Chapter 1

Monitoring Complex Rule Conditions

Tore Risch
Martin Sköld

In N.Paton (ed.): *Active Rules in Database Systems*, Springer-Verlag, New York, 1999, ISBN 0-387-98529-8.

Abstract

This chapter describes and discusses the problem of efficient checking of complex rule conditions expressed as database queries. For this several methods have been proposed that are based on the technique of *incremental evaluation*. With incremental evaluation the state of a rule condition is materialized and, after an update, the new state of the condition is defined incrementally in terms of differences to the materialized state generated by the update. First an overview of the traditional methods for incremental evaluation is given. Then a *partial differencing calculus* is defined for set algebra and is then mapped to the relational operators. Examples are given on how the calculus has been used to define an algorithm that allows trade-offs between space and time efficiency when checking complex rule conditions.

1.1 Introduction

The discussion in this chapter concerns the efficient evaluation of complex rule conditions. The discussion is applicable both to CA and ECA rules with complex rule conditions. Systems based on CA rules (e.g. AI production rule systems) have traditionally used more complex rules than ECA rules of active databases. However, as more advanced active database systems are being developed the need for efficient handling of complex rule conditions will increase for ECA rules as well.

The condition part of a rule is allowed to be more or less complex in different active database systems. If it is expressed as a general database query it can span very large parts of the database. A *naive* method of checking such a rule condition is to execute the complete rule condition whenever an event that triggers the rule has occurred. This, however, can be very costly. For example, a rule attached to update events of the salaries of employees might have a rule condition that specifies that the rule action is executed only when the sum of the incomes of all employees is larger than the salary budget. The execution cost for a query that adds together all employee salaries is proportional to the number of employees in the database. It would be very inefficient if the ADBMS would check the complete query representing the condition every time the salary of an employee is updated.

The example illustrates that for optimization of rule conditions it is not always sufficient to rely solely on conventional query optimization techniques. Special optimization techniques are needed to execute complex rule conditions with reasonable efficiency. Such optimizations can make use of special knowledge about rules. In our example, it is favorable to store the sum of all employee salaries in the database and then incrementally update the sum whenever the salary of some employee is changed. The condition checking only has to check whether the materialized sum is larger than the budget.

The example illustrates the need for an important class of optimization techniques for rule conditions based on *incremental evaluation* of rule conditions. Incremental evaluation avoids recomputing the rule condition completely for every event by incrementally computing the influence of an update event on a materialized part of a condition. In the example, when an income of an employee or manager is updated, we use the incremental difference between the old and the new income to calculate the influence on the sum on the update.

This chapter first makes an overview of some well-known incremental evaluation techniques and how they have been used in databases. In particular we discuss how well suited the various techniques are for monitoring rule conditions in active databases.

To illustrate and formalize the technique a *partial difference calculus* is presented that formally defines incremental changes to rule conditions in an active DBMS. The calculus is based on a form of incremental evaluation named *partial differencing* of rule conditions. The calculus gives us a formalism to optimize rule condition checking with respect to both space and time. Space optimization is achieved since the calculus does not presuppose materialization of all intermediate results of monitored conditions to find its previous state. As an alternative to complete materialization of the rule condition, the calculus provides a method to do a *logical rollback* from the new database state to the old one. Thus by using the calculus, a rule optimizer has the choice of not materializing when favorable.

The calculus has been applied in the implementation of an incremental algorithm that efficiently monitors complex rule conditions [SR96]. To optimize space usage the algorithm uses a *breadth-first, bottom-up* propagation based on the calculus combined with the possibility to do logical rollbacks. Time optimization is achieved by materializing some (but not all) intermediate results and then incrementally computing as little as possible at each update. The algorithm is particularly favorable for database transactions with few updates and where the rule conditions are complex and the rule checking is deferred until the end of transactions (deferred coupling mode). However, the technique can also be used for immediate coupling mode. The algorithm and the calculus are applicable both to CA (production) rules and to ECA rules.

1.2 Incremental evaluation techniques

Finite differencing [PK92] is an incremental evaluation method to calculate changes to functions in terms of changes to its arguments. As a simple example, if we have a function

```
netpay = income - taxes - fees
```

and increase the income with Δ`income` while taxes and fees stay fixed, we get the same change in `netpay`, i.e.:

Δ`netpay` = Δ`income`,

If the old state of `netpay` is materialized the new state can be computed by just incrementing `netpay` with Δ`income`. It is favorable to use such incremental evaluation if it is cheaper to look up the materialization than to compute the subtraction[1]. The example illustrates that it is often (but not always) cheaper to incrementally compute the incremental difference to the value of a function than to fully recompute it.

The example illustrates the *chain rule* [PK92] for the minus function:

If $F = X - Y$ then $\frac{\Delta F}{\Delta X} = \Delta X$ and $\frac{\Delta F}{\Delta Y} = -\Delta Y$, where $\frac{\Delta F}{\Delta X}$ denotes the change to F, given the change ΔX to X and $\frac{\Delta F}{\Delta Y}$ denotes the change to F given the change ΔY to Y. Thus we get the *partial difference* expressions $\Delta F = \frac{\Delta F}{\Delta X} = \Delta X$ if ΔY is empty and $\Delta F = \frac{\Delta F}{\Delta Y} = -\Delta Y$ if ΔX is empty.

Early work on finite differencing was done by Paige and Koenig [PK92] who used the technique for improving the efficiency of programs in the set-oriented programming language SETL by program transformations. In that work differentiation operators were defined for the basic set functions in the

[1]This is actually not the case here, but would be so if the operator had been a more expensive, e.g. a set or an aggregation operator.

SETL language. The transformed programs were faster since results of functions could then be materialized to avoid recomputations of large sets.

Paige and Koenig also discovered that finite differencing could be applied on materialization of derived data in databases [KP81]. In [KP81] finite differencing is used for materializing derived data in a functional data model. Finite differencing for maintaining materialized views in the relational model was first developed in [BLT86] where it was shown how to incrementally maintain materialized relational Select-Project-Join (SPJ) views. Work on incremental maintenance of materialized views in Datalog can be found in [GM95, KM92, DS93]. In [QW91] the relational algebra is extended with some incremental operators that can be used for differencing relational algebra expressions.

[RCBB89] proposed incremental evaluation of relational Select-Project-Join (SPJ) queries in ECA rule conditions. The motivation for this was that ECA rules with complex rule conditions need incremental evaluation techniques for efficient evaluation of the condition part. The work was based on defining an algebra for computations over database changes, Δ-relations. Each relation had an associated Δ-relation where the tuples that got added and deleted during an update operation were stored. Each SPJ-view also had Δ-relations which were computed though a *chain-rule* for SPJ queries. An incremental evaluation algorithm for rule condition monitoring was also proposed by [HD91].

In [CW91] it is shown how active rules can be used for maintaining materialized views. The rules are semi-automatically generated from the user defined views to be materialized. The generated ECA rules are parameterized to allow for a simple form of incremental evaluation.

A classical algorithm for incremental evaluation of rule conditions in AI is the RETE algorithm [For82]. It is used to incrementally evaluate rule conditions (called patterns) in the OPS5 [BFKM85] expert system shell. OPS5 is a forward-chaining production rule system where all patterns are checked using RETE. Thus, in difference to active database systems, all the instantiations of all patterns (i.e. all queries) in the current OPS5 program are incrementally maintained, and regular demand driven database queries are not supported. In RETE the system records each incremental change (insertions or deletions, called tokens) to the stored data. For patterns that reference other patterns (i.e. derived patterns) a *propagation network* is built that incrementally maintains the instances of the derived patterns. The propagation network may contain both selections (represented as *alpha nodes*) and joins (represented as *beta nodes*). The alpha nodes (selections) are always propagated before the beta nodes (joins).

The main problem with using RETE for rule matching in large databases is that RETE is very space inefficient for large databases since RETE saves all intermediate results for all rule conditions. RETE furthermore does not do join optimizations which may result in a combinatorical explosion of the size of the working memory [Mir87]. Its memory usage therefore often becomes substantially larger than the database itself. To improve the performance of RETE the TREAT [Mir87] and A-TREAT [Han92] algorithms were developed. These algorithms are shown to be more efficient for large databases [WH92].

TREAT avoids the combinatorical space explosion by using relational database optimization techniques [Mir87]. A-TREAT further reduces the memory usage by avoiding to materialize some intermediate results by defining some selection nodes in the propagation network as simple relational expressions [Han92] (named *virtual alpha nodes*). A related approach is proposed in [FRS93a] where an algorithm is presented that can take a set of rules and return the set of relational expressions that is most profitable to materialize to support efficient execution of the rules. These are examples of how to trade query execution time for space in rule condition checking.

By contrast relational differencing techniques [RCBB89] transform relational expressions into one or several incremental expressions. The nodes in the propagation network do not reflect hard-coded primitive operations as in alpha and beta nodes, but represent temporary storage of data propagated from the nodes below. The arcs represent variables in these expressions.

In the partial differencing technique described below, the propagation network will contain separate relational expressions associated with each *arc* of the network representing the specific changes coming from each input node. These partial changes are accumulated in the nodes through a special operator, called *delta-union* ($\cup_{\Delta}$). This has the advantage that the partial incremental expressions are simpler and more efficient to optimize and evaluate, in particular for deferred rules where few changes are made in the transaction [SR96, Sko97].

In Heraclitus[GHJ96] a database programming language is proposed that supports incremental evaluation by having deltas, i.e. incremental changes, as first class data types. This allows for different rule semantics to be implemented, but leaves it to the programmer to define how to incrementally evaluate database expressions. Since incremental expressions can be rather complicated it is preferable if the system could automatically generate them, rather than letting the programmer explicitly define them. The Heraclitus approach is very similar to ECA rules, which also can be used to manually maintain materialized views [SJGP90].

1.3 Differencing Relational Expressions

Let P be a relation whose values depend on the values of the relations Q and R which we call the *influents* of the *affected* relation P. Thus the definition of P is defined as some function $P = op(Q, R)$ where $op(x, y)$ is some set (or relational algebra) operator.

In *full differencing* the changes to P is defined in terms of some combination of changes to Q and R that depends on the operator op. Thus $\Delta P = op'(\Delta Q, \Delta R)$, where $op'(x, y)$ is an incremental version of $op(x, y)$. Full differencing of relational algebra was done by [RCBB89, BLT86] and of Datalog by [GM95, KM92, DS93]. The problem with full differencing of relational algebra or Datalog expressions is that the differential operator, $op'(x, y)$ is complex for many expressions such as SPJ joins and aggregation operators. It is therefore difficult to use conventional query optimization techniques to optimize the differentiated expressions. Also we notice that transactions are often small with few updates and therefore it is common that ΔP and ΔQ are not both updated in the same transaction.

Instead of full differencing we define *partial differencing* rules where changes to P are defined in terms of separate changes for each of its influents. Let $\frac{\Delta P}{\Delta Q}$ and $\frac{\Delta P}{\Delta R}$ denote changes to P given changes in Q and R, respectively. ΔP can be expressed as a function of the *partial differential* functions $\frac{\Delta P}{\Delta Q}$ and $\frac{\Delta P}{\Delta R}$. We will define how to automatically derive the partial differentials $\frac{\Delta P}{\Delta Q}$ and $\frac{\Delta P}{\Delta R}$ from the definition of P, and how to calculate the total change ΔP from the partial differentials. If there is an update of R, but not of Q, we can use the partial differential $\frac{\Delta P}{\Delta R}$ to compute $\Delta P = \frac{\Delta P}{\Delta R}(\Delta R)$. Analogous for changes to Q, but not to R, we define ΔP as $\Delta P = \frac{\Delta P}{\Delta Q}(\Delta Q)$. The partial differential operators $\frac{\Delta P}{\Delta Q}$ and $\frac{\Delta P}{\Delta R}$ give much less complicated expressions than the full differential operator op'.

Also notice that the partial differencing calculus itself does not say anything about what intermediate relational expressions (views) are materialized or not; it just tells how to calculate a change of an expression given a change to one of its influents. It is up to the rule compiler to use the calculus and to materialize intermediate results in the best possible way. In some cases the incremental expressions need to refer back to the old value of some intermediate result. If the intermediate result is materialized, this a straight-forward data access. If it is not materialized, the old value can still be computed by a so called *logical rollback* defined in the calculus below.

Partial differencing has the following properties compared to other approaches for incremental evaluation:

- Often the number of updates in a transaction is small and often one or very few tables are updated. Therefore, only one or very few partial differentials are affected and need to be checked in each transaction. Compared to using conventional finite differencing [BLT86, RCBB89] each partial differential becomes a relatively simple database query that can be optimized using traditional cost-based query optimization techniques [SAC$^+$79]. The partial differentials can be automatically generated by a rule compiler. The regular query optimizer is then applied on the partially differenced expression assuming few changes to a single influent. The cost model of the query optimizer should be adapted to support this assumption.

- Insertions are more common than deletions and the calculus for deletions is much more complicated and costly than the calculus for insertions. The partial differentials for handling insertions and deletions do not have the same structure since conditions that depend on deletions are actually historical queries that must be executed in the old database state when the deleted data were present. This makes negative differentials different, more complicated, and not easily mixable with positive ones. We therefore separately define *positive* and *negative* partial differentials, denoted $\frac{\Delta P}{\Delta_+ Q}$

and $\frac{\Delta P}{\Delta_{-} Q}$, respectively.

Based on the calculus, an algorithm has been developed [SR96] for efficient rule condition monitoring. The algorithm propagates incremental changes through a *propagation network* that describes how each monitored condition is defined in terms of its subconditions. For correct and efficient handling of both insertions and deletions in the absence of materializations the algorithm requires a *breadth-first, bottom-up* propagation through the propagation network. The propagation of deletions is performed only when records are deleted since the more complicated calculus for deletions make their partial differencing slower than insertions.

In order to significantly reduce permanent memory utilization the algorithm immediately releases intermediate change materializations as the propagation proceeds upwards in the propagation network. One can regard this as if a wavefront of change materializations that is propagated breadth-first up through the network.

A rule system has been implemented that uses the algorithm for monitoring complex rule conditions [SR96]. The default semantics of our active rules [RS92] 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. ECA-rules have also been implemented [Mac96] and the techniques presented here are applicable for evaluating complex rule conditions of ECA-rules as well; the event part just further restricts when the condition is tested. Set-oriented action execution [WF90] is supported since data instances can be passed from the condition to the action of each rule by using shared query variables. Deferred condition evaluation is supported by delaying the condition checking 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*. Then the action of the rule is executed for each instance where the rule condition is true based on the net changes of the rule condition.

Next we proceed by presenting an example incremental condition monitoring of active rules. The example will illustrate incremental evaluation by showing how our calculus is used to efficiently implement active rules in the AMOS DBMS [SR96].

1.4 Monitoring Active Rule Conditions in AMOS

Active rules have been introduced into AMOS [RS92, FRS93b] (Active Mediators Object System), an Object Relational DBMS. The data model of AMOS is based on the functional data model of Daplex [Shi81] and Iris [FAC^{+}89]. AMOSQL, the query language of AMOS, is a derivative of OSQL [Lyn91]. 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, which are equivalent to classes. Functions can be stored, derived, or foreign. Stored functions equal object attributes or base relational tables, derived functions equal methods or relational views, and foreign functions are functions written in some procedural language[2]. Stored procedures can be defined as functions that have side-effects. AMOSQL extends OSQL with active rules, a richer type system, and multidatabase functionality.

1.4.1 Rules in AMOSQL

The condition in an AMOSQL rule is a query and the action is a procedural expression.

The syntax for the CA rules is as follows:

create rule *rule-name parameter-specification* **as**
  [**for each** *variable-declaration-commalist*]
  when *predicate-expression*
  do *procedure-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*.

Let us define an example database for a factory inventory:

```
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;
```

[2]Foreign functions allow for extending the DBMS with new operations for specific database applications

```
create rule monitor_items() as
        for each item i
        when quantity(i) < threshold(i)
        do order(i,max_stock(i) - quantity(i));
```

The `monitor_items` rule monitors the quantity of all items in stock and orders new items when the quantity of some item 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.

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

```
create item instances :item1, :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();
```

The activated rule will now monitor the items and trigger if the quantity falls below the threshold (i.e. below 140 of `:item1` and below 290 of `:item2`).

1.4.2 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 then regarded as monitoring changes to the condition function [Ris89].

```
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 stored procedure that takes an item as argument and orders new items to fill the inventory.

```
create function act_monitor_items(item i) -> boolean 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`:

```
for each item i in Δcnd_monitor_items() do act_monitor_items(i);
```

We distinguish between *strict* and *nervous* rule execution semantics [SR96]. 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 (i.e. we consider exactly the changes to the condition function since the last time it was checked). With nervous semantics the rule sometimes also 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.

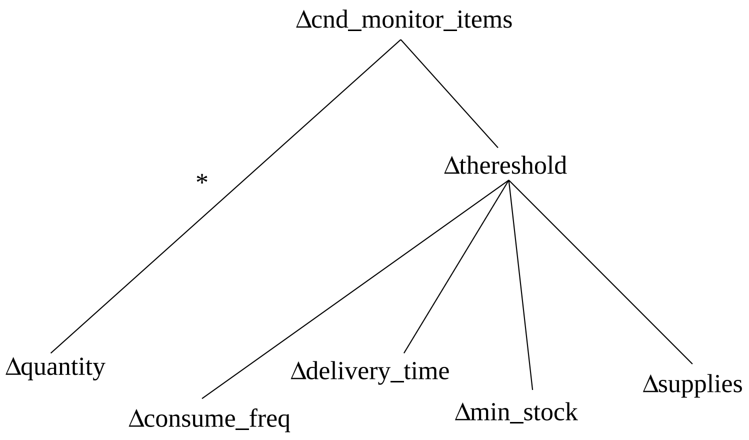

Figure 1.1: Dependency network of the rule condition

By looking at the definition of `cnd_monitor_items` we can define a *dependency network* (fig. 1.1) 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 will later 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 $\frac{\Delta cnd\_monitor\_items}{\Delta quantity}$ (the edge marked * in fig. 1.1). The dependency network is constructed from the definition of the condition function and its sub-functions.

In our system AMOSQL functions are compiled into a domain calculus language called ObjectLog [LR92], which is a variant of Datalog [Ull89] 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 examining the definitions of the functions `cnd_monitor_items` and `threshold`:

$\texttt{cnd\_monitor\_items}_{item}\texttt{(I)} \leftarrow$
$\quad\quad \texttt{quantity}_{item,integer}\texttt{(I,G1)} \wedge$

```
        threshold_item,integer(I,G2) ∧
        G1 < G2

threshold_item,integer(I,T) ←
        consume_freq_item,integer(I,G1) ∧
        delivery_time_item,supplier,integer(I,G2,G3) ∧
        supplies_item,supplier(I,G2) ∧
        G4 = G1 * G3 ∧
        min_stock_item,integer(I,G5) ∧
        T = G4 + G5
```

1.5 Partial Differencing

We first define a *difference calculus* as the theory for incremental computation of changes in set expressions using an extension of set algebra. The calculus is then mapped to relational algebra by defining partial differentials for the basic relational operators. The calculus 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* ($\sim$). 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 the section ends with partial differencing formulae of the relational algebra operators.

Separate *partial differentials* are generated for monitoring insertions and deletions for each influent of a derived relation. Positive partial differentials (insertions) are calculated in the new state of the database, while the negative partial differentials (deletions) are calculated in the old state where the deleted tuples were present in the database. The database updates are made in-place, i.e. the current database state always reflects the new state.

1.5.1 Breadth-first propagation

In some cases the old state, S_{old}, of some Δ-set is needed. This is particularly important when dealing with deletions (see [SR96, Sko97] for details).

The old state of a relation can be 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. To make the logical rollback possible the changes must be propagated in a breadth-first, bottom-up manner through a propagation network where the Δ-sets can be seen as temporary 'wave-front' materializations (fig. 1.2). This is required for the logical rollback since calculating the old state, S_{old}, requires every instance of the propagated changes that influence S, i.e. the complete new $\Delta_+ S$ and $\Delta_- S$ are needed in order to compute the complete S_{old}. Next we define how to accumulate these changes and how to generate partial differentials.

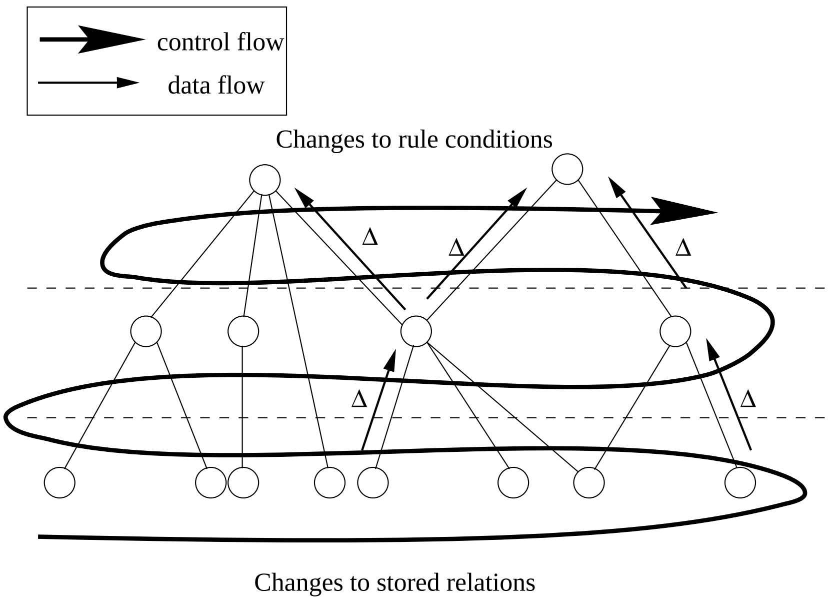

Figure 1.2: Breadth-first, bottom-up propagation

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, i.e. non-recursive functions. The algorithm propagates changes breadth-first by first executing all affected partial differentials of an edge (i.e. stored functions or base relations) and then by accumulating the changes in the nodes above. Here is an outline of the quite simple algorithm (see [Sko97] for more details):

```
for each level (starting with the lowest level)
         for each changed node (a non-empty Δ-set)
               for each edge to an above node
                    execute the partial differential(s)
                    and accumulate the result in the
                    Δ-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.

1.5.2 Differencing Base Relations

All changes to base relations, i.e. stored functions, are logged as *physical events* in an undo/redo log. If there is a change to a base relation, the physical events are accumulated in a Δ-set that reflects all *logical events* of the updated relation. Only those relations that are influents of some rule condition need Δ-sets. Before the physical update events are accumulated, a simple check is made if the base relation that was updated is influencing some activated rule condition. The Δ-sets can be discarded when the changes of the affected relations have been calculated, which saves space compared to other propagation algorithms where all the change data during the complete propagation need to be retained. Since rules are only triggered by net changes the physical events have to be added with the *delta union* operator, $\cup_\Delta$, that cancels counter-acting insertions and deletions in the Δ-set. The Δ-set for a base relation B is defined as:

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

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

$$\begin{aligned} \Delta_+ B = &\quad B - B_{old} \\ \Delta_- B = &\quad B_{old} - B \end{aligned}$$

Therefore it holds that

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

We define $\cup_\Delta$ formally as:

$$\begin{aligned} \Delta B_1 \cup_\Delta \Delta B_2 = &\quad < (\Delta_+ B_1 \cup \Delta_+ B_2) - (\Delta_- B_1 \cup \Delta_- B_2), \\ &\quad (\Delta_- B_1 \cup \Delta_- B_2) - (\Delta_+ B_1 \cup \Delta_+ B_2) > \end{aligned}$$

The $\cup_\Delta$ operator ensures that we only consider the 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:
Δ `min_stock = <{},{<:item1,100>}>`
Δ `min_stock = <{<:item1,150>},{<:item1,100>}>`
Δ `min_stock = <{},{<:item1,100>}>`
Δ `min_stock = <{},{}>`
i.e. there is no net effect of the updates.

1.5.3 Partial Differencing of Views

As for base relations, the Δ-set of a relational view 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.

Positive Partial Differentials

For a view P defined as a Horn Clause with a conjunctive body, let Ip be the set of all its influents. The positive partial differentials $\frac{\Delta P}{\Delta_+ X_i}, X_i \in I_p$ are constructed by substituting X_i in P with its positive differential $\Delta_+ X_i$.

For example, if

$$p(X,Z) \leftarrow q(X,Y) \wedge r(Y,Z)$$

then

$$\frac{\Delta p(X,Z)}{\Delta_+ q} \leftarrow \Delta_+ q(X,Y) \wedge r(Y,Z)$$

and

$$\frac{\Delta p(X,Z)}{\Delta_+ r} \leftarrow q(X,Y) \wedge \Delta_+ r(Y,Z)$$

If the old database state consists 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)`

The new state of the database 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,
$\Delta q = <\{\texttt{<1,2>}\},\{\}>$
$\Delta r = <\{\texttt{<1,4>}\},\{\}>$
Then
$\frac{\Delta p(X,Z)}{\Delta_+ q} = <\{\texttt{<1,3>}\},\{\}>$
and
$\frac{\Delta p(X,Z)}{\Delta_+ r} = <\{\texttt{<1,4>}\},\{\}>$
and joining with $\cup_\Delta$ finally gives
$\Delta p = <\{\texttt{<1,3>,<1,4>}\},\{\}>$

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:

$\texttt{cnd\_monitor\_items}_{item}\texttt{(I)} \leftarrow$
$\quad\texttt{quantity}_{item,integer}\texttt{(I,G1)} \wedge$
$\quad\texttt{consume\_freq}_{item,integer}\texttt{(I,G2)} \wedge$
$\quad\texttt{delivery\_time}_{item,supplier,integer}\texttt{(I,G3,G4)} \wedge$
$\quad\texttt{supplies}_{item,supplier}\texttt{(I,G3)} \wedge$
$\quad\texttt{G5 = G2 * G4} \wedge$
$\quad\texttt{min\_stock}_{item,integer}\texttt{(I,G6)} \wedge$
$\quad\texttt{G7 = G5 + G6} \wedge$
$\quad\texttt{G1} < \texttt{G7}$

The positive partial differential based on the influent `quantity` is defined as:

$\Delta\texttt{cnd\_monitor\_items}_{item}\texttt{(I)/}\Delta_+\texttt{quantity} \leftarrow$
$\quad\Delta_+\texttt{quantity}_{item,integer}\texttt{(I,G1)} \wedge$
$\quad\texttt{consume\_freq}_{item,integer}\texttt{(I,G2)} \wedge$
$\quad\texttt{delivery\_time}_{item,supplier,integer}\texttt{(I,G3,G4)} \wedge$
$\quad\texttt{supplies}_{item,supplier}\texttt{(I,G3)} \wedge$
$\quad\texttt{G5 = G2 * G4} \wedge$
$\quad\texttt{min\_stock}_{item,integer}\texttt{(I,G6)} \wedge$

```
G7 = G5 + G6 ∧
G1 < G7
```

The other differentials Δ`cnd_monitor_items/`Δ_+`consume_freq`, Δ`cnd_monitor_items/`Δ_+`delivery_time`, Δ`cnd_monitor_items/`Δ_+`supplies`, and Δ`cnd_monitor_items/`Δ_+`min_stock` are defined likewise. Using these partial differentials we can build a *propagation network* for `cnd_monitor_items` by associating the partial differentials with the arcs of flattened version of the dependency network in Figure 1.1.

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*[3] [FR95] this is not possible and the result is a more bushy network. Bushy networks are sometimes preferable since they can promote node sharing between nodes shared by different rule conditions.

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. See [SR96] for details.

Partial Differentials of Intersection, Union, and Set-complement

Let Δ_+P be the set of additions (positive changes) to a view P and Δ_-P be 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:

$$\begin{aligned}\Delta P_1 \cup_\Delta \Delta P_2 = \quad & < (\Delta_+P_1 \cup \Delta_+P_2) - (\Delta_-P_1 \cup \Delta_-P_2), \\ & (\Delta_-P_1 \cup \Delta_-P_2) - (\Delta_+P_1 \cup \Delta_+P_2) >\end{aligned}$$

Next we define the partial differential, $\frac{\Delta P}{\Delta X}$, 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.

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

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

[3]Late binding means that some type information can not be determined at compile-time(early binding) and must instead be determined at run-time.

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:

$$\begin{aligned} \Delta(Q \cap R) = \quad & < (\Delta_+ Q \cap R) \cup (Q \cap \Delta_+ R), \{\} > \\ & \cup_\Delta \\ & < \{\}, (\Delta_- Q \cap R_{old}) \cup (Q_{old} \cap \Delta_- R > \\ \\ \Delta(Q \cup R) = \quad & < (\Delta_+ Q - R_{old}) \cup (\Delta_+ R - Q_{old}), \{\} > \\ & \cup_\Delta \\ & < \{\}, (\Delta_- Q - R) \cup (\Delta_- R - Q) > \\ \\ \Delta(\sim Q) = \quad & < \Delta_- Q, \Delta_+ Q > \end{aligned}$$

Note that for unions any overlaps between the added (removed) tuples and the old state (new state) of the other part of the union are removed. 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 are defined as:

$$\Delta_+(Q \cap R) = \Delta_+ Q \cap R) \cup (Q \cap \Delta_+ R)$$

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 in table 1. 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. Note the table assumes set-oriented semantics and that $Q - R$ can be rewritten as $Q \cap (\sim R)$. See [Sko97] for more details.

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

Table 1. Partial differencing of the Relational Operators

1.6 Summary

This chapter presented incremental evaluation techniques for efficient monitoring of complex rule conditions. An overview of incremental evaluation techniques was given. A difference calculus was presented 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 advantage of incremental evaluation in general is the efficiency that comes from the assumption that most transactions only perform small changes to rule conditions and it is therefore cheaper to incrementally change a materialized rule condition than to recompute it in every transaction. 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. A breadth-first, bottom-up propagation algorithm is used where changes can be discarded as the propagation proceeds upwards in the propagation network. This propagation algorithm is fast, space efficient, and supports logical rollbacks.

Bibliography

[BFKM85] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5*. Adison-Wesley, 1985.

[BLT86] J.A. Blakely, P-Å. Larson, and F.W. Tompa. Efficiently updating materialized views. In *SIGMOD Conf.*, pages 61–71, 1986.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf on Very Large Data Bases*, pages 577–589. Morgan Kaufmann, 1991.

[DS93] G. Dong and J. Su. First-order incremental evaluation of datalog queries. In *4th Int'l Workshop on Database Programming Languages*, pages 295–308, 1993.

[FAC$^+$89] D. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Chan, and W.K. Wilkinson. Overview of the iris dbms. In W.Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219–250. ACM Press, 1989.

[For82] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[FR95] S. Flodin and T. Risch. Processing object-oriented queries with invertible late bound functions. In *21st Int. Conf. on Very Large Databases (VLDB'95)*, pages 335–344, 1995.

[FRS93a] F. Fabret, M. Regnier, and E. Simon. An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases. In R. Agrawal, S. Baker, and D. Bell, editors, *19th Intl. Conf. on Very Large Data Bases*, pages 455–466. Morgan Kaufmann, 1993.

[FRS93b] G. Fahl, T. Risch, and M. Sköld. Amos - an architecture for active mediators. In *Intl. Workshop on Next Generation Information Technologies and Systems (NGITS'93)*, pages 47–53, 1993.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Transactions on Database Systems*, 21(3):370–426, 9 1996.

[GM95] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering*, 18(2), 1995.

[Han92] E.N. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proc. SIGMOD*, pages 49–58. ACM, 1992.

[HD91] J. D. Harrison and S.W. Dietrich. Condition monitoring in an active deductive database. Technical Report TR-91-022, Arizona State University, 12 1991.

[KM92] A.G.D. Katiyar and I.S. Mumick. Maintaining views incrementally. Technical Report TR-91-022, AT&T Bell Laboratories, 1992.

[KP81] S. Koenig and R. Paige. A transformational framework for the automatic control of derived data. In *Proc. 7th Intl. Conf. on Very Large Data Bases*, pages 306–318. IEEE, 1981.

[LR92] W. Litwin and T. Risch. Main memory oriented optimization of oo queries using types datalog with foreign predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6), 12 1992.

[Lyn91] P. Lyngbaek. Osql: A language for object databases. Technical Report HPL-DTD-91-4, Hewlett-Packard Laboratories, 1 1991.

[Mac96] S-A. Machani. Events in an object relational database system. Technical Report LiTH-IDA-Ex-9634, University of Linköping, 1996.

[Mir87] D.P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proc. AAAI*, pages 42–47, 1987.

[PK92] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(2):402–454, 1992.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

[RCBB89] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation monitoring for active databases. In M.G.Apers and G.Wiederhold, editors, *Proc. 15th Intl. Conf. on Very Large Data Bases*, pages 455–464, 1989.

[Ris89] T. Risch. Monitoring database objects. In P.M.G. Apers and G. Wiederhold, editors, *Proc. 15th Intl. Conf. on Very Large Databases*, pages 445–453, 8 1989.

[RS92] T. Risch and M. Sköld. Active rules based on object oriented queries. *IEEE Quarterly Bulletin on Data Engineering, Special Issue on Active Databases*, 1992.

[SAC$^+$79] P. Selinger, M.M. Astrahan, R.A. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *SIGMOD Conf.*, pages 23–54. ACM, 1979.

[Shi81] D.W. Shipman. The functional data model and the data language daplex. *ACM Transactions on Database Systems*, 6(1), 3 1981.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD*, pages 281–290, 1990.

[Sko97] M. Skold. *Active Database Management Systems for Monitoring and Control.* Number Dissertation No. 494. Linköping University, 9 1997.

[SR96] M. Sköld and T. Risch. Using partial differencing for efficient monitoring of deferred complex rule conditions. In Stanley Y.W.Su, editor, *Proc. 12th Int. Conf. on Data Engineering*, pages 392–401. IEEE Computer Society Press, 1996.

[Ull89] J.D Ullman. *Principles of Database and Knowledge-Base Systems, Volume I & II.* Computer Science Press, 1989.

[WF90] J. Widom and S.J. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1990.

[WH92] Y-W Wang and E.N. Hanson. A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions. In *Proc. Data Engineering*, pages 88–97. IEEE, 1992.