

Incremental integrity checking: limitations and possibilities

Henning Christiansen and Davide Martinenghi

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

Abstract. Integrity checking is an essential means for the preservation of the intended semantics of a deductive database. Incrementality is the only feasible approach to checking and can be obtained with respect to given update patterns by exploiting query optimization techniques. By reducing the problem to query containment, we show that no procedure exists that always returns the best incremental test (aka *simplification* of integrity constraints), and this according to any reasonable criterion measuring the checking effort. In spite of this theoretical limitation, we develop an effective procedure allowing general parametric updates that, for given database classes, returns ideal simplifications and also applies to recursive databases. Finally, we point out the improvements with respect to previous methods based on an experimental evaluation.

1 Introduction

Semantic information in databases is conventionally represented under the form of integrity constraints (ICs), i.e., properties that must always be satisfied for the data to be considered *consistent*. Besides simple forms of predefined constraints, of which primary and foreign keys are the most common examples, real-world applications may involve nontrivial integrity requirements that capture complex data dependencies and “business logic”. The need for advanced integrity verification tools is testified by the introduction of several standard constructs for integrity support in the SQL language, such as *check* constraints and *assertions*. However, in spite of a long recognition of the importance of such practices, which are part of the SQL standard since 1992, today’s DBMSs still lack the ability of efficiently handling non-predefined constraints.

Maintaining compliance of data wrt ICs is an essential requirement: if data lack integrity, answers to queries cannot be trusted; furthermore, satisfaction of ICs can be exploited to improve query evaluation performance by means of so-called semantic query optimization. Databases, however, usually contain very large collections of data that quickly evolve over time. In this regard, DBMSs need to be extended with the ability to automatically verify, in an *incremental* way, that database updates do not introduce any violation of integrity.

Today’s practices based on triggers (at the database level) or hand-coding of tests (at the application level) are clearly unsatisfactory, since, by their procedural and *ad hoc* nature, they are prone to errors and difficult to maintain. The

need for automated integrity maintenance methods has attracted much research in the database as well as the logic programming and artificial intelligence communities. Main approaches to efficient integrity checking that have been proposed since the early eighties include extensions of the SLD(NF) proof procedure [18, 6], partial evaluation [10], update propagation [11], incremental view maintenance [7] and several others. The way we pursue here is the so-called *simplification* of ICs — a principle that has been recognized for more than two decades, dating back to at least [16], and then elaborated by several other authors, e.g., [11, 17, 4, 6, 20, 5]. Our work is an attempt to reconcile and generalize such ideas in a systematic way that may promote applications of deductive databases for use with current database management technology.

Simplification means to generate a set of ICs whose satisfaction in the *current* state implies the satisfaction of the original constraints in the updated state. The input of the procedure is a set of ICs to be maintained on the database as well as an update pattern describing a typology of updates that the database can receive; the produced output is the set of simplified ICs that should be checked upon reception of an update matching the given pattern. We find it important that a proposed simplification algorithm can work on parametric update patterns, not only specific updates. This means that such patterns can be simplified at design time, when only the schema exists and not yet any database state. At runtime the simplified ICs can be instantiated wrt the specific updates and tested in the actual state. The main interest of the simplification process is that the output set of ICs is as *easy* to evaluate as possible. In this sense, simplification proper is only feasible by assuming satisfaction of ICs in the state prior to the update.

We identify as “ideal” a simplification procedure that outputs a set of ICs that is minimal wrt an ordering that represents an approximation of the cost of evaluating the constraints. Although there is no ultimate criterion that, independently of the actual database state, perfectly measures the evaluation effort, natural requirements can be imposed that should be met by any sensible ordering — in particular, that “nothing to check” is the best possible simplification one can hope for. With this assumption, it can be proved that ideal simplification is equivalent to decidability of query containment, which is known not to hold in general (query containment is not decidable, e.g., already for pure DATALOG without negation). In fact, ideal simplification is possible in a class of databases if and only if query containment is decidable in that class.

In spite of this limitation, it can be argued on an experimental basis that simplification procedures that are “almost ideal” can still be of practical use and certainly improve upon non-optimized integrity checking.

2 The Simplification Problem

We adopt the notation and terminology of deductive databases, and focus on DATALOG with stratified negation [1] (aka DATALOG[¬]). Our results are, thus, also applicable in the relational setting. Predicates are divided into three pairwise disjoint sets: *intensional*, *extensional*, and *built-in* predicates. We use vector

notation to indicate sequences of terms, 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 $A \leftarrow L_1 \wedge \dots \wedge L_n$ where A is an atom and L_1, \dots, L_n are literals and with the usual understanding of variables being implicitly universally quantified; A is called the *head* and $L_1 \wedge \dots \wedge L_n$ the *body* of the clause. If the head is missing (understood as *false*) the clause is called a *denial*. A *rule* is a clause whose head is intensional, and a *fact* is a clause whose head is extensional and ground and whose body is empty (understood as *true*). Clauses are assumed to be *range restricted*, i.e., all clause variables must occur in a positive database literal in the body.

As mentioned, ICs need to be specialized for update patterns rather than for specific updates. For this purpose, we use *parameters* (written in boldface: $\mathbf{a}, \mathbf{b}, \dots$) that 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, and π is a parameter substitution for those, $E\pi$ is called a *parametric instance* of E .

Definition 1. A schema S is a pair $\langle IDB, IC \rangle$, where *IDB* (the *intensional database*) is a finite set of rules and *IC* a finite set of denials called a constraint theory. A database D on S is a pair $\langle IDB, EDB \rangle$, where *EDB* (the *extensional database*) is a finite set of facts; D is based on *IDB*. Any set $\mathcal{L} \subseteq \mathcal{S}$, where \mathcal{S} is the set of all schemata, is called a database language.

We express our definitions and operators on schemata, so that ICs are always in the context of an *IDB*; however, when the *IDB* is understood, the schema may be identified with *IC* and the database with *EDB*. When considering different schemata, we assume that they are *compatible*, i.e., they do not redefine each other's predicates. We focus on *stratified* databases [2], 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 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 say that a database $D = \langle IDB, EDB \rangle$ is *consistent* with *IC* whenever $D \models IC$ (and thus with schema $S = \langle IDB, IC \rangle$, written $D \models S$).

Definition 2. Given an *IDB* and an intensional predicate p defined in it by the rules $\{p(\vec{t}_1) \leftarrow F_1, \dots, p(\vec{t}_n) \leftarrow F_n\}$, 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 $(F_1 \wedge \vec{X} = \vec{t}_1)\rho_1 \vee \dots \vee (F_n \wedge \vec{X} = \vec{t}_n)\rho_n$, 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.

For convenience, we include *queries* in intensional predicates; when no ambiguity arises, a given query may be indicated by means of its defining formula (instead of the predicate name). The *extension* of a database predicate p in a given database D is defined as the set of tuples $\{\vec{a} \mid D \models p(\vec{a})\}$; if p is a query, we refer also to the extension as the *answer* to p in D and denote it \mathcal{A}_D^p .

Definition 3. 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. An 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 1. 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 $r(\mathbf{b}, X)$ ”. Update $U_3 = \{p(X) \Leftarrow q(X), \quad q(X) \Leftarrow p(X)\}$ exchanges the contents of p and q .

To simplify notation, 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 $p(\vec{X}) \Leftarrow p(\vec{X}) \wedge \vec{X} \neq \vec{\mathbf{a}}$.

The constraint verification problem asks, given a database D , a constraint theory Γ , such that $D \models \Gamma$, and an update U , whether $D^U \models \Gamma$ holds. Since checking $D^U \models \Gamma$ may be too expensive, a suitable reformulation of the problem is called for. With our approach we look for a constraint theory Γ^U such that $D^U \models \Gamma$ iff $D \models \Gamma^U$ and Γ^U is easier to evaluate than Γ . In other words, condition Γ^U , called a *simplification* of the original constraints Γ , should specialize the original Γ , as specific information coming from U is available, and avoid redundant checks by exploiting the fact that $D \models \Gamma$ holds. We observe that reasoning about the future database state D^U with a condition (Γ^U) that is tested in the present state D , complies with the deferred semantics of IC checking¹ and allows avoiding the execution of illegal updates completely. Formally, this is captured by the notions of conditional weakest precondition (CWP) and weakest precondition (WP).

Definition 4. Consider compatible schemata $S = \langle IDB, \Gamma \rangle$, $S' = \langle IDB', \Gamma' \rangle$ and update U . S' is a WP (resp., CWP) of S wrt U whenever $D \models \Gamma'$ iff $D^U \models \Gamma$ for any database D based on $IDB \cup IDB'$ (resp., and consistent with Γ).

A CWP is a necessary and sufficient condition for consistency of a database in the updated state to be checked in the state prior to the update (i.e., a *pre-test*). Among the CWPs, WPs do not exploit the initial consistency of the database.

¹ An update can be imagined as a sequence of operations modifying the state. With the *deferred* semantics, satisfaction of ICs is required after the whole update has executed, but not in the intermediate states.

Thus, to qualify as a simplification, a CWP must be at least as good as any WP. For this purpose, we assume, for any schema S and update U , the existence of a reference schema \bar{S}^U representing a WP of S wrt U ; we show the construction of such a WP in section 4. We further assume an ordering to sort the different CWPs so that a smallest element in this ordering represents an optimum.

Definition 5. *An ordering between schemata is a reflexive and transitive binary relation \preceq such that, for any two schemata S and S' :*

1. $\langle \emptyset, \emptyset \rangle \preceq S$ and it is decidable whether $S \preceq S'$.
2. If $S \neq S'$, either $S \prec S'$ or $S' \prec S$ ($S \prec S'$ means $S \preceq S'$ but not $S' \preceq S$).
3. For a given S , $\{S'' \mid S'' \preceq S\}$ is finite and its schemata can be enumerated.

It is essential, for definition 5, to consider as identical expressions that differ only by renaming and orders of operands of commutative and associative connectives.

Definition 6. *Given a schema S and an update U , schema S' is a simplification of S wrt U if $S' \preceq \bar{S}^U$ and S' is a CWP for S wrt U . A procedure with input S, U and output S' , written $\text{Simp}^U(S) = S'$, is a simplification procedure. The procedure is ideal if, for any S and U , there is no other simplification S'' of S wrt U s. t. $S'' \prec S'$.*

According to this definition, any CWP that is at least as small as \bar{S}^U in the \preceq ordering is considered a simplification; the minimal ones, among those, are the ideal simplifications. This distinction makes sense because, as we shall see, it is not always possible to obtain an ideal simplification in all cases, but even a non-ideal simplification can be a significant improvement wrt a non-optimized CWP. This is particularly important when the ordering somehow reflects the effort of checking the satisfaction of the constraint theory in any database state: ideal simplifications do then express the best possible way of checking ICs.

The basic idea is to start with the reference schema and optimize it as much as possible wrt the hypotheses S . For compatible schemata S, S_1, S_2 , we write $S_1 \stackrel{S}{\equiv} S_2$ to indicate that $D \models S_1$ iff $D \models S_2$ in any D consistent with S .

Definition 7. *Schema S_2 is an optimization of schema S_1 wrt schema S if $S_1 \stackrel{S}{\equiv} S_2$ and $S_2 \preceq S_1$. A procedure with input S_1, S and output S_2 , written $\text{Optimize}^S(S_1) = S_2$, which is idempotent in the sense that $\text{Optimize}^S(S_2) = S_2$, is an optimization procedure. The procedure is ideal if, for any S and S_1 , there is no other optimization S_3 of S_1 wrt S s. t. $S_3 \preceq S_2$.*

Obviously, $\text{Optimize}^S(\bar{S}^U)$ is a simplification procedure for S, U , which is ideal if Optimize is ideal, since for all CWPs S_1, S_2 of some S , we have $S_1 \stackrel{S}{\equiv} S_2$.

3 Achieving Ideal Simplification

Given two queries $\Leftarrow p(\vec{X})$ and $\Leftarrow q(\vec{X})$, the *query containment* problem (QC) asks whether \mathcal{A}_D^p is contained in \mathcal{A}_D^q for all database D . QC is already undecidable for DATALOG without negation [21], and, thus, also for DATALOG⁻. There is a direct correspondence between the problem of ideal simplification and QC.

Theorem 1. *For any database language \mathcal{L} , QC is decidable in \mathcal{L} if and only if \mathcal{L} admits an ideal simplification procedure².*

The only-if part of the proof enumerates all theories that are smaller than the reference WP and tests, by QC, whether they are CWPs until one is found. This may be impractical (although we assumed that there were finitely many such theories) so a different strategy is described at the end of this section. Analogously, an ideal optimization procedure can be constructed from a QC decision procedure (if it exists) using enumeration.

Theorem 2. *There exists an ideal optimization procedure for a language \mathcal{L} if and only if QC is decidable in \mathcal{L} .*

In order to characterize a transformed IC as an optimal simplification, it must represent a minimum in some ordering that reflects the effort of actually evaluating it. This can only be an estimate, as the actual execution times depend on the database state, which is not available at the time of the simplification process. Furthermore, it is highly dependent on the applied database technology that may perform optimizations that cannot be included in a general definition.

Several different criteria can be defined. A natural choice is a syntactic order based on the number of literals: the optimal theories are those in which this number is minimal (and when the number is the same, another standard ordering, such as the alphabetical ordering, is used). This ordering, indicated as \prec_ℓ , may appear a bit coarse, as the number of literals in, say, $\leftarrow 1 = 2$, $\leftarrow p(a)$, and $\leftarrow p(X)$ is the same. However, it applies within the class of CWPs of the input.

Semantic orderings are also possible (e.g., the weaker the theory, the better), but testing precedence is generally undecidable and it can be argued that this does not correctly reflect the evaluation effort either. The notion of *checking space* is sometimes used [17], i.e., the portion of the Herbrand base that affects the evaluation of a given constraint theory: the smaller the checking space, the better the CWP. However, there may be infinitely many theories (e.g., those that differ by tautologies) having the same checking space. For these reasons, and since no ordering can perfectly capture the notion of efficiency, we adhere to the simpler \prec_ℓ ordering. We stress that *any* criterion can only approximate optimality. For example, a syntactically minimal query does not necessarily evaluate faster than an equivalent non-minimal query in all database states; the amount of computation required to answer a query can be reduced, e.g., by adding a join with a very small relation. Several refinements can be considered, such as preferring more specific constraints. However, for all such improvements there will be cases in which efficiency is not measured precisely. For example, $\leftarrow p(X) \wedge q(Y)$ is likely to be evaluated faster than the more specific $\leftarrow p(X) \wedge q(X)$, as the former can be checked by verifying that either p or q are empty, whereas the latter introduces a join that potentially requires that all tuples in p be looked up in q . Even if we limit such criterion to the preference of ground literals to non-ground ones, we still do not capture the notion of efficiency correctly. For

² Theorems 1 and 2 and propositions 1, 2, and 3 are proved in [13].

example, $\leftarrow p(X)$ will typically run faster than $\leftarrow p(a)$, as for the former it is sufficient to verify that p is empty, whereas for the latter a lookup in p is needed.

As mentioned, finding an ideal simplification, although feasible in some cases, may be costly. We may thus less ambitiously content ourselves with a *local minimum*, i.e., a constraint theory such that no set of subclauses of its clauses is a simplification. A general procedure to find a local minimum of a given CWP Γ wrt hypotheses Δ consists in repeating the following steps as long as possible.

1. If there exists $\phi \in \Gamma$ such that $\Delta \cup (\Gamma \setminus \phi) \models \phi$ then ϕ is removed from Γ .
2. If there exists $\leftarrow L_1 \wedge \dots \wedge L_n = \phi \in \Gamma$ such that $\Delta \cup \Gamma \models \leftarrow L_1 \wedge \dots \wedge L_{i-1} \wedge L_{i+1} \wedge \dots \wedge L_n = \psi$ for some i s. t. $1 \leq i \leq n$ then ϕ is replaced by ψ .

After each step we still have a CWP and a local minimum is eventually found.

Example 2. Consider the following constraint theories.

$$\begin{aligned} \Delta &= \{ \leftarrow \neg p(X) \wedge q(X) \wedge r(X), \leftarrow p(X) \wedge \neg q(X), \leftarrow p(X) \wedge \neg r(X) \}, \\ \Gamma &= \{ \leftarrow s(X) \wedge q(X) \wedge r(X) \}, \quad \Sigma = \{ \leftarrow s(X) \wedge p(X) \}. \end{aligned}$$

We have $\Sigma \stackrel{\Delta}{\equiv} \Gamma$, as Δ is an encoding of the equivalence between $p(X)$ and $q(X) \wedge r(X)$. Both Γ and Σ are local minima of $\Gamma \cup \Sigma$ wrt Δ ; Σ is the global minimum.

In practice there is often one local minimum. However, when particular dependencies are encoded in the ICs, such as equivalences between (sets of) predicates, like in example 2, then they may differ. The procedure depicted in this section is, however, based on entailment, which is in general undecidable; furthermore, sound and complete proof procedures, based, e.g., on resolution, are not guaranteed to terminate.

Next, we describe a simplification framework implementing a practically relevant approximation of this strategy in which entailment is replaced by specialized sound and terminating proof procedures.

4 A Concrete Simplification Procedure

We now show how to construct the reference WP, given a schema and an update, which was only supposed to exist in the previous section.

Definition 8. Let $S = \langle IDB, \Gamma \rangle$ be a schema and U an update such that, for each predicate update $p(\vec{X}) \leftarrow p^U(\vec{X})$ in U , p^U is defined in IDB .

- 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 intensional predicate q^U defined in IDB^U below.
- Similarly, let us indicate with IDB^U a copy of IDB in which the same replacements are simultaneously made, and let IDB^* be the biggest subset of $IDB \cup IDB^U$ including only definitions of predicates on which Γ^U depends.

We define $\text{After}^U(S) = \langle IDB^*, \Gamma^U \rangle$.

The IDB^U used in the construction of definition 8 indicates auxiliary views that are needed in order to properly characterize the resulting constraint theory. Often no such views are strictly necessary, whereas, in some other cases (e.g., in the presence of recursion), they cannot be avoided; in the former case, we will omit the specification of the intensional database and refer to the unfolding³ of the constraint theory wrt IDB^* .

Proposition 1. *For any schema S and update U , $\text{After}^U(S)$ is a WP of S wrt U .*

In the construction of **After** we did not use the hypothesis that the initial constraint theory was satisfied in the state before the update. Since the result of **After** also refers to the same state, we use an optimization procedure receiving as input **After**'s output theory and, as hypotheses, **After**'s input theory. In other words, **After**'s result is non-optimized and we can pose $\bar{S}^U = \text{After}^U(S)$. We implement **Optimize** in terms of sound and terminating rewrite rules that remove from the input theory all denials and literals that can be proved redundant.

Given a denial ϕ , we indicate as ϕ^- its *reduction* [4], i.e., a copy of it in which all tautological (non)equalities are removed and all failing (non)equalities replace ϕ by *true*; variable-term equalities are also removed and cause the variable to be replaced by the equalled term. For example, $(\leftarrow X = a \wedge p(X))^- = \leftarrow p(a)$. Conversely, *expansion* [4] of a denial ϕ , indicated ϕ^+ , 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. For example let $(\leftarrow p(X, a, X))^+ = \leftarrow p(X, Y, Z) \wedge Y = a \wedge Z = X$. Obviously, for any denial ϕ we have $\phi^- \equiv \phi \equiv \phi^+$. We write $\Gamma \vdash_R \phi$ if there is a resolution derivation of a denial ψ from the constraint theory 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 on the size of resolvents guarantees termination, as Γ is function-free.

Definition 9. *Given the schemata S_Δ, S_Γ based on IDB , let Δ, Γ be the respective unfolding of their constraint theories wrt IDB ; $\text{Optimize}^{S_\Delta}(S_\Gamma)$ is the schema $\langle IDB, \Sigma \rangle$, where Σ is the result of applying on Γ the following rules as long as possible; ϕ, ψ are denials, Γ' is a constraint theory.*

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

The first two rules attempt the removal of a whole denial, while the last two try to remove literals from a denial, according to the strategy shown in the previous section. The described **Optimize** implements a terminating optimization procedure that can be used, with **After**, to compose a simplification procedure $\text{Simp}^U(S) = \text{Optimize}^S(\text{After}^U(S))$ for any schema S and an update U .

Example 3. We have $\text{Simp}^{\{p(\mathbf{a})\}}(\{\leftarrow p(X) \wedge q(X)\}) = \{\leftarrow q(\mathbf{a})\}$.

³ The replacement of each nonrecursive intensional predicate with its defining formula, until only extensional or recursive predicates remain; see [13] for details.

Each step in **Optimize** reduces the number of literals or instantiates them. The high complexity of **Simp** does not affect the quality of the approach, as simplification takes place at design time. This is possible thanks to the following property.

Proposition 2. *Let S, S' be schemata, U an update and π a param. substitution for U 's parameters. If S' is a CWP of S wrt U then $S'\pi$ is a CWP of S wrt $U\pi$.*

We argue that a syntactic ordering such as the one induced by the strategy for finding local minima captures efficiency for *most* cases, as will be demonstrated in our experiments. Besides, simplification also conforms to the strategy of specializing ICs as much as possible, in that variable/constant equalities are removed by substituting the variable by the constant. So, for example, a denial such as $\phi \leftarrow X = a \wedge p(X, Y) \wedge q(Y)$ is not transformed into $\leftarrow p(X, Y) \wedge q(Y)$,⁴ which has fewer literals but is arguably less efficient to evaluate than ϕ , but to $\leftarrow p(a, Y) \wedge q(Y)$, which contains fewer literals *and* is more specialized than ϕ .

4.1 Refinements for Recursion

For some of the most commonly used recursive patterns (such as left- and right-linear recursion [15]), simplification can be refined by possibly eliminating the introduction of new recursive views; work in this direction was done in [14]. A predicate r is right-linear if it is defined by the *exit* rule $r(\vec{X}, \vec{Y}) \leftarrow q(\vec{X}, \vec{Y})$ and by the *recursive* rule $r(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Z}) \wedge r(\vec{Z}, \vec{Y})$. There may in principle be several exit and recursive rules for the same predicate r , but they can always be reduced to one by introducing suitable new views.

The definition of r can always be decomposed in two parts: a nonrecursive definition $\{r(\vec{X}, \vec{Y}) \leftarrow q(\vec{X}, \vec{Y}), r(\vec{X}, \vec{Y}) \leftarrow r_p(\vec{X}, \vec{Z}) \wedge q(\vec{Z}, \vec{Y})\}$ and a transitive closure definition $\{r_p(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Y}), r_p(\vec{X}, \vec{Y}) \leftarrow p(\vec{X}, \vec{Z}) \wedge r_p(\vec{Z}, \vec{Y})\}$. The construction is symmetric when r is left-linear.

All occurrences of r in a constraint theory can now be unfolded wrt these definitions, which introduce q and r_p , the latter being the transitive closure of p (which can be thought of as a path in a directed graph of p -edges). Upon the addition U of tuple $p(\vec{a}, \vec{b})$, 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:

$$\delta_U^+ r_p(\vec{X}, \vec{Y}) \leftarrow (r_p(\vec{X}, \vec{a}) \vee \vec{X} = \vec{a}) \wedge (r_p(\vec{b}, \vec{Y}) \vee \vec{Y} = \vec{b}) \wedge \neg r_p(\vec{X}, \vec{Y}).$$

We can define $\delta_U^- r_p$ in a similar way and unfold, in **After**, predicate r_p^U wrt the definition $r_p^U(\vec{X}) \leftarrow (r_p(\vec{X}) \wedge \neg \delta_U^- r_p(\vec{X})) \vee \delta_U^+ r_p(\vec{X})$.

Similarly, **Simp** can be extended to use as extra hypotheses all transitive closure rules in S rewritten as denials, e.g., $\leftarrow \neg r_p(\vec{X}, \vec{Y}) \wedge p(\vec{X}, \vec{Y})$ and $\leftarrow \neg r_p(\vec{X}, \vec{Y}) \wedge p(\vec{X}, \vec{Z}) \wedge r_p(\vec{Z}, \vec{Y})$, for a predicate r_p .

⁴ Unless a constraint such as $\leftarrow X \neq a \wedge p(X, Y)$ is known to hold, which could then be used by a query optimizer to evaluate $\leftarrow p(X, Y) \wedge q(Y)$ as fast as $\leftarrow p(a, Y) \wedge q(Y)$.

Generally, $\delta_U^- r_p$ requires the evaluation of $\neg r_p^U$, but often $\delta_U^- r_p$ is simplified away. If both the new and the old state are available, as in some trigger implementations, r_p^U 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 with a view clearly requires extra work.

Example 4. Consider a schema S representing paths and edges of a directed graph $\{p(X, Y) \leftarrow e(X, Y), p(X, Y) \leftarrow e(X, Z) \wedge p(Z, Y)\}$ for which we impose acyclicity $\{\leftarrow p(X, X)\}$. Let $U = \{e(\mathbf{a}, \mathbf{b})\}$ be an update pattern that adds an arc. We have $\text{Simp}^U(S) = \{\leftarrow p(\mathbf{b}, \mathbf{a}), \leftarrow \mathbf{a} = \mathbf{b}\}$. Note that $\text{Simp}^U(S)$ is a much simpler test than S 's IC, as it basically requires to check whether there exists a path between two given nodes, whereas the latter implies testing the existence of a cyclic path for all the nodes in the graph.

4.2 Ideality of Simp

Definition 9 gives an approximation of the procedure described in section 3. The quality of the result depends on how well the described proof procedure implements entailment. It is known that for certain classes of languages, such as the monadic class, Herbrand's class and the one-variable class, sound and complete procedures based on resolution refinements are guaranteed to terminate. In these cases an ideal simplification can be found. The principles of subsumption and reduction are, in practice, sufficient for most cases of denial elimination, and resolution proper is only needed when the ICs encode circularity.

There are other cases in which entailment can be replaced by a terminating proof procedure. We recall that a clause is Horn if, when expressed as a disjunction of literals, it contains at most one positive literal. Then, for a set Γ of Horn denials containing no non-nullary function symbol, no parameters and no equalities, there is a terminating procedure that produces Γ 's local minimum.

Proposition 3. *The Optimize procedure always returns a local minimum when the inputs are view-less, Horn, parameter-free theories with no equalities.*

This result is in accordance with the decidability of QC for nonrecursive DATALOG [1]; it extends to Horn theories with equalities and parameters provided that proper equality axioms are added to the input set.⁵ However, QC is already undecidable for nonrecursive DATALOG⁻ [1], so we cannot hope for an ideal procedure in these cases. As for complexity, QC is known to be decidable in exponential time for nonrecursive DATALOG and subsumption is NP-complete in general [8]. The search for a local minimum in these cases is thus also exponential, since it may require solving $n + m$ QC problems, where n is the number of literals and m is the number of denials in the constraint theory. This would suggest that the problem is intractable; however, the complexity is here measured wrt the size of the query and not of the data in the database. Furthermore, simplification is a static process, therefore it is worthwhile to invest resources for compiling the constraints at design time so as to improve run time efficiency.

⁵ Actually, reduction takes care of reflexivity and expansion provides substitutivity, whereas symmetry was assumed to be an implicit syntactic property of equality; however, the transitivity axiom ($\leftarrow X \neq Y \wedge X = Z \wedge Z = Y$) needs to be added.

5 Experiments

In order to demonstrate the effectiveness of the simplification procedure, we have tested it on more complex examples. We show here our experimental results for the nonrecursive case⁶; we refer to [14] for an analysis of the recursive case. The random data sets used for the tests were generated beforehand, so that the different procedures under analysis could run on exactly the same data and thus be compared fairly. All tests were repeated 20 times, so as to have an average measure of the execution time. The symbolic simplifications shown here were obtained with an implementation of the simplification procedure [12].

We first consider the tests presented in [19] where the method of the so-called *inconsistency indicators* (II) was shown to run more efficiently than previous methods, namely [18, 11] and naive constraint checking (i.e., with no simplification). We show that, on their examples, we obtain better performance (indeed, ideal simplifications). Let S_1 be the following schema⁷:

$$\left\{ \begin{array}{l} \text{mother}(X, Y) \leftarrow \text{husband}(Z, X) \wedge \text{father}(Z, Y), \\ \text{parent}(X, Y) \leftarrow \text{father}(X, Y) \vee \text{mother}(X, Y), \\ \text{wife}(X, Y) \leftarrow \text{husband}(Y, X), \\ \text{married}(X, Y) \leftarrow \text{husband}(X, Y) \vee \text{wife}(X, Y), \\ \text{employed}(X) \leftarrow \text{occup}(X, \text{serv}), \\ \text{student}(X) \leftarrow \text{occup}(X, \text{stud}), \\ \text{dependent}(X, Y) \leftarrow \text{parent}(Y, X) \wedge \text{employed}(Y) \wedge \text{student}(X), \\ \text{dependent}(X, Y) \leftarrow \text{married}(Y, X) \wedge \text{employed}(Y) \wedge \neg \text{employed}(X), \\ \text{self}(X) \leftarrow \text{married}(Y, X) \wedge \neg \text{employed}(Y), \\ \text{guardian}(X, Y) \leftarrow \text{dependent}(Y, X) \end{array} \right\},$$

$$\left\{ \begin{array}{l} \leftarrow \text{guardian}(X, Y) \wedge \neg \text{sponsor}(X, Y), \\ \leftarrow \text{married}(X_1, Y_1) \wedge \text{student}(X_1), \\ \leftarrow \text{occup}(X_2, Y_2) \wedge \text{occup}(X_2, Z) \wedge Z \neq Y_2 \end{array} \right\}$$

The distribution of facts in the initial database considered in [19] is as follows: 177 *father* facts, 229 *husband* facts, 620 *occup* facts and 59 *sponsor* facts. We considered additions of tuples to the *father* and *husband* relations. To test whether an update $U_1 = \{\text{father}(\mathbf{a}, \mathbf{b})\}$ leads to inconsistency, the II method proposes the following tests (rewritten with our notation):

$$\{\leftarrow \neg \text{sponsor}(\mathbf{a}, \mathbf{b}) \wedge \text{guardian}(\mathbf{a}, \mathbf{b}), \leftarrow \text{guardian}(X, \mathbf{b}) \wedge \neg \text{sponsor}(X, \mathbf{b})\}.$$

These can be checked by asserting the update as a Prolog fact $\text{father}(\mathbf{a}, \mathbf{b})$ and calling the Prolog query $\text{inconsistent}(\text{father}(\mathbf{a}, \mathbf{b}))$ on the Prolog program:

```
inconsistent(father(X,Y)) :- \+ sponsor(X,Y), guardian(X,Y).
inconsistent(father(Z,Y)) :- guardian(X,Y), \+ sponsor(X,Y).
```

Their checking strategy is therefore: assert the update, then retract if inconsistency was detected. The simplification given by $\text{Simp}^{U_1}(S_1)$ is more specialized and refers only to the extensional predicates:

⁶ All tests were run on a machine with a 2.4 GHz processor, 1 GB of RAM and 80 GB of hard disk. For compatibility with the compared method, the tests are run under a Prolog system (SICStus Prolog 3.11).

⁷ We use disjunctions for compactness.

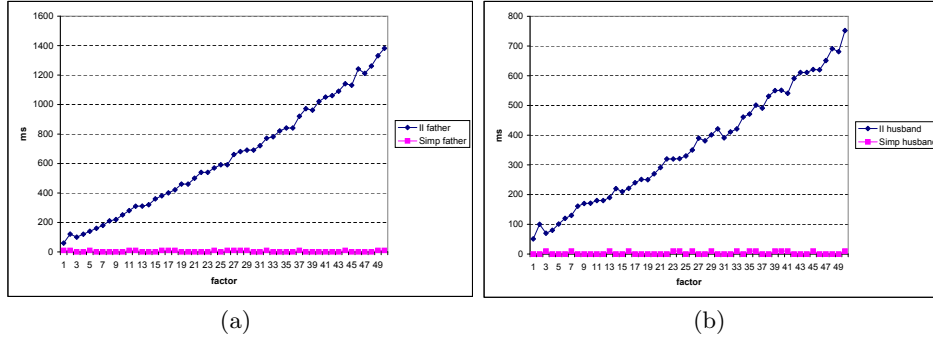


Fig. 1. Comparing Simp to the Inconsistency Indicators method

$$\begin{aligned} & \{ \leftarrow occup(\mathbf{a}, serv) \wedge occup(\mathbf{b}, stud) \wedge \neg sponsor(\mathbf{a}, \mathbf{b}), \\ & \leftarrow husband(\mathbf{a}, X) \wedge occup(X, serv) \wedge occup(\mathbf{b}, stud) \wedge \neg sponsor(X, \mathbf{b}) \}. \end{aligned}$$

Our strategy is: first test, then assert the update if inconsistency was not detected. To see whether the approaches “scale”, we ran our tests on databases that are bigger than the initial one by a given factor. Figures 1(a) and 1(b) report this factor on the X-axis and the measured average execution times (in milliseconds) for the additions of 177 *father* facts and 229 *husband* facts, respectively, with both approaches. In both tests, the performance worsens very quickly with the II method, whereas it basically remains constant with our approach.

The last example of [19] refers to the following schema S_2 :

$$\begin{aligned} \{ \{ & \leftarrow parent(X, Y) \leftarrow father(X, Y) \vee mother(X, Y), \\ & \leftarrow mother(X, Y) \leftarrow father(Z, Y) \wedge husband(Z, X), \\ & \leftarrow age(X, Y) \leftarrow civilst(X, Y, P, Q), \\ & \leftarrow dependent(X, Y) \leftarrow parent(Y, X) \wedge occup(Y, serv) \wedge occup(X, stud), \\ & \leftarrow occup(X, Y) \leftarrow civilst(X, P, Q, Y) \}, \\ & \{ \leftarrow civilst(X, Y_1, Z_1, t_1) \wedge civilst(X, Y_2, Z_2, t_2) \wedge \neg(Y_1 = Y_2 \wedge Z_1 \neq Z_2 \wedge t_1 \neq t_2), \\ & \leftarrow father(X_1, Y) \wedge father(X_2, Y) \wedge X_1 \neq X_2, \\ & \leftarrow husband(X_1, Y) \wedge husband(X_2, Y) \wedge X_1 \neq X_2, \\ & \leftarrow husband(X, Y_1) \wedge husband(X, Y_2) \wedge Y_1 \neq Y_2, \\ & \leftarrow civilst(X, Y, Z, Tax) \wedge (\neg(X > 0 \wedge X < 100000 \wedge Y > 0 \wedge Y < 125) \\ & \quad \vee (Z \neq m \wedge Z \neq f) \vee (Tax \neq stud \wedge Tax \neq ret \wedge Tax \neq biz \wedge Tax \neq serv)), \\ & \leftarrow (civilst(X, Y, Z, stud) \wedge \neg(Y < 25)) \vee (civilst(X, Y, Z, ret) \wedge \neg(Y > 60)), \\ & \leftarrow father(X, Y) \wedge (civilst(X, P, S, Q) \vee civilst(Y, P, S, Q)) \wedge S \neq m, \\ & \leftarrow husband(X, Y) \wedge (civilst(X, P, S, Q) \wedge S \neq m) \vee (civilst(Y, P, S, Q) \wedge S \neq f), \\ & \leftarrow husband(X, Y) \wedge age(X, P) \wedge age(Y, Q) \wedge (P < 20 \vee Q < 20), \\ & \leftarrow civilst(X, Y, Z, Tax) \wedge Y < 20 \wedge Tax \neq stud, \\ & \leftarrow dependent(X, Y) \wedge \neg tax(Y, X) \}. \end{aligned}$$

The update in question is a transaction of the form:

$$U_2 = \{ \leftarrow civilst(\mathbf{a}, \mathbf{p}_a, m, \mathbf{o}_a), \leftarrow civilst(\mathbf{b}, \mathbf{p}_b, f, \mathbf{o}_b), \leftarrow civilst(\mathbf{c}, \mathbf{p}_c, s_c, stud), \\ \leftarrow husband(\mathbf{a}, \mathbf{b}), \leftarrow father(\mathbf{a}, \mathbf{c}), \leftarrow tax(\mathbf{a}, \mathbf{c}) \}$$

We observe that in the example it is explicitly assumed that the added family facts were not already in the database; let us indicate this extra hypothesis as

Δ . The simplification given by the II method consists of one set of simplified constraints for every single update in U_2 . Instead, the simplification wrt the whole transaction given by $\text{Optimize}^\Delta(\text{Simp}^{U_2}(S_2))$ returns $\langle \emptyset, \emptyset \rangle$. The results of [19] have execution times that vary roughly linearly wrt the size of the database. Our simplified theory (\emptyset) is clearly a great improvement over these results, since it executes in virtually no time and guarantees, without further checking, that this transaction pattern cannot affect integrity. This example was also used in [10], where the authors, unfortunately, only compared their method to [11], but not to [19]. However, our transactional simplification is clearly unbeatable.

The same author reconsidered in [20] some of the redundancies of [19]. For the extended example discussed in [20], the schema S_3 is as follows.

$$\langle \{ \begin{array}{l} \text{mother}(X, Y) \leftarrow \text{husband}(Z, X) \wedge \text{father}(Z, Y), \\ \text{parent}(X, Y) \leftarrow \text{father}(X, Y) \vee \text{mother}(X, Y), \\ \text{agediff}(X, Y, n) \leftarrow \text{age}(X, n_1) \wedge \text{age}(Y, n_2) \wedge \text{minus}(n, n_1, n_2), \\ \leftarrow \text{parent}(X, Y) \wedge \text{agediff}(X, Y, n) \wedge n < 15 \end{array} \} \rangle$$

We tested this on the addition of *father* facts on a distribution similar to that considered for S_1 . In this case our simplifications basically correspond to the unfolding of their so-called revised inconsistency indicators (RII), so there is almost no observable difference in the execution times of the two methods. We stress, however, that the method of [20] has a much more restricted expressive power, in that the updates are limited to singleton insertions and no negations are allowed in the database. Furthermore, in this case the update was simple, so the computational effort required for assertion and retraction of facts was little; however, our approach based on early recognition of inconsistency proves yet more efficient for cases in which updates lead to illegal states (dramatically, if the transactions are complex). To see this effect we updated a small database (2 *father* facts and 2 *age* facts) with schema S_3 with an illegal *father* insertion and measured, with the RII method, an answer time approximately four times bigger than with the method based on *Simp*. This behavior is amplified as the database grows (and it is thus more expensive to add facts for the DBMS): attempting 10000 times the insertion of an illegal *father* fact on a database with approximately 5000 *father* facts took about 1s with the RII method, but only 70ms with *Simp*. This reflects the fact that with our strategy, upon an illegal update, we just perform a test, whereas the RII method requires to execute the update, perform a consistency test and then roll back the update.

The figures do not report the time employed to obtain a simplification, as this is a design time task. Yet, in our tests (with up to 20 rules, 20 ICs, 10 literals per IC and 5 literals per update) no simplification took more than 500 ms.

5.1 More on Related Work

As mentioned, the ability to check consistency of a possibly updated database *before* execution of the transaction under consideration allows avoiding inconsistent states completely, and thus rollbacks, which may require costly bookkeeping in order to restore the old state. Several approaches to simplification do not comply with this requirement [16, 11, 18, 4, 6, 19, 10]; [19] showed that his II method was

more efficient than [11, 18] and we gave evidence of great improvements obtained with `Simp wrt II`. Methods that as ours, are based on pre-tests are, e.g., [17, 9]. However, the former does not allow more than one update action in a transaction to operate on the same relation; furthermore, no mechanism corresponding to parameters is present, thus requiring to execute the procedure for each specific update. The latter provides low-cost pre-tests which are sufficient conditions that guarantee the integrity of the database; however, if the pre-tests fail (as, e.g., in simple recursive cases), nothing can be concluded about consistency and an exact test, such as ours, needs to be made. Simplification of integrity constraints with respect to given parametric update patterns resembles the notion of program *specialization*, which is the process of creating a specialized version of a given program with respect to known input data. In [10], a partial evaluation of a meta-interpreter is used to produce logic programs that correspond to simplified constraints. However, loop checks needed to ensure termination in the presence of recursion do not partially evaluate satisfactorily, resulting in an explosion of (possibly unreachable) alternatives. Integrity checking is often regarded as an instance of materialized view maintenance: integrity constraints are defined as views that must always remain empty for the database to be consistent; the book [7] provides insightful discussion on the subject. In [5], it is shown how to implement integrity constraint checking by translating first-order logic specifications into SQL triggers. It is interesting to note that the result of our transformations can be combined with similar translation techniques and thus integrated in an active database system. In this way the advantages of declarativity are combined with the efficiency of execution. The idea of embedding integrity control via semi-automatic generation of triggers (without semantic optimization) is originally due to [3].

6 Conclusion and Future Work

We applied program transformation operators to the generation of simplified ICs. A procedure was constructed that makes use of these transformations and produces the simplification searched for according to a criterion of minimality. An important contribution of this paper is the definition of the notion of ideality of a simplification procedure, its connection with the QC problem, and the analysis of different minimality criteria that can be used to characterize an ideal procedure. In particular, we showed that, in any sensible ordering in which *true* represents a minimal element, ideality of simplification corresponds to decidability of QC.

We described an implementation in terms of rewrite rules based on resolution, subsumption and replacement of specific patterns. The ability of producing a necessary and sufficient condition for checking integrity before a database update, together with the generality of the update language, constitutes the main advantage of our method with respect to earlier approaches. This was also demonstrated through a series of experiments. This work could be extended to identify more cases for which useful differential expressions exist and to integrate in the framework rewrite techniques reducing recursive problems to easier ones.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases* Addison-Wesley, 1995.
2. 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.
3. S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of VLDB 90*, pages 566–577. Morgan Kaufmann, 1990.
4. U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. on Database Syst. (TODS)*, 15(2):162–207, 1990.
5. H. Decker. Translating advanced integrity checking technology to sql. In *Database integrity: challenges and solutions*, pages 203–249. Idea Group Publishing, 2002.
6. H. Decker and M. Celma. A slick procedure for integrity checking in deductive databases. In *ICLP 94*, pages 456–469. MIT Press, Cambridge, MA, 1994.
7. A. Gupta and I. S. Mumick, editors. *Materialized views: techniques, implementations, and applications*. MIT Press, 1999.
8. D. Kapur and P. Narendran. Np-completeness of the set unification and matching problems. In *CADE*, pages 489–495, 1986.
9. S. Y. Lee and T. W. Ling. Further improvements on integrity constraint checking for stratifiable deductive databases. In *VLDB'96*, pages 495–505. Kaufmann, 1996.
10. M. Leuschel and D. de Schreye. Creating specialised integrity checks through partial evaluation of meta-interpreters. *JLP*, 36(2):149–193, 1998.
11. J. W. Lloyd, L. Sonenberg, and R. W. Topor. Integrity constraint checking in stratified databases. *JLP*, 4(4):331–343, 1987.
12. D. Martinenghi. <http://www.dat.ruc.dk/~dm/spic/index.html>, 2005.
13. D. Martinenghi. *Advanced Techniques for Efficient Data Integrity Checking*. PhD thesis, Roskilde University, Denmark, in *Datalogiske Skrifter*, 105, <http://www.ruc.dk/dat/forskning/skrifter/DS105.pdf>, 2005.
14. D. Martinenghi and H. Christiansen. Efficient integrity checking for databases with recursive views. In *ADBIS 05*, volume 3631 of *LNCIS*, pages 109–124. Springer, 2005.
15. J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *SIGMOD 89*, pages 235–242. ACM, 1989.
16. J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18:227–253, 1982.
17. X. Qian. An effective method for integrity constraint simplification. In *ICDE 88*, pages 338–345. IEEE Computer Society, 1988.
18. F. Sadri and R. Kowalski. A theorem-proving approach to database integrity. In *Foundations of Deductive Databases and Logic Programming*, pages 313–362. Kaufmann, Los Altos, CA, 1988.
19. R. Seljée. A new method for integrity constraint checking in deductive databases. *Data Knowl. Eng.*, 15(1):63–102, 1995.
20. R. Seljée and H. C. M. de Swart. Three types of redundancy in integrity checking: An optimal solution. *Data Knowl. Eng.*, 30(2):135–151, 1999.
21. O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the sixth ACM PODS symposium*, pages 237–249. ACM Press, 1987.