On the Compilation of Consistency Constraints (Extended Abstract)

Guido Moerkotte
Karl Rösch

Fakultät für Informatik
Universität Karlsruhe
D–7500 Karlsruhe 1

Abstract

We introduce a compilation technique for efficient consistency checking in deductive databases. Whereas most approaches to efficient consistency checking rely on the interpretation of the (simplified) consistency constraints we will discuss a compilation technique. In the presented approach the consistency constraints are translated into an algebra. The basic compilation procedure exhibits two main advantages. It is able to avoid the division operator in all cases and preserves the advantages of a tuple-at-a-time strategy while allowing for set-oriented processing. The basic compilation technique is then extended to capture the idea of Nicolas where the inserted or deleted facts are used to derive simplified constraints, and to check those for validity. Rather then deriving simplified constraints for each inserted or deleted fact at run time (i.e., when the consistency constraint is checked) the expressions resulting from the translation of the constraints reflect this simplifications.

1 Introduction

A database is intended to be a truthful model of a given universe of discourse (UoD) which corresponds either to a restricted part of the real world or to a mini-world. This property of being a truthful model of the UoD is referred to as integrity. As the database usually has no access to the UoD to verify its current state but instead depends on the input of human interlocutors which are bound to err a weaker notion of integrity had to be introduced: consistency. The laws and regularities observed in the UoD are modeled via consistency constraints which in general are closed first-order formulas. Then a database is called consistent if it obeys these constraints. There are two approaches to define when a database obeys the consistency constraints. Either, the constraints are a logical consequence of the database or the union of the database contents with the set of constraints is free of any contradiction.

Hypothetically any transaction may violate the consistency of a database. Thus, a consistency check has to be introduced at commit time. Efficiency is a critical feature to

these checks. Since the amount of data to be inspected during a test may be enormous the design of efficient consistency checking procedures is a major challenge. The most common approach to cut down the time needed to perform a consistency check is to decrease the amount of data visited by incorporating the idea of Nicolas ([14]) and extending it to deductive databases ([12, 11, 8, 13]). In this paper we argue that performance can be further increased by compiling the consistency constraints into an algebra. For query answering in relational databases compilation into an algebra is not only quite common but is often thought of as a prerequisite for efficiency.

Since the introduction of relational database systems in [4] many approaches to the compilation of the relational calculus (or SQL) into the relational algebra have been published (e.g. [3, 5, 6]). Recently Bry ([2]) proposed a new approach with an emphasis on the efficient treatment of quantifiers and disjunctions. His proposal shows several advantages. Mainly efficiency is gained due to the introduction of set-oriented processing while preserving the advantages of a tuple-at-a-time pipelining procedure pointed out in [15]. Further sources of efficiency are the avoidance of the division operator in many cases, and the technique of moving quantifiers inwards as far as possible. However, as we will see in section 2 this method can still be improved in several respects. The improved compilation process developed in this section will serve as a basis for the compilation and optimization of consistency constraints in the subsequent sections.

Nicolas proposal allows a major reduction of the part of the database to be inspected during the consistency check. The main idea is to utilize the difference of the original database and the database updated by the transaction. Each tuple – or fact – in this difference is then used to derive for each update relevant constraint a simplified consistency constraint. Whereas in the case of relational databases the difference can easily be computed this becomes a more severe problem in the context of deductive databases. Thus, many recent works discuss the efficient computation of this difference or of sets of literals which cover the difference (e.g. [12, 11, 13]). The difference is then used to derive the simplified constraints. Then to check the simplified constraints mostly a specialization of resolution is used (for an exception see [9]). Thus an interpretative approach to consistency checking is taken which additionally obeys the tuple-at-a-time strategy.

The paper is an extended abstract of a full paper by the authors. It does not contain the whole set of rewrite rules of the compilation procedure, proofs, the algorithm for efficiently computing the difference between two databases, and further improvements on quantifier treatment as introduced in [7]. The remainder of the paper is organized as follows. Section 2 will present our basic compilation procedure. We will define the operators needed in our algebra and the manner in which a constraint can be compiled into the algebra. Chapter 3 will give a hint how one can compile the idea of Nicolas. Section 4 concludes the paper.

2 The Basic Compilation Procedure

Before stating the basic compilation procedure we give some preliminaries and notational conventions used throughout the paper.

2.1 Preliminaries

We need the following sets of symbols: the set V of variable symbols, the set C of constant symbols, and the set P of predicate symbols. Variables are denoted by $x, y, z, \ldots$ possibly with an index. Constants are denoted by $a, b, c, \ldots$. Predicates are denoted by $p, q, r, s, \ldots$. There is an arity associated with every predicate symbol. For a predicate symbol p with arity n and constants $c_1, \ldots, c_n$ $p(c_1, \ldots, c_n)$ is a fact. A term is either a variable or a constant. Note that we do not allow function symbols. For a predicate symbol p and terms $t_1, \ldots, t_n$ $p(t_1, \ldots, t_n)$ is a positive literal, or atom. If l is a positive literal then $\neg l$ is a negative literal. A rule has the form $l_1, \ldots, l_n \Longrightarrow l_{n+1}$ for positive literals l_i. All the variables occurring in a rule are assumed to be $\forall$-quantified. We define formulas in the usual way. Every literal is a formula, if f_1 and f_2 are formulas then $f_1 \wedge f_2$, $f_1 \vee f_2$, $f_1 \Longrightarrow f_2$, and $\neg f_1$ are formulas, and for a variable symbol x $\forall x f_1$ and $\exists x f_1$ are also formulas. Be l a positive literal. Then l occurs positively in l. It occurs negatively in $\neg l$. If it occurs positively (negatively) in f_1 then it also occurs positively (negatively) in $f_1 \wedge f_2$, $f_1 \vee f_2$, and $f_2 \Longrightarrow f_1$. If l occurs positively (negatively) in f_1 then it occurs negatively (positively) in $\neg f_1$, and $f_1 \Longrightarrow f_2$. The set of free variables of a formula f is denoted by $free(f)$. For a formula of the form $\forall x_1 \ldots \forall x_n\ f'$ ($\exists x_1 \ldots \exists x_n\ f'$) we write $\forall x_1, \ldots, x_n\ f'$ ($\exists x_1, \ldots, x_n f'$). If f' is quantified we assume the first quantifier to be $\exists$ ($\forall$). For a formula f we denote by $cnf(f)$ ($dnf(f)$) its Prenexed conjunctive (disjunctive) normal form. By $\sim f$ we denote the formula which results from $\neg f$ by moving the negation inwards as far as possible. Note that quantifiers are reversed by this process, i.e., $\forall$ becomes $\exists$ and vice versa.

We use the notion of range-restrictedness as presented in [14].

Definition 2.1 (range-restrictedness) *A formula in Prenex conjunctive normal form is called range restricted iff*

- *each $\forall$-quantified variable appears in at least one negative literal (called restriction literal) in each disjunction where the variable occurs, and*
- *for each $\exists$-quantified variable x occurring in a negative literal there is a disjunction consisting only of positive literals (called restriction literals), each of them containing x.*

A database DB consists of a set of facts DB^a, a set of rules DB^d, and a set of consistency constraints DB^c, where the rules have to be range-restricted, and a consistency constraint is a closed range-restricted formula. Let l denote a fact then we define $M(DB) := \{l | DB^a \cup DB^d \models l\}$, and $C(DB) := M(DB) \cup \{\neg l | DB^a \cup DB^d \not\models l\}$. "$C(DB) \models$" is abbreviated by "$DB \models$". A database DB is called consistent iff $DB \models c$ for all $c \in DB^c$.

A substitution σ_X is a mapping of a set of variables X into the set of constants. The application of a substitution to a formula replaces every occurrence of a free variable $x \in X$ by its image under σ_X. For a formula f the application of σ_X to f is denoted by $f\sigma_X$. All variables not in X are left untouched. The empty substitution ($\sigma_\emptyset$) is denoted by ϵ. Since we will deal with sets of substitutions extensively note the difference between $\{\epsilon\}$ and $\emptyset$ which often represents the distinction between *true* and *false*.

To emphasize that f is a formula in free variables $X := \{x_1, \ldots, x_n\}$ we will write $f(x_1, \ldots, x_n)$. This is of course unnecessary for a literal. Further the expression $f[x_1, \ldots, x_n]$

is defined to evaluate to the set of substitutions $\{\sigma_X | DB \models f\sigma_X\}$. If $X = \dot{X} \dot{\cup} \bar{X}$ ($\dot{\cup}$ denotes the disjunctive union) then $f[\dot{X}] := \{\sigma_{\dot{X}} | \exists \sigma_{\bar{X}}\ DB \models f\sigma_{\dot{X}}\sigma_{\bar{X}}\}$. For a literal $l = p(x_1, \ldots, x_n)$ we write $p[x_1, \ldots, x_n]$ instead of $p(x_1, \ldots, x_n)[x_1, \ldots, x_n]$.

2.2 Compilation

Our compilation procedure is based on proposition 4 of [2] where five example formulas are translated. We agree with the translation of all the examples but the one stated in point 5 of proposition 4. There are two arguments against the suggested translation. First it is not correct, especially it does not work if one of the relations (T) is the empty relation, i.e., its extension does not contain any tuples. Second the division operator is used at this place despite the fact that it could be avoided here. Further no general method to compile any range-restricted formula is presented but instead a possible translation for six examples is given. For the treatment of disjunctions the use of an outer-join operator is suggested. We will avoid this and use the more efficient union brackets (see below) instead.

We now introduce the compilation process. An arbitrary formula is translated via the compilation mapping $\mathcal{E}$ into an expression mainly consisting of a sequence of literals connected by operators. The mapping $\mathcal{E}$ has three parameters guiding the compilation process: a set of input variables, the formula to be compiled, and a set of output variables. The compilation can be interpreted as follows. Given a set of substitutions binding the variables of the input set the result of the compilation process is an expression which evaluates to a set of bindings for the output variables s.t. the image of the formula under these substitutions is derivable from the database. More formally, if a set of variables $X = \dot{X} \dot{\cup} \bar{X}$, a set of substitutions $S[X]$, and a formula f with free variables in X are given then

$$\text{the expression } S[X]\mathcal{E}(X, f, \dot{X}) \text{ evaluates to } \{\sigma_{\dot{X}} | \sigma_{\dot{X}}\sigma_{\bar{X}} \in S[X], DB \models f\sigma_{\dot{X}}\sigma_{\bar{X}}\}.$$

The operators possibly occurring in the expression resulting from the compilation process $\mathcal{E}$ are a generalized join, union, and intersection. The generalized join operator will optionally have two output sets. The first output consists of the join of the two arguments, and the second output consists of their complement-join ([2]). For each output a set of variables specifies the kind of substitutions to be passed to the subsequent operator(s). The first output is directed to the subsequent operator whereas the second is collected by a union bracket (see below). Since we stick to a linear notation of the expressions it is sometimes necessary to exchange the two output streams, i.e., the second output is directed to the subsequent operator whereas the first one is gathered by the enclosing union bracket. Note that we allow two outputs to be specified only within an enclosing union bracket. Within a union bracket (denoted by $[\![\ldots]\!]$) we allow several operators which compute two output streams. All the outputs of these operators are collected and unioned by the union operator and after the end of the bracket (denoted by the symbol $]\!]$) feed into the pipeline again.

We recast the definition of a join into substitution terms. Additionally we allow the join to have two outputs denoted by *first output* and *second output* respectively.

Definition 2.2 ($\bowtie$) *Define the following sets of variables* $X := \{x_1, \ldots, x_n\}$, $Y := \{y_1, \ldots, y_m\}$, $\dot{X}$, $\bar{X}$, $\dot{Y}$ *and* $\bar{Y}$ *such that* $\dot{X} \dot{\cup} \bar{X} = X$, $\dot{Y} \subseteq Y \setminus X$, $\bar{Y} = (Y \setminus X) \setminus \dot{Y}$. *Fur-*

ther let e be an expression evaluating in a set of bindings for the variables X and g an expression in free variables Y. Then we define the **first** *output of the join operator in the expression* $e[x_1,\ldots,x_n] \bowtie_{\dot{X},\dot{Y}} g(y_1,\ldots,y_m)$ *as*

$$\{\sigma_{\dot{X}}\sigma_{\dot{Y}} | \exists\sigma_{\bar{X}}\ \sigma_{\dot{X}}\sigma_{\bar{X}} \in e[x_1,\ldots,x_n] \wedge \exists\sigma_{\bar{Y}}\ DB \models g\sigma_{\dot{X}}\sigma_{\bar{X}}\sigma_{\dot{Y}}\sigma_{\bar{Y}}\}.$$

The **second** *output of the join operator is defined as:*

$$\{\sigma_{\dot{X}} | \exists\sigma_{\bar{X}}\ \sigma_{\dot{X}}\sigma_{\bar{X}} \in e[x_1,\ldots,x_n] \wedge \neg\exists\sigma_{\dot{Y}\cup\bar{Y}}\ DB \models g\sigma_{\dot{X}}\sigma_{\bar{X}}\sigma_{\dot{Y}\cup\bar{Y}}\}.$$

The second output of the join operator is only computed inside the union brackets. Sometimes it will be usefull to interchange the two outputs. This is indicated by overlining the $\bowtie$ *operator, i.e., the first output of* $\overline{\bowtie}_{\dot{X},\dot{Y}}$ *is the second output of* $\bowtie_{\dot{X},\dot{Y}}$ *and vice versa.*

Note that if $\dot{Y} = \emptyset$ the join operator can be implemented more efficiently using the semi-join, and that the second output of the join is the complement-join ([2]) of its inputs which always can as efficiently be computed as a semi-join, and can additionally be computed during the computation of the join without further computational costs. Besides the union operator ($\cup$), and the intersection operator ($\cap$) we add the union-brackets to our algebra.

Definition 2.3 ([]) *Let s be a sequence of join operators and their arguments. Then the output of* $[\![s]\!]$ *is the union of all second outputs of the operators of s.*

Example 2.4 *Consider a database with*

$DB^a = \{r(a), r(b), r(c), r(d), p(b,f), p(d,h), q(c,g), q(d,h)\}$,

and the following expression

$r[x][\![\overline{\bowtie}_{\{x\},\{y\}} p(x,y) \overline{\bowtie}_{\{x\},\{y\}} q(x,y)]\!]$.

then the subexpressions evaluate as follows:

$r[x] =$
$\{\{x \leftarrow a\}, \{x \leftarrow b\}, \{x \leftarrow c\}, \{x \leftarrow d\}\}$

$r[x]\overline{\bowtie}_{\{x\},\{y\}} p(x,y) =$
1. output: $\{\{x \leftarrow a\}, \{x \leftarrow c\}\}$
2. output: $\{\{x \leftarrow b, y \leftarrow f\}, \{x \leftarrow d, y \leftarrow h\}\}$

$r[x]\overline{\bowtie}_{\{x\},\{y\}} p(x,y) \overline{\bowtie}_{\{x\},\{y\}} q(x,y) =$
1. output: $\{\{x \leftarrow a\}\}$
2. output: $\{\{x \leftarrow c, y \leftarrow g\}\}$

$r[x][\![\overline{\bowtie}_{\{x\},\{y\}} p(x,y) \overline{\bowtie}_{\{x\},\{y\}} q(x,y)]\!] =$
$\{\{x \leftarrow b\}, \{x \leftarrow c\}, \{x \leftarrow d\}\}$
(projected on x)

We are prepared to state the compilation procedure. Since it is indeterministic the possible alternatives should be evaluated by a cost model (whose development is beyond the scope of the paper), and then the best alternative should be chosen. We state the indeterministic compilation procedure in terms of rewrite rules. The definition of $\mathcal{E}$ is distributed over two definitions. The first definition contains the start and the end of the compilation process, the second definition deals with quantified formulas. Several examples are given after the two definitions.

Definition 2.5 *Let f be a formula in free variables $x_1, \ldots, x_n$. We then define:*

$\mathcal{E}(\emptyset, \forall x_1, \ldots, x_n f, \emptyset) \rightarrow \mathcal{E}(\emptyset, \sim f, \dot{X})$
$\mathcal{E}(\emptyset, \exists x_1, \ldots, x_n f, \emptyset) \rightarrow \mathcal{E}(\emptyset, f, \dot{X})$
$\mathcal{E}(v_{in}, \Lambda, v_{out}) \rightarrow \Lambda$

where $\dot{X} \subseteq \{x_1, \ldots, x_n\}$, and Λ denotes the empty word.

Definition 2.6 *Let f be a formula and $l_1, \ldots, l_l$ literals. Then we define the compilation of $\mathcal{E}(v_{in}, f, v_{out})$ according to the structure of f:*
(We do not give the definitions on $\dot{X}_i$ and $\dot{Y}_i$ which merely state to keep exactly those bindings at every operator occurrence which are needed for future operators, or the output. All the other conditions reflect the notion of range-restrictedness. Further explanations on the conditions as well as further rewrite rules are given in the full paper.)

1. $\mathcal{E}(v_{in}, \exists x_1, \ldots, x_n Q f', v_{out}) \rightarrow$
   $op_1\ l_{1'}\ \ldots\ op_l\ l_{l'}\ \mathcal{E}(v'_{in}, \exists x_{i_1}, \ldots, x_{i_m} Q f'', v_{out})$
   where:
   (a) the literals $l_{j'}$ $(1 \leq j \leq l)$ occur in every conjunction of f' and can be both positive and negative
   (b) if $l_{i'}$ is a positive literal then op_i is $\ltimes_{\dot{X}_i, \dot{Y}_i}$
   (c) if $l_{i'}$ is a negative literal then op_i is $\bar{\ltimes}_{\dot{X}_i, \dot{Y}_i}$
   (d) f'' results from f' by deleting the $l_{i'}$ $(1 \leq j \leq l)$ and subsequent simplification
2. $\mathcal{E}(v_{in}, \exists x_1, \ldots, x_n l_1 \wedge \ldots \wedge l_l, v_{out}) \rightarrow op_1\ l_{1'} \cdots op_l\ l_{l'}$
   where:
   (a) $l_{1'}, \ldots, l_{l'}$ are a permutation of $l_1, \ldots, l_l$
   (b) if $l_{i'}$ is a positive literal then op_i is $\ltimes_{\dot{X}_i, \dot{Y}_i}$
   (c) if $l_{i'}$ is a negative literal then op_i is $\bar{\ltimes}_{\dot{X}_i, \dot{Y}_i}$
3. $\mathcal{E}(v_{in}, \exists x_1, \ldots, x_n l_1 \vee \ldots \vee l_l, v_{out}) \rightarrow [\![op_1\ l_{1'} \cdots op_l\ l_{l'}]\!]$
   where:
   (a) $l_{1'}, \ldots, l_{l'}$ are a permutation of $l_1, \ldots, l_l$
   (b) if $l_{i'}$ is a positive literal then op_i is $\bar{\ltimes}_{\dot{X}_i, \dot{Y}_i}$
   (c) if $l_{i'}$ is a negative literal then op_i is $\ltimes_{\dot{X}_i, \dot{Y}_i}$
4. $\mathcal{E}(v_{in}, \forall x_1, \ldots, x_n l_1 \wedge \ldots \wedge l_l, v_{out}) \rightarrow op_1\ l_{1'} \cdots op_l\ l_{l'}$
   If the following conditions hold:
   (a) $l_{1'}, \ldots, l_{l'}$ is a permutation of $1, \ldots, l$.
   (b) If $l_{i'}$ is a positive literal then $op_{i'}$ is $\ltimes_{\dot{X}, \dot{Y}}$
   (c) If $l_{i'}$ is a negative literal then $op_{i'}$ is $\bar{\ltimes}_{\dot{X}, \dot{Y}}$.

5. $\mathcal{E}(v_{in}, \forall x_1, \ldots, x_n l_1 \vee \ldots \vee l_l, v_{out}) \rightarrow [\![op_1\ l_{1'} \cdots op_l\ l_{l'}]\!]$
   If the following conditions hold:

   (a) $l_{1'}, \ldots, l_{l'}$ is a permutation of $1, \ldots, l$.

   (b) If $l_{i'}$ is a positive literal then $op_{i'}$ is $\overline{\bowtie}_{\dot{X},\dot{Y}}$

   (c) If $l_{i'}$ is a negative literal then $op_{i'}$ is $\bowtie_{\dot{X},\dot{Y}}$.

If $v_{in} = \emptyset$ then op_1 is missing, and $l_{1'} = p_{1'}(z_1, \ldots, z_o)$ is replaced by $l'_{1'} = p_{1'}[z_1, \ldots, z_o]$. This only holds for the exist quantified cases.
In a similar manner one can define the compilation for general formulas. The complete definitions are given in the full paper.

Using $\mathcal{E}$ the examples of proposition 4 of [2] can be compiled as follows:

Example 2.7 *We now compile the examples of Bry. Recognize especially the last one which in Bry's paper needs the division operator and is, thus, incorrect for t being empty. Assume the existence of a set of bindings S for the variable x then*

- $S[x]\mathcal{E}(\{x\}, \exists y \exists z r(x,y) \wedge s(x,y,z) \wedge g(x,y,z), \{x\})$
  $\overset{def2.6(2)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) \bowtie_{\{x,y\},\{z\}} s(x,y,z) \bowtie_{\{x\},\emptyset} g(x,y,z)$

- $S[x]\mathcal{E}(\{x\}, \exists y \exists z r(x,y) \wedge s(x,y,z) \wedge \neg g(x,y,z), \{x\})$
  $\overset{def2.6(2)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) \bowtie_{\{x,y\},\{z\}} s(x,y,z) \overline{\bowtie}_{\{x\},\emptyset} g(x,y,z)$

- $S[x]\mathcal{E}(\{x\}, \exists y \exists z r(x,y) \wedge t(y,z) \wedge \neg g(x,y,z), \{x\})$
  $\overset{def2.6(2)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) \bowtie_{\{x,y\},\{z\}} t(y,z) \overline{\bowtie}_{\{x\},\emptyset} g(x,y,z)$

- $S[x]\mathcal{E}(\{x\}, \exists y \forall z r(x,y) \wedge (\neg s(x,y,z) \vee \neg g(x,y,z)), \{x\})$
  $\overset{def2.6(1)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) \mathcal{E}(\{x,y\}, \forall z(\neg s(x,y,z) \vee \neg g(x,y,z), \{x\})$
  $\overset{def2.6(5)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) [\![\bowtie_{\{x,y\},\{z\}} s(x,y,z) \bowtie_{\{x\},\emptyset} g(x,y,z)]\!]$

- $S[x]\mathcal{E}(\{x\}, \exists y \forall z r(x,y) \wedge (\neg s(x,y,z) \vee g(x,y,z)), \{x\})$
  $\overset{def2.6(1)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) \mathcal{E}(\{x,y\}, \forall z(\neg s(x,y,z) \vee g(x,y,z), \{x\})$
  $\overset{def2.6(5)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) [\![\bowtie_{\{x,y\},\{z\}} s(x,y,z) \overline{\bowtie}_{\{x\},\emptyset} g(x,y,z)]\!]$

- $S[x]\mathcal{E}(\{x\}, \exists y \forall z r(x,y) \wedge (\neg t(y,z) \vee g(x,y,z)), \{x\})$
  $\overset{def2.6(1)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) \mathcal{E}(\{x,y\}, \forall z(\neg t(y,z) \vee g(x,y,z), \{x\})$
  $\overset{def2.6(5)}{\rightarrow} S[x] \bowtie_{\{x\},\{y\}} r(x,y) [\![\bowtie_{\{x,y\},\{z\}} t(y,z) \overline{\bowtie}_{\{x\},\emptyset} g(x,y,z)]\!]$

We now state the basic propositions for the compilation process:

Proposition 2.8 *For a set S of substitutions with domain $\{x_1, \ldots, x_n\}$, and a range-restricted formula f in free variables $x_1, \ldots, x_n$ the evaluation of $S[X]\mathcal{E}(X, f, \dot{X})$ results in*

$$\{\sigma_{\dot{X}} | \exists \sigma_{\bar{X}}\ \sigma_{\dot{X}} \sigma_{\bar{X}} \in S[X] \wedge DB \models f\sigma_{\dot{X}} \sigma_{\bar{X}}\}$$

Proposition 2.9 *For a closed range-restricted formula* $f = \exists x_1, \ldots, x_n f'$ *the evaluation of* $\mathcal{E}(\emptyset, f, \{x_1, \ldots, x_n\})$ *results in*

$$\{\sigma_{x_1,\ldots,x_n} | DB \models f'\sigma_{x_1,\ldots,x_n}\}.$$

For a closed range-restricted formula $f = \forall x_1, \ldots, x_n f'$ *the evaluation of* $\mathcal{E}(\emptyset, \sim f, \{x_1, \ldots, x_n\})$ *results in*

$$\{\sigma_{x_1,\ldots,x_n} | DB \not\models f'\sigma_{x_1,\ldots,x_n}\}$$

Theorem 2.10 shows how one can deal with consistency constraints in a naive manner.

Proposition 2.10 *For a closed range-restricted formula* $f = \exists x_1, \ldots, x_n f'$

$DB \models f$ **iff** $\mathcal{E}(\emptyset, f, \emptyset)$ *evaluates to a non-empty set.*

For a range-restricted formula $f = \forall x_1, \ldots, x_n f'$

$DB \models f$ **iff** $\mathcal{E}(\emptyset, f, \emptyset)$ *evaluates to the emptyset.*

3 Compiling the Idea of Nicolas

For the rest of the paper we assume the existence of a consistent database DB_{old} whose fact base DB^a_{old} is modified by a user via a transaction resulting in the possibly inconsistent database DB_{new}. Thus we do not allow to update the rule set or the set of consistency constraints. We define the two parts of the Δ operator to capture the (possibly implicitly) added facts by Δ^+, and the (possibly implicitly) deleted facts by Δ^- respectively.

Definition 3.1 *Given two databases* DB_{old} *and* DB_{new} *an n-ary predicate p, and sets* $\dot{X} \dot{\cup} \bar{X} = \{x_1, \ldots, x_n\}$ *of variables we define the* Δ*-operator as follows:*

$$\begin{aligned}
\Delta^+_{\dot{X}}(p(x_1, \ldots, x_n)) := \{\sigma_{\dot{X}} | \exists \sigma_{\bar{X}} DB_{old} \not\models p(x_1, \ldots, x_n)\sigma_{\dot{X}}\sigma_{\bar{X}} \text{ and} \\
DB_{new} \models p(x_1, \ldots, x_n)\sigma_{\dot{X}}\sigma_{\bar{X}}\} \\
\Delta^-_{\dot{X}}(p(x_1, \ldots, x_n)) := \{\sigma_{\dot{X}} | \exists \sigma_{\bar{X}} DB_{old} \models p(x_1, \ldots, x_n)\sigma_{\dot{X}}\sigma_{\bar{X}} \text{ and} \\
DB_{new} \not\models p(x_1, \ldots, x_n)\sigma_{\dot{X}}\sigma_{\bar{X}}\}
\end{aligned}$$

All the approaches cited in the introduction rely on the computation of the Δ operators or some approximation which captures the Δ operations, i.e., they compute a set of literals subsuming each literal in $\Delta^{\pm}$. In the full paper we show how the Δ operator can be computed efficiently.

From now on we assume $\mathcal{E}$ to result in a single expression. We define the compilation process as derived by incorporating Nicolas' idea as follows. We only give the definition for literals occurring negatively in the considered consistency constraints. The other half of the definition is given in the full paper.

Definition 3.2 *Let* $f = \forall x_1, \ldots, x_n Q f'$ *be a consistency constraint. We define the following set* $\mathcal{N}(f)$ *of expressions:*

- *For each negatively occurring literal* l *in* f *s.t.* $\dot{X} := \{x'_1, \ldots, x'_m\} := free(l) \cap \{x_1, \ldots, x_n\} \neq \emptyset$

$$\Delta^+_{\dot{X}}(l)\mathcal{E}(\dot{X}, \forall x'_1, \ldots, x'_m Q f' \setminus l, \dot{X}) \in \mathcal{N}(f)$$

where if $free(l) \subseteq \{x_1, \ldots, x_n\}$ *then* $f' \setminus l$ *results from* f' *by replacing* l *by true and subsequent simplification else* $f' \setminus l := f'$.

- *For each negatively occurring literal* l *in* f *s.t.* $free(l) \cap \{x_1, \ldots, x_n\} = \emptyset$

$$\Delta^+_{\emptyset}(l)\mathcal{E}(\emptyset, f, \emptyset) \in \mathcal{N}(f)$$

We give $\mathcal{N}(f)$ for example 6 of [14].

Example 3.3 *Let* f *be the formula* $\forall x \forall y \forall z subord(x,z) \wedge subord(z,y) \Longrightarrow subord(x,y)$. *Then*

$$\begin{aligned}\mathcal{N}(f) = \{ & \Delta^+_{\{x,z\}}(subord(x,z))\mathcal{E}(\{x,z\}, \forall y subord(z,y) \Longrightarrow subord(x,y), \{x,y\}), \\ & \Delta^+_{\{z,y\}}(subord(z,y))\mathcal{E}(\{z,y\}, \forall x subord(x,z) \Longrightarrow subord(x,y), \{x,y\})\} \\ = \{ & \Delta^+_{\{x,z\}}(subord(x,z))[\![\bowtie_{\{x,z\},\{y\}} subord(z,y) \overline{\bowtie}_{\emptyset,\emptyset} subord(x,y)]\!], \\ & \Delta^+_{\{z,y\}}(subord(z,y))[\![\bowtie_{\{z,y\},\{x\}} subord(x,z) \overline{\bowtie}_{\emptyset,\emptyset} subord(x,y)]\!] \}.\end{aligned}$$

Note in the above example that the Δ^+ operator should be factored out in order to avoid duplicate computation.

Proposition 3.4 *Let* DB_{old} *be a consistent database and* DB_{new} *an updated database. For each consistency constraint* f *the following holds:* $DB_{new} \models f$ *only if each expression in* $\mathcal{N}(f)$ *evaluates to a nonempty set of substitutions.*

The *if* part of the proposition comes with the other half of the definition of $\mathcal{N}$ treating literals occurring positively in the considered consistency constraints. This is discussed in the full paper.

For consistency constraints beginning with an $\exists$-quantifier there is no simplification possible. But analogous pretests to those given in the last point of definition 3.2 can be easily stated for $\exists$-quantified constraints. More powerful pretests and quantifier treatments are presented in the full paper. There, the ideas of [7] are incorporated.

4 Conclusion

We have presented a technique to compile consistency constraints into an algebra. The domain of the algebra is the set of sets of substitutions rather than the set of relations. Besides the union and intersection operator this algebra consists of the generalized join operator which has two outputs, and the union bracket which gathers for a sequence of join operators their second outputs. The basic compilation technique exhibits two main advantages. It avoids the division operator in all cases, and allows set oriented processing in a pipelining manner while preserving the advantages of a tuple-at-a-time strategy. For efficient treatment of consistency constraints we enhanced this algebra by the Δ operator for which an efficient implementation is given in the full paper. The compilation technique was then applied to capture Nicolas' idea.

There are two main directions for further research. Potentially, after each transaction the whole set of consistency constraints has to be checked. Further, our compilation

technique generates more than one expression for each constraint. Thus, it is worthwhile to check for common subexpressions. Since the expressions in our algebra represent roughly a pipeline techniques like factorization (see e.g. [10]) should be applied to factor out common starting sequences. Most critical to the efficiency still are existentially quantified constraints when they have to be checked, e.g., if the pretests fail, since no simplifications are possible without any further information. Thus, redundant data should be used to allow for faster checking (see e.g [1]).

Acknowledgment. We thank Alfons Kemper for fruitful comments.

References

[1] P. Bernstein, B. Blaustein, and E. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *Proc. 6th Int. Conf. VLDB*, pages 126–136, 1980.

[2] F. Bry. Towards an efficient evaluation of general queries: Quantifiers and disjunction processing revisited. In *Proc. of the 18nth ACM SIGMOD*, pages 193–204, 1989.

[3] S. Ceri and G. Gottlob. Translating SQL in relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Trans. on Software Engineering*, 11(4), 1985.

[4] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[5] U. Dayal. Processing queries with quantifiers: A horticultural approach. In *ACM Symp. on Principles of Database Systems*, pages 125–136, 1983.

[6] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[7] I. Kobayashi. Validating database updates. *Information Systems*, 9(1):1–17, 1984.

[8] R. Kowalski, F. Sadri, and P. Soper. Integrity checking in deductive databases. In *Proc. 13th Int. Conf. VLDB*, pages 61–69, 1987.

[9] C. Kung. A tableaux approach for consistency checking. *in: A. Sernadas, J. Bubenko, A. Olive (eds.), Information Systems: Theoretical and Formal Aspects, North Holland*, pages 191–207, 1985.

[10] A. Lefebre and L. Vieille. On deductive query evaluation in the dedgin* system. In *Proc. 1st. Int. Conf. on Deductive and Object-Oriented Databases*, pages 225–246, 1989.

[11] J.W. Lloyd, Sonenberg, and R.W. E.A.Topor. Integrity constraint checking in stratified databases. *J. Logic Programming*, 4:331–343, 1987.

[12] J.W. Lloyd and R.W. Topor. A basis for deductive database systems. *J. Logic Programming*, 2:93–109, 1985.

[13] G. Moerkotte and S. Karl. Efficient consistency checking in deductive databases. In *2nd. Int. Conf. On Database Theory*, 1988. 118-128.

[14] J.-M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18, 1982. 227-253.

[15] S.B. Yao. Optimization of query evaluation algorithms. *ACM Trans. on Database Systems*, 4(2):133–155, 1979.