An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases

Françoise Fabret, Mireille Regnier and Eric Simon
INRIA - 78153 Le Chesnay, FRANCE

Abstract

Several incremental algorithms have been proposed to evaluate database production rule programs. They all derive from existing incremental algorithms, like RETE and TREAT, developed for rule-based systems in the framework of Artificial Intelligence. In this paper, we address a specific but crucial problem that arises with these incremental algorithms: how much data should be profitably materialized and maintained in order to speed-up program evaluation ? We show that the answer exposes to a well known tradeoff. Our major contribution is to propose an adaptive algorithm that takes as input a program of rules and returns for each rule, the set of most profitable relational expressions that should be maintained in order to obtain a good compromise. A notable feature of our algorithm is that it works for both set-oriented and instance-oriented rules. We compare our algorithms with existing incremental algorithms for database production rule programs.

Keywords: *incremental algorithms, production rules, database rule language processing, rule program optimization.*

1 Introduction

Production rules have demonstrated to be a powerful programming paradigm to specify queries, views, integrity constraints and triggers in a database system [DE88], [SLR88], [KdMS90], [SKdM92], [SJGP90], [Han92], [WCL91]. However, the problem of efficiently supporting rules in a database system still constitutes a major challenge to establish the viability of rule-based technology. A few papers have addressed this problem by adapting algorithms that were designed by AI researchers to optimize OPS5 rule programs in a main memory environment.

Most database rule systems (see [HW92]) follow a *recognize-act-cycle* similar to that of OPS5 [BFKM85]. A production rule consists of an action that must be executed whenever a condition over the database holds. In active database rule languages, an event can be associated with the (condition, action) pair. Usually, the action is a set of operations on the database, that is insertions, deletions, and updates. Executing a rule program proceeds by (i) evaluating rule's conditions against the database, (ii) choosing one rule instantiation whose condition is satisfied, (iii) executing the action of the selected rule, and repeating the cycle until a fixed-point is reached (if any). The set of tuples that satisfies a rule condition in a given database state is called a *satisfying rule instantiation.*

A critical part of rule evaluation is the match phase (phase (i) above), where satisfying rule instantiations are computed. A naive algorithm would execute the query associated with each rule's condition against the entire database on each cycle. This approach rapidly becomes intractable as soon as the number of rules and facts gets large. To overcome this problem, *incremental* algorithms that maintain state information from cycle to cycle have been proposed by AI researchers. The two main known algorithms that fall into this category are RETE [For82] and TREAT [Mir87].

The RETE algorithm maintains the result of the first matching phase into an ad-hoc data structure called a *discrimination network.* In this structure, the result of each select and join operation that occurs in every rule's condition is recorded and stored in some node. Results of selections are stored in *alpha-memory* nodes and form the input portion of the discrimination network. The results of joins are stored in *beta-memory* nodes. Nodes corresponding to the select and join operations that occur in the same rule's condition are connected

Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993

together in the form of a dataflow network. Satisfying rule instantiations are stored in the output nodes of the network.

RETE transforms the recognize-act-cycle above as follows. Phase (i) is removed from the cycle and becomes an *initialization* phase during which the discrimination network is built. A new phase is then inserted in the cycle after phase (iii). It propagates every change to the database done by a rule's action, (expected to be very small compared to the size of the database), towards the appropriate alpha and beta-memories. Thus, the set of rule instantiations is *incrementally* (or *differentially*) maintained. The TREAT algorithm essentially differs from RETE by the fact that it does not maintain beta-memory nodes. TREAT only maintains alpha-nodes and the set of all the satisfying rule instantiations.

Our starting point for this research is that incremental algorithms have an inherent *tradeoff*. Intuitively, maintaining some node N in a discrimination network is *profitable* if the work spent in computing the changes to the operands involved in the expression that computes N and in updating the current value of N from its previous value is less than that spent in computing it directly from its operands. The three important factors in this type of improvement are, (1) the ability to efficiently compute the *differential change* to N for arbitrary expressions, (2) the boundedness of this differential, and (3) the number of times N is used with regard to the number of times N is changed.

In a sense, RETE and TREAT are two opposite answers to this tradeoff. RETE must do as much work to maintain memory nodes whenever tuples are deleted from the database as it does to maintain memory nodes whenever tuples are inserted. TREAT does much less work when tuples are deleted from the database since it does not have to update the intermediate join results. Only the final set of rule instantiations has to be updated. But TREAT does do more work than RETE when tuples are inserted because rule conditions have to be reevaluated for these new tuples.

The purpose of this paper is to systematically investigate this problem of profitability of incremental algorithms for rule-based programs. Our main contribution is to propose an adaptative algorithm that given a rule program chooses the better compromise of discrimination network in between RETE and TREAT. Following previous work on the design of incremental algorithms in the framework of high-level programming languages such as SETL ([PK82], [Pai83]), our algorithm relies on a heuristic-based characterization of the notion of profitability. More, our algorithm is based on a careful analysis of the migration flow of data in a rule program.

Usually, a rule language has either set-oriented (e.g., Ariel, RDL, Starburst) or instance-oriented rules (e.g., OPS5, RPL, PRSII). We show that this difference of semantics greatly influences the choice of a good discrimination network. A notable feature of our algorithm is that it works for both set-oriented and instance-oriented rules.

This paper is structured as follows. Section 2 presents our rule language, introduces useful terminology, and relates our language to OPS5. In Section 3, we describe RETE and TREAT algorithms and exhibit their inherent tradeoff. In Section 4, we characterize autonomous expressions, that can be *easily* differentiated with respect to some changes to their operands. Then, we define when a relation is profitable to maintain and give an algorithm that extracts maximal profitable relations from a given conjunctive formula. Section 5 presents our main adaptative algorithm, which given a set of rules determines the better discrimination network to build. Section 6 compares our work with other related work. Finally, Section 7 concludes the paper.

2 The Rule Language

In order to make the presentation of our algorithms general and independent from a particular rule-based language syntax, we shall use a Datalog-like notation for rules. From a semantic point of view, we distinguish set-oriented rules from instance-oriented rules.

2.1 Terminology

We briefly review some terminology. A *fact* over a predicate Q of arity n is an expression $Q(a_1, ..., a_n)$ where each a_i is a constant. A *database schema* is a finite set of predicates. A *(database) instance* over a schema $\mathcal{S}$ is a finite set of facts over predicates in $\mathcal{S}$. If I_k is a set of facts and Q a predicate in $\mathcal{S}$, $Q^{(k)}$ denotes the set of facts over Q in I_k. A *literal* is an expression of the form $(\neg)Q(x_1, ..., x_m)$ where $m \geq 0$, Q is a predicate of arity m and each x_i is either a variable or a constant. An *eq-literal* is an expression of the form $(\neg)x_1 = x_2$ where x_1, x_2 are variables or constants. We shall use letters x, y, z, ... to denote variables, and greek letters to denote constants.

Definition 2.1 A *rule* is an expression of the form

$$B_1, ..., B_n \rightarrow A_1, ..., A_k \ (k \geq 1, n \geq 0),$$

where each A_j is a literal and each B_i is a literal or an eq-literal. A *rule program* is a finite set of rules.

We consider that there may exist a partial ordering between rules where $r \leq r'$ means that if r and r' are both firable at the same time, then r has priority over

r'. This ordering is assumed to be available at any time during program execution. We denote $(\Gamma, \leq)$ a rule program with a priority ordering between rules.

2.2 Set-oriented Rules

A rule can be interpreted using either an instance-oriented or a set-oriented semantics. We start with the last one.

Effect of a rule on a database instance: Let r be a rule: $body \rightarrow A_1, ..., A_p, \neg B_1, ..., \neg B_n$, and I a set of facts. Let r' be a ground instance of r such that (i) each positive literal of the body of r' is a fact in I and each negative literal or eq-literal in the body of r' holds, and (ii) each variable is valuated to some constant occuring in I. Then r' is said to be a *satisfying instance* of r in I. Now, the set of ground literals in the heads of all the satisfying instances of r in I is called the *effect of* r on I, denoted $effect_r(I)$.

Intuitively, the effect of rule r on I is the global effect obtained by considering the actions of all the satisfying instances of r in I.

Definition 2.2 A rule r is said to be *firable* in a state I if there exists a fact A such that either

- $A \in effect_r(I)$, $\neg A \notin effect_r(I)$ and $A \notin I$, or
- $\neg A \in effect_r(I)$, $A \notin effect_r(I)$ and $A \in I$

Thus, a rule is firable if its firing would produce a *net* change to the current database state.

Semantics of a rule program: Let $(\Gamma, \leq)$ be a rule program. This program defines a *relation* among database instances as follows. For each state I, J is *reachable* from I using Γ if there is a firable rule r in Γ such that no other firable rule has priority over r, and J consists of the facts A such that:

- A is in $I \cup effect_r(I)$ and $\neg A$ is not in $effect_r(I)$ or
- A is in I and A, $\neg A$ are both in $effect_r(I)$.

If a sequence of reachable states has a limit, it is called a *fixpoint* of the program. Notice that a program may have several fixpoints or no fixpoint at all.

Definition 2.3 Given a rule program and a state I, the *conflict set* is defined to be the set of all the firable rules in I.

2.3 Instance-oriented Rules

A set-oriented rule is fired for all its instantiations in one step whereas an instance-oriented rule is fired for one instantiation in one step. Thus, the difference is that we consider the effect of a satisfying rule instance alone. If r' is a satisfying instance of r, we denote the effect of r' on I by $effect_{r'}(I)$. We define r as a firable rule if there exists a satisfying *firable* instance r' of r that obeys the conditions of Definition 2.2., where $effect_r(I)$ is replaced by $effect_{r'}(I)$. Then, the semantics of a rule program is defined as for set-oriented rules except that r is replaced by a satisfying instance r' and $effect_r(I)$ is replaced by $effect_{r'}(I)$. Finally, the conflict set is defined to be the set of all the firable satisfying instances of rules in a state I.

2.4 Comparisons with OPS5-like Languages

OPS5 [BFKM85] is the underlying language for which RETE and TREAT have initially been designed. Compared to our language, OPS5 has three main differences: facts are timestamped, the language has a refraction-based semantics, and priorities beetwen rules are dynamic.

In spite of these differences of semantics, our final algorithm is applicable to OPS5-like languages. The main reason is that we seek to optimize the specific phase of matching rules against the database, which is common to all rule-based languages [HW92].

3 Incremental Computation Algorithms

In this section, we describe and compare the discrimination networks built by RETE and TREAT. We assume that rules are instance-oriented (as in OPS5).

3.1 RETE Network

Consider the following rule:

$$r1 \quad : \quad A(x, \alpha, z), \quad B(x, y, \beta), \quad C(\gamma, y, w) \quad \rightarrow \quad C(\gamma, x, w), \ \neg C(\gamma, y, z) \ .$$

This rule is compiled by RETE in the dataflow network depicted in Figure 1. The network has three alpha-nodes $n1$, $n2$, $n3$, and two beta-nodes $p1$, $p2$, where:

$n1 = \sigma_{A.2=\alpha} A$, $n2 = \sigma_{B.3=\beta} B$, and $n3 = \sigma_{C.1=\gamma} C$
$p1 = \Pi_{\{n1.1, n2.2, n1.3\}}(n1 \bowtie_{n1.1=n2.1} n2)$, and
$p2 = \Pi_{\{p1.1, p1.2, p1.3, n3.3\}}(p1 \bowtie_{p1.2=n3.2} n3)$
where σ, Π, and $\bowtie$ respectively denote relational select, project and join.

The incremental evaluation procedure of a rule program proceeds essentially as follows:

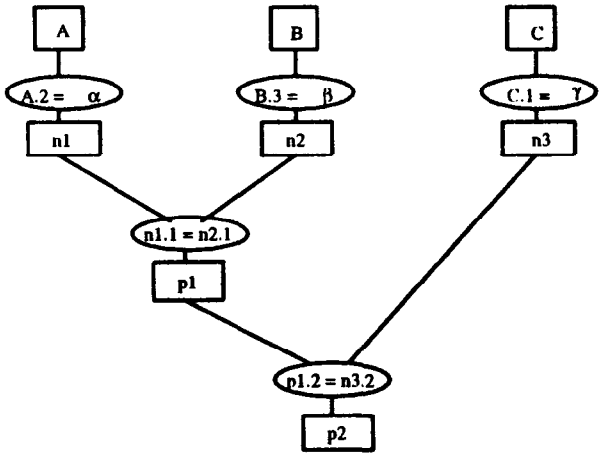

Figure 1: RETE network for rule r1

incremental evaluation of a rule program

state := initial database state;
† match rules against *state* until a firable rule is found and record the results into the discrimination network;
while there exists a firable rule[1] *r do*
  state := result of firing *r* on *state*;
  for each input node *N* of the network *do*
    if N has changed *then* propagate the change to the successor nodes of *N*;
‡ select a firable rule *r*;
end while

Compared to the original evaluation procedure used by RETE and TREAT, there is one main difference. Since we assume statically-defined priorities between rules, there is no need to construct the whole conflict set before entering the while loop. We only need to find one firable rule (step †). Thus, selecting a firable rule in step ‡ may require to match some non-already tried rules against *state* in order to compute their set of satisfying instances and hence, to expand the discrimination network.

3.2 TREAT Network

TREAT compiles rule $r1$ in a dataflow network similar to RETE. The only difference with RETE's network is that the result of the join between n_1 and n_2 is not kept. On the other hand, TREAT maintains for each rule its set of satisfying instances. This coincides with node p_2 in RETE's network. Calling p this node, we have

$p = \Pi_{\{n1.1,n2.2,n1.3,n3.3\}}\{n1 \bowtie_{n1.1=n2.1} n2 \bowtie_{n2.2=n3.2} n3)\}$.

[1] If rules have instance-oriented semantics, then r is a firable satisfying instance

In fact, TREAT also remembers in p the tuple identifiers of the matching tuples.

3.3 Tradeoff Between RETE and TREAT

The main question raised by the previous example is: "is it worthwhile to keep node p_1 ?". To answer, we need to compare the work done by each algorithm at each cycle.

RETE does an extra work to maintain p_2 when a deletion to n_3 occurs. But, some implementations of RETE optimize it by recording tuple identifiers in the nodes (as TREAT does). On the other hand, TREAT does an extra work to propagate insertions because the join betwen n_1 and n_2 needs to be recomputed at each cycle. Clearly, on this example, the tradeoff will favor RETE (even if RETE does not optimize the maintenance of p_2 with tuple identifiers).

Now, suppose that the action part of the rule is changed, yielding the following rule:

$r2 \quad : \quad A(x, \alpha, z), \quad B(x, y, \beta), \quad C(\gamma, y, w) \quad \rightarrow \quad \neg B(y, z, \beta)$

Since the body of r_2 is the same as r_1, the networks respectively built by RETE and TREAT for r_2 are as the networks for r_1. But this time, TREAT outperforms RETE because the only effect of the rule is to delete facts and these deletions require less work with TREAT. Indeed, RETE will update nodes $n2$, $p1$ and $p2$ whereas TREAT will update $n2$ and p.

The above examples hopefully make one point. There is an inherent tradeoff in incremental algorithms. Maintaining information may help for computing new rule instantiations but may hurt performance when rule instantiations are deleted from the conflict set. In general, a rule program has a non-monotonic behavior with respect to its conflict set and solving the tradeoff depends on the particular rule program.

4 Computation of Profitable Relations

Ideally, an algorithm that chooses a discrimination network for a set of rules should consider two important factors:

1. the ability to *efficiently* compute the differential change to every node in the network associated with a relational expression, given the net changes to its predecessor nodes,

2. the *profitability* of maintaining a node in the network for speeding calculations that will follow the materialization of the node.

Both conditions must be met to decide to maintain a node in the network. The first problem is addressed in Sections 4.1 and 4.2, while the second one is addressed in Section 4.3.

4.1 Autonomous Expressions

Our problem is not totally new. Previous work (e.g., [BCL89]) has addressed the problem of determining which data are useful to maintain in order to speed-up the maintenance of a materialized view when base relations change. An interesting case is when the new value of the view can be computed from its previous (stored) value and the differential changes applied to its operand relations. The view is said to be *autonomously* computable. We generalize this notion.

In what follows, a *change* to a predicate instance is either an *insertion* or a *deletion* of tuples. The *sign* of a predicate in a conjunction of literals indicates if it occurs positively or negatively. We also denote $\Phi^{(k)}$ the relation *defined* [2] by a conjunction of literals $\Phi(\vec{x})$ in state I_k. Without loss of generality, we only consider formulas Φ where each predicate occurs only once (this may be achieved by renaming).

Definition 4.1 Let Φ be a (safe) conjunction of literals and c a change to a predicate P occuring in Φ. Then Φ is *autonomous* for P wrt c if for any k, $\Phi^{(k+1)}$ can only be computed from $\Phi^{(k)}$ and c.

This definition is generalized to a set of changes.

Definition 4.2 Let Φ be a (safe) conjunction of literals and $\mathcal{C}_P$ a set of changes to P. Φ is *autonomous* for P wrt $\mathcal{C}_P$ if Φ is autonomous for P wrt every change of $\mathcal{C}_P$. Let $\mathcal{C}$ be a set of changes to some predicates of Φ. Φ is *autonomous* wrt $\mathcal{C}$ if for each predicate P and for each change c of $\mathcal{C}$, either c has no effect on P, or Φ is autonomous for P wrt c.

Sufficient conditions to detect autonomous expressions are presented below. We shall use the following notations. Let Φ be a (safe) conjunction of literals and P a predicate in Φ. We partition the variables of Φ into three sets: SV contains variables *shared* by P and some other predicate of Φ, PV contains variables that occur solely in P, and RV contains the *remaining* variables.

First and Second Autonomy Criteria: Let I_k be a state and I_{k+1} the state obtained after applying a change c to $P^{(k)}$.

1. If P is *positive* and c is a *delete*, then Φ is autonomous for P wrt c and the relation defined by Φ in state I_k is defined by:

$$\begin{cases} \Delta\Phi^{(k)} = \Phi^{(k)}(SV, PV, RV) \bowtie \Delta P^{(k)}(SV, PV), \\ \Phi^{(k+1)} = \Phi^{(k)} - \Delta\Phi^{(k)} \end{cases} \quad (1)$$

where $\Delta P^{(k)} = P^{(k)} - P^{(k+1)}$

2. If P is *negative* and c is an *insert*, then Φ is autonomous for P w.r.t c and the relation defined by Φ in state I_k is defined by formula (1), in which $\Delta P^{(k)} = P^{(k+1)} - P^{(k)}$.

Before introducing the third criteria, we need another definition.

Definition 4.3 Let $P(A_1, ..., A_n)$ be a relational schema and S be a subset of $\{A_1, ..., A_n\}$. We say that $\Pi_S P$ is *stable* wrt a set of changes $\mathcal{C}$, if for any state I_k, $\mathcal{C}$ has no effect on $\Pi_S P^{(k)}$.

For instance, P is stable on $\{x\}$ wrt the changes induced by rule r: $R(x,z), P(x,y) \rightarrow P(x,z)$.

Third Autonomy Criteria: If P is *positive*, c is an *insert* and $\Pi_{SV} P$ is *stable* wrt Γ, then Φ is autonomous for P w.r.t c and the relation defined by Φ in state I_k is defined by

$$\begin{cases} \Delta\Phi^{(k)} = (\Pi_{\{SV,RV\}}\ \Phi^{(k)}(SV, PV, RV)) \bowtie \Delta P^{(k)}(SV, PV), \\ \Phi^{(k+1)} = \Phi^{(k)} \cup \Delta\Phi^{(k)} \end{cases} \quad (2)$$

where $\Delta P^{(k)} = P^{(k+1)} - P^{(k)}$.

Example 4.1: Take $\Phi = R(x,z) \bowtie P(x,y)$ and rule $R(x,z), P(x,y) \rightarrow P(x,z)$. Given a state I_k, when r fires, the set of tuples $\Delta P^{(k)}$ inserted into $P^{(k)}$ is such that $\Pi_{\{x\}}\Delta P^{(k)} \subseteq \Pi_{\{x\}}\ P^{(k)}$. By the third autonomy criteria, Φ is autonomous for P wrt the changes induced by r, and $\Phi^{(k+1)}$ can be *easily* computed using formula (2).

Remark however, that if r has a set-oriented semantics then the rule can only be fired once because after the first firing no new tuples can be inserted into P. Thus, maintaining $\Phi^{(k)}$ will not be *profitable*. On the other hand, if r has an instance-oriented semantics then r can be fired many times before reaching a fixpoint. Hence, maintaining $\Phi^{(k)}$ will be *profitable*.

4.2 Producer-filter Decomposition of an Expression

In this section, we make use of the notion of producer-filter view initially introduced in [Bry89]. We provide a different definition of a producer-filter and relate this notion to the previous concept of autonomous expression. We first use a motivating example.

Example 4.2: Take the expression:

$$\Phi_1 = A(x,y),\ B(y,z),\ C(x,u),\ \neg D(z,x),\ \neg E(z,y)$$

[2] In the sense of [Ull89], p. 107

and suppose we have a set of changes $\mathcal{C}$ consisting of deletions from A, insertions into C that keep $\Pi_{\{C.1\}}C$ invariant, insertions into D, and insertions and deletions to E. According to the three autonomy criteria, Φ_1 is autonomous wrt all changes of $\mathcal{C}$ but deletions from E. Now, take formula:

$$\Phi_2 = A(x,y),\ B(y,z),\ C(x,u),\ \neg D(z,x)$$

Φ_2 is clearly autonomous wrt to $\mathcal{C}$. Suppose that $T(x,y,z,u)$ is the relation defined by Φ_2, and let us define $\tilde{T}(x,y,z,u,f_E)$ as: $(T \bowtie E) \times \{true\} \cup (T \neg\bowtie E) \times \{false\}$. The relation defined by Φ_1 is simply computed by $\Pi_{\{x,y,z,u,\}}(\sigma_{f_E=false}\ \tilde{T})$. Furthermore, when a tuple t is deleted from E then all tuples of $\tilde{T}$ that match with t have their f_E attribute set to $false$.

Thus, maintaining relation $\tilde{T}$ is ***easy*** when deletions to E occur and can be *profitable* to compute the relation defined by Φ_1.

Definition 4.4 Let Φ be a (safe) conjunction of literals. Assume that $\Phi = \phi \wedge \psi$, where ϕ is safe and all ψ variables are bound by positive literals in ϕ. Then ϕ (resp. ψ) is said to be a *producer* (resp. a *filter*) for Φ, and $(\phi,\ \psi)$ is said to be a *producer-filter* decomposition.

Definition 4.5 Let $(\phi,\ \psi)$ be a *producer-filter* decomposition of a formula Φ. Let $\psi = (\neg)B_1(\vec{x}_1),\ \ldots,\ (\neg)B_n(\vec{x}_n)$, and $T(A_1,\ldots A_k)$ the relation defined by ϕ. We define the relation $\tilde{T}$ with schema $\{A_1,\ldots A_k\} \cup \{f_1,\ldots,f_n\}$, where $Dom(f_i) = \{true, false\}$ as:

$$\tilde{T} = T \ \overline{\ltimes}\ B_1 \ldots \ \overline{\ltimes}\ B_n$$

where $\overline{\ltimes}$ denotes the semi-outer join operator [Ull89].

Intuitively, for each tuple t of T (the producer), there is a tuple in $\tilde{T}$ that indicates whether t satisfies the condition of the filter (formula ψ).

Proposition 4.1 Let (ϕ,ψ) be a producer filter decomposition of Φ and $\tilde{T}$ the associated relation. Let P be a predicate of Φ and $\mathcal{C}$ a set of changes. If either P occurs in ψ, or Φ is autonomous [3] for P wrt $\mathcal{C}$ then $\tilde{T}$ is autonomous for P wrt $\mathcal{C}$.

In the previous example, $(\Phi_2, \neg E)$ is a *producer filter* decomposition of Φ_1 which is autonomous wrt the set of changes we considered.

4.3 Algorithm to Compute Profitable Relations

We now characterize profitable relations and provide an algorithm that computes the maximal profitable relations from a given conjunction of literals.

[3] for a sake of simplicity we shall abusively use the word autonomous for a relation

Definition 4.6 A relation R is said Φ-*profitable* with respect to a set of changes $\mathcal{C}$ if:

- R is defined by a subexpression ϕ of Φ that is autonomous wrt $\mathcal{C}$, or R is an autonomous relation associated with a producer-filter decomposition of Φ wrt $\mathcal{C}$,
- no cartesian product is used to build R.

Given a formula, there may be many profitable subexpressions but not all of them will be interesting to maintain. Take $\Phi = A(x,y), B(y,z), C(z,x)$, and assume that the set of changes $\mathcal{C}$ consists of insertions into A that keep $\Pi_{A.2}A$ invariant, deletions from B, and insertions into C that keep $\Pi_{C.1}C$ invariant. Then A, B, C, $A \bowtie B$, $B \bowtie C$ are all Φ-*profitable* subexpressions wrt $\mathcal{C}$. Nevertheless, memorizing and maintaining all these relations is clearly not a good compromise. Our algorithm only selects *maximal* relations. In our case, it will randomly choose to maintain either A and $B \bowtie C$, or C and $A \bowtie B$.

Maximal Profitable Relations Algorithm
input: a safe conjunction of literals Φ, and a set of changes $\mathcal{C}$
output: $\mathcal{A}$ = a set of Φ-profitable relations such that every operand relation belongs to only one expression;

$\mathcal{M} \leftarrow \emptyset$; /* a set of autonomous expressions */
step 1 :
build a connection graph $\mathcal{G}$ where vertices are literals of Φ and there is an edge between literals P_1 and P_2 if they have at least a variable in common;
step 2 :
repeat
  $P \leftarrow$ *choose_source* $(\phi, \mathcal{G}, \mathcal{C})$;
  /* return a literal of Φ that is non marked in $\mathcal{G}$ */
  mark P in $\mathcal{G}$;
  $g \leftarrow$ *choose_max_subexpression* $(\mathcal{G}, \mathcal{C}, P, P)$;
  /* construct a maximal autonomous expression from P */
  $\mathcal{M} \leftarrow \mathcal{M} \cup \{g\}$;
until all nodes of $\mathcal{G}$ with positive literals are marked;
step 3 :
$\mathcal{F} \leftarrow$ {unmarked literals in $\mathcal{G}$} $\cup$ {$f \in \mathcal{M}$ s.t. f is a singleton and there is some p in $\mathcal{M}$ s.t. (p,f) is a producer-filter decomposition of $p \wedge f$ and its associated relation is autonomous wrt $\mathcal{C}$ };
$\mathcal{P} \leftarrow$ *make_producer* $(\mathcal{M}\text{-}\mathcal{F})$;
/* an element of $\mathcal{P}$ is a producer-filter decomposition (p, f) s.t. p is in $\mathcal{M}$-$\mathcal{F}$ and f is initialized with $\emptyset$. */
for f in $\mathcal{F}$ do
  $(p, f') \leftarrow$ *select_producer-filter* $(f, \mathcal{P}, \mathcal{C})$;
  /* $(p, f') \in \mathcal{P}$ s.t. (p, f) is a producer-filter decomposition and its associated relation is autonomous wrt $\mathcal{C}$. When several elements of $\mathcal{P}$ are candidate, we choose the most selective one; if none p is randomly selected.*/
  if p exists then add f to f';
od

step 4:
$\mathcal{A} \leftarrow \emptyset$;
for (p, f) in $\mathcal{P}$ do
if $f \neq \emptyset$ or p is not a single literal then add the relation associated with (p, f) to $\mathcal{A}$;
fi
od
return $\mathcal{A}$;

At Step 2 of the algorithm, we choose a node, called the *source*, in the predicate connection graph and then construct a maximal autonomous expression from this node. Note that there may be several autonomous expressions that can be built from a given node. The heuristics used to choose the source and compute the *preferred* autonomous expression is detailled in Appendix A. In general, we are interested in the selectivity of a literal (some arguments can be constants) because we want to compute a profitable relation with the smallest size.

At the end of Step 2, $\mathcal{M}$ contains a set of autonomous expressions but some nodes of $\mathcal{G}$ may remain unmarked (negative literals). In Step 3, $\mathcal{F}$ is constructed with potential filters to some autonomous expressions of $\mathcal{M}$. Each autonomous expression that is not in $\mathcal{F}$ becomes a producer-filter decomposition in $\mathcal{P}$. Then, we try to integrate every literal of $\mathcal{F}$ within the filter of some producer-filter decomposition in $\mathcal{P}$.

In the last step of the algorithm, we output relations that are associated with *interesting* producer-filter decompositions, i.e., those where the filter is not empty or the producer is not reduced to a single literal.

5 The COSMA Algorithm

5.1 Output Token Relations

Before presenting our algorithm we need to define specific relations, called *output token* relations, that play a key role in the computation of a rule program. Intuitively, given a rule r and a literal $l = (\neg)P(\vec{x})$ in its head, the output token relation associated with r and l records the net effect of r on the instance of P, for a particular database state. Such relations are essential because they enable to know if a rule is firable: at least one of its output token relations must be not empty. More, when a rule fires, output token relations serve to compute the effect of the rule.

The definition of an output token relation varies according to the semantics of rules.

Definition 5.1 Let r be a set-oriented rule, $l = (\neg)P(\vec{x})$ a literal in the head of r, and I a database state. The output token relation, noted $OT(r, l)$, associated with r and l is the set of ground instances of P in $effect_r(I)$ that represent a net change to $I(P)$.

For instance-oriented rules, the definition slightly differs because we have to identify to which satisfying rule instance is associated each tuple of OT. In fact, every tuple in the relation defined by the body of a rule, represents a satisfying instance of that rule.

Definition 5.2 Let r be an instance-oriented rule, $l = (\neg)P(\vec{x})$ a literal in the head of r, and I a database state. Each tuple in $OT(r, l)$ is the concatenation of two tuples, t and t' where t is a tuple in the relation defined by the body of r that corresponds to a satisfying instance r' of r, and t' is a ground instance of P in $effect_{r'}(I)$ that represents a net change to $I(P)$.

5.2 Migration Flow of Data in a Program

As we mentioned before, our algorithm relies on the knowledge of the changes to any predicate that are induced by the program at a particular point in time. Extracting this knowledge from a rule program requires to perform a careful analysis of the migration flow of data within the rule program. Because we assume statically-defined priorities between rules, this analysis can be done statically.

In order to capture the migration flow of data, we construct a labelled directed graph $\mathcal{I}$ called an *Influence graph*. Rules are the vertices of $\mathcal{I}$ and there is an arc from r to r' if there are predicates occurring in the head of r that also occur in r'. In the following, we say that r has an *influence* over r'. The label associated with (r, r') is a set of expressions of the form $l\ \theta\ l'$ where l (resp. l') is a literal of the head of r (resp. head of r') and θ is in $\{+, -\}$. The expression $l + l'$ (resp. $l - l'$) means that when r fires, its effect wrt l has for consequence to add (resp. to remove) tuples into (resp. from) $OT(r, l')$.

Example 5.1: Consider the following program Γ:

r_1 : A(x,α,y), B(y,β,z), E(z,v,γ), $\neg$F(x,z) $\rightarrow$ C(x,z)
r_2 : C(x,y), D(y,z,u) $\rightarrow$ D(z,u,x)
r_3 : B(x,y,β), $\neg$D(δ,y,x), E(z,u,x) $\rightarrow$ D(δ,z,x), $\neg$C(y,x)
r_4 : A(x,y,z), D(δ,x,y), C(z,t), F(z,w) $\rightarrow$ $\neg$A(y,t,x), F(z,y)

Table 5.1 represents the *Influence graph* of program Γ, rows [resp. columns] represent r [resp. r'] :

Definition 5.3 Let $\mathcal{I}$ be the influence graph of a program Γ and $r_0 r_1 \cdots r_n$ a path in $\mathcal{I}$. If there exists a sequence $(l_0\ \theta_0\ l_1)(l_1\ \theta_1\ l_2) \ldots (l_{n-1}\ \theta_{n-1}\ l_n)$ where $(l_i\ \theta_i\ l_{i+1})$ is an expression in the label of (r_i, r_{i+1}), then this sequence is said to be a *propagation path* from r_0 to r_n. If for each $i \in \{0, \ldots, (n-1)\}$, $\theta_i = +$, then the propagation path is *positive*. If $\theta_{n-1} = -$, then the propagation path is *negative*.

	r_1	r_2	r_3	r_4
r_1		{C + D}	{C+ ¬C }	{ **C + ¬A, C+F** }
r_2		{D + D }	{D − D, D − ¬ C }	{ **D + ¬A , D + F** }
r_3	{ ¬C + C }	{D + D, **¬C − D** }	{D − D, D −¬C }	{ **¬C − ¬A, ¬C − F , D + ¬A, D + F** }
r_4	{ **¬A − C, F − C** }			{ ¬A − ¬A, ¬A − F, F + ¬A, F + F }

Table 5.1: Influence graph of Γ

The priority ordering between rules enables to simplify an influence graph by removing some labels. Assume for instance that r_1, r_2 and r_3 have priority over r_4 in the previous example. Rule r_4 can only be fired when no other rule is firable. Since $OT(r_4, \neg A)$ has not yet been computed any firing of r_1, r_2 and r_3 has no effect on $OT(r_4, \neg A)$. Furthermore, if r_4 fires, neither r_1 or r_2 or r_3 becomes firable. Thus, r_4 has no effect on $OT(r_1, C)$. Hence, the labels of the arcs (r_1, r_4), (r_2, r_4), (r_3, r_4) and (r_4, r_1) are *irrelevant* and can be discarded, thereby changing the propagation paths in $\mathcal{I}$. This is formalized below.

Definition 5.4 Let $(\Gamma, \leq)$ be a program, $\mathcal{I}$ its influence graph, and $p = l\ \theta\ l'$ an element of the label of (r, r'). Then p is *relevant* iff

- r and r' are not comparable wrt $\leq$, or
- r' has priority over r and p is a positive propagation path from r to r', or
- r has priority over r' and there exist a rule r'' that has not priority over r' and a positive propagation path, p', from r'' to r, such that $p'\ p$ is a propagation path from r'' to r'.

Irrelevant portions of the labels can be discarded from the influence graph. To illustrate, in Table 5.1 bold characters represent the label expressions that can be discarded.

5.3 The Algorithm

Given a program $(\Gamma, \leq)$, with either an instance-oriented or a set-oriented semantics, COSMA is a one-pass algorithm which produces for each rule, which relations should be materialized, how to compute these relations and how to differentially maintain them.

COSMA incrementally builds an *influence graph* and simultaneously computes the relevant label expressions. When all the labels of the edges ending at some rule r are fully computed, the influence graph provides the necessary knowledge about the possible set of changes, say $\mathcal{C}$, to the predicates occuring in r. Then, r is annotated with two specific sets noted *Effect* and *Subexp* using the *Annotate_Rule* algorithm.

```
COSMA Algorithm :
  input: a program (Γ, ≤) with a set-oriented or instance-
    oriented semantics;
  output : Γ where each rule has been annotated;

  Step 1:
  I ← ∅ ;
  /* Initialisation of the influence graph */
  Step 2:
  while there exists a non annotated rule in Γ do
    M ← {r ∈ Γ s.t. ∀ r' ∈ Γ if r' has priority
      over r then r' is annotated }
    for r in M do
      N ← { rules that have an influence over r };
      for r' in N do
        expand I with edge (r', r) ;
        compute the relevant label expressions of (r', r);
      od;
      Annotate_Rule (I, r);
    od;
  od;
```

An element of *Subexp* is a pair $(E, \partial E/\partial \mathcal{C})$ such that E is a subexpression of the expression associated with $OT(r, l)$, for some l in the head of r, and $\partial E/\partial \mathcal{C}$ denotes the differential expression that incrementally maintain the relation defined by E wrt a set of changes $\mathcal{C}$. All the relations defined by the E-expressions of *Subexp* will be materialized. A differential expression consists itself of a pair $(\partial E^+/\partial \mathcal{C},\ \partial E^-/\partial \mathcal{C})$, where the first (resp. second) element computes tuples that must be added (resp. deleted) to the relation defined by E, say T. If no insertions (resp. deletions) can occur to T then the first (resp. second) element is said to be *undefined*.

For every literal l in the head of r, there is a triple $(E, \partial E/\partial \mathcal{C}, \text{Mat})$ in *Effect*. In each triple, E represents the expression that computes $OT(r, l)$ in terms of the materialized relations defined in *Subexp* and other (non-materialized) relations. If $OT(r, l)$ is materialized then $\partial E/\partial \mathcal{C}$ is the differential expression that maintains OT wrt $\mathcal{C}$. In this case, the differential will consist of a pair $(\partial E^+/\partial \mathcal{C},\ \partial E^-/\partial \mathcal{C})$ as we mentioned before. Otherwise, $\partial E/\partial \mathcal{C}$ is the expression that computes the new value of OT in terms of the differential changes to its operand relations. Here, the differential consists

of a single expression. Finally, *Mat* is a boolean that indicates if OT(r,l) is materialized.

- *Steps 2 and 3* of the Annotate_Rule algorithm compute *Subexp*. Given a rule r and a literal l in its head, our algorithm uses the following simple rule to decide which subexpressions should be computed in *Subexp*.

R1. If $OT(r,l)$ is not autonomous wrt $\mathcal{C}$ then memorizing maximal $OT(r,l)$_profitable relations is useful.

The rationale for this heuristic rule is that when OT(r,l) is not autonomous wrt $\mathcal{C}$, $OT(r,l)$ needs to be computed more than once whatever the semantics of the rule is. Thus, it is *useful* to memorize subexpressions that may speed up repetitive computations of OT(r,l).

- *Step 4* of the algorithm computes the second part of the annotation, *Effect*. An important decision taken by the algorithm is to decide if an output token relation must be materialized. We use the following heuristic rule:

R2. If a rule r is instance-oriented then memorizing $OT(r,l)$ is useful for every literal l in the head of r.

With instance-oriented rules, when a rule fires, only one instantiation, say r', i.e., one tuple of OT is used. Intuitively, the effect of the rule may have for consequence to add new instantiations in OT and/or to invalidate previous instantiations of OT. If OT is not materialized then it has to be entirely recomputed at each firing of r because the next instantiations cannot be computed solely in terms of r' and its consequences. Thus, OT represents the most recent computation that can be differentiated before the choice of an instantiation to fire. On the other hand, with set-oriented rules, the OT relations are emptied after each firing of the rule and hence there is no need to materialize them. Furthermore, for set_oriented rules, we have:

Proposition 5.1 Let $(\Gamma, \leq)$ be a program with set oriented semantics, r a rule, $l = (\neg)P(\vec{x})$ a literal in the head of r. If the formula associated with $OT(r,l)$ is autonomous wrt the changes to P induced by the program, then $OT(r,l)$ will be computed once. Furthermore, if this is the case for every literal in the head of r, then r is fired at most once.

Annotate_Rule Algorithm :
input: an influence graph $\mathcal{I}$ and a rule r;
output: r is annotated with two sets: *Effect* and *Subexp*;

Step 1:
  $\mathcal{C} \leftarrow$ {changes on predicates occuring in r};
  /* $\mathcal{C}$ is easily derived from $\mathcal{I}$ */
Step 2:
  $H \leftarrow \emptyset$;
  /* H contains output token expressions which are not autonomous wrt $\mathcal{C}$ */
  for each literal l of the head of r do
    $\Psi \leftarrow$ expression associated with $OT(r,l)$;
    if Ψ is *not* autonomous wrt $\mathcal{C}$ then add Ψ to H;
  od ;
Step 3:
  if $H \neq \emptyset$ then
    if H is a singleton $\{\Psi\}$ then $\Phi = \Psi$
    else $\Phi =$ expression associated with the body of r;
    fi;
    $\mathcal{A} \leftarrow$ Maximal_Profitable_Relation (Φ, $\mathcal{C}$);
    Subexp $\leftarrow$ $\{(E, \partial E/\partial\mathcal{C})$ s.t. $E \in \mathcal{A}\}$;
  else *Subexp* $\leftarrow \emptyset$;
Step 4:
  Effect $\leftarrow \emptyset$;
  for each literal l of the head of r do
    $E =$ expression associated with $OT(r,l)$ in terms of relations defined by expressions in *Subexp*;
    if r is instance_oriented then
      calculate $\partial E/\partial\mathcal{C}$ for maintaining $OT(r,l)$;
      Mat $\leftarrow$ true;
    else
      Mat $\leftarrow$ false;
      if E is not autonomous wrt $\mathcal{C}$
      then calculate $\partial E/\partial\mathcal{C}$ for computing the new value of $OT(r,l)$
      else $\partial E/\partial\mathcal{C} \leftarrow \emptyset$;
        /* OT(r,l) is computed once only */
      fi;
    fi;
    add $(E, \partial E/\partial\mathcal{C})$ to *Effect* ;
  od;

5.4 An Example

We apply COSMA to the program $(\Gamma, \leq)$ of section 5.2. We assume that rules are set-oriented and that r_1, r_2 and r_3 have priority over r_4. The rule annotations returned by COSMA are as follows :

r_1 Subexp :
$e1 = \Pi\{x,z\}(\ \sigma_{A.2=\alpha}A(x,\_,y)) \bowtie (\sigma_{B.2=\beta}B(y,\_,z)) \bowtie (\sigma_{E.3=\gamma}E(z,v,\_))$
$\neg \ltimes$ F(x,z) $\overline{\ltimes}$ C(x,z)
/* T1 is the relation defined by e1 */
$\partial e1^+/\partial C = (\Pi_{\{x,z\}}$ T1 $\ltimes\ \Delta^-$C(x,z)) $\times$ false
$\partial e1^-/\partial C =$ T1 $\ltimes\ \Delta^-$C(x,z)
Effect :
$e2 = \Pi_{\{x,z\}}(\ \sigma_{f_C=false}$ e1)
$\partial e2/\partial C = \Pi_{\{x,z\}}\ \Delta^+ T1$
Materialize : false

r_2 Effect :
e3= $\Pi_{\{z,u,x\}}$(C(x,y) $\bowtie$ D1(y,z,u)) $\neg \ltimes$ D2(z,u,x)
$\partial e3/\partial C = \Pi_{\{z,u,x\}}(\Delta^+ C(x,y) \bowtie$ D1(y,z,u)) $\neg \rtimes <$ D2(z,u,x)

$\partial e3/\partial D = \Pi_{\{z,u,x\}}(C(x,y) \bowtie \Delta^+D1(y,z,u)) \neg\rhd\lhd D2(z,u,x)$
Materialize : false

r_3 Subexp :
$e4=(\sigma_{B.3=\underline{\beta}}(B(x,y,\_)) \neg\rhd\lhd (\sigma_{D.1=\delta} D1(\_,y,x)) \bowtie E(z,u,x) \ltimes C(y,x)$
/* T2 is the relation defined by e4 */
$\partial e4^+/\partial C = (\Pi_{\{x,z\}} T2 \ltimes \Delta^+C(y,x)) \times true$
$\partial e4^-/\partial C = T2 \ltimes \Delta^+C(y,x)$
Effect :
$e5 = \Pi_{\{y,x\}}(\sigma_{fC=true} T2)$
$\partial e5/\partial C = \Pi_{\{y,x\}} \Delta^+T2$
$e6 = \Pi_{\{z,x\}}((\sigma_{B.3=\beta}(B(x,y,\_)) \neg\ltimes (\sigma_{D.1=\delta} D1(\_,y,x)) \bowtie E(z,u,x)) \neg\ltimes (\sigma_{D.2=\delta} D2(\_,z,x))$
$\partial e5/\partial C$ = undefined
Materialize : false

r_4 Effect :
$e7 = \Pi_{\{y,t,x\}}(A1(x,y,z) \bowtie (\sigma_{D.1=\delta} D(\_,x,y)) \bowtie C(z,t) \bowtie F1(z,w)) \bowtie A2(y,t,x)$
$e8 = \Pi_{\{x,y\}}(A1(x,y,z) \bowtie (\sigma_{D.1=\delta} D(\_,x,y)) \bowtie C(z,t) \bowtie F1(z,w)) \neg\ltimes F2(z,y)$
Materialize : false

In Figure 3, we represent the discrimination network for every output token relation associated with a rule of program. In each diagram, there is one input node per literal in the rule and one output token relation. Circle boxes denote relational expressions, square boxes denote materialized relations, and other nodes denote non-materialized relations.

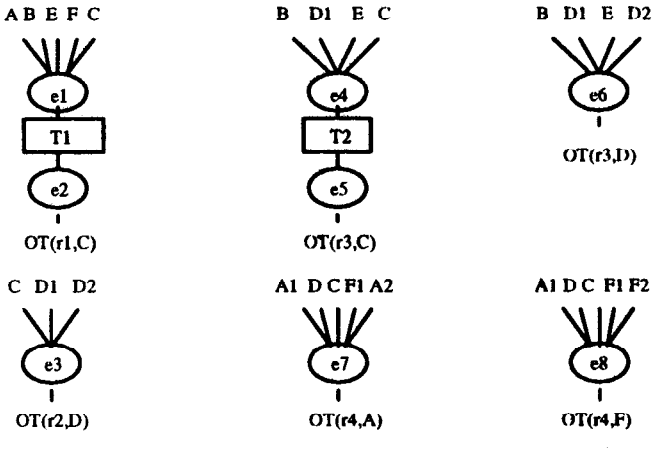

Figure 2: Discrimination network for Example 5.1

Take rule r_1, the only changes on its predicates consist of deletions from C according to the influence graph of table 5.2. The expression associated with $OT(r_1, C)$ is $\Psi = A(x,\alpha,y), B(y,\beta,z), E(z,v,\gamma), \neg F(x,z), \neg C(x,z)$. Step 2 of the Annotate_Rule algorithm detects that Ψ is not autonomous wrt deletions in C. Step 3 computes the maximal OT_profitable relations wrt changes to C and returns the relation defined by $e1$ which is the relation associated with the producer_filter decomposition of Ψ $((A(x,\alpha,y), B(y,\beta,z), E(z,v,\gamma), \neg F(x,z)), (\neg C(x,z)))$. Then $\partial e1^+/\partial C$ and $\partial e1^-/\partial C$ are computed. Next, Step 4 proceeds. Since the rule is set_oriented, its output token relation is not materialized. The expression that computes the initial value of $OT(r_1, C)$ in terms of $T1$ (the relation defined by e1) is calculated and yields $e2$.

Subsequent values of $OT(r_1, C)$ are then computed in terms of $T1$ and Δ^-C using $\partial e2/\partial C$, where Δ^-C represents the delta relation containing deletions to C that occurred since last firing of r_1. When deletions occur in C, they are propagated to $T1$. Deletions to $T1$ are directly performed to $T1$ while insertions to $T1$ are recorded separately into Δ^+T1. In fact, Δ^+T1 is a delta relation, in the sense of [SKdM92], representing the net additions to T1.

For rule r_1, only one delta relation has to be maintained: Δ^-C. Similarly, for the other rules, our algorithm enables to detect which delta relations are necessary to maintain. In addition to enable an incremental computation of the rules, these relations play the role of logs. Indeed, delta relations enable to implement a deferred update strategy for processing the rules. The idea is that updates to the input level of a rule network can be logged into delta relations and their propagation into the network can be delayed until the rule is tried to fire.

6 Related Work

A few incremental algorithms have been proposed to evaluate production rules in a database system.

A_TREAT [Han92] uses a TREAT-like discrimination network to record the result of matching rule bodies against the database. However, in order to save memory space, A_TREAT only memorizes the results of selection predicates in α_memory nodes when the selectivity of the selection is high. If the selectivity the selection is low, A_TREAT replaces the corresponding α_memory by a *virtual* α_memory where it memorizes the selection predicate of the node instead of the result of the selection itself. Much attention has been devoted to the efficient calculation of the new value of the relation defined by a rule body by taking advantage of the query optimizer and the use of attribute indexes. On the other hand, our algorithm focuses on choosing which appropriate discrimination network should be built. With regard to this problem A_TREAT only question

the profitability of maintaing a selection node based on its selectivity.

DBCond [SLR88] maintains a RETE discrimination network into a flattened data structure. Take a rule r : $A(x,\alpha,z), B(x,y,\beta), C(\gamma,y,z) \rightarrow head$, and relations $A(A_1,A_2,A_3)$, $B(B_1,B_2,B_3)$ and $C(C_1,C_2,C_3)$. Then, DBcond will maintain one relation per literal in the body of r, called COND relations. For instance, COND_A will contain the projections $\Pi_{B_1}(B)$ and $\Pi_{C_3}(C)$, as well as the projection on B_1 and C_3 of tuples in the join between B and C. The main feature of this algorithm is that when a change occurs to A then COND-A suffices to determine if rule r is firable (although it does not compute the effect of the rule). However, all affected COND_relations have to be maintained upon a change to A (in our case, COND_B and COND_C relations). Thus, the tradeoff pointed out earlier in this paper applies exactly as for RETE. Our algorithm is able to choose a better discrimnation network which could then be implemented using the DBCond technic.

In [SZ91], the authors propose an efficient technique to materialize the relation defined by a rule body. Suppose that the rule body is an expression: $E = Q_1 \bowtie \cdots, \bowtie Q_n$. Then the relation, say T, defined by E has one attribute for every surrogate attribute of Q_i plus all Q_i-join attributes that occur in E. The result of every intermediate join is stored into T. For instance, the result of the join between Q_1 and Q_2 will yield tuples in T where the value of every attribute that is neither a Q_1 or Q_2 surrogate attribute or a join attribute between Q_1 and Q_2 is a *null* value. It is worth noting that such a relation is autonomous wrt any change to the Q_i's. Here again, our algorithm can be used to reduce the size of T by retaining only those intermediate results that are profitable to maintain.

7 Conclusion

We have presented an original adaptive algorithm that takes as input a program of rules and returns a discrimination network representing an expected good compromise. The discrimination network is then used to perform an incremental evaluation of the rule program. In order to decide if a subexpression must be materialized as a node in the discrimination network, our algorithm analyzes two main factors:

1. the ability to *efficiently* compute the differential change to this node, given the net changes to its predecessor nodes induced by the rule program,

2. the *profitability* of maintaining a node in the network for speeding calculations that will follow its materialization.

Our algorithm works for set-oriented and instance-oriented rules which are the two most common kinds of rules used in database rule languages.

We believe that our algorithm fills a hole in the literature on incremental algorithms for production rule languages in databases. As we said before, most algorithms are derived either from RETE or TREAT. Specific implementation techniques have been designed to efficiently process a RETE or TREAT discrimination network, but we are not aware of any previous work that questioned the pertinence of building a RETE or TREAT network. Our algorithm does so and capitalizes on various database algorithms, in particular in the area of maintenance of materialized views.

References

[BCL89] Jose A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, 1989.

[BFKM85] L. Brownston, R. Farrel, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to rule Based Programming.* Addison-Wesley, 1985.

[Bry89] F. Bry. Towards an Efficient Evaluation of General Queries: Quantifier and Disjunction Processing Revisited. In *Proceedings of the 1989 ACM SIGMOD Int. Conf.*, Portland, Oregon, 1989.

[DE88] L.M. Delcambre and J.N. Etheredge. The Relational Production Language: a Production Language for Relational Database. In *Proceedings of 2-nd International Conference on Expert Database System*, Redwood City, 1988.

[For82] C. Forgy. Rete, a fast algorithm for the many patterns many objects match problem. *J. Artificial Intelligence*, 19:17–37, 1982.

[Han92] E. Hanson. Rule Condition Testing and Action Execution in Ariel. *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, June 1992.

[HW92] E. Hanson and J. Widom. An Overview of Production Rules in Database Systems. Research Report RJ 9023 (80483), IBM, Oct. 1992.

[KdMS90] G. Kiernan, Ch. de Maindreville, and E. Simon. Making Deductive Database a Practical Technology: a Step Forward. In *Proc. ACM SIGMOD International Conference on Management of Data*, Atlantic City, May 1990.

[Mir87] D.P. Miranker. Treat: A better match algorithm for AI production systems. In *Proceedings of the National Conference on Artificial Intelligence*, Seattle, Washington, 1987.

[Pai83] R. Paige. Transformational programming–applications to algorithms and systems. In *Proc. Tenth ACM Symp. on Principles of Programming Language*, Jan 1983.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamanios. On Rules, Procedures, Caching and Views in Data Base Systems. In *Proc. International Conference SIGMOD*, Atlantic City, May 1990.

[SKdM92] E. Simon, J. Kiernan, and C. de Maindreville. Implementing High Level Active Rules on top of a Relational DBMS. In *Proc. International Conference on Very Large Databases*, Vancouver, British Columbia, Aug. 1992.

[SLR88] T. Sellis, C. Lin, and L. Raschid. Implementing large production systems in a DBMS environment: Concepts and algorithms. In *Proc. International Conference SIGMOD*, Chicago, May 1988.

[SZ91] A. Seguev and J. Leon Zhao. Data Management for Large Rule Systems. In *Seventh International Conference on Data Engineering*, kobe, Japan, April 1991.

[Ull89] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1989.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to starburst. In *Proc. International Conference on Very Large Databases, Barcelona*, Spain, 1991.

Appendix

choose_source Algorithm
input: a conjunction Φ, its predicate connexion graph $\mathcal{G}$, and a set of changes $\mathcal{C}$;
output: P = a positive literal of Φ st its vertex is not marked in $\mathcal{G}$;
let P denote an unmarked positive literal ;
let $\mathcal{C}_P$ denote the set of changes to P in $\mathcal{C}$;
apply the ordered list of statements until one P is found :
/* searches for an invariant wrt $\mathcal{C}$ */
1. return the most selective (henceforth, m.s.) P s.t. $\mathcal{C}$ has no effect on P;
/* seek for a literal whose defined relation is stable on some attributes */
2. return m.s. P s.t. Φ is autonomous for P wrt $\mathcal{C}_P$ by 3^{rd} autonomy criterium ;
/* seek for a literal whose defined relation can only decrease due to $\mathcal{C}$ */
3. return m.s. P s.t. Φ is autonomous for P wrt $\mathcal{C}_P$ by 1^{st} autonomy criterium ;
4. return m.s. P s.t. Φ is autonomous for P wrt $\mathcal{C}_P$ by any criteria ;
5. return m.s. P ;

choose_max_subexpression Algorithm
input: a conjunction Φ, its predicate connexion graph $\mathcal{G}$, a set of changes $\mathcal{C}$, P a literal of Φ , and g an autonomous expression wrt $\mathcal{C}$;
output: g = an autonomous expression wrt $\mathcal{C}$;
/* the literals of g are marked in $\mathcal{G}$ */
if P is positive then compute
$\mathcal{V}_P$ = {unmarked literal V st V node is connected to P in $\mathcal{G}$ and $g \wedge V$ is autonomous wrt $\mathcal{C}$} ;
while $\mathcal{V}_P \neq \emptyset$ do
1. choose an adjacent vertex V in $\mathcal{V}_P$:
let $\mathcal{C}_V$ denote the set of changes to V in $\mathcal{C}$;
apply the ordered list of statements until one V is found :
1. return negative V s.t. $\mathcal{C}$ has no effect on V;
2. return m.s. positive V s.t. $\mathcal{C}$ has no effect on V ;
3. return m.s. V s.t. Φ is autonomous for V wrt $\mathcal{C}_V$ by 2^{nd} autonomy criterium ;
4. return m.s. V s.t. Φ is autonomous for V wrt $\mathcal{C}_V$ by 1^{st} autonomy criterium ;
5. return m.s. V s.t. Φ is autonomous for V wrt $\mathcal{C}_V$ by 3^{rd} autonomy criterium ;
6. return m.s. V ;
2. extend g with V and mark V;
3. visit $\mathcal{G}$ from V and expand g :
$g \leftarrow g \wedge$ *choose_max_subexpression* $(\Phi, \mathcal{G}, \mathcal{C}, V, g)$;
update $\mathcal{V}_P$ according to g ;
od
endif
return g;