but for relevant cases of recursion, simplified integrity constraints are obtained that can be checked more efficiently than the original ones and without auxiliary views. Notably, these simplified tests are derived at design time and can be executed before the particular database update is made and without simulating the updated state. In this way all overhead due to optimization or execution of compensative actions at run time is avoided. It is argued that, in the recursive case, earlier approaches have not achieved comparable optimization with the same level of generality.

1 Introduction

Recursive views are generally regarded as a welcome extension to relational databases, as they allow a large class of query problems to be formulated within a declarative query language. To this end, we can mention flexible query answering based on taxonomies stored in the database, and various kinds of path-finding problems, such as network routing and travel planning. The introduction of recursion (since 1999 in the SQL standard [13] as *stratified linear* recursion based on fixpoint semantics) naturally raises a need for a satisfactory treatment of recursion in integrity constraints (ICs) which, in real-world applications, usually include complex data dependencies and "business logic". In this respect, database management systems should provide means to automatically verify, in an efficient way, that database updates do not introduce any violation of integrity. Maintaining compliance of data with respect to ICs is a crucial database issue, as, if data consistency is not guaranteed, then query answers cannot be trusted.

ICs are properties that must hold throughout the existence of a database for it to represent a meaningful set of data. While a complete check of integrity is prohibitive in any realistic case, it gives good sense to search for incremental strategies checking only the consequences of a database update, based on the hypothesis that the database was consistent before the update itself. This principle, called simplification, has been studied at least since [24], both for relational and deductive databases. The majority of existing methods either disregard recursion completely or disallow recursively defined relations to occur in ICs. Earlier approaches to simplification produce constraints that need to be checked in the updated database. We emphasize, however, that it is possible to decide, in the current state, whether a proposed update will introduce inconsistency (i.e., if it were executed). The framework we propose can handle a very general class of updates specified with a rule-based language that allows one to express parametric update patterns. Such patterns are used on the ICs at design time, when only the schema exists and not yet any database state, in order to generate simplified parametric constraints. Later, at runtime, the parametric constraints are instantiated with the specific update values and tested in the actual state.

The main contributions of this paper are as follows. *(i)* We formalize the general problem of finding simplified, incremental integrity checks for databases with recursive views, based on our previous contribution for the non-recursive case [6]. *(ii)* We develop a terminating procedure, based on the identification of specific recursive patterns, that generates efficient simplified tests that are necessary and sufficient conditions for integrity of the updated database. The method allows more general updates and provides finer results than previous approaches. The procedure takes in input a parametric update pattern and a set of ICs and produces, as output, a set of optimized ICs.

The paper is organized as follows. The simplification framework is shown in section 2 and refined for recursion in section 3; its ability to handle recursive cases is demonstrated through a series of examples in section 4. A detailed comparison of methods that handle recursion is given in section 5, followed by experimental evaluation in section 6. Concluding remarks are provided in section 7.

2 A Framework for Simplification of Integrity Constraints

2.1 Basic Notions

For simplicity, we apply notation and concepts from deductive databases, more specifically Datalog programs with stratified negation [10], but we stress that our results are also applicable in a relational setting, since translation techniques from Datalog to SQL are available [7]. In particular, we assume familiarity with the notions of *predicates* (p, q, $\ldots$), *constants* (a, b, $\ldots$), *variables* (x, y, $\ldots$), function-free *terms*, *atoms*, *literals*, *logical formulas*, *substitutions*, *renaming*, *instances* of formulas and *subsumption.* Sequences of terms are indicated by vector notation, e.g., $\vec{t}$. Substitutions are written as $\{\vec{x}/\vec{t}\}$ in order to indicate which variables are mapped to which terms. A *clause* is a formula $A \leftarrow L_1 \wedge \cdots \wedge L_n$ where A is an atom and $L_1, \ldots, L_n$ are literals and with the usual understanding of variables being implicitly universally quantified; A is called the *head* and $L_1 \wedge \cdots \wedge L_n$ the *body* of the clause. If the head is missing (understood as *false*) the clause is called a *denial*; if the body is missing (understood as *true*) it is a *fact*; all other clauses are called *rules*. Clauses are assumed to be *range restricted*, i.e., all clause variables must occur in a positive database literal in the body.

As stressed in the introduction, ICs need to be specialized for update patterns rather than for specific updates. In order to integrate this in our framework, a special category of symbols called *parameters* is introduced. Parameters are written in boldface ($\mathbf{a}, \mathbf{b}, \ldots$) and can appear anywhere in a formula where a constant is expected. Parameters behave like variables that are universally quantified at a metalevel; they are not expected to be part of any actual database nor of any query or update actually given to a database, but we may have parametric expressions of these categories.

Unique name axioms are assumed for (non-parametric) constants, i.e., distinct constants denote distinct values. A *parameter substitution* is a mapping from parameters to constants. Whenever E is an expression containing parameters $\vec{\mathbf{a}}$, and π is a parameter substitution of the form $\{\vec{\mathbf{a}}/\vec{c}\}$, $E\pi$ denotes the expression that arises from E when each occurrence of a parameter is replaced by its value specified by π; $E\pi$ is called a *parametric instance* of E.

Definition 1 (Database). *A* (database) schema *consists of disjoint sets of* extensional *and* intensional *predicates (collectively called* database predicates*) and a pair $\langle IDB, IC\rangle$, where IDB is a finite set of range restricted rules defining intensional predicates and IC a finite set of denials called a* constraint theory*. A* database *with schema $\langle IDB, IC\rangle$ is a triple $\langle IDB, IC, EDB\rangle$, where EDB is a finite set of facts of extensional predicates.*

When the schema is understood, the database may be identified with EDB. By virtue of the one-to-one correspondence between these logical notions and relational databases, we will use interchangeably the notions of intensional predicate and *view*, extensional predicate and *relation*, fact and *tuple*.

Definition 2 (Recursion). *For two predicates p and q, p* derives *q (written $p \hookrightarrow q$) if p occurs in the body of a rule whose head predicate is q. Let $\hookrightarrow^+$ be the transitive closure of $\hookrightarrow$. A predicate p is* recursive *iff $p \hookrightarrow^+ p$.*

As in [4], we can limit our attention to bilinear systems (those whose rules have at most two predicates mutually recursive with the head predicate), as any stratified program can be rewritten as an equivalent bilinear program. We only focus on *stratified* databases [1], that do not allow mixing negation and recursion. We refer to the semantics of the *standard model*, and write $D \models \phi$, where D is a (stratified) database and ϕ is a closed formula, to indicate that ϕ holds in D's standard model. The notation $A \models B$ is extended to parametric expressions with the meaning that it holds for all its parametric instances; similarly for $\equiv$ and "iff". We view satisfaction of ICs *by entailment* [10].

Definition 3. *A database $D = \langle IDB, IC, EDB\rangle$ is* consistent *whenever $D \models IC$.*

Definition 4 (Defining formula). *Given an IDB and an intensional predicate p defined in it by the rules $\{p(\vec{t_1}) \leftarrow F_1, \ldots, p(\vec{t_n}) \leftarrow F_n\}$, where the $\vec{t_i}$'s are sequences of terms and the F_i's are conjunctions of literals, the* defining formula *of p is $(F_1 \wedge \vec{x} = \vec{t}_1)\rho_1 \vee \ldots \vee (F_n \wedge \vec{x} = \vec{t}_n)\rho_n$, where $\vec{x}$ is a sequence of new distinct variables and each ρ_i is a renaming giving fresh new names to the variables of F_i not in $\vec{x}$. The variables in $\vec{x}$ are the* distinguished variables *of the defining formula; all other variables in it are the* non-distinguished variables.

Example 1. Let D be a database representing an acyclic directed graph and let S be its schema $\langle IDB, \Gamma\rangle$, where

$$\begin{aligned} IDB = \{\ & p(x,y) \leftarrow e(x,y),\\ & p(x,y) \leftarrow e(x,z) \wedge p(z,y)\}. \end{aligned}$$

and $\Gamma = \ \{\leftarrow p(x,x)\}$. Direct connection of nodes is stored in relation $e/2$. Directed paths are expressed by $p/2$. Acyclicity of the graph is imposed by Γ. The defining formula of p is $e(x,y) \vee (e(x,z) \wedge p(z,y))$, where x, y are distinguished variables and z is a non-distinguished variable.

For convenience, we include *queries* in intensional predicates; when no ambiguity arises, a given query may be indicated by means of its defining formula.

Definition 5 (Update). *A* predicate update *for an extensional predicate p is an expression of the form $p(\vec{x}) \Leftarrow p'(\vec{x})$ where $\Leftarrow p'(\vec{x})$ is a query; p is said to be* affected *by the update. A* (database) update *is a set of predicate updates for distinct predicates. For a given database D and an update U, the* updated *database D^U is as D, but for every extensional predicate p affected by a predicate update $p(\vec{x}) \Leftarrow p'(\vec{x})$ in U, the subset $\{p(\vec{t}) \mid D \models p(\vec{t})\}$ of EDB is replaced by the set $\{p(\vec{t}) \mid D \models p'(\vec{t})\}$.*

This definition subsumes others that separately specify the added and deleted parts of a predicate. As mentioned, updates can be parametric as input to the transformations to follow.

Example 2. Update $U_1 = \{p(x) \Leftarrow p(x) \vee x = a\}$ describes the addition of fact $p(a)$, whereas $U_2 = \{r(x,y) \Leftarrow (r(x,y) \wedge x \neq \mathbf{a}) \vee (r(\mathbf{a},y) \wedge x = \mathbf{b})\}$ is parametric and means "change any $r(\mathbf{a},x)$ into $r(\mathbf{b},x)$". If $\mathbf{a}$ and $\mathbf{b}$ are instantiated to the same constant, U_2 is immaterial.

In order to simplify the notation for tuple additions and deletions, we write in the following $p(\vec{\mathbf{a}})$ as a shorthand for $p(\vec{x}) \Leftarrow p(\vec{x}) \vee \vec{x} = \vec{\mathbf{a}}$ and $\neg p(\vec{\mathbf{a}})$ for $p(\vec{x}) \Leftarrow p(\vec{x}) \wedge \vec{x} \neq \vec{\mathbf{a}}$.

2.2 Weakest Preconditions

In order to capture the effect of an update U on a constraint theory Γ we introduce the After operator below, which returns a formula that evaluates, in the present state, in the same way as Γ would evaluate in the updated state. In order to make the definition precise, we need to make use of *unfolding* to repeatedly replace every non-recursive intensional predicate by its defining formula until only extensional or recursive predicates appear in the constraint theory.

Definition 6 (Unfolding). *Let Γ be a formula and IDB a set of rules defining predicates $p_1, \ldots, p_n$. Let $F_i(\vec{x}_i, \vec{y}_i)$ be the defining formula of p_i in IDB, where $\vec{x}_i$ are the distinguished and $\vec{y}_i$ the non-distinguished variables.* $\mathsf{Unfold}_{IDB}(\Gamma)$ *is the formula obtained by replacing as long as possible, in Γ, each occurrence of an atom of the form $p_i(\vec{t})$ by $(\exists \vec{y}_i F_i(\vec{t}, \vec{y}_i))$, for each non-recursive predicate p_i defined in IDB.*

Definition 7. *Let U be an update, IDB a set of rules of a schema S and Γ a constraint theory.*

- *Let us indicate with Γ^U a copy of Γ in which any atom $p(\vec{t})$ whose predicate is affected by a predicate update $p(\vec{x}) \Leftarrow p^U(\vec{x})$ in U is simultaneously replaced by the expression $p^U(\vec{t})$ and every intensional predicate q is replaced by a new predicate q^U.*
- *Similarly, let us indicate with IDB^U a copy of IDB in which the same replacements are simultaneously made.*

We define $\mathsf{After}^U_S(\Gamma) = \mathsf{Unfold}_{IDB \cup IDB^U}(\Gamma^U)$.

Without including details, it may be assumed that After performs standard rewriting in order to have the resulting formula in denial form. The subscript S is always omitted when clear from the context[1].

Example 3. Consider the updates of example 2. Let Γ_1 be $\{\leftarrow p(x) \wedge q(x)\}$ (p and q are mutually exclusive). We have, then:

$$\mathsf{After}^{U_1}(\Gamma_1) = \{ \leftarrow p(x) \wedge q(x), \\ \leftarrow q(a) \quad \}.$$

For $\Gamma_2 = \{\leftarrow r(c, x) \wedge q(x)\}$, we have:

$$\mathsf{After}^{U_2}(\Gamma_2) = \{ \leftarrow r(c, x) \wedge c \neq \mathbf{a} \wedge q(x), \\ \leftarrow r(\mathbf{a}, x) \wedge c = \mathbf{b} \wedge q(x) \ \}.$$

Note that these (non)equalities cannot be evaluated; if both parameters are instantiated to the same constant, the result collapses to Γ_2 (the update is neutral).

The characteristic property of the After transformation is captured by the notion of weakest precondition, i.e., a test that can be checked in the present state but indicating properties of the new state.

Definition 8 (Weakest precondition). *Let Γ and Γ' be constraint theories referring to the same schema S, and U an update. Then Γ' is a* weakest precondition (WP) *of Γ wrt U whenever $D \models \Gamma'$ iff $D^U \models \Gamma$ for any database state D with schema S.*

[1] Note, however, that, in the body of the resulting formula, some of the conjuncts might be expressions of the form $\neg\exists\vec{x}[\ldots]$, with nested levels of existentially quantified variables. Although the framework can be adapted to these cases, for reasons of space we will focus on standard denials.

Proposition 1. *For any constraint theory Γ and update U, $\mathsf{After}^U(\Gamma)$ is a WP of Γ wrt U; for any other Ψ which is a WP of Γ wrt U, we have $\Psi \equiv \mathsf{After}^U(\Gamma)$.*

To simplify means then to optimize a WP based on the invariant that the constraint theory holds in the present state.

Definition 9 (Conditional WP). *Let Γ and Γ' be constraint theories referring to the same schema S, and U an update. Then Γ' is a* conditional weakest precondition (CWP) *of Γ wrt U whenever $D \models \Gamma'$ iff $D^U \models \Gamma$ for any database state D consistent with Γ.*

A WP is also a CWP but not necessarily the other way round. For instance, $\{\leftarrow q(a)\}$ is a CWP (but not a WP) of Γ_1 wrt U_1 of example 3.

2.3 Optimizing Transformations on Integrity Constraints

An essential step in the simplification process is the achievement of constraints that are easier to evaluate than the original ICs. Several measures of the evaluation cost exist: the checking space [26] (the tuples to be accessed in order to evaluate the constraint), the "weakness" of the constraint theory [26], the number of literals in it [5], its level of instantiation [7]. However, all these criteria are only estimates of the effort that is needed to evaluate an IC, as the actual execution time will also depend on the database state as well as on the physical data structure. Furthermore, due to theoretical limitations, no procedure can produce an optimal constraint theory in all cases (for any of the above measures).

In order to remove as many unnecessary checks as possible from After's output, such as redundant denials and sub-formulas, we define a transformation Optimize that simplifies a given constraint theory using a set of trusted hypotheses. Typically, the input to Optimize is After's output theory and the hypotheses are After's input theory. Optimize applies sound and terminating rewrite rules to remove from the input theory all denials and literals that can be proved redundant. *Reduction* [11] is used to eliminate redundancies within a single denial.

Definition 10 (Reduction). *For a denial ϕ, the reduction ϕ^- of ϕ is the result of applying on ϕ the following rules as long as possible, where c_1, c_2 are distinct constants, $\mathbf{a}$ is a parameter, t a term, A an atom, C, D (possibly empty) conjunctions of literals,* vars *indicates the set of variables occurring in its argument and* dom *the set of variables in a substitution domain.*

$$\begin{array}{lll}
\leftarrow c_1 = c_2 \wedge C & \Rightarrow & \mathit{true} \\
\leftarrow c_1 \neq c_2 \wedge C & \Rightarrow & \leftarrow C \\
\leftarrow t \neq t \wedge C & \Rightarrow & \mathit{true} \\
\leftarrow t = t \wedge C & \Rightarrow & \leftarrow C \\
\leftarrow x = t \wedge C & \Rightarrow & \leftarrow C\{x/t\} \\
\leftarrow x \neq t \wedge C & \Rightarrow & \leftarrow C \ \textit{if}\ \{x,t\} \cap \mathrm{vars}(C) = \emptyset \ \textit{and}\ t \ \textit{is not}\ x \\
\leftarrow \mathbf{a} = c_2 \wedge C & \Rightarrow & \leftarrow \mathbf{a} = c_2 \wedge C\{\mathbf{a}/c_2\}^2 \\
\leftarrow A \wedge \neg A \wedge C & \Rightarrow & \mathit{true} \\
\leftarrow C \wedge D & \Rightarrow & \leftarrow D \ \textit{if}\ \exists\sigma \ \textit{s.t.}\ C\sigma \ \textit{subclause of}\ D \ \textit{and}\ \mathrm{dom}(\sigma) \cap \mathrm{vars}(D) = \emptyset
\end{array}$$

[2] We assume that each equality is only processed once.

Obviously, for any denial ϕ we have $\phi^- \equiv \phi$. The last rule (*subsumption factoring* [9]) includes the elimination of duplicate literals. The *expansion* [10] of a clause, indicated with a "+" superscript, replaces every constant in a database predicate (or variable already occurring elsewhere in database predicates) by a new variable, and equals it to the replacing item.

Example 4. Let $\phi = \leftarrow p(x,a,x)$. Then $\phi^+ = \leftarrow p(x,y,z) \wedge y = a \wedge z = x$.

For some classes of constraints, such as sets of Horn clauses[3], a resolution-based procedure limiting the size of resolvents to the size of the biggest denial is known to be refutation-complete[4], i.e., it derives *false* iff the set is unsatisfiable. We refer to [27] for the resolution principle and other related notions.

Definition 11. *For a constraint theory Γ, the notation $\Gamma \vdash_R \phi$ indicates that there is a resolution derivation of a denial ψ from Γ^+ such that in each resolution step the resolvent has at most n literals and ψ^- subsumes ϕ, where n is the number of literals of the largest denial in Γ^+.*

The boundedness we have imposed guarantees termination, as Γ is function-free.

Proposition 2. $\vdash_R$ *is sound and terminates on any input.*

Definition 12. *Given two constraint theories Δ and Γ,* $\mathsf{Optimize}_\Delta(\Gamma)$ *is the result of applying the following rewrite rules on Γ as long as possible; ϕ, ψ are denials, Γ' is a constraint theory, $\sqcup$ is disjoint union.*

$$\begin{array}{lll}
\{\phi\} \sqcup \Gamma' & \Rightarrow & \Gamma' \textit{ if } \phi^- = \textit{true} \\
\{\phi\} \sqcup \Gamma' & \Rightarrow & \Gamma' \textit{ if } (\Gamma' \cup \Delta) \vdash_R \phi \\
\{\phi\} \sqcup \Gamma' & \Rightarrow & \{\phi^-\} \cup \Gamma' \textit{ if } \phi \neq \phi^- \\
\{\phi\} \sqcup \Gamma' & \Rightarrow & \{\psi^-\} \cup \Gamma' \textit{ if } (\{\phi\} \sqcup \Gamma' \cup \Delta) \vdash_R \psi \textit{ and } \psi^- \textit{ strictly subsumes } \phi
\end{array}$$

The last rewrite rule allows the removal of literals from a denial; the other rules are self-explanatory.

Proposition 3 (Correctness of $\mathsf{Optimize}$). $\mathsf{Optimize}_\Delta(\Gamma)$ *terminates for any Γ, Δ and $D \models \Gamma$ iff $D \models$* $\mathsf{Optimize}_\Delta(\Gamma)$ *in any database D consistent with Δ.*

Definition 13. *For a schema $S = \langle IDB, \Gamma \rangle$ and an update U, let $\Delta = \mathsf{Unfold}_{IDB}(\Gamma)$. We define $\mathsf{Simp}^U_S(\Gamma) = \mathsf{Optimize}_\Delta(\mathsf{After}^U_S(\Gamma))$.*

From the previous results we get immediately the following.

Proposition 4. *Let $S = \langle IDB, \Gamma \rangle$ be a schema and U an update. Then $\mathsf{Simp}^U_S(\Gamma)$ is a CWP of Γ wrt U.*

Example 5. With Γ_1 and U_1 from example 2, we have $\mathsf{Simp}^{U_1}(\Gamma_1) = \{\leftarrow q(a)\}$.

[3] Here denials with at most one negative literal.
[4] With *factoring*, *paramodulation* for inequalities and the *reflexivity* axiom [16].

Each step in Optimize reduces the number of literals or instantiates them. Simp is indeed guaranteed to reach a minimal result (by the *subsumption theorem* [25]) for all constraint classes for which $\vdash_R$ is refutation complete[5]. The high complexity of Simp (subsumption alone is in general NP-complete [15]) does not affect the quality of the approach, as simplification takes place at design time (runtime simplification could indeed outweigh the optimization gained), which is justified by the following property.

Proposition 5. *Let Γ be a constraint theory, U an update, and π a parametric substitution. Then* $(\mathsf{Simp}^U(\Gamma))\pi \equiv \mathsf{Simp}^{U\pi}(\Gamma\pi)$.

The present technique is based on an a priori knowledge of the update patterns allowed by a database designer. However, if such patterns are not given in advance, the method is still applicable. We may, e.g., generate all simplifications corresponding to single additions or deletions of any database relation and, thus, obtain optimized behavior for these cases.

3 Refinements for Ordered Linear Recursion

In Simp, recursive predicates in ICs are replaced by new recursive predicates. For an important class of linear recursion that embraces some of the most commonly used recursive patterns (such as left- and right-linear recursion [23]), known as *ordered linear recursion* (OLR) [29], the simplification process can be refined, by possibly eliminating the introduction of new recursive views.

Definition 14. *A predicate r is an OLR predicate if it is defined as follows*

$$\begin{array}{l} \{\ r(\vec{x},\vec{y}) \leftarrow q(\vec{x},\vec{y}) \\ \ \ \ r(\vec{x},\vec{y}) \leftarrow p(\vec{x},\vec{z}) \wedge r(\vec{z},\vec{y})\ \}, \end{array} \tag{1}$$

where p and q are predicates on which r does not depend and $\vec{x},\vec{y},\vec{z}$ are disjoint sequences of distinct variables. The first rule is the exit rule*, while the other is the* recursive rule*.*

There may in principle be several exit rules and recursive rules for the same OLR predicate r; however, these can always be reduced to one single exit rule and recursive rule by introducing suitable new views. Note thus that p and q need not be base predicates.

We first transform the definition of r as to decompose it in two parts: a nonrecursive definition and a transitive closure definition (r_p below). If p and q are the same predicate, then no transformation is needed, as the definition of r is already the transitive closure of p. Otherwise we replace r's definition with the following, equivalent set of rules:

$$\begin{array}{l} \{\ r(\vec{x},\vec{y}) \leftarrow q(\vec{x},\vec{y}) \\ \ \ \ r(\vec{x},\vec{y}) \leftarrow r_p(\vec{x},\vec{z}) \wedge q(\vec{z},\vec{y}) \\ \ \ \ r_p(\vec{x},\vec{y}) \leftarrow p(\vec{x},\vec{y}) \\ \ \ \ r_p(\vec{x},\vec{y}) \leftarrow p(\vec{x},\vec{z}) \wedge r_p(\vec{z},\vec{y})\ \}. \end{array} \tag{2}$$

[5] Outside these classes, there are (practically unlikely) cases where the simplification may contain some redundancies.

Note that the argument is perfectly symmetric when r's recursive rule is of the form $r(\vec{x},\vec{y}) \leftarrow r(\vec{x},\vec{z}) \wedge p(\vec{z},\vec{y})$. In this case the second rule in (2) becomes $r(\vec{x},\vec{y}) \leftarrow q(\vec{x},\vec{z}) \wedge r_p(\vec{z},\vec{y})$ and r_p is defined as before.

All occurrences of r in a constraint theory can now be unfolded wrt the first two rules in (2), which introduce q and r_p, the latter being the transitive closure of p. Intuitively, it is easy to characterize the set of tuples that are added to r_p upon addition of a p-tuple, as r_p can be thought of as a representation of paths of a directed graph of p-edges. Suppose that update U is the addition of tuple $\langle \vec{\mathbf{a}}, \vec{\mathbf{b}} \rangle$ to p, then all added r_p paths are those that pass by the new p-arc and that were not there before the update. If $\delta^+_U r_p(\vec{x},\vec{y})$ indicates that there is a new path from $\vec{x}$ to $\vec{y}$ after update U, this can be expressed as:

$$\delta^+_U r_p(\vec{x},\vec{y}) \leftarrow (r_p(\vec{x},\vec{\mathbf{a}}) \vee \vec{x} = \vec{\mathbf{a}})) \wedge (r_p(\vec{\mathbf{b}},\vec{y}) \vee \vec{y} = \vec{\mathbf{b}})) \wedge \neg r_p(\vec{x},\vec{y}),$$

However, U is not necessarily a single tuple update, so $\delta^+_U r_p$ needs, in general, to be characterized in terms of r_p in the updated state.

Definition 15. *Let U be an update and r_p the transitive closure of non-recursive predicate p in schema $S = \langle IDB, \Gamma \rangle$; let r_p^U, p^U, IDB^U be defined as to obtain $\mathsf{After}^U_S(\Gamma)$ in definition 7. Let $OLR(r_p, S)$ be the following set of rules:*

$$\begin{aligned}
\{r_p^U(\vec{x},\vec{y}) &\leftarrow (r_p(\vec{x},\vec{y}) \wedge \neg\delta^-_U r_p^U(\vec{x},\vec{y})) \vee \delta^+_U r_p^U(\vec{x},\vec{y}),\\
\delta^+_U r_p(\vec{x},\vec{y}) &\leftarrow (r_p^U(\vec{x},\vec{w}_1) \vee \vec{x} = \vec{w}_1) \wedge (r_p^U(\vec{w}_2,\vec{y}) \vee \vec{y} = \vec{w}_2) \wedge\\
&\qquad \delta^+_U p(\vec{w}_1,\vec{w}_2) \wedge \neg r_p(\vec{x},\vec{y}),\\
\delta^-_U r_p(\vec{x},\vec{y}) &\leftarrow (r_p(\vec{x},\vec{w}_1) \vee \vec{x} = \vec{w}_1) \wedge (r_p(\vec{w}_2,\vec{y}) \vee \vec{y} = \vec{w}_2) \wedge\\
&\qquad \delta^-_U p(\vec{w}_1,\vec{w}_2) \wedge \neg r_p^U(\vec{x},\vec{y}),\\
\delta^+_U p(\vec{x}) &\leftarrow p^U(\vec{x}) \wedge \neg p(\vec{x}),\\
\delta^-_U p(\vec{x}) &\leftarrow \neg p^U(\vec{x}) \wedge p(\vec{x})\}
\end{aligned}$$

If, in $OLR(r_p, S)$, $\delta^+_U p(\vec{w}_1,\vec{w}_2) \equiv \vec{w}_1 = \vec{c}_1 \wedge \vec{w}_2 = \vec{c}_2 \wedge A$, where A is a conjunction of literals and c_1, c_2 are constants, then the second rule is replaced by

$$\begin{aligned}
\delta^+_U r_p(\vec{x},\vec{y}) \leftarrow (r_p(\vec{x},\vec{c}_1) \vee \vec{x} = \vec{c}_1) \wedge (r_p(\vec{c}_2,\vec{y}) \vee \vec{y} = \vec{c}_2) \wedge\\
\vec{w}_1 = \vec{c}_1 \wedge \vec{w}_2 = \vec{c}_2 \wedge A \wedge \neg r_p(\vec{x},\vec{y}).
\end{aligned} \tag{3}$$

The notation $OLR(S)$ indicates the rules obtained from $IDB \cup IDB^U$ by replacing the clauses defining each transitive closure predicate r_p^U with $OLR(r_p^U, S)$.

Definition 16. *Let U be an update, $S = \langle IDB, \Gamma \rangle$ a schema and Γ^U be defined as in definition 7. Let S^* be the same as S but in which, for all OLR predicate r, its definition (1) is replaced as in (2). Then $\mathsf{AfterRec}^U_S(\Gamma)$ is defined as $\mathsf{Unfold}_{OLR(S^*)}(\Gamma^U)$.*

Proposition 6. *For any constraint theory Γ and update U, $\mathsf{AfterRec}^U_S(\Gamma)$ is a WP of Γ wrt U.*

$\mathsf{OptimizeRec}$ is as $\mathsf{Optimize}$, but for any transitive closure predicate r_p, it also considers that $r_p(\vec{t_1}, \vec{t_2})$ subsumes $r_p(\vec{t_1}, \vec{x}) \land r_p(\vec{x}, \vec{t_2})$ if $\vec{x}$ does not occur elsewhere[6]. $\mathsf{SimpRec}^U_S(\Gamma)$ is defined as $\mathsf{OptimizeRec}_\Delta(\mathsf{AfterRec}^U_S(\Gamma))$, where Δ contains $\mathsf{Unfold}_S(\Gamma)$ plus the set of all transitive closure rules in S rewritten as denials, e.g., for a predicate r_p defined as in (2) the constraints are $\leftarrow \neg r_p(\vec{x}, \vec{y}) \land p(\vec{x}, \vec{y})$ and $\leftarrow \neg r_p(\vec{x}, \vec{y}) \land p(\vec{x}, \vec{z}) \land r_p(\vec{z}, \vec{y})$. Proposition 4 extends to $\mathsf{SimpRec}$.

The characterization of $\delta^-_U r_p$ given in proposition 6 requires the evaluation of $\neg r^U_p$. However, in many interesting cases $\delta^-_U r_p$ is going to be simplified away. The new views introduced by $\mathsf{AfterRec}$ can be completely disregarded if r^U_p does not occur in the simplified constraints. If both the new and the old state are available, as in some trigger implementations, r^U_p can be evaluated as "r_p in the new state". However, these are precisely the cases where the simplification was, to some extent, unsuccessful, as accessing or simulating the new state clearly requires extra work.

4 Examples

We first observe that many important problems can be reduced to OLR.

Example 6. In [22] the following recursive predicate b is described:

$$\{\ b(x,y) \leftarrow k(x,z) \land b(z,y) \land c(y),\quad b(x,y) \leftarrow d(x,y)\ \},$$

where b stands for "buys", k for "knows", c for "cheap" and d for "definitely buys". These definitions can be rewritten [14] as:

$$\begin{aligned}\{\ & b'(x,y) \leftarrow k(x,z) \land b'(z,y),\\ & b'(x,y) \leftarrow k(x,z) \land d(z,y) \land c(y),\\ & b(x,y) \leftarrow b'(x,y),\\ & b(x,y) \leftarrow d(x,y)\ \}.\end{aligned}$$

Replacing the body of b''s exit rule with a new view e, makes b' OLR:

$$\begin{aligned}\{\ & b'(x,y) \leftarrow k(x,z) \land b'(z,y),\\ & b'(x,y) \leftarrow e(x,y),\\ & e(x,y) \leftarrow k(x,z) \land d(z,y) \land c(y),\\ & b(x,y) \leftarrow b'(x,y),\\ & b(x,y) \leftarrow d(x,y)\ \}.\end{aligned}$$

The next example will be used in section 5 to compare the present work with previous methods.

Example 7. Consider the database from example 1. Let $U = \{e(\mathbf{a}, \mathbf{b})\}$ be an update pattern that adds an arc. We have

$$\begin{aligned}\mathsf{AfterRec}^U_S(\Gamma) &\equiv \{\leftarrow (p(x,x) \land \neg\delta^-_U p(x,x)) \lor \delta^+_U p(x,x)\}\\ &\equiv \{\leftarrow p(x,x) \land \neg\delta^-_U p(x,x),\\ &\qquad \leftarrow \delta^+_U p(x,x)\}.\end{aligned}$$

[6] It also subsumes $p(\vec{t_1}, \vec{x}) \land r_p(\vec{x}, \vec{t_2})$ and $r_p(\vec{t_1}, \vec{x}) \land p(\vec{x}, \vec{t_2})$.

When $\mathsf{OptimizeRec}$ is applied to $\mathsf{AfterRec}_S^U(\Gamma)$, every unfolding of the first constraint is removed (it is subsumed by the original constraint in Γ). Furthermore, $\delta_U^+ e(x,y)$ bounds both x and y, as $\delta_U^+ e(x,y) \equiv \neg e(\mathbf{a},\mathbf{b}) \wedge x = \mathbf{a} \wedge y = \mathbf{b}$. Therefore we can replace $\delta_U^+ p$ as in (3)

$$\delta_U^+ p(x,y) \equiv (p(x,\mathbf{a}) \vee x = \mathbf{a}) \wedge (p(\mathbf{b},y) \vee y = \mathbf{b}) \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(x,y),$$

which unfolds in the remaining $\leftarrow \delta_U^+ p(x,x)$ expression as follows:

$$\begin{array}{ll} \{ \leftarrow p(x,\mathbf{a}) \wedge p(\mathbf{b},x) \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(x,x), & \\ \;\; \leftarrow p(\mathbf{b},\mathbf{a}) \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(\mathbf{b},\mathbf{b}), & \\ \;\; \leftarrow p(\mathbf{b},\mathbf{a}) \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(\mathbf{b},\mathbf{b}), & \\ \;\; \leftarrow \mathbf{a} = \mathbf{b} \wedge \neg e(\mathbf{a},\mathbf{b}) \wedge \neg p(\mathbf{a},\mathbf{a}) & \}. \end{array}$$

The second and third constraints are identical, and therefore either can be removed. The $\neg p(-,-)$ literals are removed in $\mathsf{OptimizeRec}$ via resolution with the constraint in Γ. Similarly, the $\neg e(\mathbf{a},\mathbf{b})$ literals, in all constraints but the first one, can be removed by reduction and resolution via the intermediate $\vdash_R$-derivations of $\leftarrow e(x,x)$ and $\leftarrow e(x,z) \wedge p(z,x)$. Finally, the first IC is removed as $p(\mathbf{b},\mathbf{a})$ subsumes $p(\mathbf{b},x) \wedge p(x,\mathbf{a})$.

$$\mathsf{SimpRec}_S^U(\Gamma) = \{ \leftarrow p(\mathbf{b},\mathbf{a}), \;\; \leftarrow \mathbf{a} = \mathbf{b} \;\; \}.$$

Note that $\mathsf{SimpRec}_S^U(\Gamma)$ is a much simpler test than Γ as it basically requires to check whether there exists a path between two given nodes, whereas Γ implies testing the existence of a cyclic path for all the nodes in the graph.

A straightforward SQL translation of this simplified result (with p defined as a `WITH` view with columns `c1` and `c2`) is, e.g., the following query

```
SELECT "ko" FROM p WHERE (p.c1=$B AND p.c2=$A) OR $A=$B
```

in which `$A` and `$B` are replaced by the corresponding parameter values and an empty answer indicates consistency, whereas `ko` indicates inconsistency.

Example 8. [6 continued] Consider a schema S defining the *IDB* of example 6 and a scenario in which a given person p does not want to buy cheap products, expressed by $\Gamma = \{\leftarrow b(p,x) \wedge c(x)\}$. Suppose that a person meets another person who is definitely going to buy something. This event can be represented by the update $U = \{k(\mathbf{a},\mathbf{b}), d(\mathbf{b},\mathbf{c})\}$. We have [7]:

$$\begin{array}{ll} \mathsf{SimpRec}_S^U(\Gamma) = \{ & \leftarrow c(\mathbf{c}) \wedge [p = \mathbf{a} \vee p = \mathbf{b} \vee k'(p,\mathbf{a}) \vee k'(p,\mathbf{b})], \\ & \leftarrow c(x) \wedge [p = \mathbf{a} \vee k'(p,\mathbf{a})] \wedge \{d(\mathbf{b},x) \vee [k'(\mathbf{b},z) \wedge d(z,x)]\}\}, \end{array}$$

where k' is the transitive closure of k. The result indicates that U introduces an inconsistency whenever:

- $\mathbf{c}$ is cheap, and p is or (in)directly knows $\mathbf{a}$ or $\mathbf{b}$, or
- p is or (in)directly knows $\mathbf{a}$, and $\mathbf{b}$ definitely buys (or (in)directly knows someone who does) something cheap.

[7] For readability, the resulting formula is presented with disjunctions and rearranged via other trivial, cosmetic steps. Calculations are not shown due to space constraints.

5 Related Works

Several authors have provided results directly related to integrity checking. Most methods have been explicitly designed for relational databases with no views or disallow recursion in ICs; we refer to the survey [21] for references falling under these categories. We also point out that integrity checking is often regarded as an instance of materialized view maintenance: ICs are defined as views that must always remain empty for the database to be consistent. The database literature is rich in methods that deal with relational view/integrity maintenance; the book [12] and the survey [8] provide insightful discussion on the subject.

We now compare our approach with the methods that apply to recursive databases and show that our results have wider applicability and are at least as good. We discuss example 7 and use constants a, b instead of parameters $\mathbf{a}$, $\mathbf{b}$ for compatibility with these methods.

The technique described in [19] requires the calculation of two sets, P and N, that represent the positive and, respectively, negative potential updates generated by a given update. A set Θ is then computed, which contains all the mgus of the atoms in P and N with the atoms of corresponding sign in the IC. For example 7, we have $P = \{e(a,b), p(x,y)\}$, $N = \emptyset$ and $\Theta = \{y/x\}$. The updated database is consistent iff every condition $\Gamma\theta$ holds in it, for all $\theta \in \Theta$, Γ being the original constraint theory. Unlike our method, in this case the obtained condition is identical to Γ and therefore there is no simplification.

In [17], the authors determine low-cost pre-tests which are sufficient conditions that guarantee the integrity of the database. If the pre-tests fail, then integrity needs to be checked with an exact method, such as ours. A set of literal/condition pairs, called *relevant set*, is calculated. If the update in question unifies with any of the literals in the relevant set and the attached condition succeeds, then the pre-test fails; otherwise we are sure that the update cannot falsify the ICs. For example 7 the relevant set is $\{p(x,x)/true, e(x,x)/true, e(x,z)/true, p(z,x)/e_N(x,z)\}$ (e_N refers to e in the updated state). The update $e(a,b)$ unifies with $e(x,z)$, whose associated condition trivially succeeds, therefore the pre-test fails and an exact test needs to be executed.

In [18] partial evaluation of a meta-interpreter is used to produce logic programs that correspond to simplified constraints. The partial evaluator is given a meta-interpreter that constitutes a general integrity checker and produces as output a version of the meta-interpreter specialized to specific update patterns to be checked in the updated state (and employing the hypothesis that integrity holds before the update). The method *could* work for recursive databases, if a perfect partial evaluator were at disposal, but a loop check needs to be included in the program to ensure termination. This does not partially evaluate satisfactorily, resulting in an explosion of (possibly unreachable) alternatives.

With the method described in [5], which is based on the notion of *partial subsumption*, database rules are annotated with *residues* to capture the relevant parts that are concerned by the ICs. When doing semantic query optimization, such parts can often allow faster query evaluation times. However, when it comes to integrity checking, the method typically leaves things unchanged in the presence of recursive rules. In example 7 we need to calculate the residue of the constraint in Γ associated with the extensional relation e. The partial subsumption algorithm stops immediately, as no resolution step is possible, thus resulting in no simplification at all.

Seljée's *inconsistency indicators* (IIs) [29] are based on incremental expressions for OLR. We have improved on his method as follows. Firstly, our update language is more general, allowing compound updates and any kind of bulk operation expressible with rules (IIs cannot handle example 8). Secondly, the simplified constraints produced by SimpRec only need to consult the present database state, whereas IIs require, in general, the availability of both the old and the new state, even in the non-recursive case. For the treatment of recursion IIs impose a number of restrictions on the language (no negation, no existentially quantified variables) that we do not need. For example 7 the II, to be checked *after* the update, is $\leftarrow (p(b,x) \vee b = x) \wedge (p(x,a) \vee a = x)$. We evaluate the performance of this result in section 6.

In [3], integrity checking is regarded as an instance of *update propagation*, i.e., the problem of determining the effect of an update on a set of rules. The method extends the database with rules that express the incremental evaluation of the new state and the ICs themselves are defined by rules. A *soft consequence* operator [2] is then used to compute the model of this augmented database. Instead of a symbolic simplification of the original constraints, this method rather provides an efficient way for evaluating the new state. In this respect, it can be seen as orthogonal to ours, at least when our method does not eliminate references to the new state.

6 Experiments

In order to demonstrate the effectiveness of the simplification procedure, we have tested it on random update sets for example 7. Our tests were run on a machine with a 2 GHz processor, 1 GB of RAM and 80 GB of hard disk, using DES 1.1 [28], which is a Datalog system featuring full recursive evaluation and stratified negation. DES is implemented on top of Prolog; we could therefore program our tests in Prolog and simulate insertions by means of `assert` and deletions by `retract`. The DES query engine is optimized with memoization techniques for answering queries based on previous answers. In this context we always pose the same query $\leftarrow p(x,x)$ to check whether the graph is acyclic, and therefore answers can be reused for subsequent queries. Our method greatly improves performance even in the presence of an already optimized system.

Average execution times are indicated in milliseconds (within a time frame of 50 seconds) and the number of attempted insertions of edge facts is indicated on the x-axis. Each figure reports the execution times needed to update the database and check its consistency according to:

- The un-optimized IC (diamonds).
- The II produced by Seljée's method [29] (squares).
- The formula $\leftarrow p(b,a)$ (II*), produced by improving the II manually (crosses).

– The simplification obtained with Simp (triangles). Note that in this case consistency is checked before the update.

The third curve (a “perfect” post-test), although not generated by any known method, was included for comparison with the test-before-update strategy. In particular, in figure 1 we randomly generated 1500 arcs between 1000 different nodes, whereas in figure 2 we only used 50 different nodes. In the former case the formation of cycles is less likely and the times are generally better. In the latter, however, updates are much more likely to be rejected (44% of the updates were rejected in total, while only 12% in the former case); Simp in this case performs significantly better, with improvements around 20% even wrt the manually produced formula. The interpretation of these results is in accordance with the following observations:

– The comparison between the performance of the optimized and un-optimized checks shows that the optimized version is always more efficient than the original one.
– In both the un-optimized and II methods many more paths need to be computed, which is an expensive operation.
– The gain of early detection of inconsistency, which is a distinctive feature of our approach, is unquestionable in the case of illegal updates. In such a case, with our optimized strategy, the simplified constraint immediately reports an integrity violation wrt the proposed update, which is therefore *not* executed. On the other hand, the other methods require to execute the update, perform a consistency test and then roll back the update.

Note that the extra burden due to the execution and subsequent rollback of an illegal update is even more evident for compound updates, such as those of example 8; in these cases the benefits of a pretest wrt a post-test are even greater. We observe that the above comparisons did not take into account the time spent to produce the optimized constraints[8], as these can be generated at schema design time and thus do not interfere with run time performance.

7 Conclusion and Future Work

We have described a simplification framework for integrity checking in databases with recursive views. A general methodology based on the introduction of new recursive views has been described. This allows checking in the state before the update whether the database will be consistent in the updated state. While for recursive problems we cannot guarantee, in general, that the resulting test will be more efficient than the original one, this is indeed the case for the important class of OLR problems, for which differential expressions can be easily derived that indicate the incremental variations of the recursive predicate.

[8] All symbolic simplifications in this paper were obtained with an experimental implementation of the simplification procedure [20].

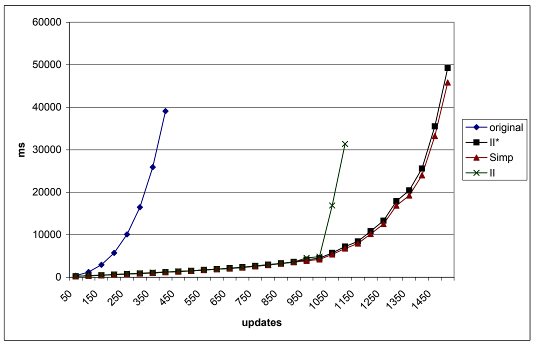

Fig. 1. Sparse data

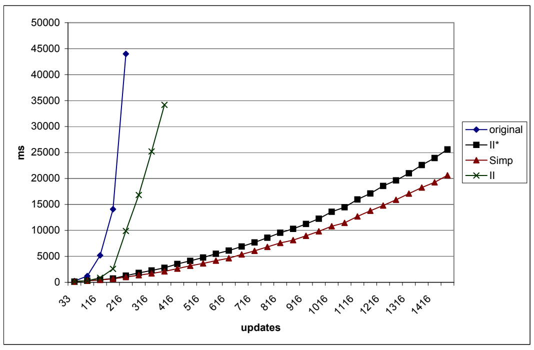

Fig. 2. Dense data

The simplified ICs can be regarded as queries and can therefore make use of all known traditional query optimization methods, including specific techniques for recursive queries evaluation, such as, e.g., magic sets.

There are numerous ways to extend this work. First of all, more cases for which useful differential expressions exist could be identified; regular-chain programs are a likely candidate. The literature is rich in decidable rewriting techniques that reduce recursive problems to easier ones; these could be integrated in the framework.

References

1. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming.*, pages 89–148. Morgan Kaufmann, 1988.
2. A. Behrend. Soft stratification for magic set based query evaluation in deductive databases. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 102–110. ACM Press, 2003.
3. A. Behrend. *Soft Stratification for Transformation-Based Approaches to Deductive Databases*. PhD thesis, University of Bonn, 2004.
4. T. Catarci and I. F. Cruz. On expressing stratified datalog. In *2nd ICLP Workshop on Deductive Databases and Logic Programming*, pages 85–100, 1994.
5. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems (TODS)*, 15(2):162–207, 1990.
6. H. Christiansen and D. Martinenghi. Simplification of database integrity constraints revisited: A transformational approach. In *LOPSTR'03*, volume 3018 of *LNCS*, pages 178–197. Springer, 2004.
7. H. Decker. Translating advanced integrity checking technology to sql. In *Database integrity: challenges and solutions*, pages 203–249. Idea Group Publishing, 2002.
8. G. Dong and J. Su. Incremental Maintenance of Recursive Views Using Relational Calculus/SQL. *SIGMOD Record*, 29(1):44–51, 2000.
9. N. Eisinger and H. J. Ohlbach. Deduction systems based on resolution. In *Handbook of Logic in Artificial Intelligence and Logic Programming - Vol 1: Logical Foundations.*, pages 183–271. Clarendon Press, Oxford, 1993.

10. P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. In *Logics for Databases and Information Systems*, pages 265–306, 1998.
11. J. Grant and J. Minker. Integrity constraints in knowledge based systems. In *Knowledge Engineering Vol II, Applications*, pages 1–25. McGraw-Hill, 1990.
12. A. Gupta and I. S. Mumick, editors. *Materialized views: techniques, implementations, and applications.* MIT Press, 1999.
13. INCITS. *Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation) - INCITS/ISO/IEC 9075-2-1999.* 1999.
14. Y. E. Ioannidis and E. Wong. Towards an algebraic theory of recursion. *J. ACM*, 38(2):329–381, 1991.
15. D. Kapur and P. Narendran. Np-completeness of the set unification and matching problems. In *CADE*, pages 489–495, 1986.
16. D. Knuth and P. Bendix. Simple word problems in universal algebras. *Computational Problems in Abstract Algebras*, pages 263–297, 1970.
17. S. Y. Lee and T. W. Ling. Further improvements on integrity constraint checking for stratifiable deductive databases. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 495–505. Morgan Kaufmann, 1996.
18. M. Leuschel and D. de Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *JLP*, 36(2):149–193, 1998.
19. J. W. Lloyd, L. Sonenberg, and R. W. Topor. Integrity constraint checking in stratified databases. *JLP*, 4(4):331–343, 1987.
20. D. Martinenghi. A simplification procedure for integrity constraints. `http://www.dat.ruc.dk/~dm/spic/index.html`, 2004.
21. E. Mayol and E. Teniente. A survey of current methods for integrity constraint maintenance and view updating. In *Advances in Conceptual Modeling: ER '99 Workshops*, volume 1727 of *LNCS*, pages 62–73. Springer, 1999.
22. J. F. Naughton. Minimizing function-free recursive inference rules. *J. ACM*, 36(1):69–91, 1989.
23. J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and mult-lineare rules. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 235–242. ACM Press, 1989.
24. J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.
25. S.-H. Nienhuys-Cheng and R. de Wolf. The equivalence of the subsumption theorem and the refutation-completeness for unconstrained resolution. In *ASIAN*, pages 269–285, 1995.
26. X. Qian. An effective method for integrity constraint simplification. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 338–345. IEEE Computer Society, 1988.
27. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
28. F. Sáenz-Pérez. Datalog educational system v1.1. user's manual. Technical Report 139-04, Faculty of Computer Science, UCM, 2004. Available from http://www.fdi.ucm.es/profesor/fernan/DES/.
29. R. Seljée. *A Fact Integrity Constraint Checking System for the Validation of Semantic Integrity Constraints after Updating Consistent Deductive Databases.* PhD thesis, Tilburg University, 1997.