Using Delta Relations to Optimize Condition Evaluation in Active Databases*

Elena Baralis[1] and Jennifer Widom[2]

[1] Politecnico di Torino - Torino, Italy
baralis@polito.it
[2] Stanford University - Stanford, USA
widom@cs.stanford.edu

Abstract. We give a method for improving the efficiency of condition evaluation during rule processing in active database systems. The method derives, from a rule condition, two new conditions that can be used in place of the original condition when a previous value (true or false) of the original condition is known. The derived conditions are generally more efficient to evaluate than the original condition because they are *incremental*—they replace references to entire database relations by references to *delta relations*, which typically are much smaller. Delta relations are accessible to rule conditions in almost all current active database systems, making this optimization broadly applicable. We describe an attribute grammar based approach that we have used to implement our condition rewriting technique.

1 Introduction

Active database systems allow users to specify *event-condition-action* rules that are processed automatically by the database system in response to data manipulation by users and applications. A rule's *event* specifies what causes the rule to become triggered; typical (simple) triggering events are data modification or data retrieval. A rule's *condition* is a further qualification of a triggered rule, usually expressed as a predicate or query over the database. A rule's *action* is performed when the rule is triggered and its condition is true; actions usually are sequences of arbitrary database commands. Most current active database systems, both research prototypes and commercial systems, use this rule paradigm; see [19].

Rule processing in active database systems usually consists of an iterative cycle in which: (1) a triggered rule is selected; (2) the rule's condition is evaluated; (3) if the condition is true the rule's action is executed. In this paper we give a method for optimizing step (2) in this cycle for active databases that use the relational model.[3] Our method is based on the following two assumptions:

* This work was partially performed while the authors were at the IBM Almaden Research Center, San Jose, CA. At Stanford this work was supported by the Anderson Faculty Scholar Fund and by equipment grants from Digital Equipment Corporation and IBM Corporation.

[3] Adapting the method to active OODB's is planned for further work; see Section 6.

1. During the rule processing cycle, the rule processor may be aware of the previous value of a rule's condition (true or false), either because the rule was evaluated earlier during rule processing, or because the rule is enforcing an integrity constraint that was consistent at the beginning of the transaction.
2. The rule language permits references to *delta relations*, which contain the data that was modified in a relation since the last time the rule was evaluated or since the beginning of the transaction.[4]

Although these assumptions may appear strong, we note that many of the prominent relational active database systems do satisfy the assumptions, including *A-RDL* [15], *Ariel* [9], *Heraclitus* [8], and *Starburst* [20]. Furthermore, for those active database systems that do not exactly satisfy this paradigm, e.g., *Postgres* [17], relatively straightforward modifications of our method should be applicable.

Given a rule r with condition C, our method statically derives from C two "optimized" conditions, $PT(C)$ (the *Previously True* condition) and $PF(C)$ (the *Previously False* condition). When rule r is selected at run time, if it is known that r's condition was previously true, then $PT(C)$ is evaluated instead of C. Similarly, if it is known that r's condition was previously false, then $PF(C)$ is evaluated instead of C. $PT(C)$ and $PF(C)$ reference delta relations where C references entire relations, so $PT(C)$ and $PF(C)$ are likely to be much more efficient to evaluate than C.

For generality (and for ease in proving correctness) we present our method using a rule condition language based on relational algebra. The adaptation of our method for a condition language based on SQL or Quel is straightforward. We also specify our method as an attribute grammar; this allows a direct implementation of the method using a compiler-generator such as YACC [10].

1.1 Previous Related Work

There is a clear connection between our work and the well-studied problem of *incremental evaluation*, especially as addressed in [12, 13, 16]. [13] proposes an incremental optimization technique for rule conditions, with similar goals to ours. However, rule conditions are restricted to Select-Project-Join (SPJ) expressions, and all relations are required to have user-accessible tuple identifiers. In contrast, our rule conditions are more general than SPJ expressions (see Section 3), and user-level tuple identifiers are not required. [12] proposes a set of transformations that compute incremental changes to arbitrary relational expressions. Although the methods in [12] do apply to (a small variation on) the problem we consider, the application itself is not direct and is rather difficult to understand. Furthermore, the method in [12] sometimes determines that there have been additions to and deletions from an expression when in fact the net effect of the changes is null. This produces an unnecessary reevaluation of the relational expression,

[4] We assume that delta relations can be accessed efficiently, since in most systems delta relations are stored or indexed in main memory. We also assume that delta relations typically are much smaller than the corresponding entire relations.

while our method correctly detects that there has been no change. [16] presents a "partial differentiation" technique to derive incremental rule conditions. Like [12], the approach in [16] may produce a superset of the actual changes; in [16] an extra "filtering" step is introduced to eliminate the extra changes. In all three of [12, 13, 16], update modifications are modeled as deletions followed by insertions. We handle update modifications directly, which in some cases produces more efficient incremental expressions than those in [12, 13, 16], particularly when only certain attributes are updated (see our γ transformation in Section 4). Note also that our attribute grammar specification is a unique approach that leads directly to an implementation.

In [14], a classification is given of techniques to optimize the performance of transaction execution when transactions include active rules. One of the suggested techniques is to detect in advance when a rule will have no effect on the database because the rule's condition is guaranteed to be false. This optimization is more effective than ours when it is applicable, but it applies only in very special cases, and it requires compile-time analysis of transaction code, which our method does not.

Some active database systems use methods based on *Rete* or *TREAT* networks [18] for efficient condition evaluation, e.g., [7, 9]. Unfortunately, these methods apply only to rule languages where references to a relation R implicitly reference delta relations for R, and to rule conditions that are restricted to SPJ expressions. We consider more general conditions, and we determine those scenarios in which R can be replaced by its delta relations. Commercial active rule languages appear to be following our model, so our techniques should be applicable in very practical settings.

Finally, note that in [6] we suggest techniques similar to those we present here, but only a very restricted case is described. In this paper we elaborate the suggestions of [6] in a general context.

1.2 Outline of the Paper

In Section 2 we give a more rigorous description of active database rule processing and we formalize the notion of delta relations. In Section 3 we define our condition specification language and we provide some examples. Section 4 is the core technical section: it contains our method for rule condition rewriting and several examples of the method. Section 5 specifies an implementation of the method using an attribute grammar. Finally, Section 6 concludes and proposes improvements and extensions to our technique.

2 Rule Processing and Delta Relations

Consider an active database system in which a set of event-condition-action rules are defined as described in Section 1. Suppose further that a set of user or application modifications are performed on the database, then rule processing is invoked. The pseudo-code in Figure 1 describes the general behavior of the

```
S = initial DB state
user or application modifications
S' = new DB state
repeat until no rules are triggered:
    select a triggered rule r
    evaluate r's condition based on S' and delta relations
    if true, execute r's action based on S' and delta relations
    S' = new DB state
```

Fig. 1. Active database system behavior

system. Note that issues such as the "granularity" of rule processing (i.e., when rule processing is invoked relative to triggering events) and the method for selecting among multiple triggered rules do not affect our method. Furthermore, our method also applies to rule languages in which triggering events are implicit rather than explicit, e.g., [9, 14, 15, 16].

When a rule's condition is evaluated and its action is executed, this occurs with respect to a database *transition*, i.e., the changes that have occurred since some previous database state. We consider a semantics in which each rule uses the transition since that rule was last selected, or since the original state (state S in Figure 1) if the rule has not yet been selected during rule processing. While this is the semantics taken by many active database systems, systems with slightly different semantics may require corresponding modifications to our method.

Delta relations encapsulate the changes that have occurred during a rule's transition, and they may be referenced in a rule's condition and action. For each relation R we assume four delta relations:

- *inserted*(R) contains the tuples inserted into R during the transition.
- *deleted*(R) contains the tuples deleted from R during the transition.
- *old-updated*(R) contains the pre-transition values of the tuples modified in R during the transition.
- *new-updated*(R) contains the current (i.e., new) values of the tuples modified in R during the transition.

In addition, *new-updated* and *old-updated* may be restricted to sets of attributes. Let $A_1, \ldots, A_n$ be attributes of relation R. Then:

- *old-updated*($R, \{A_1, \ldots, A_n\}$) contains the pre-transition values of the tuples in R for which at least one of $A_1, \ldots, A_n$ was modified during the transition.
- *new-updated*($R, \{A_1, \ldots, A_n\}$) contains the current values of the tuples in R for which at least one of $A_1, \ldots, A_n$ was modified during the transition.

Typically, delta relations reflect the *net effect* of database modifications; that is, they contain only the net result of successive actions over the same tuple.

This concept of net effect is used widely, e.g., [9, 15, 21], so we assume it here. Note that one implication of using net effects is that *inserted*(R), *deleted*(R), *old-updated*(R), and *new-updated*(R) are disjoint.

We introduce four abbreviations that are used throughout the remainder of the paper:

(1) $\Delta^{+}(R) = \mathit{inserted}(R) \cup \mathit{new\text{-}updated}(R)$
(2) $\Delta^{-}(R) = \mathit{deleted}(R) \cup \mathit{old\text{-}updated}(R)$
(3) $\Delta^{+}(R, \{A_1, \ldots, A_n\}) = \mathit{inserted}(R) \cup \mathit{new\text{-}updated}(R, \{A_1, \ldots, A_n\})$
(4) $\Delta^{-}(R, \{A_1, \ldots, A_n\}) = \mathit{deleted}(R) \cup \mathit{old\text{-}updated}(R, \{A_1, \ldots, A_n\})$

Note that in (3) and (4), if the attribute list is empty, then Δ^{+} and Δ^{-} degenerate to *inserted*(R) and *deleted*(R). We informally refer to (both versions of) Δ^{+} and Δ^{-} as *incremental* and *decremental changes,* respectively.

Sometimes our optimized conditions require access to the "old" value of a relation, i.e., the relation's pre-transition value. We denote the old value of a relation R as R^{O}. While some active database rule languages provide a feature for accessing R^{O} directly, others do not. However, R^{O} always can be derived from the current value of R and R's delta relations, based on the equivalence:

$$R^{O} = (R - \Delta^{+}(R)) \cup \Delta^{-}(R)$$

3 Condition Language

In active database rule languages, conditions are sometimes expressed as predicates and sometimes as queries, where in the latter case usually the interpretation is that the condition is true iff the query produces a non-empty result. It can easily be shown that the two representations are equivalent [2]; we represent conditions as predicates. The grammar of our condition language is given in Figure 2. The language is powerful enough to describe any condition expressible in relational algebra or calculus extended with aggregate functions, with the exception of duplicates and ordering conditions.

In the grammar's productions, terminal symbol `R` stands for a relation name and `R.A` for an attribute of relation `R`. The meaning of the language is mostly self-explanatory. Condition $\exists(\mathit{Rexp})$ is satisfied iff relational expression *Rexp* produces one or more tuples, while condition $\neg\exists(\mathit{Rexp})$ is satisfied iff *Rexp* produces no tuples. We assume a set semantics, i.e., no duplicates. Note the following points:

- Although not included in the grammar explicitly, joins can be expressed using cross-product and selection.
- A selection that is a boolean combination of comparisons can be expressed in our language by using the following equivalences:

$$\sigma_{c_1 \wedge c_2} \mathit{Rexp} = \sigma_{c_1}(\sigma_{c_2} \mathit{Rexp})$$
$$\sigma_{c_1 \vee c_2} \mathit{Rexp} = \sigma_{c_1} \mathit{Rexp} \cup \sigma_{c_2} \mathit{Rexp}$$

$$
\begin{array}{lll}
Cond & ::= & \exists(Rexp) \mid \neg\exists(Rexp) \\
 & \mid & Cond_1 \wedge Cond_2 \mid Cond_1 \vee Cond_2 \mid (Cond) \\
Rexp & ::= & \mathtt{R} \mid Rexp_1 \cup Rexp_2 \\
 & \mid & Rexp_1 \times Rexp_2 \mid Rexp - SimpleRexp \\
 & \mid & \sigma_{Compare} Rexp \mid \pi_{AList} Rexp \\
 & \mid & Aggr(Attr_1[, Attr_2])Rexp \mid (Rexp) \\
SimpleRexp & ::= & \mathtt{R} \mid SimpleRexp_1 \cup SimpleRexp_2 \\
 & \mid & SimpleRexp_1 \times SimpleRexp_2 \mid \sigma_{Compare} SimpleRexp \\
 & \mid & \pi_{AList} SimpleRexp \mid (SimpleRexp) \\
Compare & ::= & Term_1 \ Op \ Term_2 \\
Term & ::= & Attr \mid \mathtt{Const} \\
Op & ::= & > \mid < \mid \leq \mid \geq \mid = \mid \neq \\
Aggr & ::= & \mathtt{sum} \mid \mathtt{avg} \mid \mathtt{min} \mid \mathtt{max} \mid \mathtt{count} \\
AList & ::= & Attr_1, \ldots, Attr_n \\
Attr & ::= & \mathtt{R.A}
\end{array}
$$

Fig. 2. Condition language syntax

- Negation of selection predicates can be expressed by repeatedly applying DeMorgan's laws and then negating the innermost comparisons (i.e., $\leq$ becomes $>$, $=$ becomes $\neq$, etc.).
- *Terms* may be arithmetic expressions over attributes and constants without affecting our method, although for simplicity we have omitted this feature from our grammar.
- The *Aggr* operation is for handling the aggregate functions `sum`, `avg`, `min`, etc. It extends a given relational expression with a new attribute containing the computed value of the aggregate function. The function is computed over the attribute $Attr_1$ and grouping may optionally be performed by specifying the $Attr_2$ attribute; see [4] for a detailed description of this construct.
- "Simple relational expressions" (*SimpleRexp*) are introduced to restrict the expressions that may appear as the second operand of a difference. The need for this restriction is explained in Section 4.[5]
- Most active database rule languages permit explicit references to delta relations in rule conditions. Therefore, technically we should include *Rexp* productions for *inserted* (R), *deleted* (R), etc. in our grammar. However, since there is no need to attempt optimization for these references, for clarity and simplicity we omit them from the grammar; adding them is trivial.

3.1 Examples

We give two examples of conditions expressed in our language, which are adapted from a case study in [3, 6] involving an electrical power distribution network;

[5] This restriction does not reduce the expressive power of our condition language with respect to relational algebra; see [2].

these examples use the two relations WIRE(wire-id,from,to,voltage) and TUBE(tube-id,from,to,protected), which we abbreviate as W and T, respectively.

Example 3.1: Informally: Some unprotected tube contains a high voltage ($> 5k$) wire. In our condition language:

$$\exists(\sigma_{\texttt{W.<from,to>=T.<from,to>}}(\sigma_{\texttt{W.voltage>5k}}\texttt{W} \times \sigma_{\texttt{T.protected=false}}\texttt{T}))$$

where $\sigma_{\texttt{W.<from,to>=T.<from,to>}}$ is an abbreviation for $\sigma_{\texttt{W.from=T.from}}\sigma_{\texttt{W.to=T.to}}$.

Example 3.2: Informally: Some tube contains no wires. More precisely, there is some tube such that no wire has the same from and to attributes. In our condition language:

$$\exists(\texttt{T} - \pi_{\texttt{schema(T)}}(\sigma_{\texttt{T.<from,to>=W.<from,to>}}(\texttt{T} \times \texttt{W})))$$

where we use schema(T) as an abbreviation for a list of all the attributes of T.

4 Derivation of Optimized Conditions

Let C be a condition expressed using the language of Section 3. Suppose we want to evaluate C with respect to a database state S' and the transition from some previous database state S. Further suppose that the result of C in state S is known, i.e., C was either true or false in S.[6] In this case we use one of two optimized conditions in place of C:

- $PF(C)$ (PF for "Previously False") is chosen when the outcome of the previous evaluation of C was false.
- $PT(C)$ (PT for "Previously True") is chosen when the outcome of the previous evaluation of C was true.

In general, $PF(C)$ only provides a useful optimization for conditions with existential quantifications ($\exists$) and disjunction, while $PT(C)$ only provides a useful optimization for conditions with negative existential quantification ($\neg\exists$) and conjunction. Intuitively, this is for the following reasons. When the condition was previously false, an existential quantification tells us that no data satisfied the relational expression. Hence we can check if data now satisfies the relational expression by checking changed data only. However, suppose we have a negative existential quantification. Then, since the condition was previously false, some data did previously satisfy the relational expression. In this case it is impossible to tell, by examining changed data only, whether the relational expression is now empty. If a previously false condition contains disjuncts, then we know that all

[6] Recall from Figure 1 that this information is available whenever C's rule has been selected previously or the value of C was known in the initial state before user modifications.

C	$PF(C)$	$PT(C)$
$\exists(Rexp)$	$\exists(\vartheta(Rexp))$	$\exists(Rexp)$
$\neg\exists(Rexp)$	$\neg\exists(Rexp)$	$\neg\exists(\vartheta(Rexp))$
$Cond_1 \wedge Cond_2$	$Cond_1 \wedge Cond_2$	$PT(Cond_1) \wedge PT(Cond_2)$
$Cond_1 \vee Cond_2$	$PF(Cond_1) \vee PF(Cond_2)$	$Cond_1 \vee Cond_2$
$(Cond)$	$(PF(Cond))$	$(PT(Cond))$

Table 1. The PF and PT optimized conditions

disjuncts were false and we can optimize each one based on that knowledge. However, if the condition contains conjuncts, then we don't know which conjuncts were previously false, and optimization is impossible. The same argument, in converse, holds for previously true conditions.[7]

In $PF(C)$ and $PT(C)$, each reference to a relation R is replaced by one of its corresponding incremental or decremental changes, $\Delta^+(R)$ or $\Delta^-(R)$, whenever possible. We define $PF(C)$ and $PT(C)$ by means of transformation rules based on the structure of C according to the grammar of Figure 2. The rules are given in Tables 1, 2, and 3; the rules are applied inductively to derive $PF(C)$ and $PT(C)$ for an arbitrarily complex condition. The correctness of these rules is shown in [2]. We now provide more intuition for the rules, including an explanation for certain details of Tables 1, 2, and 3.

$Rexp$	$\vartheta(Rexp)$
$\mathtt{R}$	$\Delta^+(\mathtt{R})$
$Rexp_1 \cup Rexp_2$	$\vartheta(Rexp_1) \cup \vartheta(Rexp_2)$
$Rexp_1 \times Rexp_2$	$(\vartheta(Rexp_1) \times Rexp_2) \cup (Rexp_1 \times \vartheta(Rexp_2))$
$Rexp_1 - SimpleRexp_2$	$(\vartheta(Rexp_1) - SimpleRexp_2) \cup (Rexp_1 \cap \vartheta'(SimpleRexp_2))$
$\sigma_{Compare} Rexp$	$\sigma_{Compare}(\vartheta(Rexp))$
$\pi_{AList} Rexp$	$\pi_{AList}(\vartheta(Rexp))$
$Aggr(Attr_1[, Attr_2])Rexp$	$Aggr(Attr_1[, Attr_2])Rexp$
$(Rexp)$	$(\vartheta(Rexp))$

Table 2. The ϑ transformation for relational expressions

In Table 1, the rules for $PF(C)$ and $PT(C)$ use a transformation ϑ, which is applied to relational expressions. Table 2 contains the transformation rules for

[7] Certainly we might do somewhat better here, e.g., keep track of which conjuncts/disjuncts were previously true/false, keep track of which data previously satisfied the condition, or handle certain special cases. We plan to investigate these improvements as future work.

$SimpleRexp$	$\vartheta'(SimpleRexp)$
R	Δ^{-}(R)
$SR_1 \cup SR_2$	$(\vartheta'(SR_1) - SR_2) \cup (\vartheta'(SR_2) - SR_1)$
$SR_1 \times SR_2$	$(\vartheta'(SR_1) \times SR_2^O) \cup (SR_1^O \times \vartheta'(SR_2))$
$\sigma_{Compare}SR$	$\sigma_{Compare}(\vartheta'(SR))$
$\pi_{AList}SR$	$\pi_{AList}(\vartheta'(SR)) - \pi_{AList}SR$
(SR)	$(\vartheta'(SR))$

Table 3. The ϑ' transformation for simple relational expressions

ϑ applied to an arbitrary relational expression *Rexp*, while Table 3 contains an additional transformation ϑ' applied to an arbitrary simple relational expression *SimpleRexp*. (Recall that simple relational expressions are those relational expressions that can appear as the second operand of a difference. Hence, ϑ' is applied in the fourth line of Table 2.) Intuitively, ϑ applied to a relational expression *Rexp* produces an optimized expression $Rexp'$ that computes the incremental changes to *Rexp*. When ϑ is applied to an expression with the difference operation, decremental changes to the second operand cause incremental changes to the entire expression. Hence, ϑ' is a "negated" version of ϑ that computes these decremental changes. Decremental changes can be computed only on *monotonic* relational expressions. This explains our introduction of simple relational expressions: simple relational expressions exclude difference and aggregate operators, which are non-monotonic.

Observe the following points:

- For convenience, we use $\cap$ in ϑ applied to $Rexp_1 - SimpleRexp_2$. Expression $Rexp_1 \cap Rexp_2$ is equivalent to $Rexp_1 \times Rexp_2$ with a selection condition equating all corresponding attributes and appropriate projection.
- The computation of an aggregate function over a relational expression always requires the entire relational expression result (not just an incremental portion), thus aggregate expressions cannot be optimized in the general case. However, conditions containing aggregate function expressions as operands still may be optimized in their other operands.
- Note that in ϑ, the treatment of projection and union are substantially simpler than in general incremental query evaluation such as [12].
- ϑ' on cartesian products refers to "old" relational expression SR^O. Here we are using SR^O as an abbreviation for SR with all relation names R replaced by R^O, denoting the old value of R. (Recall that if old relations are not directly accessible, they can be derived from delta relations as described in Section 2.)
- For ϑ' applied to projections, if the attribute list *AList* contains a key for the expression SR (as is often the case), then our formula can be simplified to $\pi_{AList}(\vartheta'(SR))$.

While we expect the sizes of $\Delta^+(R)$ and $\Delta^-(R)$ in $PF(C)$ and $PT(C)$ to be small (much smaller than R), we can further reduce the sizes of $\Delta^+(R)$ and $\Delta^-(R)$ by ignoring all updates to attributes that do not influence the outcome of the condition. Let R' denote a specific reference to relation R in a condition C. We compute the *relevant attribute set* $\rho(R', C)$, where $\rho(R', C)$ is the set of all attributes in R whose updates can affect the outcome of condition C with respect to reference R'. Let $Rexp(R')$ be any relational expression (or simple relational expression) in C containing the reference R'. For each attribute A_i of R, $A_i \in \rho(R', C)$ iff:

1. A_i appears in a selection comparison over $Rexp(R')$ (predicate *Compare* in $\sigma_{Compare}$), or
2. A_i is used for aggregate computation or grouping on $Rexp(R')$ ($Attr_1$ or $Attr_2$ in *Aggr* operation), or
3. there is a union or difference applied to $Rexp(R')$.

Thus, given an original condition C and optimized condition C' (either $PF(C)$ or $PT(C)$), we further optimize C' as $\gamma(C')$, where γ replaces every reference $\Delta^+(R')$ or $\Delta^-(R')$ in C' by $\Delta^+(R', \rho(R', C))$ or $\Delta^-(R', \rho(R', C))$. This additional optimization has not been suggested previously (to the best of our knowledge); it can be very effective in practice, as shown by the examples in the next section.

The proof of correctness of the $PF(C)$ and $PT(C)$ transformations is given by Theorem 1. Due to space constraints, the proof is omitted; it appears in [2].

Theorem 1: Let C denote any condition specified using the language of Figure 2. Then:

(a) If C was false in state S, C is true in state S' iff $PF(C)$ is true in state S'.
(b) If C was true in state S, C is true in state S' iff $PT(C)$ is true in state S'. □

4.1 Examples

We show the optimized conditions for the two examples introduced in Section 3.1.

Example 4.1: Some unprotected tube contains a high voltage ($> 5k$) wire. In our condition language:

$$C = \exists(\sigma_{\texttt{W.<from,to>=T.<from,to>}}(\sigma_{\texttt{W.voltage>5k}}\texttt{W} \times \sigma_{\texttt{T.protected=false}}\texttt{T}))$$

The optimized conditions are:

$$PF(C) = \exists(\sigma_{join}(\sigma_{volt}\, incr\text{-}W \times \sigma_{unprot}\texttt{T}) \cup (\sigma_{volt}\texttt{W} \times \sigma_{unprot}\, incr\text{-}T)))$$
$$PT(C) = C$$

where *join* is $\texttt{W.<from,to>} = \texttt{T.<from,to>}$, *volt* is $\texttt{W.voltage} > \texttt{5k}$, *unprot* is $\texttt{T.protected} = \texttt{false}$, *incr-W* is $\Delta^{+}(\texttt{W}, \{\texttt{voltage}, \texttt{from}, \texttt{to}\})$, and *incr-T* is $\Delta^{+}(\texttt{T}, \{\texttt{from}, \texttt{to}, \texttt{protected}\})$. Note that this same example could not be optimized using the method presented in [6].

Example 4.2: Some tube contains no wires. In our condition language:

$$C = \exists(\texttt{T} - \pi_{\texttt{schema(T)}}(\sigma_{\texttt{T.<from,to>=W.<from,to>}}(\texttt{T} \times \texttt{W})))$$

Since this example has a difference operator, transformation ϑ' is used as well as ϑ. Recall that $\texttt{W}^{\texttt{0}}$ and $\texttt{T}^{\texttt{0}}$ refer to the pre-transition (old) states of W and T, respectively. Note also that here delta relations cannot be restricted to updates on specific attributes (because of the difference operator), and that we are applying the simplified formula for ϑ' applied to projections since the projected attributes form a key. The optimized conditions are:

$$PF(C) = \exists((\Delta^{+}(\texttt{T}) - \pi_t(\sigma_j(\texttt{T} \times \texttt{W}))) \cup (\texttt{T} \cap \pi_t(\sigma_j((\Delta^{-}(\texttt{T}) \times \texttt{W}^{\texttt{0}}) \cup (\texttt{T}^{\texttt{0}} \times \Delta^{-}(\texttt{W}))))))$$
$$PT(C) = C$$

where t is $\texttt{schema(T)}$, and j is $\texttt{T.<from,to>} = \texttt{W.<from,to>}$. As in Example 4.1, this condition could not be optimized using the method in [6].

Numerous additional examples can be found in [2].

5 Attribute Grammar Implementation

Now we address the issue of implementing our approach. Suppose a new rule r with condition C is added to the database (or an existing rule's condition is to be optimized). The following steps are followed:

1. C is translated into our condition language.
2. The optimized conditions $PF(C)$ and $PT(C)$ are generated.
3. $PF(C)$ and $PT(C)$ are translated back to the rule's condition language.[8]
4. At run-time, whenever possible, the evaluation of C is replaced by the evaluation of $PF(C)$ or $PT(C)$.

Note that steps 1–3 are *static*: they are performed only once, when rule r is first defined. At run-time, the system evaluates an optimized condition (step 4) just as it would have evaluated the original condition.

We consider in detail the implementation of step 2. The ϑ and γ transformations can be computed during the parsing of a condition to yield the optimized conditions. Thus, we have chosen to implement the derivation of optimized conditions using an *attribute grammar*. In an attribute grammar, each symbol is

[8] Note from Section 4 that our process is highly unlikely to yield constructs in $PF(C)$ and $PT(C)$ that are not available in the language used to specify the original rule condition.

$$
\begin{array}{lll}
1. & Cond ::= & \exists(Rexp) \\
 & & \mathsf{Rexp}.\gamma := \emptyset \\
 & & \mathsf{Cond.PF} := \exists(\mathsf{Rexp}.\vartheta) \\
 & & \mathsf{Cond.PT} := \exists(\mathsf{Rexp.E}) \\
 & & \mathsf{Cond.C} := \exists(\mathsf{Rexp.E}) \\
2. & \quad\quad | & \neg\exists(Rexp) \\
 & & \mathsf{Rexp}.\gamma := \emptyset \\
 & & \mathsf{Cond.PF} := \neg\exists(\mathsf{Rexp.E}) \\
 & & \mathsf{Cond.PT} := \neg\exists(\mathsf{Rexp}.\vartheta) \\
 & & \mathsf{Cond.C} := \neg\exists(\mathsf{Rexp.E}) \\
3. & \quad\quad | & Cond_1 \wedge Cond_2 \\
 & & \mathsf{Cond.PF} := \mathsf{Cond_1.C} \wedge \mathsf{Cond_2.C} \\
 & & \mathsf{Cond.PT} := \mathsf{Cond_1.PT} \wedge \mathsf{Cond_2.PT} \\
 & & \mathsf{Cond.C} := \mathsf{Cond_1.C} \wedge \mathsf{Cond_2.C} \\
4. & \quad\quad | & Cond_1 \vee Cond_2 \\
 & & \mathsf{Cond.PF} := \mathsf{Cond_1.PF} \vee \mathsf{Cond_2.PF} \\
 & & \mathsf{Cond.PT} := \mathsf{Cond_1.C} \vee \mathsf{Cond_2.C} \\
 & & \mathsf{Cond.C} := \mathsf{Cond_1.C} \vee \mathsf{Cond_2.C} \\
5. & \quad\quad | & (Cond_1) \\
 & & \mathsf{Cond.PF} := (\mathsf{Cond_1.PF}) \\
 & & \mathsf{Cond.PT} := (\mathsf{Cond_1.PT}) \\
 & & \mathsf{Cond.C} := (\mathsf{Cond_1.C})
\end{array}
$$

Fig. 3. Attribute grammar for the *Cond* production

allowed to have a fixed number of associated values, called attributes, and each grammar production has a set of attribute evaluation rules. Attributes can be used to pass information up a syntax tree: these are called *synthesized attributes*, and the evaluation rules associated with each production describe how the attributes' left-hand-side (LHS) values are computed from their right-hand-side (RHS) values. If instead the information flows down the syntax tree, these are called *inherited attributes*, and their evaluation rules describe how RHS attribute values are computed as a function of LHS values. For a more detailed description of attribute grammars refer to [1].

Our attribute grammar computes the ϑ and γ transformations at the same time. At the end of the parsing process, the grammar produces the optimized conditions $PF(C)$ and $PT(C)$ as attributes. The grammar uses one inherited attribute (γ) and eight synthesized attributes (ϑ, ϑ', PF, PT, A, C, E, EO), as follows:

- The ϑ and ϑ' synthesized attributes implement the ϑ and ϑ' transformations. At any time during the parsing process, attribute ϑ for a relational expression *Rexp* contains ϑ applied to *Rexp*. Attribute ϑ' is similarly defined for simple relational expressions.
- The PF and PT synthesized attributes are used to build the complete optimized condition, using as building blocks the optimized conditions provided

6. $Rexp ::= \mathtt{R}$

$\mathsf{R}.\gamma := \pi_{\mathsf{schema(R)}}(\mathsf{Rexp}.\gamma)$

$\mathsf{Rexp}.\vartheta := \Delta^{+}(\mathtt{R}, \{\mathtt{R}.\gamma\})$

$\mathsf{Rexp.E} := \mathsf{R}$

7. $\mid Rexp_1 \cup Rexp_2$

$\mathsf{Rexp_1}.\gamma := \mathsf{All(Rexp_1)}$

$\mathsf{Rexp_2}.\gamma := \mathsf{All(Rexp_2)}$

$\mathsf{Rexp}.\vartheta := \mathsf{Rexp_1}.\vartheta \cup \mathsf{Rexp_2}.\vartheta$

$\mathsf{Rexp.E} := \mathsf{Rexp_1.E} \cup \mathsf{Rexp_2.E}$

8. $\mid Rexp_1 \times Rexp_2$

$\mathsf{Rexp_1}.\gamma := \mathsf{Rexp}.\gamma$

$\mathsf{Rexp_2}.\gamma := \mathsf{Rexp}.\gamma$

$\mathsf{Rexp}.\vartheta := (\mathsf{Rexp_1}.\vartheta \times \mathsf{Rexp_2.E}) \cup (\mathsf{Rexp_1.E} \times \mathsf{Rexp_2}.\vartheta)$

$\mathsf{Rexp.E} := \mathsf{Rexp_1.E} \times \mathsf{Rexp_2.E}$

9. $\mid Rexp_1 - SimpleRexp_2$

$\mathsf{Rexp_1}.\gamma := \mathsf{All(Rexp_1)}$

$\mathsf{SimpleRexp_2}.\gamma := \mathsf{All(SimpleRexp_2)}$

$\mathsf{Rexp}.\vartheta := (\mathsf{Rexp_1}.\vartheta - \mathsf{SimpleRexp_2.E}) \cup (\mathsf{Rexp_1.E} \cap \mathsf{SimpleRexp_2}.\vartheta')$

$\mathsf{Rexp.E} := \mathsf{Rexp_1.E} - \mathsf{SimpleRexp_2.E}$

10. $\mid \sigma_{Compare} Rexp_1$

$\mathsf{Rexp_1}.\gamma := \mathsf{Rexp}.\gamma \oplus \mathsf{Compare.A}$

$\mathsf{Rexp}.\vartheta := \sigma_{\mathsf{Compare}}(\mathsf{Rexp_1}.\vartheta)$

$\mathsf{Rexp.E} := \sigma_{\mathsf{Compare}}(\mathsf{Rexp_1.E})$

11. $\mid \pi_{AList} Rexp_1$

$\mathsf{Rexp_1}.\gamma := \mathsf{Rexp}.\gamma$

$\mathsf{Rexp}.\vartheta := \pi_{\mathsf{AList}}(\mathsf{Rexp_1}.\vartheta)$

$\mathsf{Rexp.E} := \pi_{\mathsf{AList}}(\mathsf{Rexp_1.E})$

12. $\mid Aggr(Attr_1[, Attr_2])Rexp_1$

$\mathsf{Rexp_1}.\gamma := \mathsf{Rexp}.\gamma \oplus \mathsf{Attr_1.A} \oplus \mathsf{Attr_2.A}$

$\mathsf{Rexp}.\vartheta := \mathsf{Aggr(Attr_1, Attr_2)Rexp_1.E}$

$\mathsf{Rexp.E} := \mathsf{Aggr(Attr_1, Attr_2)Rexp_1.E}$

13. $\mid (Rexp_1)$

$\mathsf{Rexp_1}.\gamma := \mathsf{Rexp}.\gamma$

$\mathsf{Rexp}.\vartheta := (\mathsf{Rexp_1}.\vartheta)$

$\mathsf{Rexp.E} := (\mathsf{Rexp_1.E})$

Fig. 4. Attribute grammar for the *Rexp* production

by the ϑ attribute. At the end of the parsing process, these attributes contain the optimized conditions $PF(C)$ and $PT(C)$, respectively.

- The γ inherited attribute allows us to progressively build top-down the relevant attribute set ($\rho(R, C)$ from Section 4), by adding the attributes involved in all predicates and aggregate functions while descending the parse tree. When the relation terminal symbol is reached ($\mathtt{R}$), the reduced delta relation is defined using the relevant attribute set contained in the γ attribute.

14. $SRexp$ ::= R
$\mathsf{R}.\gamma := \pi_{\mathsf{schema}(\mathsf{R})}(\mathsf{SRexp}.\gamma)$
$\mathsf{SRexp}.\vartheta' := \Delta^{-}(\mathtt{R}, \{\mathtt{R}.\gamma\})$
$\mathsf{SRexp.E} := \mathtt{R}$
$\mathsf{SRexp.EO} := \mathtt{R}^{0}$

15. | $SRexp_1 \cup SRexp_2$
$\mathsf{SRexp_1}.\gamma := \mathsf{All}(\mathsf{SRexp_1})$
$\mathsf{SRexp_2}.\gamma := \mathsf{All}(\mathsf{SRexp_2})$
$\mathsf{SRexp}.\vartheta' := (\mathsf{SRexp_1}.\vartheta' - \mathsf{SRexp_2.E}) \cup (\mathsf{SRexp_2}.\vartheta' - \mathsf{SRexp_1}.E)$
$\mathsf{SRexp.E} := \mathsf{SRexp_1.E} \cup \mathsf{SRexp_2.E}$
$\mathsf{SRexp.EO} := \mathsf{SRexp_1.EO} \cup \mathsf{SRexp_2.EO}$

16. | $SRexp_1 \times SRexp_2$
$\mathsf{SRexp_1}.\gamma := \mathsf{SRexp}.\gamma$
$\mathsf{SRexp_2}.\gamma := \mathsf{SRexp}.\gamma$
$\mathsf{SRexp}.\vartheta' := (\mathsf{SRexp_1}.\vartheta' \times \mathsf{SRexp_2.EO}) \cup (\mathsf{SRexp_1.EO} \times \mathsf{SRexp_2}.\vartheta')$
$\mathsf{SRexp.E} := \mathsf{SRexp_1.E} \times \mathsf{SRexp_2.E}$
$\mathsf{SRexp.EO} := \mathsf{SRexp_1.EO} \times \mathsf{SRexp_2.EO}$

17. | $\sigma_{Compare} SRexp_1$
$\mathsf{SRexp_1}.\gamma := \mathsf{SRexp}.\gamma \oplus \mathsf{Compare.A}$
$\mathsf{SRexp}.\vartheta' := \sigma_{\mathsf{Compare}}(\mathsf{SRexp_1}.\vartheta')$
$\mathsf{SRexp.E} := \sigma_{\mathsf{Compare}}(\mathsf{SRexp_1.E})$
$\mathsf{SRexp.EO} := \sigma_{\mathsf{Compare}}(\mathsf{SRexp_1.EO})$

18. | $\pi_{AList} SRexp_1$
$\mathsf{SRexp_1}.\gamma := \mathsf{SRexp}.\gamma$
$\mathsf{SRexp}.\vartheta' := \pi_{\mathsf{AList}}(\mathsf{SRexp_1}.\vartheta') - \pi_{\mathsf{AList}}(\mathsf{SRexp_1.E})$
$\mathsf{SRexp.E} := \pi_{\mathsf{AList}}(\mathsf{SRexp_1.E})$
$\mathsf{SRexp.EO} := \pi_{\mathsf{AList}}(\mathsf{SRexp_1.EO})$

19. | $(SRexp_1)$
$\mathsf{SRexp_1}.\gamma := \mathsf{SRexp}.\gamma$
$\mathsf{SRexp}.\vartheta' := (\mathsf{SRexp_1}.\vartheta')$
$\mathsf{SRexp.E} := (\mathsf{SRexp_1.E})$
$\mathsf{SRexp.EO} := (\mathsf{SRexp_1.EO})$

Fig. 5. Attribute grammar for the *SimpleRexp* production

– The A, C, E, and EO synthesized attributes are needed to aid the propagation process. Attribute A allows us to extract from a predicate or aggregate definition the list of relevant attributes. Attributes C, E, and EO propagate up the parse tree the definition of the original condition, and the current and old relational expressions, so that they are available each time they are required in the definition of $PF(C)$, $PT(C)$, ϑ, and ϑ'.

In Figures 3, 4, and 5, the attribute grammars for the *Cond*, *Rexp*, and *SimpleRexp* productions are given, while the remaining productions are given in Figure 6. The renaming of tokens appearing in both the LHS and RHS productions, although not necessary, improves the readability of the grammar. The $\oplus$

20. $Compare$	$::= Term_1\ Op\ Term_2$	
	Compare.A := Term_1.A $\oplus$ Term_2.A	
21. $Term$	$::= Attr$	
	Term.A := Attr.A	
22.	| `Const`	
	Term.A := $\emptyset$	
23. Op	$::= > \mid < \mid \leq \mid \geq \mid = \mid != $	
24. $Aggr$	::= `sum` | `avg` | `min` | `max` | `count`	
25. $AList$	$::= Attr_1, \ldots, Attr_n$	
26. $Attr$	::= `R.a`	
	Attr.A := R.a	

Fig. 6. Attribute grammar for the remaining productions

operator performs list concatenation; it is used to build attribute lists. Note that not every production needs to compute every attribute, since different attributes have different scopes; for details see [1].

We have built a prototype condition rewriter using the parser-generator tools *LEX* [11] and *YACC* [10]. We would like to incorporate our condition rewriting facility into an active database rule system, most likely Starburst [20]. This should be relatively straightforward: Delta relations as used in this paper are directly available in Starburst. The Starburst SQL-based condition language easily translates to and from our condition language. We can generate the optimized conditions and store them with the original condition in the Starburst *Rule Catalog* [20]. It is then sufficient to add to the run-time rule processor the logic for: (a) storing a previous outcome of condition evaluation for each rule, and (b) choosing an optimized condition for evaluation in place of the original condition whenever possible. Based on the implementation architecture of the Starburst Rule System [20], both of these tasks can be performed easily and efficiently.

6 Conclusions and Future Work

We have described a method for improving the condition evaluation phase of active database rule processing. Our method is based on rule conditions expressed in an extension of relational algebra; this provides both a logical formulation and a framework that can apply to multiple rule languages. We have also specified an implementation of our approach based on an attribute grammar.

As future work we plan to:

- Improve our handling of aggregate functions.
- Extend our condition language to more succinctly express certain constructs, so that we can improve our rewriting for these constructs (e.g., negative subqueries, which are now expressed as relational difference).

- Handle special cases where $PF(C)$ and $PT(C)$ can optimize conditions currently not optimized (recall Section 4).
- Consider similar methods in the context of deductive and object-oriented active database systems (e.g., we hope to use these methods in the *IDEA* project [5], where they should apply in a straightforward way).
- Investigate query optimization strategies that exploit references to very small relations, such as the delta relations used in our optimized conditions.
- Implement our method in the Starburst Rule System and experiment with its practical effectiveness under a variety of database and rule processing loads.

Although we believe that our optimized conditions will be much cheaper to evaluate in most scenarios, there may be some cases in which the original condition is cheaper. Our optimized conditions generally have a more complex structure than the original conditions (i.e., a larger number of subconditions), and there may be some run-time situations in which delta relations are not significantly smaller than their corresponding complete relations. Hence, to address this issue, we plan to develop and exploit an appropriate cost model. The cost model will be based on, e.g., the cardinality of the referenced relations versus their delta relations, and the selectivity of predicates appearing in the condition. The cost model then can be used to predict whether or not it will actually be beneficial to evaluate an optimized condition in place of the original condition.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley, Reading, Massachusetts, 1986.
2. E. Baralis. *An Algebraic Approach to the Analysis and Optimization of Active Database Rules.* PhD thesis, Politecnico di Torino, Torino, Italy, Feb. 1994.
3. E. Baralis, S. Ceri, G. Monteleone, and S. Paraboschi. An intelligent database system application: The design of EMS. In *Proc. of the First Int. Conf. on Applications of Databases*, LNCS 851, Vadstena, Sweden, June 1994. Springer-Verlag.
4. S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Trans. Softw. Eng.*, 11(4):324–345, Apr. 1985.
5. S. Ceri and R. Manthey. Consolidated specification of Chimera, the conceptual interface of Idea. Technical Report IDEA.DD.2P.004, Politecnico di Milano, Milan, Italy, June 1993.
6. S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of the Sixteenth Int. Conf. on Very Large Data Bases*, pages 566–577, Brisbane, Australia, Aug. 1990.
7. F. Fabret, M. Regnier, and E. Simon. An adaptive algorithm for incremental evaluation of production rules in databases. In *Proc. of the Nineteenth Int. Conf. on Very Large Data Bases*, pages 455–466, Dublin, Ireland, Aug. 1993.
8. S. Ghandeharizadeh, R. Hull, D. Jacobs, et al. On implementing a language for specifying active database execution models. In *Proc. of the Nineteenth Int. Conf. on Very Large Data Bases*, Dublin, Ireland, Aug. 1993.

9. E. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, pages 49–58, San Diego, California, June 1992.
10. S. Johnson. YACC — yet another compiler compiler. Technical Report CSTR 32, Bell Laboratories, Murray Hill, New Jersey, 1975.
11. M. Lesk. LEX — a lexical analyzer generator. Technical Report CSTR 39, Bell Laboratories, Murray Hill, New Jersey, 1975.
12. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. on Knowledge and Data Eng.*, 3(3):337–341, Sept. 1991.
13. A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation monitoring for active databases. In *Proc. of the Fifteenth Int. Conf. on Very Large Data Bases*, pages 455–464, Amsterdam, The Netherlands, Aug. 1989.
14. E. Simon, F. Fabret, F. Llirbat, and D. Tombroff. Triggers and transactions: Performance issues. In *Proc. of the PUC Rio DB Workshop on New Database Research Challenges*, pages 53–64, Rio de Janeiro, Brazil, Sept. 1994.
15. E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational DBMS. In *Proc. of the Eighteenth Int. Conf. on Very Large Data Bases*, pages 315–326, Vancouver, British Columbia, Aug. 1992.
16. M. Sköld and T. Risch. Compiling active object-oriented relational rule conditions into partially differentiated relations. Technical Report LiTH-IDA-R-94-10, Linköping University, Linköping, Sweden, Mar. 1994.
17. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 281–290, Atlantic City, New Jersey, May 1990.
18. Y.-W. Wang and E. Hanson. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proc. of the Eighth Int. Conf. on Data Engineering*, Phoenix, Arizona, Feb. 1992.
19. J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, San Francisco, California, 1995.
20. J. Widom, R. Cochrane, and B. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proc. of the Seventeenth Int. Conf. on Very Large Data Bases*, pages 275–285, Barcelona, Spain, Sept. 1991.
21. J. Widom and S. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 259–270, Atlantic City, New Jersey, May 1990.

This article was processed using the LaTeX macro package with LLNCS style