Towards an Incremental Condition Evaluation Strategy for Active Deductive Databases

John Harrison and Suzanne Wagner Dietrich

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406

Abstract

An "active" database management system (DBMS) is characterized by its ability to automatically monitor user-defined conditions and react when a particular condition is satisfied. With traditional (passive) database systems, sub-optimal approaches must be used to provide this capability. Efficient condition evaluation is critical to obtain satisfactory performance in an active database system. Condition evaluation differs from query evaluation in terms of the anticipatory knowledge available for optimization and the origin and propagation of bindings. To minimize the cost of condition evaluation, an incremental approach is preferred. This paper takes the first step towards the development of a condition evaluation strategy for active deductive databases, by applying the HiPAC approach for incremental condition evaluation to deductive database systems. The applicability of the HiPAC approach is demonstrated by presenting an incremental condition evaluation strategy for resatisfiable conditions expressed in nonrecursive Datalog without negation. The strategy is then extended to handle nonresatisfiable conditions.

I. Introduction

An "active" database management system (DBMS) automatically monitors user-defined conditions and reacts when a particular condition is satisfied. Neither the monitoring of the pre-defined conditions, nor the initiation of the various actions resulting from the satisfaction of a condition, require user intervention. A conventional DBMS is passive, since queries and transactions are executed only when explicitly requested by a user or application program.

Active services have been identified as a key requirement of future applications of databases [LAG89, STO90]. Some applications that require timely responses to critical situations include computer integrated manufacturing (CIM), power and data distribution network control, program trading, battle management and chemical and nuclear process control. As an illustrative example of the use of active services, consider a database that contains investment management information. The information maintained might include the recent price history of selected stocks, commodities and precious metals. An investor may wish to be notified if some complex situation arises

that might affect a current investment or indicate a new investment opportunity. The investor may want the database to continually monitor specific economic conditions and then automatically initiate a purchase or sale of a stock or security in a timely manner.

As noted by several researchers [DAY88b, CHA89, CHN90], traditional database systems use two approaches for monitoring conditions (also called *situations*). The first approach is to poll the database to check when a condition becomes true. The second approach is to encode the condition evaluation function as part of application programs that access the database. Polling is sub-optimal because the repetitive queries represent a waste of system resources. In addition, if the time interval between each identical polling query is large then the system may not detect condition satisfaction in a timely manner, resulting in unsatisfactory system performance. Alternatively, if the time interval is too small, system performance will degrade as a result of having to process large quantities of unnecessary queries.

The second approach involves augmenting the application programs that access the database to check for condition satisfaction. This approach passes the burden of condition monitoring to the application programmer and affects software development. Embedding condition monitoring code in application programs inhibits the development of optimizations that would otherwise be available if the conditions were managed by the database.

Active database systems support alerters, triggers and integrity constraints. Conceptually, the primary difference between these three types of active services is the possible actions that result from the associated condition becoming satisfied. If the condition associated with an alerter becomes satisfied then the action is to notify a user or application program. If the condition associated with a trigger becomes satisfied then the action involves the propagation of an update. When the condition associated with an integrity constraint becomes satisfied, the typical action is to abort the transaction. Other options to restoring integrity include modifying the database (via a trigger) or, alternatively, modifying the constraints. One measurement of active DBMS sophistication is the degree of expressiveness the system offers the user in terms of the ability to describe a condition. A trade-off exists between how expressive a condition is and how expensive it is to monitor. If the conditions are expressive then the user can pose a more comprehensive class of conditions and can benefit more from active service support. The disadvantage of supporting more expressive conditions is that the conditions will likely be more expensive to evaluate. Ideally, a designer of an active DBMS would like to achieve efficient condition monitoring while allowing the user to utilize very expressive conditions.

Deductive databases offer a logic-based language that can be utilized to declaratively express complex conditions. For example, consider an alerter that could be applied in a power distribution application. It states, "Alert the network manager if the connection is broken between the main station and remote station number one". A connection between two stations exists if there is a direct connection between the two stations *or* there is an indirect connection via one or more other stations. The condition is represented by the Datalog expression occurring before the arrow:

not(connection_between(main_station,remote_station_1)) →
alert(network_manager, "remote_station_1_inaccessable").

The recursive *connection_between* relationship can be expressed declaratively using a transitive closure specification:

connection_between(Station1, Station2) :-
direct_connection(Station1,Station2).
connection_between(Station1, Station2) :-
direct_connection(Station1,Station),
connection_between(Station,Station2).

The challenge is to provide efficient support for active services within this declarative framework.

This paper describes an approach towards incorporating active service support in a deductive database that is based on an algebra and data structure that were used in the HiPAC (High Performance Active Database System) project [DAY88a, DAY88b, HSU88, CHA89, MCC89, ROS89, CHN90]. The objective of the HiPAC project is to investigate active, time-constrained database management [MCC89]. The HiPAC system is object-oriented but is described as utilizing a rule subsystem that includes a condition evaluator. The algebra and data structure were presented in [ROS89] and a summary of relevant portions will be provided later in this paper. The ability to define conditions on derived objects (termed *virtual conditions*) is a useful capability [DAY88a]. The HiPAC condition evaluator supports virtual conditions that were limited to select-project-join (SPJ) expressions [CHA90]. Since Datalog allows for more expressive conditions, the application of active service support to deductive databases will result in a technique that can be used to extend the HiPAC condition evaluator. Here, we consider conditions expressed in non-recursive Datalog without negation.

Other research has been aimed at problems relating to active service support in deductive databases. Unfortunately, this work is focused primarily to the area of integrity constraint checking [LLO87,SAD88]. In [SAD88] an approach for checking integrity constraints in a deductive database is presented that uses an extension of the SLDNF proof procedure. Although the extension offers a method for handling implicit insertions and deletions , it does not explicitly handle implicit *modifications*. In addition, since the work focused solely on integrity constraint checking, where the goal is to obtain grounds for update rejection, finding any proof of inconsistency is all that is required. However, when implementing certain types of alerters and triggers (e.g. those with *nonresatisifable* conditions), a derivation of a proof is irrelevant if another proof already exists. Nonresatisfiable conditions, which are discussed later in this paper, represent a class of conditions that are not treated in literature relating to integrity constraint checking.

The remainder of this paper is organized as follows: The differences between condition monitoring and query evaluation, as applied to deductive databases, is given in Section 2. Section 3 provides a description of the data structure, known as delta relations, that supports incremental condition evaluation. A straightforward, incremental condition evaluation (ICE) strategy for nonrecursive Datalog without negation is presented in Section 4. This evaluation strategy monitors resatisfiable conditions. An extension of the monitoring approach is given in Section 5 for nonresatisfiable conditions. The paper concludes with a discussion of the future research directions for providing efficient active service support in deductive databases.

II. Condition Evaluation vs. Query Evaluation

In a deductive DBMS, conditions and queries share an equivalent declarative representation, namely the conjunction of atomic formulae. However, the methods used to evaluate them must be different. Conditions represent a characteristic of the database state that the user wishes to identify. The database state changes after every update. Therefore, each update forces some degree of evaluation of all conditions. If condition evaluation was performed in a naive fashion, the DBMS would incur an unacceptably large overhead cost as a result of each update. Condition evaluation involves reasoning about two different database instances. This differs from (retrieval-only) query evaluation, which is initiated when a query is introduced to the system. With query evaluation, only the current database state is examined.

With condition evaluation, the DBMS can exploit anticipatory knowledge obtained from analyzing the pre-defined conditions. This information can be utilized to increase the efficiency of condition evaluation. This differs from query evaluation where little, if any, anticipatory knowledge is available. Typically, the DBMS has little information available to use for forecasting what query the user may enter.

Another difference between condition evaluation and query evaluation is the origin and the propagation of the constraining bindings. Often, when a query is presented to the DBMS, constants can be extracted to constrain the query. In the case of resolution-based query evaluation techniques [DIE87, WAR90], these constants would propagate, via the unification mechanism, through relevant rules towards the base relations to avoid unnecessary computation. In the case of evaluation techniques based on rule-rewriting followed by semi-naive bottom-up evaluation [BAN86,ULL89], the rules in the database are rewritten to allow constraining bindings originating from the query to propagate towards the base relations to inhibit irrelevant computation.

The query evaluation strategies described do not represent a solution to the condition evaluation problem because, in the condition evaluation problem, the constraining bindings originate from the base relation update and then must be used to constrain the evaluation of the conditions. These bindings propagate towards the conditions that are defined, directly or indirectly, in terms of the updated base relation. This motivates the concept of a condition evaluation strategy (CES). Given a set of conditions and a set of updates, the CES will determine if proofs are formed or lost for expressions that are created by unifying variables occurring in the conditions with constants that are generated as a result of the updates.

III. Delta Relations

The description that follows explains the data structure that is used in the HiPAC approach to ICE. The data structure is called a *delta relation* (abbreviated *Δrelation*). A brief explanation of the data structure is presented here along with notation that will be used in following sections. The concepts presented in this section were developed in [ROS89].

The purpose of a Δrelation is "to provide a single object that captures an arbitrary change to a relation" [ROS89]. A Δrelation represents the net change to a stored or derived relation. Separate algorithms for handling insertions, deletions and modifications are not needed. Neither are algorithms for recombining the results [ROS89]. The

Δrelation is a powerful construct for unifying the effects of database insertions, deletions and modifications.

Consider a relation X with schema $X(X\_tid,X\_att_1,X\_att_2,...X\_att_n)$. Let X_att_i where $1 \le i \le n$, be the attributes of X and let *X_tid* be a unique immutable tuple identifier. Define ΔX as a relation with schema **ΔX**:

$$\mathbf{\Delta X}(\sim X\_tid,\sim X\_att_1,\sim X\_att_2,\ldots,\sim X\_att_n,\ X\_tid\sim,X\_att_1\sim,X\_att_2\sim,\ldots,X\_att_n\sim)$$

The attribute names that *begin* with a "~" (tilde prefix) represent old attribute values and attribute names that *end* with a "~" (tilde suffix) represent new attribute values. There are renaming functions defined for attributes given a schema **X**:

$$pretilde(X) = \{\sim tid_X,\sim X\_att_1,\sim X\_att_2,\ldots,\sim X\_att_n\}$$
$$postilde(X) = \{tid_X\sim,X\_att_1\sim,X\_att_2\sim,\ldots,X\_att_n\sim\}$$

If X is a relation, the function *pretilde(X) (postilde(X))* returns a relation with the same tuples as X but with all attribute names of the schema preceded (followed) by a single tilde. The function *untilde(X)* removes tildes from all attribute names. There are additional functions defined that operate on a Δrelation:

$removals\sim(\Delta X) \equiv P_{pretilde(X)}(\Delta X)$ $\quad$ $additions\sim(\Delta X) \equiv P_{postilde(X)}(\Delta X)$

$removals(\Delta X) \equiv$ untilde(removals~(ΔX)) $\quad$ $additions(\Delta X) \equiv$ untilde(additions~(ΔX))

The definition of these functions is extended such that if a tuple $\Delta x \in \Delta X$ is passed to one of these functions, it is treated as a relation ΔX with a single tuple Δx.

For any Δrelation tuple, the tuple identifier cannot be null on both the pretilde side and the postilde side. The function *insertions(ΔX)* returns every tuple in ΔX where the value of all attributes, including the tid, that occur in the schema produced by pretilde(ΔX) is nil. The function *deletions(ΔX)* returns every tuple in ΔX where the value of all attributes that occur in the schema produced by postilde(ΔX) is nil. The function *modifications(ΔX)* returns every tuple in ΔX where the value of the tid on both the pretilde and postilde side is not nil. Note, that if a tuple represents a modification then both tids will be equal.

IV. An Initial Condition Evaluation Strategy

Since the database state changes after every update and we assume that updates will occur frequently, condition evaluation must be efficient. One way to reduce the cost of evaluating a set of conditions is to evaluate them *incrementally*. The remaining subsections describe a basic incremental condition evaluation strategy (ICES). First, an ICES is presented that can incrementally evaluate a condition consisting of two literals representing extensionally defined (EDB) predicates followed by an example. Next, we modify the ICES to support *n* literals representing either EDB or fully-bound evaluable

predicates. Finally, we add support for conditions defined in terms of non-recursive intensionally defined (IDB) predicates.

Conditions Consisting of Two EDB Literals

To incrementally evaluate a condition whose body consists solely of EDB literals, we can utilize a revised version of the HiPAC incremental join operation that was reported in [ROS89]. Assume we are given two relations (call them X and Y), their corresponding Δrelations and a join predicate $\theta(\mathbf{X},\mathbf{Y})$ that involves attributes of schemas **X** and **Y**. Let XY represent the join of X and Y using $\theta(\mathbf{X},\mathbf{Y})$. Function *IncrJoin* will compute a Δrelation that indicates changes to XY, resulting from the changes to both X and Y. The function is shown in Figure 1.

Essentially, the algorithm performs three separate computations and then combines the results. First, the changes to relation X (i.e. ΔX) are examined to determine which updates will result in a change to the join. Each update in ΔX is matched with a tuple from relation Y that is not going to be removed as a result of the changes to relation Y. Additions and removals from ΔX that match the retained tuples of Y are stored in variable *Delta_join_Kept*. A similar procedure, to determine which updates in ΔY will result in a change to the join, is performed to obtain *Kept_join_Delta*. Finally, the changes to X and Y are examined to detect whether additions to both relations or removals from both relations result in a change to the join. The result of this computation is reflected in variable *Delta_join_Delta*.

This revision of the incremental join algorithm resolves an anomaly with the initial definition that allowed redundant tuples to occur in the result. This was the result of an oversight that occurred when applying the algebra introduced in [ROS89] to derive the operator definition. Closer examination revealed that the algebra itself is correct and derives the revised definition presented here. Further description of both versions of the incremental join operator and a proof of the revised operator appear in [HAR91].

We now present an example that illustrates the incremental join operator given above. Consider the relations PART, STOCK and their join PART_STOCK given in Figure 2. PART_STOCK was created using the join predicate, PART.Pid = STOCK.Prt. The PART relation indicates that part *Pid* has the name *Pname*. The STOCK relation indicates that there are *Pamt* units of part *Prt* currently in stock. The unique tuple identifier in relation STOCK is Prt. Now, assume the following Δrelations exist for PART and STOCK:

ΔPART(~Pid,~Pname,Pid~,Pname~) =

(~Pid,	~Pname,	Pid~,	Pname~)
(p123	widget	p123	wicket)

ΔSTOCK(~Prt,~Pamt,Prt~,Pamt~) =

(~Prt,	~Pamt,	Prt~,	Pamt~)
(p123	100	p123	200)
(p234	200	p234	250)

The tuple in PART specifies that part 'p123' is to have its name changed from 'widget' to 'wicket'. The first tuple in ΔSTOCK specifies that the number of units of part 'p123' is to be increased from 100 to 200. The second tuple specifies that the number of units of part


```
Function IncrJoin(X,Y : relation; ΔX,ΔY : Δ_relation;
                  Join_pred : join_predicate) : Δ_relation;
begin
    Rem_ΔX_join_Y      = join(removals(ΔX), (Y - removals(ΔY)), Join_pred);
    Add_ΔX_join_Y      = join(additions(ΔX), (Y - removals(ΔY)), Join_pred);
    Delta_join_Kept    = outer_join( pretilde(Rem_ΔX_join_Y),
                                     postilde(Add_ΔX_join_Y),
                                     (~tid_X = tid_X~ AND ~tid_Y = tid_Y~));

    X_join_Rem_ΔY      = join((X - removals(ΔX)), removals(ΔY), Join_pred);
    X_join_Add_ΔY      = join((X - removals(ΔX)), additions(ΔY), Join_pred);
    Kept_join_Delta    = outer_join( pretilde(X_join_Rem_ΔY),
                                     postilde(X_join_Add_ΔY),
                                     (~tid_X = tid_X~ AND ~tid_Y = tid_Y~));

    Rem_ΔX_join_Rem_ΔY = join(removals(ΔX), removals(ΔY), Join_pred);
    Add_ΔX_join_Add_ΔY = join(additions(ΔX), additions(ΔY), Join_pred);
    Delta_join_Delta = outer_join( pretilde(Rem_ΔX_join_Rem_ΔY),
                                   postilde(Add_ΔX_join_Add_ΔY),
                                   (~tid_X = tid_X~ AND ~tid_Y = tid_Y~));

    IncrJoin = Delta_join_Kept ∪ Kept_join_Delta ∪ Delta_join_Delta;
end.
```

Figure 1: An Incremental Join Algorithm

PART(Pid,Pname)

(Pid,	Pname)
(p123,	widget)
(p234,	switch)
(p345,	bolt)
(p456,	roller)
(p567,	frame)

STOCK(Prt,Pamt)

(Prt,	Pamt)
(p123,	100)
(p234,	200)
(p345,	300)
(p456,	400)
(p567,	500)

PART_STOCK(Pid,Pname,Prt,Pamt)

(Pid,	Pname,	Prt,	Pamt)
(p123,	widget,	p123,	100)
(p234,	switch,	p234,	200)
(p345,	bolt,	p345,	300)
(p456,	roller,	p456,	400)
(p567,	frame,	p567,	500)

Figure 2: Sample Relations

'p234' is to be increased from 200 to 250. Now consider the incremental join of relations PART and STOCK, again using the join predicate, PART.Pid = STOCK.Prt. Using the relations given above and the definition of the incremental join operator, the following relations would be computed.

Delta_join_Kept = ∅

Kept_join_Delta =

```
(~Pid,~Pname,~Prt, ~Pamt,Pid~,Pname~,Prt~,Pamt~)
(p234,switch,p234,  200, p234,switch,p234, 250)
```

Delta_join_Delta =

```
(~Pid,~Pname,~Prt, ~Pamt,Pid~,Pname~,Prt~,Pamt~)
 (p123,widget,p123, 100, p123,wicket,p123, 200)
```

IncrJoin([PART,STOCK,ΔPART, ΔSTOCK], PART.Pid = STOCK.Prt) =

```
(~Pid,~Pname,~Prt,~Pamt,Pid~,Pname~,Prt~,Pamt~)
(p123,widget,p123, 100, p123,wicket,p123, 200)
(p234,switch,p234, 200, p234,switch,p234, 250)
```

Note that the first tuple that occurs in ΔPART_STOCK (as a result of using IncrJoin) represents a modification to the tuple with tid 'p123' ∈ PART_STOCK. The modification indicates that the fields *Pname,* with value 'widget', and *Pamt* , with value 100, should be changed to have 'wicket' as *Pname* and 200 as *Pamt.* The second tuple represents a modification to the tuple with tid 'p234' in PART_STOCK. The modification indicates that the field *Pamt* should be changed from 200 to 250. These changes correctly correspond to the Δrelations ΔPART and ΔSTOCK. Also note that, here, the union operator (∪) represents the less expensive *disjoint* union operation.

Conditions Consisting of *N* EDB or Fully-Bound Evaluable Literals

Function *IncrJoin* can be utilized to incrementally evaluate conditions consisting of more than two EDB literals. If we have a rule with *n* literals, corresponding to EDB relations R_1 through R_n, and *n* corresponding Δrelations, ΔR_1 through ΔR_n, the condition can be incrementally evaluated by performing the following steps:

$\Delta R_{1,2} = \text{IncrJoin}(R_1 , R_2 , \Delta R_1 , \Delta R_2 , \theta(\mathbf{R_1},\mathbf{R_2}))$

$\Delta R_{1,2,3} = \text{IncrJoin}(R_{1,2} , R_3 , \Delta R_{1,2} , \Delta R_3 , \theta(\mathbf{R_1},\mathbf{R_2},\mathbf{R_3}))$

where $R_{1,2} = \text{join}(R_1,R_2, \theta(\mathbf{R_1},\mathbf{R_2}))$

...

$\Delta R_{1,2,\ldots,n} = \text{IncrJoin}(R_{1,2,\ldots,n-1} , R_n , \Delta R_{1,2,\ldots,n-1} , \Delta R_n , \theta(\mathbf{R_1},\mathbf{R_2},\ldots,\mathbf{R_n}))$

where $R_{1,2,\ldots,n-1} = \text{join}(R_{1,2,\ldots,n-2} , R_{n-1} , \theta(\mathbf{R_1},\mathbf{R_2},\ldots,\mathbf{R_{n-1}}))$

For readers familiar with the differential approach to query evaluation [BAL87], note that this technique is not applicable to ICE because the Δrelations represent non-monotonic changes to the relations.

In this simple approach it is necessary to have access to each partial join for use as the first argument of each call to function *IncrJoin.* Access to tuples contained within the partial join is required by function *IncrJoin* to compute the value of variable *Kept_join_Delta.* Two ways this access can be facilitated is by either recomputing or maintaining each partial join. Maintaining the partial join is prohibitive because of the large cost in terms of space. Recomputing the partial join is suboptimal because the bindings that occur in the Δrelation of the second argument to function IncrJoin (i.e. ΔY) can be used to significantly constrain the computation required to obtain the portion of the partial join that is necessary to compute *Kept_join_Delta.* A technique that does not require materialization and also utilizes available bindings to optimize the access to necessary portions of the partial join is incorporated in procedure *ICES*, described below.

Procedure *ICES* will incrementally evaluate a datalog condition that consists of either EDB or fully-bound evaluable literals. The literals are processed in a cascading fashion (i.e. $[[[[\text{LIT}_1, \text{LIT}_2], \text{LIT}_3], \text{LIT}_4], \ldots, \text{LIT}_n]$). In an actual implementation, an optimizer could reorder the literals. Procedure *ICES* avoids the need to access the various partial joins (e.g. $\text{LIT}_{1,2}$,$\text{LIT}_{1,2,3}$, etc.) by posing successively larger conjunctions of literals as datalog queries to the database. Each query is constrained by the bindings that occur in the Δrelation of a literal in the rule and the accumulated Δrelation for the rule body up to that point.

For example, if the algorithm is currently processing literal LIT_k, a query is issued consisting of literals 1 through k-1, constrained by bindings occurring in ΔLIT_k and removals($\Delta\text{LIT}_{1,\ldots,k-1}$). This query is issued in lieu of computing the partial join and is used by procedure *ICES* to compute the variable *Kept_join_Delta.* Note that literals 1 to k-1 can be fully-bound evaluable. Another query is issued to compute *Delta_join_Kept.* Assume, again, we are processing literal LIT_k. Here a query is issued to evaluate literal k, constrained by bindings occurring in removals(ΔLIT_k) and $\Delta\text{LIT}_{1,\ldots,k-1}$.

We assume the existence of a function *eval(Bindings,Literals,Exclusions)* that performs the queries as described above. *Literals* is a conjunction of literals that represent the partial join. *Bindings* is a relation from which bindings can be extracted that will constrain the evaluation of *Literals* and therefore eliminate the computation of unnecessary tuples of the partial join. The relation *Exclusions* is used to disqualify certain tuples that result from the evaluation of *Literals.* Tuples that result from evaluating *Literals* but are present in relation *Exclusions* are disregarded. If a left-to-right sideways information passing (SIP) strategy was used, function *eval* would evaluate a query of the form:

:- bindings, LIT_1, LIT_2, ... , LIT_n, *not* exclusions.

Since function *eval* queries the database to access the relevant portions of the partial join, the entire partial join does not need to be maintained or recomputed to obtain *Kept_join_Delta.* As a convenience, function *eval* can also be used to compute *Delta_join_Kept.*

If a literal represents a fully-bound evaluable predicate then evaluating it cannot result in the addition of any columns of bindings to the resulting Δrelation for the rule

body. Therefore, to process a fully-bound evaluable literal, procedure *ICES* need only filter the Δrelation constructed by incrementally evaluating the previous literals. In addition, a fully-bound evaluable literal will always have a null Δrelation since unlike EDB and IDB predicates, it cannot change as a result of an update. As a result, both *Kept_join_Delta* and *Delta_join_Delta* will always be null and need not be computed.

Procedure ICES is shown in Figure 3. The arguments passed to the procedure include a rule body (RB) consisting of *n* literals denoted by LIT_1 through LIT_n that represent EDB or evaluable predicates. Procedure *ICES* also receives a Δrelation corresponding to each non-evaluable literal of the rule. These Δrelations are denoted by ΔLIT_i where *i* is the literal's position in the rule body.

Conditions Containing Intensionally-Defined Predicates

Conditions defined in terms of non-recursive intensionally defined (IDB) predicates can be handled in a straightforward manner. Before run-time, the conditions are *unfolded* [SAT84]. Each condition containing an IDB predicate is rewritten to obtain a set of conditions that do not contain any IDB predicates. All that is required is to backward-substitute rule bodies for rule heads. We assume that the rules in the database are rectified as described in [ULL88]. The substitution is repeated recursively until a set of conditions, expressed using only EDB relations, is produced corresponding to each original condition. No IDB predicate will occur in any of the rewritten conditions. Procedure ICES can then be used to incrementally evaluate the resulting set of conditions.

An alternative for supporting conditions defined in terms of IDB predicates is to employ a bottom-up rule evaluation strategy. Consider a predicate P that is defined by several rules where in each of the rule bodies the exists a literal corresponding to an updated base relation. Each rule body would then have to be evaluated incrementally, resulting in a corresponding (possibly null) Δrelation describing the changes to P as indicated by that particular rule. Next, Δrelations would be produced for each of the rule heads by projecting attributes from the Δrelations formed for the rule bodies. Since the incremental projection operator described in [ROS89] has the restriction that the tuple identifier must be projected out with any additional attributes the user requests, the system would have to store these distinguishing attribute values in the corresponding Δrelation. The next task would be to propagate these Δrelations to rules that contain a literal corresponding to P in the body. These newly identified rules would then be evaluated to obtain more changes. This process would continue recursively as long as new Δrelations were produced. At that point, the conditions themselves would be incrementally evaluated and ICE would be complete.

With this approach, condition rewriting is avoided but a complication is introduced. Consider again the Δrelations produced for each rule of predicate P. Because of the restriction involving the projected tids, these Δrelations could have different schemas. Merging the Δrelations to form one unified Δrelation that represents all changes to P poses a challenge . This complication encourages the use of the unfolded conditions.


```
Procedure ICES(in RB: rule_body; ΔLIT : set of Δrelations; out ΔRB : Δrelation);
begin /* ICES */
    Curlit = 1;
    ΔRB_Curlit = ΔLIT_Curlit;          /* if only one literal in rule body, result is ΔLIT_1 */

        while Curlit ≤ num_of_lits(RB) do
        begin
          ++Curlit;                                   /* consider the next literal */
          Rem_ΔPrev_join_Current = eval(removals(ΔRB_Curlit - 1),[LIT_Curlit],removals(ΔLIT_Curlit));
          Add_ΔPrev_join_Current = eval(additions(ΔRB_Curlit - 1),[LIT_Curlit],removals(ΔLIT_Curlit));
          Delta_join_Kept        = outer_join(pretilde(Rem_ΔPrev_join_Current),
                                              (postilde(Add_ΔPrev_join_Current),
                                              (~tid_Prev= tid_Prev~ AND ~tid_Curlit = tid_Curlit~));

          if evaluable(LIT_Curlit) then
                    ΔRB_Curlit = Delta_join_Kept
          else
          begin /* non-evaluable */

                Prev_join_Rem_ΔCurrent =
                        eval(removals(ΔLIT_Curlit),[LIT_1,...,LIT_Curlit-1],removals(ΔRB_Curlit - 1));
                Prev_join_Add_ΔCurrent =
                        eval(additions(ΔLIT_Curlit),[LIT_1,...,LIT_Curlit-1],removals(ΔRB_Curlit - 1));

                Kept_join_Delta          = outer_join(pretilde(Prev_join_Rem_ΔCurrent),
                                                      postilde(Prev_join_Add_ΔCurrent),
                                                      (~tid_Prev= tid_Prev~ AND ~tid_Curlit = tid_Curlit~));

                Rem_ΔPrev_join_Rem_ΔCurrent  = join(removals(ΔRB_Curlit - 1),
                                                    removals(ΔLIT_Curlit), Join_pred);
                Add_ΔPrev_join_Add_ΔCurrent  = join(additions(ΔRB_Curlit - 1),
                                                    additions(ΔLIT_Curlit), Join_pred);
                Delta_join_Delta = outer_join(
                                                    pretilde(Rem_ΔPrev_join_Rem_ΔCurrent),
                                                    postilde(Add_ΔPrev_join_Add_ΔCurrent),
                                                    (~tid_Prev= tid_Prev~ AND ~tid_Curlit = tid_Curlit~));

                ΔRB_Curlit = Delta_join_Kept ∪ Kept_join_Delta ∪ Delta_join_Delta;

          end; /* non-evaluable */
        endwhile;

    ΔRB = ΔRB_Curlit;

end. /* ICES */
```

Figure 3: Procedure ICES

V. Resatisfiable vs. Nonresatisfiable Conditions

We now focus on the subtle distinction between two types of conditions, namely those that are *resatisfiable* and those that are *nonresatisfiable*. Resatisfiable conditions can be viewed as event oriented. For example, consider a condition that states, "An inventory shortage exists if the stock level of some item falls below the predetermined minimal amount that is required to be kept on hand". Assume that there is an inventory item called a widget and that there are currently 400 in stock. Also assume that the predetermined minimal amount of widgets that are required to be kept on hand is 500. If the condition is labeled as resatisfiable, each time a change occurs to either the stock level or the predetermined minimal amount and that change does not rectify the shortage, the system would report the inventory shortage again. In this example, this means that if the stock level of widgets dropped from 400 to 300 and then to 200, the condition would be reported as being satisfied after each reduction (i.e. each event) and two additional notifications would be issued.

Nonresatisfiable conditions can be viewed as being state oriented. If the condition is nonresatisfiable, then once the shortage is initially detected no additional reports will be issued until a change occurs to either the stock level or the minimal amount that resolves the shortage (i.e. changes the state). This means that, once the condition has been reported as being satisfied, further reductions of widget inventory would not result in any additional notification.

By using the technique already described, resatisfiable conditions can be monitored. The information is contained in the Δrelations resulting from ICE. Each addition represents a set of bindings that, when unified with variables in the condition definition, will form a provable expression in the new database state. The new database state will reflect the updates to the base relations. This does not imply, however, that the expression was not already provable in the original database state. It merely indicates that a new proof exists for the expression. Each removal represents a set of bindings that, when unified with variables in the condition definition, forms an expression that has lost a method of proof in the new database state. This does not imply, however, that the expression is not provable in the new database state. It merely indicates that one less proof exists.

Each addition occurring in a Δrelation corresponding to a nonresatisfiable condition represents a set of bindings that, when unified with variables in the condition definition, will form a provable expression in the new database state that was not already provable in the original database state. Each removal represents a set of bindings that, when unified with variables in the condition definition, forms an expression that is no longer provable in the new database state.

The Δrelations resulting from ICE must be processed further to support nonresatisfiable conditions. The additions that occur in Δrelations resulting from ICE must be unified with the condition and then posed as queries to the initial database state. The removals must be unified with the condition and then posed as queries to the new database state. Fortunately, both states are available because updates to the various base relations were placed into Δrelations to be used in the condition evaluation process but do not have to be incorporated to the base relations to perform ICE. Below we describe additional processing for the Δrelations that is required for nonresatisfiable conditions.

Each expression formed with bindings from an addition is posed as a query before the updates to the base relations are incorporated to check if the expression is already provable. If a tuple of addition bindings, when unified with the condition, forms a provable expression then the tuple is discarded. If the expression is not provable, the tuple is considered as satisfying the condition for the first time.

Now, consider the removals. By definition, each tuple that appears as a removal has already been proven. All that is needed is to perform the actual updates to the base relations and then perform a check to determine if the expression formed with the removal tuple is still provable. If so, the removal can be discarded since the expression is still provable. If the expression is no longer provable then it represents the last expression formed with the bindings from the removal tuple that satisfied the condition.

The procedure *Process_Nonresatisfiable_Conditions*, shown in Figure 4, implements the concepts above. Let the set of Δrelations ΔD be the result of using procedure ICES on the set of all nonresatisfiable conditions.

```
Procedure Process_Nonresatisfiable_Conditions(ΔD)
begin
  For each Δd_i ∈ ΔD do
    For each tuple t ∈ Δd_i do
      if additions(t) ≠ ∅ AND provable(additions(t)) then addition(t) ← ∅;

  {Perform updates to base relations}

  For each Δd_i ∈ ΔD do
    For each tuple t ∈ Δd_i do begin
      if removals(t) ≠ ∅ AND provable(removals(t) then removals(t) ← ∅;
      if removals(t) = ∅ AND additions(t) = ∅      then Δd_i = Δd_i - t;
    end;
  {report changes as specified in ΔD}
end;
```

Figure 4: Procedure Process_Nonresatisfiable_Conditions

VI. Summary and Future Work

We have described an approach towards incorporating active service support in a deductive database that is based on incremental condition evaluation and *delta relations*. This approach was used successfully in the HiPAC project and it was our objective to determine the feasibility of applying this technology towards the development of an active deductive database. Our results demonstrate the usefulness of the technology for incrementally evaluating conditions defined in terms of non-recursive datalog without negation. Algorithms for incrementally evaluating conditions that are defined in terms of recursive and negated predicates are currently under investigation. When refined, these algorithms may be used to enhance the capability of the HiPAC condition evaluation component as well as to provide support for an active *deductive* database system.

Acknowledgements

The authors would like to thank Arnon Rosenthal and Huibin Zhao for providing material explaining the theory of incremental changes. This work was supported by an Arizona State University, Office of the Vice President of Research, Research Assistantship Award.

References

[BAL87] Balbin, I. and Ramamohanarao, K., "A Generalization of the Differential Approach to Recursive Query Evaluation", Journal of Logic Programming, 1987:4:259-262.

[BAN86] 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.

[CHA89] Chakravarthy, S., "Rule Management and Evaluation: An Active BMS Perspective", *SIGMOD RECORD, Special Issue on Rule Management and Processing in Expert Database Systems*,Vol. 18, No. 3, September 1989, pp. 20-28.

[CHA90] Chakravarthy, Sharma, personal communication, September 25, 1990.

[CHN90] Chakravarthy, S. and Nesson, S., "Making an Object-Oriented DBMS Active: Design, Implementation, and Evaluation of a Prototype",*Proc. of the Intl. Conf. on Extending Database Technology*, Venice, March 1990.

[DAY88a] Dayal, U., et. al., "The HiPAC Project: Combining Active Databases and Timing Constraints", *SIGMOD RECORD,* Vol. 17, No. 1, March 1988, pp. 51-70.

[DAY88b] Dayal, U., "Active Database Management Systems", *Proc. of the Third Int. Conf. on Data and Knowledge Bases*, Jerusalem, June 1988, pp. 150-170.

[DIE87] Dietrich, S. W., "Extension Tables: Memo Relations in Logic Programming", *IEEE Symposium on Logic Programming,* San Francisco, CA, 1987, pp. 264-272.

[HAR91] Harrison, John and Dietrich, Suzanne W., "An Incremental Join Operator for Active Databases", Technical Report TR-91-012, Department of Computer Science, Arizona State University

[HSU88] Hsu, M., Ladkin, R. and McCarthy, D., "An Execution Model for Active Database Management Systems", *Third International Conference on Data and Knowledge Bases,* Jerusalem, June 1988, pp. 171-179.

[LAG89] The Laguna Beach Participants, "Future Directions in DBMS Research", *SIGMOD RECORD,* Vol. 18, No. 1, March 1989, pp. 17-26.

[LLO87] Lloyd, J. W., Sonenberg, E. A., and Topor, R. W., "Integrity Constraint Checking in Stratified Databases", *J. Logic Programming*, 4:331-343, 1987.

[MCC89] McCarthy, D. and Dayal, U., "The Architecture of an Active Data Base Management System", *Proc. 1989 ACM SIGMOD Conference on Management of Data*, Portland, OR, 1989, pp. 215-224.

[ROS89] Rosenthal, A., Chakravarthy, S., Blaustein, B., and Blakely, J., "Situation Monitoring for Active Databases", *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, 1989.

[SAD88] Sadri, F. and Kowalski, R., "Theorem-Proving Approach to Database Integrity", Appears in: *Foundations of Deductive Databases and Logic Programming* (ed. Jack Minker), Morgan Kaufmann Pub., Los Altos, CA,1988, pp. 313-362.

[STO90] Stonebraker, M., et. al., "Third-Generation Database System Manifesto", *SIGMOD RECORD,* Vol. 19, No. 3, September 1990, pp. 31-44.

[TAM84] Tamaki, H., and Sato, T., "Unfold/Fold Transformation of Logic Programs", *Proceedings of the Second International Symposium on Logic Programming,* Sweden, July 1984, pp. 127-138.

[ULL88] Ullman, J., *Principles of Database and Knowledge-base Systems,* Vol. 1, Computer Science Press, Rockville, MD,1988.

[ULL89] Ullman, J., "Bottom-up Beats Top-down for Datalog", *Principles of Database Systems*, 1989, pp. 140-149.

[WAR90] Warren, D. S. , "The XWAM: A Machine that integrates Prolog and Deductive Database Query Evaluation", *Proceedings of the Architecture Workshop/North American Conference on Logic Programming* (ed. J. Mills), Austin, TX,1990.

Research and Practical Issues in Databases

Proceedings of the 3rd Australian Database Conference

Melbourne 3–4 February 1992

Editors

B Srinivasan
Department of Computer Science
Monash University
Australia

J Zeleznikow
Applied Computing Research Institute
La Trobe University
Australia

World Scientific
Singapore • New Jersey • London • Hong Kong