Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions

Martin Sköld, Tore Risch
Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden,
e-mail: {marsk,torri}@ida.liu.se

Presented at the 12th International Conference on Data Engineering (ICDE'96)'
New Orleans, Louisiana, February 1996

Abstract

This paper presents a difference calculus for determining changes to rule conditions in an active DBMS. The calculus has been used for implementing an algorithm to efficiently monitor rules with complex conditions. The calculus is based on *partial differencing* of queries derived from rule conditions. For each rule condition several partially differentiated queries are generated that each considers changes to a single base relation or view that the condition depends on. The calculus considers both insertions and deletions. The algorithm is optimized for deferred rule condition monitoring in transactions with few updates. The calculus allows us to optimize both space and time. Space optimization is achieved since the calculus and the algorithm does not presuppose materialization of monitored conditions to find its previous state. This is achieved by using a *breadth-first, bottom-up* propagation algorithm and by calculating previous states by doing a *logical rollback*. Time optimization is achieved through incremental evaluation techniques. The algorithm has been implemented and a performance study is presented at the end of the paper.

1 Introduction

When introducing rules into a database it is crucial that the overall performance of the database is not impaired significantly. *Rule monitoring* is the activity of monitoring changes to the state of rule conditions. A *naive* method of detecting changes is to execute the complete condition when an event that triggers the rule has occurred. This, however, can be very costly, since a rule condition can span over large portions of the database.

This paper presents a technique for efficient monitoring of active rules integrated with a query language of an Object Relational database system. The technique is especially designed for *deferred* rules, i.e. rules where the rule execution is deferred until a check phase that usually occurs when transactions are committed. The technique can also be used for immediate rule processing[4], but this is outside the scope of the paper.

A *difference calculus* will be defined for computations of changes to the results of database queries and views. Queries and relational views are regarded as functions over sets of tuples and the calculus for monitoring changes is regarded as an extension of set algebra. Let P be a function dependent on the functions Q and R, denoted the *influents* of the *affected* function P. The problem of *finite differencing*[14] is how to calculate changes to P, ΔP, in terms of changes to its influents. With *partial differencing,* changes to P are defined as the combination of the changes to P originating in the changes to each of its influents. Thus, ΔP is a function of the *partial differential* functions $\Delta P/\Delta Q$ and $\Delta P/\Delta R$. In this paper we define how to automatically derive the partial differentials $\Delta P/\Delta Q$ and $\Delta P/\Delta R$, and how to calculate ΔP from them. The calculus is mapped to relational algebra by defining partial differentials for the basic relational operators. Partial differencing has the following properties compared to other approaches:

- We assume that the number of updates in a transaction is usually small and often very few (or only one) tables are updated. Therefore, very few partial differentials are affected in each transaction. Each partial differential generated by the rule compiler is a relatively simple database query which is optimized using traditional query optimization techniques [22]. The optimizer assumes few changes to a single influent.

- We separately define *positive* and *negative* partial differentials, denoted $\Delta P/\Delta_+ Q$ and $\Delta P/\Delta_- Q$, respectively, since monitored conditions are often only dependent on insertions in influents (not on deletions), as will be shown. Furthermore, the partial differentials for handling insertions and deletions do not have the same structure. Conditions that depend on deletions are actually historical queries that must be executed in the database state when the deleted data were present. This makes negative differentials different and not easily mixable with positive ones.

- The calculus allows us to optimize both space and time. Space optimization is achieved since the calculus and the algorithm does not presuppose materialization of monitored conditions to find their previous state. Instead it gives a choice between materialization and computation of the old state from the new one, given all the state changes. Time optimization is achieved through incremental evaluation techniques.

- Based on the calculus, an algorithm has been developed for efficient rule condition monitoring by propagation of incremental changes through a *dependency network*. For correct handling of deletions in the absence of materializations and for efficient execution, a *breadth-first, bottom-up* propagation is made through the network of both insertions and (only when applicable) deletions. The algorithm reduces memory utilization by only temporarily saving the intermediate changes appearing during the propagation.

- For explainability, one can easily determine which influents actually caused a rule to trigger and if it was triggered by an insertion or a deletion. It is straight forward to determine this by remembering which partial differentials were actually executed in the triggering.

The method is implemented and a performance measurement has been made. We have implemented both our incremental algorithm and a 'naive' condition monitoring algorithm that recomputes the whole rule condition every time an update has been made to an influent affecting a condition. The performance evaluation shows that for transactions with few updates our incremental algorithm scales better over the database size than the naive method. For transactions with many updates to several influents the method is not as efficient as naive evaluation, but with a factor that is constant over the size of the database.

The default semantics of our active rules [19] uses the *CA model* where each rule is a pair, <Condition,Action>, where the condition is a declarative database query, and the action is a database procedural expression. The method can be used for ECA-rules as well; the event part just further restricts when the condition is tested. Set-oriented action execution[24] is supported since data can be passed from the condition to the action of each rule by using shared query variables. Condition evaluation is delayed until a *check phase* usually at commit time. In the check phase, change propagation is performed only when changes affecting activated rules have occurred, i.e. no overhead is placed on database operations (queries or updates) that do not affect any rules. After the change propagation, one triggered rule is chosen through a *conflict resolution method*[1]. Then the action of the rule is executed for each instance for which the rule condition is true based on the net changes of the rule condition.

The paper proceeds as follows. In section 2 related work is presented. In section 3 active rules in our DBMS are introduced with a running example used throughout the rest of the paper. In section 4 the calculus of partial differencing is introduced, first intuitively and then formally. In section 5 the propagation algorithm is presented. In section 6 a performance measurement of the algorithm is presented. In section 7 some possible refinements to the implementation are presented. Finally in section 8 a summary and future work is presented.

2 Related Work

Incremental evaluation of database queries was presented as *finite* differencing in [14] and in [3] as a technique for continuously maintaining derived data in materialized views. The technique was adopted for rule condition monitoring in HiPac[4][20], Ariel[11], PARADISER[5], and [7]. Recent work on incremental maintenance of materialized views can be found in [10][13] and on incremental evaluation of Datalog programs in [6].

Our work differs from the above in that we deal with the problem of *partial* differencing of database queries, i.e. automatic generation of several separate partial differentials from a given rule condition rather than one large incremental expression. Furthermore, we also deal with deletions and incremental evaluation of deferred rule conditions.

In [17] the relational algebra is extended with incremental expressions. In [1] a method is presented that derives two optimized conditions, *Previously True* and *Previously False*, based on a materialization of a simple truth value of a condition. Since our rules are set-oriented we need to consider sets of truth values.

In contrast to the work above we will also present a space and time efficient propagation algorithm based on our calculus. By using *breadth-first, bottom-up* propagation to correctly and efficiently propagate both positive and negative changes without retaining space consuming materializations of intermediate views our algorithm differs from the PF-algorithm [12]. The materialized views can be very large and can even be considerably larger than the original database, e.g. where cartesian products or unions are used. This may exhaust memory or buffers when many conditions are monitored and the database is large. Ariel[11] uses a propagation algorithm called TREAT[16] and avoids materialization of intermediate results, but in a more restricted way than in our approach, without using any formal calculus.

1. Conflict resolution is the process of choosing one single rule when more than one rule is triggered.

3 Monitoring Active Rule Conditions

Active rules have been introduced into AMOS[8][19] (Active Mediators Object System), an Object Relational DBMS. The data model of AMOS is based on the functional data model of Daplex[21] and Iris[9]. AMOSQL, the query language of AMOS, is a derivative of OSQL. The data model of Iris is based on objects, types, and functions. In AMOS the data model is extended with rules. Everything in the data model is an object, including types, functions, and rules. All objects are classified as belonging to one or several types, i.e. classes. Functions can be stored, derived, or foreign. Stored functions equal object attributes or base tables, derived functions equal methods or relational views, and foreign functions are functions written in some procedural language[1]. Procedures can be defined as functions that have side-effects. AMOSQL extends OSQL with active rules, a richer type system, and multidatabase functionality.

3.1 Rules in AMOSQL

Condition Action (CA) rules have been introduced into AMOSQL. The condition is an AMOSQL query and the action is an AMOSQL procedural expression.
The syntax for rules is as follows:

create rule *rule-name parameter-specification* **as**
  when *for-each-clause* | *predicate-expression*
  do *procedure-expression*
where
for-each-clause ::=
    for each *variable-declaration-commalist*
    where *predicate-expression*

The *predicate-expression* can contain any boolean expression, including conjunction, disjunction, and negation. Rules are activated and deactivated separately for different parameters.

The semantics of a rule is as follows: If an event of the database changes the truth value for some instance of the condition to *true*, the rule is marked as *triggered* for that instance. If something happens later in the transaction which causes the condition to become false again, the rule is no longer triggered. This ensures that we only react to net changes, i.e. *logical events*. A non-empty result of the query that represents the condition is regarded as *true* and an empty result is regarded as *false*.

A classical example for active databases is that of monitoring the quantity of items in an inventory. When the quantity of an item drops below a certain threshold, new items are to be automatically ordered. The definitions in AMOSQL would be:

```
create type item;
create type supplier;
create function quantity(item) -> integer;
create function max_stock(item) -> integer;
create function min_stock(item) -> integer;
create function consume_freq(item)
              -> integer;
create function supplies(supplier) -> item;
create function delivery_time(item,supplier)
              -> integer;
create function threshold(item i) -> integer
    as
    select consume_freq(i) *
        delivery_time(i, s) + min_stock(i)
    for each supplier s where supplies(s) = i;
create rule monitor_item(item i) as
    when quantity(i) < threshold(i)
    do order(i, max_stock(i) - quantity(i));
create rule monitor_items() as
     when for each item i
     where quantity(i) < threshold(i)
     do order(i,max_stock(i) - quantity(i));
```

The `monitor_item` rule monitors the quantity of a specific item in stock and orders new items when the quantity drops below the threshold, considering the time to get new items delivered. The procedure `order` does the actual ordering.

The consume-frequency defines how many instances of a specific item are consumed on average per day. The `monitor_items` rule monitors all items instead of just one at a time. This rule will be used as an example throughout the rest of the paper.

Next we populate the database and activate the rule `monitor_items`:

```
create item instances :item1[2], :item2;
set max_stock(:item1) = 5000;
set max_stock(:item2) = 7500;
set min_stock(:item1) = 100;
set min_stock(:item2) = 200;
set consume_freq(:item1) = 20;
set consume_freq(:item2) = 30;
create supplier instances :sup1, :sup2;
set supplies(:sup1) = :item1;
set supplies(:sup2) = :item2;
set delivery_time(:item1, :sup1) = 2;
set delivery_time(:item2, :sup2) = 3;
activate monitor_items();
```

This will ensure that the quantity of items of type 1 is always kept between 5000 and 100, and new items will be delivered if the quantity drops below 140. The quantity of

1. In AMOS foreign functions can be written in Lisp or C.

2. These are interface variables and are not part of the stored database.

items of type 2 will be kept between 7500 and 200, and new items will be ordered if the quantity drops below 290.

3.2 Rule Compilation

The rule compiler generates the condition function `cnd_monitor_items` from the condition of the rule `monitor_items`. This function returns all the items with quantities below the threshold. Condition monitoring is regarded as monitoring changes to the condition function[18].

```
create function cnd_monitor_items() -> item
    as
    select i for each item i
    where quantity(i) < threshold(i);
```

The action part of the rule generates a procedure that takes an item as argument and orders new items to fill the inventory.

```
create function act_monitor_items(item i)
    -> boolean[1] as
    order(i, max_stock(i) - quantity(i));
```

At run-time the `act_monitor_items` procedure will be applied to the set of *changes* calculated from the differential denoted Δ`cnd_monitor_items`:

```
act_monitor_items(Δcnd_monitor_items());
```

AMOSQL is a stream-oriented language so the procedure is executed for every changed value of the condition. We distinguish between *strict* and *nervous* rule execution semantics. With strict semantics the action procedure is executed *only* when the truth value of the monitored condition changes from false to true in some transaction. With nervous semantics the rule sometimes triggers when there has been an update that causes the rule condition to become true without having been false previously. Nervous semantics is often sufficient; however, in our example strict semantics is preferable since we only want to order an item once when it becomes low in stock. Note that before the action part of a triggered rule is executed a conflict resolution method is applied.

By looking at the definition of `cnd_monitor_items` we can define a dependency network (fig. 1) that specifies what changes can affect the differential Δ`cnd_monitor_items`. Each edge in the dependency network defines the influence from one function to another. With each edge we also associate the partial differentials that calculate the actual influence from a particular node. For instance, Δ`quantity` is an influent of Δ`cnd_monitor_items` with a partial differential Δ`cnd_monitor_items`/Δ`quantity` (the edge marked * in fig. 1). The dependency network is constructed from the definition of the condition function and its sub-functions.

1. A procedure that does not explicitly return anything implicitly returns a boolean.

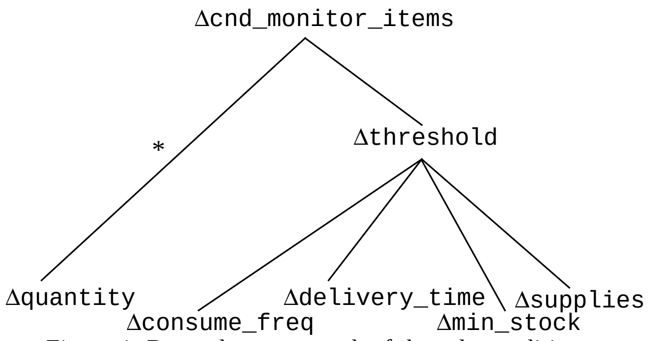

Figure 1: Dependency network of the rule condition

In our system AMOSQL functions are compiled into a domain calculus language called ObjectLog[15], which is a variant of Datalog where facts and Horn Clauses are augmented with type signatures. In AMOS stored functions are compiled into facts (base relations) and derived functions are compiled into Horn Clauses (derived relations). In our example the system can deduce the dependency network by looking the definitions of the functions `cnd_monitor_items` and `threshold`:

$$
\begin{aligned}
&\texttt{cnd\_monitor\_items}_{\texttt{item}}(\texttt{I}) \leftarrow \\
&\quad \texttt{quantity}_{\texttt{item,integer}}(\texttt{I,\_G1}) \wedge \\
&\quad \texttt{threshold}_{\texttt{item,integer}}(\texttt{I,\_G2}) \wedge \\
&\quad \texttt{\_G1 < \_G2}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{threshold}_{\texttt{item,integer}}(\texttt{I,T}) \leftarrow \\
&\quad \texttt{consume\_freq}_{\texttt{item,integer}}(\texttt{I,\_G1}) \wedge \\
&\quad \texttt{delivery\_time}_{\texttt{item,supplier,integer}}(\texttt{I,\_G2,\_G3}) \wedge \\
&\quad \texttt{supplies}_{\texttt{item,supplier}}(\texttt{I,\_G2}) \wedge \\
&\quad \texttt{\_G4 = \_G1 * \_G3} \wedge \\
&\quad \texttt{min\_stock}_{\texttt{item,integer}}(\texttt{I,\_G5}) \wedge \\
&\quad \texttt{T = \_G4 + \_G5}
\end{aligned}
$$

4 The Calculus of Partial Differencing

The calculus of partial differencing is our basis for incremental evaluation of rule conditions. It formalizes update event detection and incremental change monitoring. The calculus is based on the usual set operators *union* ($\cup$), *intersection* ($\cap$), *difference* (-), and *complement* (~). Three new operators are introduced, *delta-plus* (Δ_+), *delta-minus* (Δ_-), and *delta-union* ($\cup_\Delta$). Δ_+ returns all tuples added to a set over a specified period of time, and Δ_- all tuples removed from the set. A *delta-set* (Δ-set) is defined as a disjoint pair $<\Delta_+S, \Delta_-S>$ for some set S and $\cup_\Delta$ is defined as the union of two Δ-sets. The calculus is general and in section 4.6 partial differencing of the relational algebra operators is shown.

Separate *partial differentials* are generated for monitoring insertions and deletions for each influent of a derived relation. The intuition is to calculate positive partial differentials (monitoring insertions) in the new state of the database. The negative partial differentials (monitoring deletions) are calculated in the old state since this was when the deleted tuples were present in the database.

The old state of a relation is calculated from the new state by performing a *logical rollback* that inverts all the updates. Given the value of S_{new} we can calculate S_{old} by inverting all operations done to S, i.e. by using $S_{old} = (S_{new} \cup \Delta_- S) - \Delta_+ S$. The calculus is based on accumulating all the relevant updates to base relations during a transaction. These accumulated changes are then used to calculate the partial differentials of derived relations. Changes are propagated in a breadth-first, bottom-up manner through a propagation network where the Δ-sets can be seen as temporary 'wave-front' materializations. Calculating the old state, S_{old}, requires all the propagated changes that influence S, i.e. the complete $\Delta_+ S$ and $\Delta_- S$.

Our algorithm guarantees that all changes to influents of an affected relation are propagated before the changes to the affected relation are propagated further. Therefore, by propagating breadth-first, bottom-up we can calculate the old states (S_{old}) of relations by doing a logical rollback. Next we define how to accumulate these changes and how to generate partial differentials.

4.1 Differencing of Base Relations

All changes to base relations, i.e. stored functions, are logged in a logical undo/redo log. During database transactions, before these physical update events are written to the log, a check is made if a stored base relation was updated that might change the truth value of some activated rule condition. If so, the *physical events* are accumulated in a Δ-*set* that reflects all *logical events* so far of the updated relation. Only those functions that are influents of some rule condition need Δ-sets. The Δ-sets can be discarded when the changes of the affected relations have been calculated, which saves space. Since rules are only triggered by logical events the physical events have to be added with the *delta union* operator, $\cup_\Delta$, that cancels corresponding insertions and deletions in the Δ-set. The Δ-set for a base relation B is defined as:

$\Delta B = <\Delta_+ B, \Delta_- B>$,

where $\Delta_+ B$ is the set of added tuples to B and $\Delta_- B$ is the set of removed tuples, they are defined as:

$\Delta_+ B = B - B_{old}$ and[1]

$\Delta_- B = B_{old} - B$, and thus

$B_{old} = (B \cup \Delta_- B) - \Delta_+ B$

We define $\cup_\Delta$ formally as:

$$\Delta B_1 \cup_\Delta \Delta B_2 = <(\Delta_+ B_1 - \Delta_- B_2) \cup (\Delta_+ B_2 - \Delta_- B_1), (\Delta_- B_1 - \Delta_+ B_2) \cup (\Delta_- B_2 - \Delta_+ B_1) >$$

1. The current database always reflects the new state

The operator works correctly when there is no net effect of updates to a function. Updates to stored functions are made by first removing the old value tuples and then adding the new ones. For example, let us update the minimum stock of some item twice assuming that `min_stock` was originally 100:

```
set min_stock(:item1) = 150;
set min_stock(:item1) = 100;
```

This produces the physical update events:

```
-(min_stock,:item1,100),
+(min_stock,:item1,150),
-(min_stock,:item1,150),
+(min_stock,:item1,100).
```

The Δ-set for `min_stock` changes accordingly with:

```
Δmin_stock = <{},{(:item1,100)}>
Δmin_stock = <{(:item1,150)},{(:item1,100)}>
Δmin_stock = <{},{(:item1,100)}>
Δmin_stock = <{},{}>
```

i.e. there is no net effect of the updates.

4.2 Partial Differencing of Views

As for base relations, the Δ-set of a relational view, i.e. a derived function, is defined as a pair:

$\Delta P = <\Delta_+ P, \Delta_- P>$

We need to define how to calculate the Δ-set of an affected view in terms of the Δ-sets of its influents. To motivate our calculus we next exemplify change monitoring of views for positive changes (adding) and negative changes (removing), respectively. We then show how to combine partial differentials into the final calculus.

4.3 Positive Partial Differentials

For a view P defined as a Horn Clause with a conjunctive body, let I_p be the set of all its influents. The positive partial differentials $\Delta P/\Delta_+ X_i$, $X_i \in I_p$ (for insertions only) are constructed by substituting X_i in P with its positive differential $\Delta_+ X_i$.

For example, if

```
p(X, Z) ←
        q(X, Y) ∧
        r(Y, Z)
```

then

```
Δp(X, Z)/Δ+q ←
        Δ+q(X, Y) ∧
        r(Y, Z)
```

and

```
Δp(X, Z)/Δ+r ←
        q(X, Y) ∧
        Δ+r(Y, Z)
```

If DB_{old} consist of the stored relations (facts) `q(1, 1)`, `r(1, 2)`, `r(2, 3)`, then we can derive `p(1, 2)`.

A transaction performs the updates
`assert q(1, 2), assert r(1, 4)`

DB_{new} now becomes `q(1, 1)`, `q(1, 2)`, `r(1, 2)`, `r(1, 4)`, `r(2, 3)`, and we can derive `p(1, 2)`, `p(1, 3)`, `p(1, 4)`

The updates give the Δ-sets,
```
Δq = <{(1,2)},{}>
Δr = <{(1,4)},{}>
```
Then `Δp(X, Z)/Δ₊q = <{1,3},{}>`
and `Δp(X, Z)/Δ₊r = <{1,4},{}>`
and joining with $\cup_\Delta$ finally gives
`Δp = <{(1,3),(1,4)},{}>`

The AMOSQL compiler expands as many derived relations as possible to have more degrees of freedom for optimizations. The condition function of our running example will be expanded to:
```
cnd_monitor_items_item(I) ←
   quantity_item,integer(I,_G1) ∧
   consume_freq_item,integer(I,_G2) ∧
   delivery_time_item,supplier,integer(I,_G3,_G4)∧
   supplies_item,supplier(I,_G3) ∧
   _G5 = _G2 * _G4 ∧
   min_stock_item,integer(I,_G6) ∧
   _G7 = _G5 + _G6 ∧
   _G1 < _G7
```

The positive partial differentials based on the influents `quantity` and `consume_freq` are defined as:
```
Δcnd_monitor_items_item(I)/Δ₊quantity ←
   Δ₊quantity_item,integer(I,_G1) ∧
   consume_freq_item,integer(I,_G2) ∧
   delivery_time_item,supplier,integer(I,_G3,_G4)∧
   supplies_item,supplier(I,_G3) ∧
   _G5 = _G2 * _G4 ∧
   min_stock_item,integer(I,_G6) ∧
   _G7 = _G5 + _G6 ∧
   _G1 < _G7

Δcnd_monitor_items_item(I)/Δ₊consume_freq ←
   quantity_item,integer(I,_G1) ∧
   Δ₊consume_freq_item,integer(I,_G2) ∧
   delivery_time_item,supplier,integer(I,_G3,_G4)∧
   supplies_item,supplier(I,_G3) ∧
   _G5 = _G2 * _G4 ∧
   min_stock_item,integer(I,_G6) ∧
   _G7 = _G5 + _G6 ∧
   _G1 < _G7
```

The other differentials `Δcnd_monitor_items/Δ₊delivery_time`, `Δcnd_monitor_items/Δ₊supplies`, and `Δcnd_monitor_items/Δ₊min_stock` are defined likewise. Using these partial differentials we can build a *propagation network* for `cnd_monitor_items`

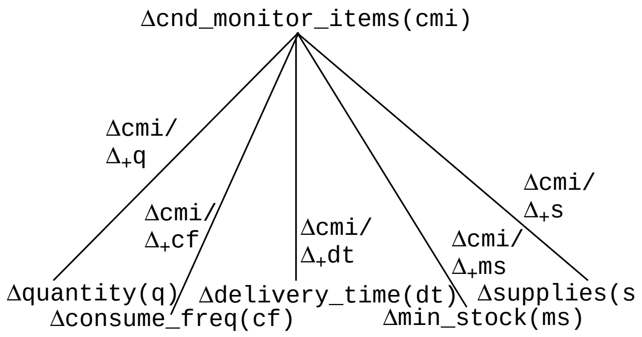

Figure 2: Propagation network of the rule condition

(fig. 2). This is basically the dependency network (fig. 1) augmented with partial differentials. One difference to fig. 1 is that the propagation network for `cnd_monitor_items` is flat since the AMOS query compiler expands functions as much as possible. In the case of *late binding*[1] this is not possible and the result is a more bushy network
In section 7.1 we show how sub-expressions can be reused to produce a more bushy network.

Note that the examples above only deal with conjunctions in the bodies of the Horn Clauses. In ObjectLog disjunctions are introduced in the body only and not as separate Horn Clauses as in traditional Datalog[2]. Disjunctions, i.e. unions, are treated in section 4.5

4.4 Negative Partial Differentials

Often the rule condition depends only on positive changes, as for the `monitor_items` rule. However, for negation and aggregation operators, negative changes must be propagated as well. For strict rule semantics, propagation of negative changes is also necessary for rules whose actions negatively affect other rules' conditions.

In our example in section 4.3 the two partial differentials of the relation P with regard to the negative changes of Q and R are defined as:
```
Δp(X, Z)/Δ₋q ←
        Δ₋q(X, Y) ∧
        r_old(Y, Z)
```
and
```
Δp(X, Z)/Δ₋r ←
        q_old(X, Y) ∧
        Δ₋r(Y, Z)
```

1. Late binding means that some type information can not be determined at compile-time (early binding) and must instead be determined at run-time.
2. In ObjectLog separate Horn Clauses are generated for different AMOSQL functions that are overloaded on the type signatures of a single function name. Since only one function is chosen at run-time, this is not a disjunction.

where $R_{old} = (R \cup \Delta_-R) - \Delta_+R$ and where Q_{old} is defined likewise.

These can be calculated by a logical rollback (fig. 3) or by materialization.

Let DB_{old} consist of the stored relations (facts) `q(1, 1)`, `r(1, 2)`, `r(2, 3)`, from `p` defined above we can now derive `p(1, 2)`. A transaction performs the updates:

```
assert q(1, 2), assert r(1, 4),
retract r(1, 2), retract r(2, 3)
```

DB_{new} is now `q(1, 1)`, `q(1, 2)`, `r(1, 4)`, and we can derive `p(1, 4)`. The updates give the Δ-sets,

```
Δq = <{(1,2)},{}>
Δr = <{(1,4)},{(1,2),(2,3)}>.
```

Then `Δp(X, Z)/Δ+q = <{},{}>`,
`Δp(X, Z)/Δ+r = <{(1,4)},{}>`,
`Δp(X, Z)/Δ-r = <{},{(1,2)}>`,
and joining with $\cup_\Delta$ gives
`Δp = <{(1,4)},{(1,2)}>`.

Note that if we did not use the old state of `q` (`q`$_{old}$) in `Δp(X,Z)/Δ-r` we would get
`Δp = <{(1,4)},{(1,2),(1,3)}>`, which is clearly wrong.

4.5 The Calculus of Partial Differentials

Let Δ_+P be the set of additions (positive changes) to a view P and Δ_-P the set of deletions (negative changes) from P. As before, the Δ-set of P, ΔP, is a pair of the positive and the negative changes of P:

$$\Delta P = <\Delta_+P, \Delta_-P>$$

As for base relations, we formally define the *delta-union*, $\cup_\Delta$, over differentials as:

$$\Delta P_1 \cup_\Delta \Delta P_2 = <(\Delta_+P_1 - \Delta_-P_2) \cup (\Delta_+P_2 - \Delta_-P_1), (\Delta_-P_1 - \Delta_+P_2) \cup (\Delta_-P_2 - \Delta_+P_1)>$$

Next we define the *partial differential*, $\Delta P/\Delta X$, that incrementally monitors changes to P from changes of each influent X. *Partial differencing* of a relation is defined as generating partial differentials for all the influents of the relation. The net changes of the partial differentials are accumulated (using $\cup_\Delta$) into ΔP.

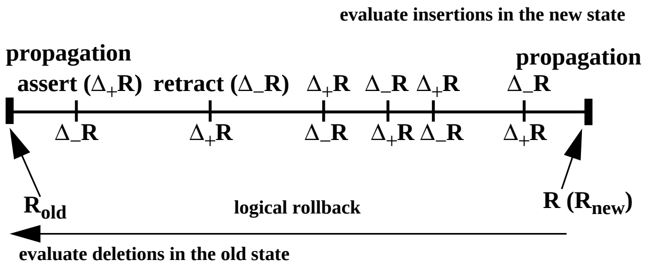

Figure 3: Calculating the old state by a logical rollback

Let I_p be the set of all relations that P depends on. The Δ-set of P, ΔP, is then defined by:

$$\Delta P = \cup_\Delta \frac{\Delta P}{\Delta X} = \cup_\Delta <\frac{\Delta P}{\Delta_+X}, \frac{\Delta P}{\Delta_-X}>, \forall X \in I_p$$

For example, if P depends on the relations Q and R then:

$$\Delta P = \frac{\Delta P}{\Delta Q} \cup_\Delta \frac{\Delta P}{\Delta R} = <\frac{\Delta P}{\Delta_+Q}, \frac{\Delta P}{\Delta_-Q}> \cup_\Delta <\frac{\Delta P}{\Delta_+R}, \frac{\Delta P}{\Delta_-R}>$$

To detect changes of derived relations we define intersection (conjunction), union (disjunction), and complement (negation) in terms of their differentials as:

$$\Delta(Q \cap R) = <(\Delta_+Q \cap R) \cup (Q \cap \Delta_+R), \{\}> \cup_\Delta <\{\}, (\Delta_-Q \cap R_{old}) \cup (Q_{old} \cap \Delta_-R)>$$

$$\Delta(Q \cup R) = <(\Delta_+Q - R_{old}) \cup (\Delta_+R - Q_{old}), \{\}> \cup_\Delta <\{\}, (\Delta_-Q - R) \cup (\Delta_-R - Q)>$$

$$\Delta(\sim Q) = <\Delta_-Q, \Delta_+Q>$$

From the expressions above we can easily generate the simpler expressions in the case of, e.g. insertions only. For example, when only considering insertions, changes to intersections is defined as:

$$\Delta_+(Q \cap R) = <(\Delta_+Q \cap R) \cup (Q \cap \Delta_+R), \{\}>$$

4.6 Partial Differencing of the Relational Operators

The calculus of partial differencing can easily be applied to the relational algebra to incrementally evaluate its operators. This is illustrated by the table in fig. 4. This was generated by separating the expressions above for insertions and deletions and by using the definitions of the relational operators in terms of set operations. See [23] for more details.

P	$\frac{\Delta P}{\Delta_+Q}$	$\frac{\Delta P}{\Delta_+R}$	$\frac{\Delta P}{\Delta_-Q}$	$\frac{\Delta P}{\Delta_-R}$
$\sigma_{cond}Q$	$\sigma_{cond}\Delta_+Q$		$\sigma_{cond}\Delta_-Q$	
$\pi_{attr}Q$	$\pi_{attr}\Delta_+Q$		$\pi_{attr}\Delta_-Q$	
$Q \cup R$	$\Delta_+Q - R_{old}$	$\Delta_+R - Q_{old}$	$\Delta_-Q - R$	$\Delta_-R - Q$
$Q - R$*	$\Delta_+Q - R$	$Q \cap \Delta_-R$	$\Delta_-Q - R_{old}$	$Q_{old} \cap \Delta_+R$
$Q \times R$	$\Delta_+Q \times R$	$Q \times \Delta_+R$	$\Delta_-Q \times R_{old}$	$Q_{old} \times \Delta_-R$
$Q \bowtie R$	$\Delta_+Q \bowtie R$	$Q \bowtie \Delta_+R$	$\Delta_-Q \bowtie R_{old}$	$Q_{old} \bowtie \Delta_-R$
$Q \cap R$	$\Delta_+Q \cap R$	$Q \cap \Delta_+R$	$\Delta_-Q \cap R_{old}$	$Q_{old} \cap \Delta_-R$

Figure 4: Partial differencing of the Relational Operators
*) $Q - R = Q \cap \sim R$

5 The Propagation Algorithm

A breadth-first, bottom-up propagation algorithm has been implemented to support the partial differencing calculus. In the implementation Δ-sets are represented as temporary materializations done in the propagation algorithm and are discarded as the propagation proceeds upwards. Changes, i.e. Δ-sets, which are not referenced by any partial differentials further up in the network are discarded. This assumes that there are no loops in the network, which is not so with *recursive* relations[1]. The algorithm propagates changes breadth-first by first executing all affected partial differentials of an edge and then by accumulating the changes in the nodes above (fig. 5). Here is an outline of the quite simple algorithm (see [23] for more details):

```
for each level (starting with the lowest level)
    for each changed node (a non-empty $\Delta$-set)
        for each edge to an above node
            execute the partial differential(s)
            and accumulate the result in the
            $\Delta$-set of the node above using $\cup_\Delta$
```

The Δ-sets of each node are cleared after the node has been processed, i.e. after the partial differentials that reference the Δ-sets have been executed.

6 Performance Measurements

A performance measurement was performed using two implementations of rule condition evaluation, one based on naive evaluation and another based on partial differencing. The benchmarks were based on monitoring the `monitor_items` rule defined previously and with full expansion of rule conditions. Several benchmarks were run[2] and with encouraging results[23]. We present results from the two most significant of them here. The first one considers few changes per transaction to one partial differential. This is considered the normal case and is shown to be very efficient to monitor using partial differencing. The second one considers many changes to several partial differentials. This is considered a worst case situation, which is more efficient to monitor naively, but which is still monitored with an acceptable efficiency using partial differencing.

6.1 Few Changes to One Partial Differential

In transactions where few updates were performed to monitored conditions, the cost of evaluating conditions using incremental change monitoring was shown to be independent of the size of the database in most cases (fig. 6). In the case of naive change monitoring the cost is linear to the size of the database. In the measurements 100 transactions were run where each transaction only changed the quantity of one item. The test runs were done by using databases populated with between 1 and 10000 items. This causes change to only one partial differential in each transaction in the incremental change monitoring. The reason for this can be seen in fig. 2 where changes to quantities (Δ`quantity`) will be propagated by executing only the partial differential Δ`cnd_monitor_items/` Δ_+`quantity`. By contrast, the naive method goes through all the quantities of all the items in the database.

6.2 Massive Changes to Several Partial Differentials

In this benchmark each transaction changed the quantity, the delivery time, and the consume frequency of all

1. The algorithm can be extended to handle linear recursion by revisiting nodes below and using fixed point techniques. Work on recursion can be found in [12].

2. All measurements were made on a HP9000/710 with 64 Mbyte of main memory and running HP/UX.

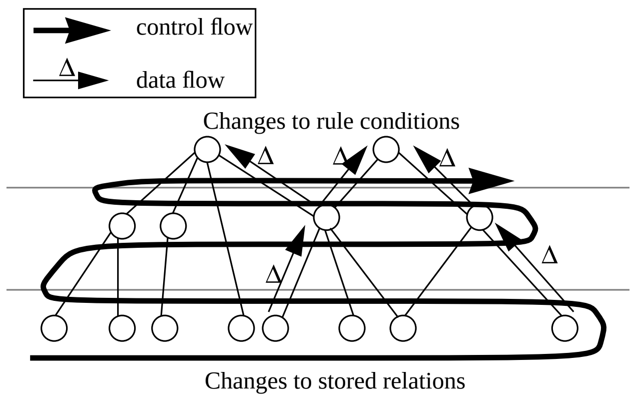

Figure 5: Breadth-first, bottom-up propagation

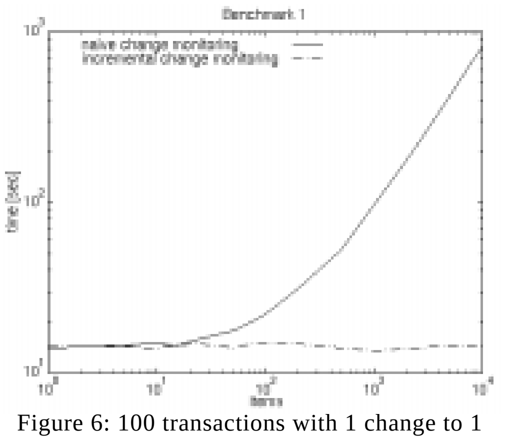

Figure 6: 100 transactions with 1 change to 1 partial differential

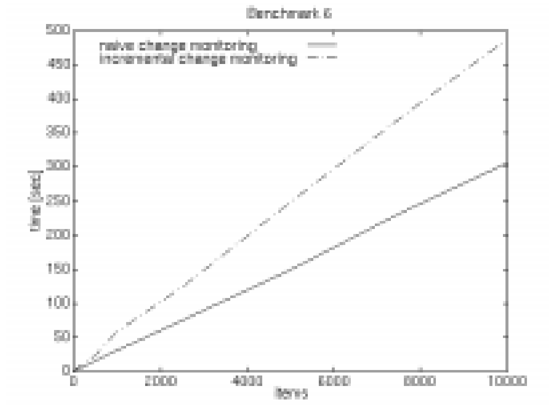

Figure 7: 1 transaction with n changes to 3 partial differentials

items. This caused changes to three out of the five partial differentials in each transaction in the incremental change monitoring. As shown in fig. 2 the partial differentials Δ`cnd_monitor_items`/Δ_+`quantity`, Δ`cnd_monitor_items`/Δ_+`delivery_time`, and Δ`cnd_monitor_items`/Δ_+`consume_freq` will all need to be executed, which results in overlapping execution. In the naive version these overlaps in the execution do not appear. As shown in fig. 6 massive changes to several partial differentials perform worse than naive change monitoring but only with a constant factor of about 1.6.

7 Refinements

There are many refinements that can be done to the implementation.

7.1 Optimizations and Node Sharing

Optimizations such as reusing sub-expressions are possible by restricting the way AMOSQL functions are expanded when being compiled into ObjectLog. There is a trade-off between expansion for better query optimization and node sharing for more efficient change propagation. This is an area for further research.

To get a propagation network analogous to that in fig. 1 we could choose to define `cnd_monitor_items` in terms of two differentials instead:

Δcnd_monitor_items$_{\text{item}}$(I)/Δ_+quantity $\leftarrow$
  Δ_+quantity$_{\text{item,integer}}$(I, _G1) $\wedge$
  threshold$_{\text{item,integer}}$(I, _G2) $\wedge$
  _G1 < _G2

Δcnd_monitor_items$_{\text{item}}$(I)/Δ_+threshold $\leftarrow$
  quantity$_{\text{item,integer}}$(I, _G1) $\wedge$
  Δ_+threshold$_{\text{item,integer}}$(I, _G2) $\wedge$
  _G1 < _G2

The Δ`threshold` function would then be defined in terms of four partial differentials and become an intermediate node in the network. This would be beneficial if the threshold function is referenced in other rule conditions as well since this would enable node sharing.

7.2 Strict and Set-oriented Semantics

Partial differentials that contain selections might produce Δ-sets that are too large. This is acceptable for positive changes and nervous semantics. For strict semantics these tuples have to be removed by checking the old state of the selection. Negative partial differentials might also produce a Δ-set that is too large, i.e. deletions of tuples that are still present in the new state of the database. Unlike for positive changes, this is more serious as it might cause rules not to trigger on positive changes since these have been cancelled by incorrectly propagated negative changes. To avoid this, for negative changes we have to check if the tuple is still present in the new state of the database. If this is not done, the rules might under-react, which is unacceptable.

Note that we assume *set-oriented semantics* since this is the most natural semantics for rule conditions. Partial differencing can be defined for *bag-oriented semantics* as well, but this is outside the scope of this paper. Partial differencing of the relational operators for bag-oriented semantics is not as straight forward as for set-oriented semantics. Some work on differencing where bag-oriented semantics is assumed can be found in [13].

With set-oriented semantics, when there are changes to more than one influent the definitions in fig. 4 might give a set of changes that are too large, i.e. containing duplicates. These will, however, be removed by $\cup_\Delta$. Since $\cup_\Delta$ is not commutative for set-oriented semantics, $\cup_\Delta$ has to be performed in the same order as the changes originally occurred in the transaction. For strict semantics of unions a check is made that positive/negative changes are propagated only if the other part of the union was/is not present.

8 Conclusions

The paper presented a *difference calculus* for incremental evaluation of queries, based on database updates. The calculus defines *partial differentials* of rule conditions as separate queries that each considers changes to a single relation that influences a monitored rule condition. The benefits of using partial differencing include optimization for both time and space, and explainability. The advantage of incremental evaluation in general is the efficiency that comes from the assumption that most transactions only perform small changes to rule conditions. Partial differencing has the additional advantages that only a few (or just one) partial differentials are normally executed in each transaction. The partial differentials are much simpler and more efficient than the combined full differentials, in particular when combining partial differentials for both positive (insertions) and negative (deletions) changes. The calculus also defines how to calculate the old database state without materializing. This is important for saving space when differencing negative changes since the intermediate results can sometimes be very large.

Partial differentials can also be used to discriminate between different reasons why a rule was triggered, i.e. the influents of a rule condition can be traced and different actions can be taken in the rules depending on why they were triggered. In systems based on ECA-rules this is accomplished by defining separate rules for each situation with different event parts, but with the same conditions. This causes code duplication. By giving access to the results of partial differentials in the action part of a CA-rule it is possible perform different actions depending on what has happened.

As was shown in the performance measurements, the partial differencing is not always optimal. For transactions with many updates affecting monitored relations naive evaluation can be more efficient, but only with a constant factor. Further research is needed on detecting situations where naive evaluation should be chosen and how to mix naive and incremental evaluation into the same execution mechanism in a *hybrid* evaluation method. Another interesting research area is the possibility of incremental evaluation of foreign functions through user defined differentials. Other future work includes extending the calculus to handle aggregates and recursion.

9 References

[1] Baralis E., Widom J.: Using Delta Relations to Optimize Condition Evaluation in Active Databases, *RIDS'95(Rules in Database Systems)*, Springer Lecture Notes in Computer Science, pp. 292-308, Athens, Greece, Sept., 1995

[2] Bernstein P.A., Blaustein B.T., and Clarke E.M.: Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data, *VLDB 6*, Oct.1980, pp.126-136.

[3] Blakeley J.A., Larson P-Å., Tompa F.W.: Efficiently Updating Materializing Views, *ACM SIGMOD conf.*, Washington D.C., 1986, pp. 61-71.

[4] Dayal U., McCarthy D., The architecture of an Active Database Management System, *ACM SIGMOD conf.*, 1989, pp. 215-224

[5] Dewan H. M., Ohsie D., Stolfo S. J., Wolfson O., Da Silva S.: Incremental Database Rule Processing in PARADISER, *Journal of Intelligent Information Systems*, 1:2, 1992

[6] Dong G., Su J.: First-Order Incremental Evaluation of Datalog Queries, *Proc. of the Fourth Int'l Workshop on Database Programming Languages - Object Models and Languages, Springer-Verlag*, Aug. 30, 1993, pp. 295-308

[7] Fabret F., Regnier M., Simon E.: An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases, *Proc. 19th VLDB conf.*, Dublin 1993

[8] Fahl G., Risch T., Sköld M.: AMOS - An Architecture for Active Mediators, *Intl. Workshop on Next Generation Information Technologies and Systems (NGITS '93)* Haifa, Israel, June 1993, pp. 47-53

[9] Fishman D. et. al: Overview of the Iris DBMS, *Object-Oriented Concepts, Databases, and Applications*, ACM press, Addison-Wesley Publ. Comp., 1989

[10] Gupta A., Mumick I. S.: Maintenance of Materialized Views: Problems, Techniques and Applications, *IEEE Data Engineering bulletin*, Vol. 18, No. 2, 1995

[11] Hanson E. N.: Rule Condition Testing and Action Execution in Ariel, *ACM SIGMOD conf.*, 1992, pp. 49-58

[12] Harrison J. V., Dietrich S. W.: Condition Monitoring in an Active Deductive Database, Arizona State University, *ASU Technical Report* TR-91-022 (Revised), Dec. 1991

[13] Katiyar A. G. D., Mumick I. S.: Maintaining Views Incrementally, *AT&T Bell Laboratories, Technical Report* 921214-19-TM, Dec. 1992

[14] Koenig S., Paige R.: A Transformational Framework for the Automatic Control of Derived Data. *Proc. VLDB conf.* 1981, pp. 306-318

[15] Litwin W., Risch T.: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering* Vol. 4, No. 6, December 1992

[16] Miranker D. P.: TREAT: A Better Match Algorithm for AI Production Systems, *AAAI 87 Conference on Artificial Intelligence*, Aug. 1987, pp. 42-47

[17] Qian X., Wiederhold G.: Incremental Recomputation of Active Relational Expressions, *IEEE Transactions on Knowledge and Data Engineering* Vol. 3, No. 3 December 1991, pp. 337-341

[18] Risch T.: Monitoring Database Objects, *Proc. VLDB conf.* Amsterdam 1989

[19] Risch T., Sköld M.: Active Rules based on Object Oriented Queries, *IEEE Data Engineering bulletin*, Vol. 15, No. 1-4, Dec. 1992, pp. 27-30

[20] Rosenthal A., Chakravarthy S., Blaustein B., Blakely J.: Situation Monitoring for Active Databases, *VLDB conf.* Amsterdam, 1989

[21] Shipman D. W.: The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems*, 6(1), March 1981

[22] Selinger P., Astrahan M. M., Chamberlin R.A., Lorie R. A., Price T.G.: Access Path Selection in a Relational Database Management System, *ACM SIGMOD conf.*, Boston, MA, June 1979, pp. 23-54

[23] Sköld M.: Active Rules based on Object Relational Queries, - Efficient Change Monitoring Techniques, *Lic. Thesis, Linköping University*, LiU-Tek-Lic 1994:38, Sept. 1994

[24] Widom J., Finkelstein S.J.: Set-oriented production rules in relational database system, *ACM SIGMOD conf.*, Atlantic City, New Jersey 1990, pp. 259-270