Integrating Active and Deductive Rules

John V. Harrison
Department of Computer Science, University of Queensland
Brisbane, QLD Australia

Suzanne W. Dietrich
Department of Computer Science and Engineering, Arizona State University
Tempe, AZ U.S.A.

Abstract

This paper describes how active and deductive rules can be integrated to form an expressive representation for declaring and reasoning about events and conditions. Specifically, this paper describes an extension and optimizations to the PF algorithm, which detects events, i.e. updates, that affect derived relations (or views), even when the derived relation is recursively defined. This capability improves the expressiveness of the *event-condition-action* (ECA) rules, which can then be used to detect more complex events and express conditions that reason with the updates to both stored and derived relations. The updates to the derived relations can be detected *without* having to materialize the derived relations. The *PF* algorithm can detect these updates when certain modifications to the definition of the derived relation are made. This approach has been implemented to form an event detector and condition evaluator for an *active deductive* database. These enhancements increase the sophistication of an active database since ECA rules can be defined that react to a larger scope of real-world situations.

1 Introduction

An *active* database management system (DBMS) automatically monitors user-defined situations and reacts when these situations are detected. Neither the monitoring of the pre-defined situations, nor the initiation of the various actions resulting from the detection of a situation, require user intervention. The services provided by an active database are often termed *active services*. Some support for active services is present in several commercial database systems, e.g., Oracle, Sybase, Ingres.

Active services include alerters, triggers and integrity constraints. An *alerter* notifies a user or process that a situation has been detected. If the situation associated with a *trigger* is detected then another database transaction may be created and executed. An *integrity constraint* aborts a transaction when a situation is detected that implies an integrity violation has occurred.

All three mechanisms can be represented uniformly using *active rules*, also referred to as *event-condition-action* (ECA) rules. An ECA rule is a declarative representation of the expression: *when event E occurs, check condition C, and if true, then execute action A*. When the event and condition component are unified into one expression, the expression is referred to as a *situation*.

A deductive database, which is based on the *Datalog* data model [23], extends a relational database by including inherent support for deductive rules with the

power of recursion. Deductive databases provide a logic-based language, called *Datalog*, that can be utilized to declaratively express deductive rules along with all other system components such as data, queries, views and the data manipulation language. Datalog, when enhanced with support for stratified negation, is a strictly more expressive query language than relational algebra. This is due to the fact that Datalog can be used to directly express recursive queries and view definitions.

Many researchers in active databases consider updates to relations a fundamantal event type. However, some experimental active database systems restrict event detection to updates to stored relations [22], or to derived relations defined using only a subset of the relational operators [9]. Other approaches support event detection to derived relations defined using all of the relational operators but not recursion [3, 6, 11, 19, 21]. Active rules can be more expressive, hence capture more real-world situations, if events, i.e., updates, affecting derived relations defined using all of the relational operators, recursion and aggregation can be detected.

Consider the following example. Let the derived (IDB) relation *LINK*, which is defined by the Datalog predicate *link*, contain tuples that represent the reachability between source and destination stations on a network. The stored (EDB) data referenced in the definition of *LINK* is represented by a single relation, namely *DIRECT_LINK*. Let *link* be comprised of two deductive rules that represent a simple left-recursive transitive closure, which is given below:[1]

$$link(X,Y) \leftarrow direct\_link(X,Y).$$
$$link(X,Y) \leftarrow link(X,Z),\ direct\_link(Z,Y).$$

If the user is interested in detecting lost links in the network, both direct *and indirect*, the capability of detecting events affecting the derived relation *LINK* is required. The event detection algorithm described here can detect this type of complex event without incurring the cost of materializing the derived relation. Alternatively, the algorithm can be employed when the application requires materialization of the derived relation,

Let *V* be the definition of a derived relation, which can also be considered as a *view* definition. Let *R* be a stored relation that represents the materialization of *V*. If an event, e.g., an update, to *V* is detected, then the action component of the ECA rule can perform the update to *R* to maintain *R*'s validity.

Another reason for supporting ECA rules that detect events on derived relations is to increase the availability of active services where database security is a concern. Consider the case where a derived relation, i.e., view, has been defined for a user X to prohibit X from having access to the stored relations comprising V's definition. If X does not have the capability to detect events to V nor has permission to detect events to the stored relations because of a lack of access, X can not utilize active services.

If updates to derived relations can be detected, they can both be reasoned with during condition evaluation and available during action execution. Several experimental systems, along with the proposed SQL3 standard, allow for conditions to refer to both the *old* and *new* values of a tuple corresponding to an update event. If update events affecting derived relations can be detected, the conditions can then reason with the old and new values of the derived tuples as well. They can also access both states of derived relations to support additional expressiveness.

[1] The proposed standard for SQL3 supports such views defined using recursion, namely views defined by a SQL statement involving *recursive union*.

This paper describes how deductive and active rules were integrated to create an event detector for derived relations and to create a unified representation for both events and conditions. The updates to derived relations can be detected *without* materializing the (possibly recursive) derived relation regardless of whether the events result from updates to stored relations or modifications to the definition of the derived relation. The approach has been implemented to form an event detector and condition evaluator for an *active deductive* database.

In section 2, basic terminology is introduced. An overview of event detection in the presence of derived relations and its relationship to condition evaluation is given. In section 3, an algorithm is described that identifies events to derived relations when events occur to the stored relations. In section 4, the representation for conditions is presented and some examples are provided. In section 5, a series of optimizations that improve the efficiency of detecting the satisfaction of event-condition pairs are described. The paper concludes with a summary and a discussion of future research directions.

2 Basic Concepts

Situation monitoring in an active database differs from query processing since it involves reasoning with two database instances, i.e., the state before and after the updates, whereas query processing occurs in the context of a single database instance. The approach employed for situation monitoring must be more general than traditional integrity constraint checking algorithms. This is because a fundamental assumption of these algorithms, namely that every expression representing an IC is false before the algorithm is executed, cannot be made.

The approach for event detection and condition monitoring described in this paper relies on a procedure for computing the difference between two consecutive database states. This difference represents the changes that must be made to the initial database to obtain the updated database. The approach also relies on a procedure that reasons with the computed changes to detect condition satisfaction.

Let DB be a database formed by the union of a set of extensionally defined, i.e., stored relations (EDB) and a set of intensionally defined, i.e., derived relations (IDB). Let relation $R \in$ DB. Let $\mathcal{U}$ represent an update to the EDB.

The database state before $\mathcal{U}$ is performed is referred to as *old*. The database state after $\mathcal{U}$ is performed is referred to as *new*. Let the difference between R in the old state (R_{Old}) and R in the new state (R_{New}) be termed the "delta set" (abbreviated Δset) for R and be represented using the notation ΔR. A Δset consists of two distinct (possibly empty) subsets. The first, labeled ΔR_{add}, consists of tuples that must be added to the *old* relation to obtain the *new* relation. The second, labeled ΔR_{rem}, consists of tuples that must be removed from the *old* relation to obtain the *new* relation.

A Δset is defined for each updated relation, including both EDB and IDB relations. The Δset ΔE for an updated EDB relation E consists of ΔE_{add}, which are the additions to E appearing in $\mathcal{U}$, and ΔE_{rem}, which are the removals from E appearing in $\mathcal{U}$. The Δset ΔI for an updated IDB relation I consists of the changes in the IDB relation, which can be computed by the difference between the materialization of the IDB relation in the old state and the materialization of the same relation in the new state.

These concepts are formalized using the definitions below, which assume that the predicates that define the IDB relations are not updated. Let *materialize(IDB_Rels, DB_State)* denote the materialization of the set of IDB relations specified by *IDB_Rels* using the EDB indicated by *DB_State*.

Definition. Let EDB_{Old} refer to an arbitrary EDB before $\mathcal{U}$ is performed. Let EDB_{New} refer to the same EDB after $\mathcal{U}$ is performed. Let p denote a predicate representing an arbitrary IDB relation P.

$$\begin{aligned}
DB_{Old} &= EDB_{Old} \cup IDB(EDB_{Old}) \\
DB_{New} &= EDB_{New} \cup IDB(EDB_{New}) \\
\Delta P_{rem} &= materialize(p, Old) - materialize(p, New) \\
\Delta P_{add} &= materialize(p, New) - materialize(p, Old) \\
\Delta P &= \{\Delta P_{rem}, \Delta P_{add}\}
\end{aligned}$$

□

The event detection/condition evaluation process described in this paper is illustrated in Figure 1, which indicates that two tasks are performed when updates to the base relations are made. The first task is to identify the updates, i.e., Δsets, to the set of IDB relations that are referenced in the user-defined conditions. This step corresponds to detecting events that affect derived relations. The second task is to evaluate the user-defined conditions, which can contain references to the Δsets and each state of the database.

The first task could be accomplished using the straightforward definition above as an algorithm. However, this would result in a very inefficient implementation. A more efficient technique would compute the changes using an *incremental* approach. An update propagation algorithm can be used for this purpose and is the subject of the next section.

The second task involves reasoning with the computed changes to detect satisfied *event-condition* (E-C) pairs, which are implemented using additional Datalog rules. The body of these rules may include references to the Δsets computed during update propagation and also the database predicates, which the user designates to be evaluated over either the old or the new database state.

3 Detecting Update Events on Derived Data

This section describes an approach for detecting updates to derived data using an algorithm known as *Propagation/Filtration (PF)*. This algorithm performs update propagation to identify the events to derived relations that are defined using recursive Datalog with stratified negation[2]. A description of the basic algorithm, which supports stratified negation, appears in [13].[3] Here we describe an extension that detects updates to derived relations when there are modifications to the definition of the derived relations, A short review of the basic algorithm is given below to provide a basis for this material and material appearing in later sections.

The dependency graph [23] $\mathcal{DG}$ for a Datalog program $\mathcal{D}$ can be used to determine the IDB relations defined by $\mathcal{D}$ that may have been updated as a result of the updates to specific EDB relations. An IDB relation I may have been updated as a result of updates to the EDB relations if the predicate i defining I depends, directly

[2]The approach has been extended to support *group stratified reducible aggregates*[17] but is not described here due to space limitations. The extension is described in [14]

[3]A complete description, correctness proofs and performance measurements can be found in [10].

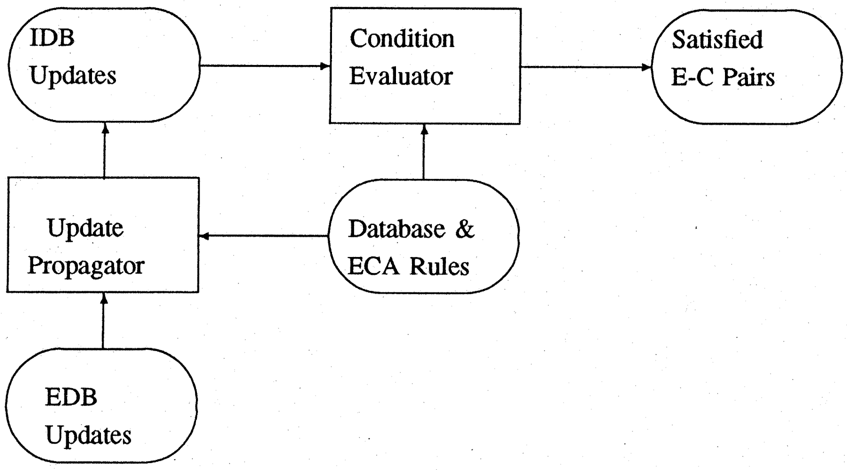

Figure 1: Event Detection and Condition Evaluation

or indirectly, on one or more of the set of updated EDB relations $\mathcal{E}_u$.

Definition. Let e_u represent the predicate defining an arbitrary EDB relation E_u where $E_u \in \mathcal{E}_u$. Let $p \Rightarrow q$ represent a path in $\mathcal{DG}$ from p to q. An IDB relation I, defined by the predicate i, is a *candidate* for update if:

$$e_u \Rightarrow i \in \mathcal{DG}.$$

An IDB relation is *unaffected* if it is not a candidate. □

The objective of the update propagation algorithm is to compute the updates for all candidate relations. Since candidate relations may be defined by several rules, only those rules that depend on candidate relations or updated EDB relations need to be considered in the update propagation process.

Definition. Let *c-rule* represent a rule defining an arbitrary candidate predicate. If *c-rule* contains one or more literals in its rule body corresponding to either a candidate predicate or an updated EDB relation, then *c-rule* is a *candidate rule.*

The *PF* algorithm computes the updates for all candidate relations using the candidate rules. The candidate rules are processed by performing a *propagation* phase followed by a *filtration* phase. A high-level description of each phase is given below.

3.1 The Propagation Phase

During the propagation phase, candidate rules are evaluated when the relations that correspond to subgoals appearing in the body of the rule are updated. The evaluation is constrained using bindings taken from these updates. The result of the evaluation is a set of tuples representing possible updates to the candidate relation.

Definition. The set of tuples generated for an IDB relation as a result of a propagation phase is termed an *approximation.* Each tuple in the approximation is termed a *potential IDB update.*

To obtain an approximation from a candidate rule, a query consisting of the literals appearing in the rule body is invoked over either DB$_{Old}$ or DB$_{New}$. Consider a rule r defining an IDB relation P where both additions and removals have been identified for a relation L corresponding to a literal l appearing in the body of r. To propagate the additions to L to the IDB relation P, the rule body is evaluated using DB$_{New}$. The evaluation is constrained using bindings from ΔL_{add}. The result of this evaluation is a relation whose schema contains all of the variables that occur in the rule body. The relation is projected onto the set of attributes corresponding to the set of variables that appear in the head literal. A similar procedure is performed to process the removals, however, the rule body is evaluated using DB$_{Old}$ and the evaluation is constrained using bindings from ΔL_{rem}. The result of the projection is the approximation for P.

Note that for both additions and removals the subgoal representing the literal l can be removed from the query for efficiency, since the updates to L used to constrain the query bind all variables appearing in subgoal l. This forms a query that tests a tuple that has already been determined to be an actual IDB update and, therefore, represents redundant computation.

If the rule body contains several literals, each representing either a candidate relation or an updated base relation, then a separate query is issued for each literal. In addition, a separate query is issued for the additions to, and the removals from, each relation. In the worst case situation, where a rule defining a predicate p has k subgoals each corresponding to a relation where both addition and removal updates have been identified, $2k$ queries would be issued during the propagation phase to obtain all potential updates for p. In our implementation, a multiple query optimizer identifies common subexpressions in the queries, which all involve the same set of predicates, thereby significantly reducing the actual computation performed.

3.2 The Filtration Phase

The propagation phase propagates the changes to the extensional relations up through the rules and identifies potential changes to the derived relations. Potential changes are filtered to identify actual changes. For example, a potential addition represents a derivation of a tuple t. If t is provable in the database state before the updates, then the potential addition is filtered and is not reflected as an actual change to the database. Similarly, a potential removal represents the deletion of a derivation for a tuple t and if t is still provable in the database state after the updates, then the filter phase does not identify the potential removal as an actual removal.

Thus, the filtration phase of *PF* refines the approximation of potential updates to IDB relations identified during the propagation phase. Potential IDB updates that cannot be proven are removed from the approximation. Each potential addition is posed as a query to the database using DB$_{Old}$. Each potential removal is posed as a query to the database using DB$_{New}$. Tuples returned as a result of the query do not represent a change in the database state so they are deleted from the approximation.

Definition. A potential IDB removal is termed *disqualified* if it is provable in DB_{New}. A potential IDB addition is disqualified if it is provable in DB_{Old}. A potential IDB update that is not disqualified is termed an *actual IDB update.*

Actual IDB updates are saved in global Δsets and are available for use in subsequent invocations of the algorithm, including recursive ones. Note that duplicate elimination is easily performed when updates are tabled in a Δset. This also serves to reduce the size of subsequent approximations.

3.3 Event Detection and Deductive Rule Updates

In this section, we address event detection when the definition of the derived relation is modified. When the derived relation is defined using deductive rules, these modifications are in the form of *rule updates*. Rule updates are defined here as either the addition or removal of a rule from the database. This changes the definition of a derived relation. The derived relation can be viewed as being comprised of a n-way union where each rule defines one operand of the union. An update to the definition of a candidate IDB relation can result in the generation of actual IDB updates to the IDB relation that the rule defines.

Consider the case where a rule update is performed on a non-recursive IDB relation I. When a rule r defining I is added to the database, an unconstrained query consisting of the conjunction of literals representing the rule body is issued over DB_{New}. The result is an approximation that is filtered using DB_{Old}. Conversely, the body of a deleted rule is issued over DB_{Old} to obtain the approximation and DB_{New} is used for filtration.

The query issued over either state includes all literals from the rule body and there are no bindings to constrain computation. Actual IDB updates computed for P resulting from a rule update are propagated using the *PF* algorithm in the same manner as those resulting from base relation updates. The rule update algorithm can be applied to recursive IDB relations. There is a constraint, however, as to the order in which rules are removed. When removing rules defining a recursive predicate, the base rule must be removed last to avoid an ill-defined recursive specification. When adding rules defining a recursive predicate, the order of addition is inconsequential. The complete algorithm for rule updates can be found in [12].

4 Event-Condition Pairs as Deductive Rules

The previous section described how the *PF* algorithm is used to detect updates to derived relations. The Δsets correspond to the set of identified events. When satisfied, the E-C pairs trigger the action component of the ECA rule. A method for evaluating these E-C pairs is described. If evaluating a rule produces a non-null result then the E-C pair has been satisfied.

To represent the E-C pairs, a set of Datalog predicates called *ec-preds* are created. Each *ec-pred* is defined using one or more *ec-rules*. The body of an ec-rule may include special literals, termed *Δset literals*, that represent the Δsets produced during update propagation.

In addition to containing Δset literals, ec-rules can also include literals that represent both stored and derived relations. Since both states of each type of relation

is accessible, the user must specify either the old or new state. The choice is indicated by a *state designator* that appears in the literal. The state designator directs the evaluator as to what database state that evaluation of the predicate is to occur. Also note that, like rules comprising IDB predicates, ec-rules may contain evaluable literals, e.g., $\leq, =, \geq$.

Consider the following example, which illustrates the concepts above. The ec-pred *connection_lost* represents an E-C pair that detects when any connection between two stations on a network is lost.

$$connection\_lost(X,Y) \leftarrow \Delta link_{rem}(X,Y).$$

It consists of one ec-rule that refers to the Δset computed for the IDB relation *LINK*. The E-C pair is satisfied once for each removal computed by the *PF* algorithm for *LINK*. In this example, the condition is null.

A second example involves a derived relation defining a high salaried employee as one whose annual salary and bonus equals or exceeds fifty thousand dollars. The definition appears below:

$$\begin{aligned} & high\_salaried\_employee(E\_Name, Tot\_Salary) \leftarrow \\ & \quad base\_salary(E\_Name, E\_Salary), yearly\_bonus(E\_Name, E\_Bonus), \\ & \quad Tot\_Salary = E\_Salary + E\_Bonus, Tot\_Salary \geq 50,000. \end{aligned}$$

Let the attribute *E_Name* uniquely identify an employee. Assume the manager wants to detect the situation where a high-salaried employee receives an increase in total salary that is greater than or equal to ten percent, regardless of the classification of the income. The following ec-pred *large_raise_given(E_Name)* defines the E-C pair. The Δset literals are used to detect the updates to the derived relation and the evaluable literal ($\geq$) represents the condition.

$$\begin{aligned} & large\_raise\_given(E\_Name) \leftarrow \\ & \quad \Delta high\_salaried\_employee_{rem}(E\_Name, Old\_sal), \\ & \quad \Delta high\_salaried\_employee_{add}(E\_Name, New\_sal), \\ & \quad New\_sal \geq Old\_sal * 1.1. \end{aligned}$$

Consider an inventory management example. The ec-pred *inventory_shortage*, shown below, identifies parts that need to be reordered when the required amount to be maintained drops below the currently available amount for that part. Additions to the *part* relation, which may have been caused by either a decrease in the available amount or from the addition of a new part, are compared to the required amount, which may have also been updated requiring the reference to the new state.

$$\begin{aligned} & inventory\_shortage(Part) \leftarrow \\ & \quad \Delta part_{add}(Part, Avail\_Amt), stock\_required_{New}(Part, Req\_Amt), \\ & \quad Avail\_Amt \leq Req\_Amt. \end{aligned}$$

The *PF* algorithm need only compute updates for candidate relations that are directly or indirectly referenced in Δset literals. This provides an initial opportunity to customize the *PF* algorithm for its use in derived relation event detection since the *PF* algorithm need only compute updates for these relevant candidate relations. The concept is formalized below.

Definition. A candidate relation is termed *relevant* if it is represented by a Δset literal appearing in an ec-pred. If R_{rel} is a relevant candidate relation and:

$$E \Rightarrow R_{rel} \in \mathcal{DG}$$

then E is a relevant candidate relation. A candidate rule that defines a relevant candidate relation is termed a *relevant* candidate rule.

Evaluating the Conditions After Event Detection

Tuples that are produced as a result of evaluating a ec-pred indicate the satisfaction of an E-C pair. All ec-preds representing situations of interest are evaluated after completion of the update propagation algorithm. The evaluation can be performed using a conventional recursive query evaluation strategy [1]. A minor modification would be made to allow the strategy to ignore removed facts when evaluating in the new state and to ignore new facts when evaluating in the old state.

5 Optimizing Complex Event Detection

In this section, optimizations are described that improve the efficiency of the event detector. The optimizations are motivated from deductive database query evaluation strategies but require adaptation for use in event detection.

5.1 Dependency Graph Analysis

The efficiency of the *PF* algorithm can be improved by reducing irrelevant computation. The computation of actual IDB updates that cannot contribute to the formation of ec-pred tuples can be identified and eliminated using the dependency graph.

The *PF* algorithm need only compute updates for candidate relations that are directly or indirectly referenced in Δset literals. All other updates represent irrelevant computation. The concept of a *relevant* candidate relation is formalized below.

Definition. A candidate relation is termed *relevant* if it is represented by a Δset literal appearing in an ec-pred. If R_{rel} is a relevant candidate relation and:

$$E \Rightarrow R_{rel} \in \mathcal{DG}$$

then E is a relevant candidate relation. A candidate rule that defines a relevant candidate relation is termed a *relevant* candidate rule.

In certain cases, updates to a relevant candidate relation can be identified as irrelevant with respect to the definition of the E-C pair that directly or indirectly references the relation.

Definition. An actual IDB update or an EDB update is *condition relevant* if it participates, directly or indirectly, in the formation of a ec-pred tuple. Conversely, actual IDB updates or EDB updates that do not participate, directly or indirectly, in the formation of a ec-pred tuple are *condition irrelevant.*

The optimization involves analyzing the program's dependency graph to identify paths that the *PF* algorithm would use to propagate condition irrelevant tuples. The objective of the optimization is to inform the *PF* algorithm to remove the paths from consideration.

Intuitively, the implementation of the optimization involves pushing the addition/removal specifier, which appears in Δset literals, down through the paths in the dependency graph towards the base relations. This will identify if either additions to, or removals from, a base relation are condition irrelevant.

If a Δset literal Δp_{Utype} appears in a condition, it indicates that the corresponding relation P is relevant candidate. The update type specification $Utype$ indicates whether removals from, or additions to, the relevant candidate relation are of interest. The Δset tuples produced for a relation R that represent additions to R are of no consequence to the references in the ec-preds that refer to removals from R. Therefore, if all references to the ΔR that appear in the ec-preds indicate additions then certain base relation updates, namely those that result only in removals to R, need not be propagated. An equivalent relationship holds for Δset tuples produced for a relation R that represent additions.

After the completion of path analysis, the update types, i.e., additions or removals, for each relation that cannot contribute to the formation of a condition relevant tuple will have been identified. Any update classified as one of these types is ignored by the condition monitor and will not be propagated.

The optimization can reduce irrelevant computation in instances where negation is present. Since no significant cost is incurred at run-time, its application is advantageous in virtually all instances.

5.2 Optimizing Rule Evaluation

Both the propagation and filtration phases of the *PF* algorithm invoke queries to the database. Each query issued during the propagation phase consists of a subset of the literals that appear in a rule body. The *sideways informationpassing* strategy (SIP)[2] chosen for query evaluation has a direct effect on the efficiency of the algorithm. Informally, a sip implements the decision as to how bindings will be utilized during each step of query evaluation. In our prototype, which is implemented in Prolog, the different sip's are implemented by reordering the subgoals appearing in the Prolog queries.

The sip optimization dynamically computes an optimal sip for every query issued during both the propagation and filtration phases. The heuristic used to create the sips is the traditional "bound-is-easier" assumption [24]. This assumption implies that the more arguments that are bound in a call to a subgoal, the less expensive it will be to compute the result of the call.

Consider the non-linear recursion example shown in Figure 2. The predicate sg has a non-linear recursive definition but, for ease of comprehension, defines the traditional *same generation* relationship. Assume that updates were made to relation PAR and this has resulted in updates $\mathcal{U}$ to relation SG. Since the PF algorithm derives additional IDB updates from those already computed, the set $\mathcal{U}$ will be used for a subsequent propagation phase, which will now be examined in detail.

The second rule defining predicate sg contains subgoals representing SG. It must therefore be used for propagation. The body of the second rule is issued as a query three times, once for each occurrence of the sg literal in the rule. Each query will consist of the body of the rule less one of the three sg literals.

```
sg(X,Y)  :- par(X,P),par(Y,P).
sg(X,Y)  :- sg(X,X1),par(X1,XP),sg(XP,YP),par(Y1,YP),sg(Y1,Y).
```

Figure 2: A Non-linear Recursion

The first query, issued without the subgoal $sg(X,X1)$, is given below.

$$par^{bf}(X1, XP),\ sg^{bf}(XP, YP),\ par^{fb}(Y1, YP),\ sg^{bf}(Y1, Y)$$

Variables X and $X1$ will be bound using constants extracted from $\mathcal{U}$. The adornment of the call to each subgoal comprising the query is indicated by a superscript.

Note that both subgoals in the above query that refer to predicate sg are called with adornment *bf*. The rules below show the adornments associated with the subgoals when the query to subgoal sg^{bf} is made.

$sg^{bf}(X,Y)$:- $par^{bf}(X, P), par^{fb}(Y, P)$.
$sg^{bf}(X,Y)$:- $sg^{bf}(X, X1), par^{bf}(X1, XP), sg^{bf}(XP, YP),$
$par^{fb}(Y1, YP), sg^{bb}(Y1, Y)$.

Note that all of the calls to predicate sg that appear in the rule bodies will be at least partially constrained, i.e., a b appears in the adornment.

Now consider the second query where variables XP and YP are bound using $\mathcal{U}$. This query consists of the rule body without the literal $sg(XP,YP)$ and is given below. Again, note the adornment of the call to each subgoal representing predicate sg.

$$sg^{ff}(X, X1),\ par^{bb}(X1, XP),\ par^{fb}(Y1, YP),\ sg^{bf}(Y1, Y)$$

The first subgoal in the query is called with adornment *ff*. This introduces a gross inefficiency since this unconstrained call to predicate sg would result in the complete materialization of the predicate. This would dramatically raise the expense of computing the updates.

Subgoal reordering based on the "bound-is-easier" assumption can correct this inefficiency. The query below has the subgoals reordered. The adornments are updated to reflect the reordering.

$$par^{fb}(X1,XP),\ par^{fb}(Y1,YP),\ sg^{bf}(Y1,Y),\ sg^{fb}(X,X1)$$

Unfortunately, this first pass at optimizing the rule via subgoal reordering is not enough to avoid the inefficiency. Even though the adornment of the call to $sg(X,X1)$ has been restricted from *ff* to *fb*, the call to $sg^{fb}(X,X1)$ reintroduces the inefficiency. The rules below show the adornments associated with the subgoals when the query to subgoal sg^{fb} is made.

$sg^{fb}(X,Y)$:- $par^{ff}(X,P), par^{bb}(Y,P)$.
$sg^{fb}(X,Y)$:- $sg^{ff}(X,X1), par^{bf}(X1,XP), sg^{bf}(XP,YP),$
$par^{fb}(Y1,YP), sg^{bb}(Y1,Y)$.

Note that the *ff* adornment again appears; this time in the first subgoal of the second rule. However, the inefficiency can again be corrected by a second pass at reordering the subgoals. The rules for $sg^{fb}(X,Y)$, which have reordered subgoals so that no *ff* adornment appears, are given below:

$sg^{fb}(X,Y)$:- $par^{bf}(Y, P), par^{fb}(X, P)$.
$sg^{fb}(X,Y)$:- $sg^{fb}(Y1, Y), par^{bf}(Y1, YP), sg^{fb}(XP, YP),$
$par^{fb}(X1, XP), sg^{fb}(X, X1)$.

The solution to avoiding the inefficiency is to create optimized versions of certain predicates in the IDB. Each version has the subgoals of its rules ordered to correspond to the sip that allows for the most efficient evaluation with respect to a single adornment. To be consistent with the semantics of negation, negated subgoals are ordered such that their evaluation will be delayed until their arguments are fully bound. Evaluable predicates are also delayed until fully bound.

The subgoal reordering could be performed statically using an IDB preprocessor that would rewrite the rules of each predicate for as many possible adornments for which the predicate could be called. This is what is performed in the Magic Sets[1] approach to achieve the unique binding property. Unfortunately, significant rewriting is called for since a predicate with n arguments has 2^n possible adornments. This approach may be suboptimal since it is unlikely that all of these "sip-optimized" versions will be required during computation. The costs of computing and storing all of them may not be required.

One way to avoid the costs of statically creating the sip-optimized versions of every predicate in the database is to dynamically create only the versions required by the *PF* at execution time. This introduces overhead during execution of the condition monitoring instead of the preprocessing phase. In an implementation of the condition monitor designed for practical use, a more sophisticated rule-rewriter should be employed by the IDB preprocessor that analyzes both the IDB rules and the Δset literals and will only generate sip-optimized versions of predicates that are likely to be utilized.

The non-linear same-generation example was utilized to illustrate the potential benefit of the subgoal reordering optimization. As can be observed from the example, the reduction in cost of query evaluation as a result of subgoal reordering can be significant. Without the optimization, an unconstrained query is posed to the database. This unconstrained call increases the cost of computing the updates for the particular relation to that of the naive approach.

Since all sip-optimized predicates can be created before runtime, very little cost is incurred during update propagation as a result of this optimization. In this example, the predicate sg has only two arguments resulting in the creation of four sip-optimized versions. Unless the relations in the database are extremely small, the improved performance of the *PF* algorithm will generally outweigh the cost of storing the sip-optimized versions of the predicate. In addition, the sip-optimized rules can be used during query evaluation. The cost of creating and storing the sip-optimized rules is amortized over both event-condition monitoring and query evaluation.

5.3 Multiple Query Optimization

A database system that utilizes multiple query optimization (MQO) exploits similarities that may exist in a set of queries. When the system receives the query set it analyzes the queries to detect if any common subqueries exist. Evaluating common subqueries more than once represents redundant work. The heuristic that justifies MQO is that the queries will likely share subqueries and that overall query response time will decrease. Unfortunately, if no common subqueries exist, query response time may increase because time is spent analyzing the queries.

The *PF* algorithm may issue many queries to compute the Δsets. The queries are issued during both the propagation and filtration phases. The worst case situation for update propagation involves a rule that has n subgoals where each subgoal corresponds to a relation that has been updated as a result of both additions and removals. In this situation, $2n$ queries would need to be issued, where n queries would be issued in the *new* database state and n queries would be issued in the *old* database state. Each of these queries will involve essentially the same set of predicates hence the queries will be very similar. A call made to the same predicate by different queries represents a common subquery if the arguments of each query will unify with the arguments of the others. This behavior is consistent with the heuristic justifying MQO and motivates its use.

Depending on the database, it is possible that the call to a predicate resulting from one query subsumes one or more calls invoked by other queries. In this situation, result sharing is possible. The reduction in redundant computation can be dramatic if p_i is recursive.

The invocation of multiple related queries motivates the development of a multiple query optimization (MQO) algorithm to increase the efficiency of the *PF* algorithm. As indicated in [7], memoing inherently implements the MQO task of *common subexpression identification* [5, 20]. Each occurrence of a literal defining an IDB predicate represents the subexpressions defined by the conjunction of literals in the bodies of the rules defining the predicate.

Our prototype employs a top-down recursive query evaluation strategy known as *EQ**¬[12]. This strategy, which is an optimized version of *ET** [7], utilizes memoing to insure completeness. This memoing feature facilitates the implementation of MQO. The *EQ**¬ algorithm detects *completed* calls. A completed call has an extension, which has been computed by the *EQ**¬ algorithm, that is complete. Completed calls that are detected by the algorithm are not recomputed. Instead, the answers for the call, which are retained in the extension table, are returned to the caller. If a call is made to a predicate that is not subsumed by an earlier call, the call is tagged complete after evaluation and the results are made available to subsequent callers.

The MQO optimization benefits both the propagation and filtration phases of the *PF* algorithm. Answers obtained from DB_{New} (DB_{Old}) when computing potential additions (removals) during a propagation phase are available to reduce the effort required to test potential removals (additions) in a subsequent filtration phase. Tuples stored in a memo table for unaffected predicates, obtained during a propagation or filtration phase, can be utilized regardless of the state in which the query is posed.

Four memo tables are maintained in the prototype. Two are used to record the *calls* made to both the initial and updated database states. Two more are used to record the *answers* computed in each database state resulting from the calls.

5.4 Partial Evaluation of Deductive Rules

Lakhotia and Sterling describe partial evaluation, as it applies to logic programming, as follows. Given a program $\mathcal{P}$ and a goal $\mathcal{G}$, the result of partially evaluating $\mathcal{P}$ with respect to the goal $\mathcal{G}$ is the program $\mathcal{P}'$ such that for any substitution θ, evaluating $\mathcal{G}\theta$ results in the same answers with respect to both $\mathcal{P}$ and $\mathcal{P}'$. The objective of partial evaluation is to produce a $\mathcal{P}'$ on which $\mathcal{G}\theta$ can be evaluated more efficiently than on $\mathcal{P}$ [16]. A theoretical foundation for partial evaluation is given in [18].

CP: *possible_vessel_failure(V)* $\leftarrow$ *Δabnormal_vessel_condition$_{add}$(V).*

IDB: *abnormal_vessel_condition(V)* $\leftarrow$
vessels_in_use(V), vessel_spec_violation(V).

vessel_spec_violation(V) $\leftarrow$ *temp(V,Current_Temp), max_temps(V,Max_Temp), Current_Temp* $>$ *Max_Temp.*
vessel_spec_violation(V) $\leftarrow$ *pressure(V,Current_Pres), max_pressure(V,Max_Pres), Current_Pres* $>$ *Max_Pres.*

Figure 3: Conditions and IDB before Partial Evaluation Optimization

Partial evaluation can be used to form optimized deductive rules that increase the efficiency of event detection in the case of derived relations. The optimization applies when constants appear as arguments in the Δliterals, which appear as ec-pred subgoals. A Δliteral for some predicate p serves as the goal $\mathcal{G}$. The Datalog definition of predicate p serves as the program $\mathcal{P}$. Condition relevant updates are computed more efficiently using the partially evaluated predicate p' than they are using p.

Informally, the partially evaluated predicate p' is created by pushing the constants appearing in the Δliteral down through the rules that directly or indirectly define the predicate p. The constants bind variables in the rules and are used to form new, restricted versions of the predicates. These new predicates are essentially copies of the initial database predicates. However, they are bound using the constants from the ec-preds and may have less arguments and even less subgoals.

The Δset literal $\Delta p'$, where p' represents the partially evaluated version of p, is substituted for Δp in the ec-pred. The IDB relation P is no longer *relevant candidate* with respect to the Δliteral being processed. If P is not referenced directly or indirectly by any other Δliteral then P will no longer be relevant candidate and will not require update propagation. Instead, the IDB relation P' is now *relevant candidate*. Updates will now be propagated using the definition of the partially evaluated predicate representing relation P'. The predicate P' is less expensive to monitor since it will not propagate irrelevant tuples. The result of this optimization is a reduction in computation performed by the *PF* algorithm as well as a reduction in the cost of evaluating the ec-preds during condition evaluation.

Example

The following simple example illustrates the partial evaluation concept. The condition states that a vessel V may have failed if its current condition is abnormal. An abnormal condition is defined as one where either the temperature is higher or internal pressure is lower than is considered normal. Figure 3 shows the condition and the IDB before the optimization.

Now, assume that the user is solely concerned with vessel *v5* since it alone contains flammable chemicals. The constant *v5* is substituted for the variable V in the ec-pred. The updated ec-pred appears below.

possible_vessel_failure(v5) $\leftarrow$ *Δabnormal_vessel_condition$_{add}$(v5).*

Since a constant, namely *v5*, appears as an argument in the Δliteral:

$$\Delta abnormal\_vessel\_condition,$$

the partial evaluation optimization is applicable. Initially, a partially evaluated version of the predicate *abnormal_vessel_condition* is created. The predicate has one rule:

$$abnormal\_vessel\_condition\_v5 \leftarrow vessels\_in\_use(v5),\ vessel\_spec\_violation(v5).$$

Note that the arity of the new predicate is less than the original since there is no need to pass the constants that were used to form the partially evaluated version. Predicate *vessels_in_use* represents an EDB relation so no further partial evaluation is necessary. However, predicate *vessel_spec_violation* represents an IDB relation so partial evaluation can continue. The partially evaluated predicate *vessel_spec_violation_v5* is defined by the following two rules:

$$\begin{aligned}&vessel\_spec\_violation\_v5 \leftarrow temp(v5, Current\_Temp),\\ &\qquad max\_temps(v5, Max\_Temp),\ Current\_Temp > Max\_Temp.\\ &vessel\_spec\_violation\_v5 \leftarrow pressure(v5, Current\_Pres),\\ &\qquad max\_pressure(v5, Max\_Pres),\ Current\_Pres > Max\_Pres.\end{aligned}$$

Using this revised definition, only updates that affect vessel *v5* are propagated. For example, updates to EDB relation *temp* that will not unify with *temp(v5,Current_Temp)* are identified as condition irrelevant and will not be propagated. The initial definitions of the IDB predicates remain in the database after the optimization but since they no longer define relevant candidate relations, they will not be utilized for update propagation.

When the partial evaluation optimization can be applied, a performance increase, in terms of speed is achieved. The optimization requires that additional rules be added to the database. A tradeoff exists between the costs of maintaining the additional rules and the increased speed of update propagation.

It is expected that in a practical application, speed would be the primary concern. The optimization can reduce irrelevant computation in instances where constants appear in the Δliterals. Since the costs of partial evaluation and rule rewriting are incurred at compile-time and the optimization can only improve the performance of the algorithm at run-time, the application of the optimization is justified.

5.5 Minimizing the Approximation

Potential IDB updates that have already been filtered and determined to be actual IDB updates need not be filtered again. In addition, potential IDB updates that have been disqualified during the filtering phase need not be filtered again. Memoing can be used to reduce the size of the approximation by identifying tuples that are known to be actual IDB updates or disqualified potential IDB updates. This reduces the computation that must be performed during any subsequent filtration phase and results in an efficiency improvement.

In order to ensure termination and to identify the events to the derived relations, the actual IDB updates computed by the *PF* algorithm are retained in the Δsets. A potential IDB update computed for an IDB relation P during a propagation phase that already occurs in ΔP as a result of prior computation need not be tested for disqualification. As evidenced by its occurrence in ΔP, it would pass the filtration phase, incurring the cost of filtering, but simply be discarded later as a duplicate. The Δset serves as a memo table to reduce the tuples tested during the filtration phase and therefore reduces redundant computation.

Potential IDB updates that have been disqualified as a result of previous invocations of the *PF* algorithm can also be used to prefilter the approximation. These disqualified updates can be used to identify disqualified tuples that appear in future approximations. Identified tuples are removed from the approximation to avoid unnecessary computation during the filtration phase.

6 Discussion

The sophistication of an active database system can be increased by allowing complex events to be detected and expressive conditions to be defined. This paper described how updates to derived relations could be detected using the *PF* algorithm. The updates can be detected without having to materialize the derived relations. In addition, the updates are detected even when the definition of the derived relation is modified via the addition or deletion of deductive rules. We describe how conditions can be defined that offer a unified representation for reasoning with derived relations, stored relations and evaluable predicates. These conditions can also reference both the old and new values of tuples representing updates to either stored or derived relations. The result is an expressive representation for events and conditions that can specify more complex situations.

Integrity constraint enforcement [26], computing the differences between database states [15], managing derived relations [4, 8] and performing change computation [25] can be directly supported using the *PF* algorithm. Performance improvements can be obtained by employing the optimizations described here.

For future work, a suitable execution model will be coupled with the event-condition monitor to form an *active deductive* DBMS. In addition, since the *PF* algorithm only queries the database, we are investigating whether parallelism can be used to improve the performance of the event detector thereby providing better support for real-time applications.

References

[1] Bancilhon, F., Maier, D., Sagiv, Y. and Ullman, J., "Magic Sets and Other Strange Ways to Implement Logic Programs", *Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, Washington, DC, 1986, pp. 1–15.

[2] Beeri, C. and Ramakrishnan, R., "On the Power of Magic", *Journal of Logic Programming*, October , 1991:10, pp. 255–299.

[3] Ceri, S. and Widom, J., "Deriving Production Rules for Incremental View Maintenance", *Proc. 17th Intl. Conf. on Very Large Database Systems*, Barcelona, September, 1991, pp. 577–589.

[4] Ceri, S. and Widom, J., "Deriving Incremental Production Rules for Deductive Data", IBM Research Report RJ 9071 (80884) November, 1992.

[5] Chakravarthy, U. S. and Minker, J., " Multiple Query Processing in Deductive Databases using Query Graphs", *Proc. of the 12th Intl. Conf. on Very Large Data Bases*, Kyoto, August 1986.

[6] Chakravarthy, S. and Garg, S., "Extended Relational Algebra (ERA): for Optimizing Situations in Active Databases", Technical Report UF-CIS TR-91-24, CIS Department, University of Florida, Gainesville, November 1991.

[7] Dietrich, S. W., "Extension Tables: Memo Relations in Logic Programming", *IEEE Symposium on Logic Programming*, San Francisco, CA, 1987, pp. 264–272.

[8] Gupta, A., Mumick, I. S. and Subrahmanian, V. S., "Maintaining Views Incrementally", *Proceedings of the 1993 ACM SIGMOD*, Washington, DC, May 1993.

[9] Hanson, E., Chaabouni, M., Kim, C. and Wang, Y., "Rule Condition Testing and Action Execution in Ariel", *Proceedings of the 1992 ACM SIGMOD International Conference of Management of Data*, San Diego, CA, June 1992, pp. 49–58.

[10] Harrison, J. V., and Dietrich, S. W., "Condition Monitoring using Update Propagation in an Active Deductive Database", Arizona State University Tech. Rep. TR-91-027 (Revised), December, 1991, To appear: *J. Info. Sys.*

[11] Harrison, J. V. and Dietrich, S. W., "Towards an Incremental Condition Evaluation Strategy for Active Deductive Databases", In Proceedings of Databases '92, Third Australian Database Conference, Melbourne, Australia, February 1992. pp. 81–95.

[12] Harrison, J. V., "Condition Monitoring in an Active Deductive Database", Ph.D. Dissertation, Arizona State University, July, 1992.

[13] Harrison, J. V. and Dietrich, S. W., "Maintaining Materialized Views in Deductive Databases: An Update Propagation Approach", Proceedings of the Deductive Database Workshop held in conjunction with the Joint International Conference and Symposium on Logic Programming, Washington, D.C., November, 1992, pp. 56–65.

[14] Harrison, J. V., "Monitoring Complex Events defined using Aggregates in an Active Deductive Database", University of Queensland Tech. Rep. 268, May, 1993 (revised).

[15] Kuchenhoff, V.,"On the efficient computation of the difference between consecutive database states", In *Proc. of the Second Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, Munich, Germany, December 1991.

[16] Lakhotia, A. and Sterling, L., "ProMiX: a Prolog Partial Evaluation System", In *The Practice of Prolog*, Sterling, L. (eds), MIT Press, Cambridge, 1990, pp. 137–179.

[17] Lefebvre, A., "Towards an Efficient Evaluation of Recursive Aggregates in Deductive Databases", *Proc. of FGCS'92*.

[18] Lloyd, J. W. and Shepherdson, J. C., "Partial Evaluation in Logic Programming", *Journal of Logic Programming*, 11:217-242, 1991.

[19] Lohman, G., Lindsay, B., Pirahesh, H. and Schiefer, K. B.,"Extensions to Starburst: Objects, Types, Functions, and Rules", *Communications of the ACM*, Vol. 34, No. 10, October 1991, pp. 94–109.

[20] Park, J. and Segev, A., "Using Common Subexpressions to Optimize Multiple Queries", *Proc. of Seventh IEEE Conf. on Data Engineering*, 1988, pp. 311–319.

[21] Schreier, U., Pirahesh, H., Agrawal, R. and Mohan, C., "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS", *Proceedings of the 17th Intl. Conf. on Very Large Databases (VLDB)*, Barcelona, Spain, 1991, pp. 469–478.

[22] Stonebraker, M. and Kemnitz, G, "The Postgres Next-Generation Database Management System", *Communications of the ACM*, Vol. 34, No. 10, October 1991, pp. 78–92.

[23] Ullman, J., *Principles of Database and Knowledge-base Systems*, Vol. 1, Computer Science Press, Rockville, MD, 1988.

[24] Ullman, J., *Principles of Database and Knowledge-base Systems*, Vol. 2, Computer Science Press, Rockville, MD, 1989.

[25] Urpi', T. and Olive, A., "Events and Event rules in Active Databases", *Special Issue on Active Databases, Bulletin of the Technical Committee on Data Engineering*, December, 1992 Vol. 15, No. 1-4.

[26] Vieille, L., Bayer, P. and Kuchenhoff, V., "Integrity Checking and Materialized Views Handling by Update Propagation in the EKS-V1 System", ECRC Technical Report TR-KB-35, ECRC, Munich, Germany, June 1991.

Norman W. Paton and M. Howard Williams (Eds.)

Rules in Database Systems

Proceedings of the 1st International Workshop on Rules in Database Systems, Edinburgh, Scotland, 30 August–1 September 1993

Published in collaboration with the British Computer Society

BCS

Springer-Verlag
London Berlin Heidelberg New York
Paris Tokyo Hong Kong
Barcelona Budapest