Situation Monitoring for Active Databases†

Arnon Rosenthal[1], Sharma Chakravarthy[1], Barbara Blaustein[2]
Xerox Advanced Information Technology
Email: <lastname>@xait.xerox.com

José Blakeley
Texas Instruments, Information Technologies Laboratory
MS 238, P.O. Box 655474, Dallas, TX 75265 Email: blakeley@csc.ti.com

Abstract

We present a basis for efficiently evaluating the situation (event and condition) portion of situation/action rules, either in an active database or in a standalone situation monitor. A common framework handles situations involving both database changes and nondatabase situations. We introduce ΔRelations to represent net changes to a stored or derived relation. We define an operator that computes ΔRelations for derived relations. Evaluation of expressions involving changes is optimized by defining incremental forms of relational operators and by providing a chain rule that extends incremental computation to data derived by arbitrary expressions.

1. Introduction

An *active* Database Management System monitors *situations* triggered by *events* representing database updates or occurrences external to the database. This paper is concerned with ways of specifying situations and evaluating them efficiently. The techniques described in this paper were developed as part of the HiPAC (High Performance Active) DBMS, a prototype active DBMS [DAYA88a, DAYA88b, CHAK89], parts of which have been implemented. However, the algebra and transformations described in this paper can be applied more generally. The situation evaluation mechanism can either be a component integrated tightly with a DBMS, or coupled with a heterogeneous array of applications or databases that signal it when specified events occur.

We use the relational model to specify situations and operations — all data objects will be relations, and all operators will map relations to relations. The relational model is not essential to our approach, but it simplifies the representation of database changes, allows use of the relational algebra, and avoids the need to explain the semantics of a particular object model.

1.1 Expressing Events and Situations

The problem addressed in this paper is the expression and evaluation of a single situation. A *situation* is an *event/condition* pair. An event may be a combination of more primitive pre-defined events. Informally, a situation describes a logical condition to be evaluated when one or more of a set of pre-defined events occur. Each situation is associated with a set of actions to be taken depending on the result of evaluating the condition.

An *event* is typed and associated with a relation scheme that describes certain data relevant to the event. In Example 1.1, the event of checking the status of a network link is related to a relation scheme including link identification, occurence, and severity information. An occurrence of an event is reported in a *signal* — a message that includes a *signal relation* (conforming to the event's associated relation scheme). The signal relation contains information about this particular event occurrence, and

[1]Address: 4 Cambridge Center, Cambridge MA 02142
[2]Address: 1800 Diagonal Road, Alexandria VA 22314

†This work was supported by the Defense Advanced Research Projects Agency and by Rome Air Development Center under contract No. F30602-87-C-0029. The views and conclusions contained in this paper are those of the authors and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Rome Air Development Center, or the U.S. Government.

may also include some information obtained from the database at the time of the event. We require that signal relations be nonempty. (If an event does not generate data, a dummy value is passed.)

The *condition* part of a situation is a relational expression whose inputs are: 1)signal relations from events in the situation's event set, and 2)zero or more database relations. The condition is evaluated when one or more events in the situation's event set have occurred. The situation *fires* (i.e., the action is invoked) if the condition result is a nonempty relation; the condition result is made available to the action.

1.2 The Evaluator

The situation evaluation component of the system, the *evaluator*, is kept simple by limiting its concerns. We depend on *event detectors* to detect events and send signals. (Any piece of software/hardware that sends event signals is an event detector.) The signals are collected and sent to the evaluator. The evaluator produces a signal that causes invocation of the appropriate action(s). The signal relation of this signal may be used by the action(s).

The major data structure used by the evaluator is a directed acyclic *signal graph* (shortened here to *graph*), that represents a situation. This is an operator graph, in which nodes represent relations: a leaf is bound either to an event (and its signal relation) or to a database relation. To aid in execution, nodes and edges on paths upward from event leaves are called *active*; the remainder are called *passive*. The *value* associated with each node is defined (though not necessarily computed) by the usual operator graph rule of bottom-up execution.

Because situation evaluation imposes overhead on every database update and other primitive event, it must be very efficient. The challenge is that the number of situations may be large, and some may require disk accesses or extensive calculations.

We show two situations that might be defined for a network-control application. We assume that a status-checking program examines the state of links, and occasionally sends a signal "Link_Status", listing all interesting test results. Note that this situation involves an event that does not necessarily do a database update, and that the condition uses an ordinary relational operator.

Example 1.1: **Severe Failures**

```
Event: Link_Status with relation scheme
         [Link#, Occurrence_Type, Severity]
```

Condition: $\texttt{Select}_{(\text{severity}>5)}$ (Link_Status)

For the signal relation {(link1, occ_a, 1), (link2, occ_b,6)}, the condition evaluates to {(link2, occ_b, 6)}. The situation fires, and makes this result available to the action.

For the next example, assume that the database contains a relation Link_Technology[L#, Mode] that describes the implementation of each link. The join query below combines the incoming signal with database information.

Example 1.2: **Severe Failures on Microwave Links.**

```
Event: Link_Status with relation scheme
         [Link#, Occurrence_Type, Severity]
```

Condition:
Join[$\texttt{Select}_{(\text{Severity}>5)}$(Link_Status),
$\texttt{Select}_{(\text{Mode}=\text{'microwave'})}$(Link_Technology),
Link_Status.Link# = Link_Technology.L#]

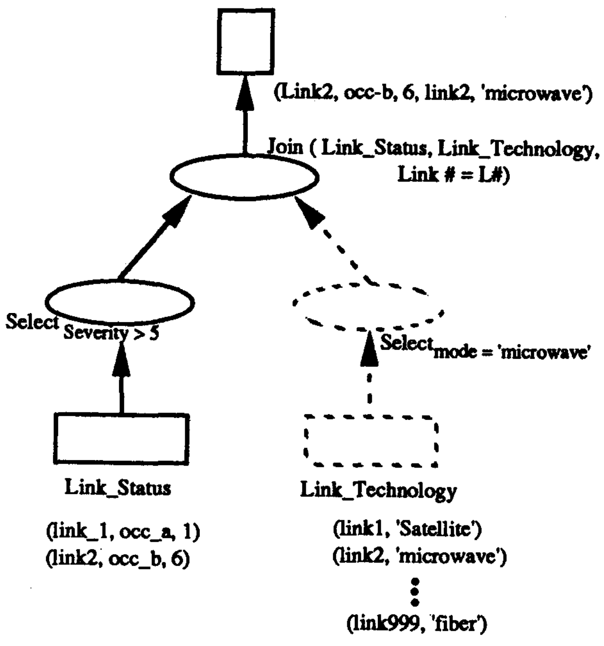

Figure 1: Signal Graph for Example 1.2

Figure 1 shows the signal graph for Example 1.2. The figure also shows the node values that would be computed during graph execution. Note that values need not be computed for all the passive nodes (shown with dashed lines). For example, if Link_Technology is a large relation, it is not advisable to explicitly evaluate $\texttt{Select}_{\text{Mode='microwave'}}$ (Link_Technology). Note also that if the output of $\texttt{Select}_{(\text{Severity}>5)}$ (Link_Status) had been empty, it would be unnecessary to examine any node above it.

Section 2 describes operators that express situations involving database changes. Section 3 discusses optimization techniques for such situations, and general techniques for optimizing signal graphs. Section 4 discusses related work and presents our conclusions.

2. Database Changes

We now define a relation, called a Δ*Relation*, that can represent changes to another relation, and operators for manipulating such relations. We look at the *net* additions, deletions, and modifications to the relation instance. We do not look at individual operations. The goal is a unified treatment of changes, not separate treatments of insertions, deletions, and modifications resulting in several algorithms for generating and combining individual results. Section 2.1 describes conventions satisfied by Δrelations that represent changes, and gives some basic manipulation operators. Section 2.2 discusses an operator that examines updates to some of the objects underlying a derived relation (e.g., view), and determines the corresponding change to a derived relation. Section 2.3 describes efficient ways of implementing this operator, both for familiar relational operators and for arbitrary relational expressions.

Let **R** denote a *relation scheme* specifying a set of attributes and let R denote a *relation* (or, more precisely, a relation instance of **R**) consisting of a set of tuples whose values are taken from the domains of the set of attributes of **R**. *t.A* denotes the value of attribute A for tuple t. Each tuple has an attribute (denoted *tid*) that provides a unique immutable identifier. We use relational operators \m Select, Project (which discards tuples that are entirely null), Join, Outerjoin, Difference, and Union. The unique tid makes it easier to connect tuples that hold values of the same object before and after changes; we will assume that every updatable relation includes the tid column that provides a unique immutable identifier. For readability, pairs of parentheses may be denoted either as "(...)" or as "[...]".

2.1 Representation and Basic Manipulations

To provide a single object that captures an arbitrary change to a relation, we introduce the concept of a Δ*Relation*. For each relation scheme **R** ≡ (tid, A_1,. . . . A_n), define scheme **ΔR** to be (˜tid, ˜A_1,. . . ,˜A_n, tid˜, A_1˜,. . . . , A_n˜). When a ΔRelation represents changes, attribute names with a ˜-suffix (e.g., A˜) refer to new attribute values, and attribute names with a ˜-prefix (e.g., ˜A) refer to old attribute values. In any tuple t, if both t.tid˜ and t.˜tid are non-null, they must be equal; if t.tid˜ is null, so are all t.A_i˜; if t.˜tid is null, so are all t.˜A_i, $1 \le i \le n$; ˜t.tid and t˜.tid cannot both be null; no tid can appear in multiple rows (even in different columns).

ΔRelations can represent tuples (or objects) where only the tid field is nonnull. $\texttt{Join}_{\text{tid}}$ is used to indicate a join that uses predicate [˜tid=tid˜].

Example 2.1 shows a ΔRelation that expresses changes due to a transaction. A ΔRelation can be partitioned horizontally, i.e., expressed as the disjoint union of insertions, deletions, and modifications. (For insertions, the ˜-prefix attributes are null; for deletions the ˜-suffix attributes are null).

Example 2.1: Consider the Employee relation EMP and the transaction T:

EMP	tid	Name	Salary
	@123	Joe	30k
	@456	Lynn	40k
	@321	Ann	28k

Transaction T:

```
{Modify(@123, new_tuple = (@123, Joe, 33k));
 Insert   (@789,  Ed,   25k);
 Delete  (@456)}
```

The following ΔRelation (here called ΔEMP) captures the changes that T makes to EMP:

˜tid	˜Name	˜Salary	tid˜	Name˜	Salary˜
@123	Joe	30k	@123	Joe	33k
-	-	-	@789	Ed	25k
@456	Lynn	40k	-	-	-

To express conditions that involve changes, we provide several operators that manipulate ΔRelations. Here we describe some basic operators; the next section uses them to define a more powerful construct.

We first define renaming functions that add or delete tildes from attribute names in a relation scheme. Let **R** = {tid, A_1, A_2, ..., A_n}. Then **pretilde(R)** = {˜tid, ˜A_1, ˜A_2, ..., ˜A_n}, and **postilde(R)** = {tid˜, A_1˜, A_2˜, ..., A_n˜}. So Δ**R** = **pretilde(R)** ∪ **postilde(R)**.

We define analogous renaming functions on relation instances. If R is a relation, pretilde(R) is the relation with the same tuples, but with all attribute names preceded by a single tilde (and similarly for postilde(R)). For relations where all attributes have pretildes, or all have postildes, untilde removes tildes from all attribute names.

The next set of operators project the old or new tuple values from a ΔRelation. Some of the operators are illustrated in Example 2.2. **Removals˜** takes a ΔRelation and returns the relation consisting of tuples removed by either deletion or modification. **Removals** returns the corresponding relation without the ˜-prefix on attribute names. **Additions˜** returns the relation consisting of tuples added by either insertion or modification, and **Additions** returns the corresponding relation without ˜-suffix. Formally,

Removals˜(ΔR) ≡ **Project**[ΔR, **pretilde(R)**]
Removals(ΔR) ≡ **untilde**[**Removals˜**(ΔR)]
Additions˜(ΔR) ≡ **Project**[ΔR, **postilde(R)**]
Additions(ΔR) ≡ **untilde**[**Additions˜**(ΔR)]

Example 2.2

Removals˜(ΔEMP)

˜tid	˜Name	˜Salary
@123	Joe	30k
@456	Lynn	40k

Additions(ΔEMP)

tid	Name	Salary
@123	Joe	33k
@789	Ed	25k

The main use of ΔRelations is to represent the net effect of a collection of updates to a relation. In such cases, if the "before value" is the relation R, the updates will be represented as ΔR, and the updated relation will be denoted R'. We now define operators that relate these three values.

Definition: The pair (R, ΔR) is *composable* if:
(1) The relation scheme for ΔR is **pretilde(R)** ∪ **postilde(R)**;
(2) **Removals**(ΔR) ⊆ R, ; and
(3) **Additions**(ΔR) ∩ R = ∅.

Composability requires that the ΔRelation contain only the net updates. The restrictions can be loosened somewhat, at the cost of some extra complications for **Compose** and other operators. If the restrictions are sufficiently loosened, a ΔRelation will resemble an undo/redo log.

Definition: Suppose the pair (R, ΔR) is composable. Then define
Compose(R, ΔR) ≡ (R - **Removals**(ΔR)) ∪ **Additions**(ΔR).
The result will often be denoted R'.

Example 2.3: **Given EMP and ΔEMP as in Example 2.1:**

EMP' ≡ **Compose**(EMP, ΔEMP)

EMP'	tid	Name	Salary
	@123	Joe	33k
	@789	Ed	25k
	@321	Ann	28k

Definition: **Δdifference** takes any pair (R_1, R_2) of relations on the same scheme, and produces a ΔRelation that describes their differences. Null values are used to pad tuples that appear in only R_1 - R_2 or R_2 - R_1. It can be expressed as the Outerjoin of tuples in R_1 - R_2 (supplying the ˜-prefix attributes) and R_2 - R_1 (supplying the ˜-suffix attributes). That is,
Δdifference(R_1,R_2) ≡
Outerjoin$_{tid}$[pretilde(R_1 - R_2), postilde(R_2-R_1)]

The following lemmas show some of the algebraic properties of these operators.

Lemma 1A:
ΔR= **Outerjoin**$_{tid}$[**Removals˜**(ΔR), **Additions˜**(ΔR)]

Lemma 1B: For any relations R_1 and R_2 having the same scheme, **Compose**[R_1, **Δdifference**(R_1, R_2)] = R_2

2.2 The Changes Operator

We now introduce a new high-level operator, called **Changes**, to express how a derived relation (e.g., view) changes when at least one of its input relations changes. Let E denote an algebraic expression that defines a derived relation, and let **E** denote the scheme of the relation produced by E. The output of the **Changes** operator is a ΔRelation with scheme Δ**E**. **Changes** can be useful in writing situations, and is easy for a compiler to recognize and optimize.

Notation: Let $E(R_1, \ldots, R_n)$ denote an expression defined on relation schemes $\mathbf{R_1}\ldots\mathbf{R_n}$; let each R_i denote an arbitrary relation; let ΔR_i denote a relation that is composable with R_i, and let R'_i denote Compose(R_i, ΔR_i). A pair $L = \{(R_i, 1 \le i \le n), (\Delta R_i, 1 \le i \le n\}$ is called a *substitution list*.

Definition:
Changes(E; L) $\equiv$ Δdifference($E(R_1,\ldots,R_n)$, $E(R'_1,\ldots,R'_n)$)

Lemma 2: Let ΔR be composable with R. Then
Changes[R; (R, ΔR)] = ΔR

The following lemma states that **Changes**(E; L) correctly computes the changes to the relation derived by E.

Lemma 3:
$E(R'_1,\ldots,R'_n)$=Compose($E(R_1,\ldots,R_n)$, **Changes**(E; L))

Example 2.4 **High-Salaried Employees**
Consider the EMP and ΔEMP relations in Example 2.1. To monitor changes to the view High_Salary_EMP $\equiv$ Select$_{(\mathrm{Salary}>32k)}$(EMP), define the situation:

Event: **Update to EMP**
Condition: **Changes(High_Salary_EMP; [EMP, ΔEMP])**

Substituting the definition of High_Salary_EMP, the condition becomes **Changes(Select$_{(\mathrm{Salary}>32k)}$; [EMP, ΔEMP])**. The result of evaluating this condition for the updates of Example 2.1 is given by the relation

ˉtid	ˉName	ˉSalary	tidˉ	Nameˉ	Salaryˉ
-	-	-	@123	Joe	33k
@456	Lynn	40k	-	-	-

If the user wants to monitor all changes to a view, there is no need to specify explicitly which base-relations' updates should be monitored — in fact, encapsulation is improved if the system rather than the user fills in that list. The explicit form allows specification of a situation that monitors changes caused only by certain kinds of events (e.g., an abstract event Hire_Employee).

3. Optimizations

Efficiency is critical to situation monitoring, but evaluation of **Changes** directly from its definition is very inefficient, requiring retrieval or materialization of R_i and R'_i, for all i. This section describes three ways in which the evaluator's performance can be improved. The first two subsections define transformations on expressions involving the **Changes** operator. Section 3.1 presents efficient "incremental" implementations of **Changes**(F; L) where F is **Select**, **Project**, or **Join**; Section 3.2 treats the case where F is an arbitrary expression. It describes a *chain rule* that obtains an incremental form of F by composing incremental forms of the operators within F. Section 3.3 discusses issues involved in determining execution strategies for signal graphs.

3.1 Incremental Operators

For many operations F, **Changes**(F; L) can be computed more efficiently than by evaluating the definition for **Changes**. These computations use ΔRelations heavily (and sometimes exclusively), instead of computing on database relations, which tend to be much larger.

We study instantiations of **Changes**(F; L) for particular operations F. Define the *incremental form* for F as the operator **IncrF**(substitution list) --> (ΔRelation) where **IncrF**(L) $\equiv$ **Changes**(F; L)

If the optimizer has been informed of an efficient implementation for **IncrF**, it will use that in place of Changes(F; L). Many authors (e.g., [KOEN81, BLAK86]) have defined efficient incremental forms of **Select**, **Project**, and **Join** to deal with insertions or with deletions or with modifications. Here we show efficient implementations of **IncrSelect**, **IncrProject**, and **IncrJoin** that handle an entire ΔRelation as a unit.

An extensible active DBMS would allow users to define new incremental forms. For example, the definer of a new function could be invited to provide an incremental form, which the DBMS would then register with the optimizer.

Incremental Select: Let pred denote a predicate defined on **R**, and $\texttt{Select}_{\text{pred}}$ denote the associated selection operator. Let ˜pred and pred˜ denote the predicates obtained from pred by replacing each attribute by its ˜-prefix (respectively, ˜-suffix) form.

The following formulas give an implementation of IncrSelect that does not reference the base relation. Let $\Delta S \equiv \texttt{IncrSelect}_{\text{pred}}(R, \Delta R))$.

$S1 = \texttt{Select}_{\tilde{}\text{pred}}(\texttt{Removals}\tilde{}(\Delta R))$
$S2 = \texttt{Select}_{\text{pred}\tilde{}}(\texttt{Additions}\tilde{}(\Delta R))$
$\Delta S = \texttt{Outerjoin}_{\text{tid}}(S1, S2)$ /* by lemma 1A */

Note that S1 and S2 represent $\texttt{Removals}\tilde{}(\Delta S)$ and $\texttt{Additions}\tilde{}(\Delta S)$, respectively. The expression for **IncrSelect** can be implemented by one pass over ΔR, determining for each tuple of ΔR whether it induces an insertion, deletion, or modification to Select(R). The proof that the above formulas are indeed implementations of $\textbf{Changes}(\textbf{Select}_{\text{predicate}}$ (R): [R, ΔR]) appears in the appendix.

The optimizer transforms signal graphs, replacing **Changes** by efficient incremental forms. Figure 2a shows the initial signal graph, and Figure 2b shows the final signal graph, for **Changes**(High_Salary_Emp: [Emp, ΔEMP]) from Example 2.4.

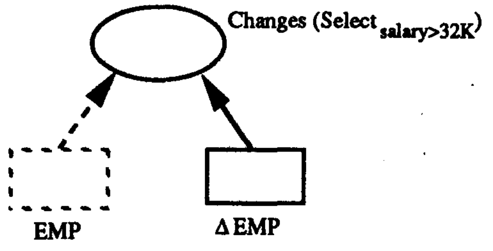

Figure 2a: Initial Signal Graph of Situation in Example 2.4

Incremental Project. Let $A = \{A_1,...A_k\}$ denote a subset of the attributes in **R**, where tid ∈ A. IncrProj(ΔR, A) ≡ **Changes**[Project(R, A): (R, ΔR)]. Let the predicate "differ" state that for some attribute A_i in A, $t.\tilde{}A_i$ differs from $t.A\tilde{}_i$. Then an efficient implementation of **IncrProj** is to select tuples for which at least one attribute has changed, i.e.,

$\texttt{IncrProj}(\Delta R, A) = \texttt{Select}_{\text{differ}}[\texttt{Project}(\Delta R, \tilde{}A \cup A\tilde{})]$

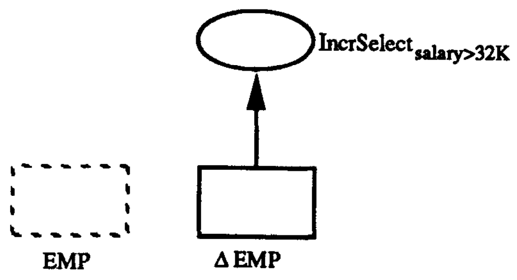

Figure 2b: Optimized Signal Graph of Situation in Example 2.4

Incremental Join. In the case of join, changes to the resulting relation can be induced by changes to either of the individual operands or to both of them. First, suppose that just R_1 changes.

Let join_pred denote a predicate on $\mathbf{R_1}$ and $\mathbf{R_2}$; and let $\text{tid}_1 \in \mathbf{R_1}$ and $\text{tid}_2 \in \mathbf{R_2}$, respectively. Let ˜join_pred (join_pred˜) denote the predicate obtained from join_pred by replacing each attribute by its ˜-prefix (˜-suffix) form. Then

$\texttt{LeftIncrJoin}(\Delta R_1, R_2, \texttt{join\_pred}) =$
 $\texttt{Outerjoin}($
  $\texttt{Pretilde}[\texttt{Join}(\texttt{Removals}(\Delta R_1), R_2, \text{join\_pred})]$,
  $\texttt{Postilde}[\texttt{Join}(\texttt{Additions}(\Delta R_1), R_2, \text{join\_pred})]$,
  $(\tilde{}\text{tid}_1 = \text{tid}_1\tilde{})$ AND $(\tilde{}\text{tid}_2 = \text{tid}_2\tilde{}))$

$\texttt{RightIncrJoin}(R_1, \Delta R_2, \text{join\_pred})$ is defined similarly, for the case where just R_2 changes. **IncrJoin**, defined below, is used when both R_1 and R_2 change. To capture the interaction of simultaneous changes to R_1 and R_2, the notation $\texttt{Both}(\Delta R_1, \Delta R_2, \text{join\_pred})$ denotes
$\texttt{Outerjoin}[$
 $\texttt{Join}(\texttt{Removals}\tilde{}(\Delta R_1), \texttt{Removals}\tilde{}(\Delta R_2), \tilde{}\text{join\_pred})$,
 $\texttt{Join}(\texttt{Additions}\tilde{}(\Delta R_1), \texttt{Additions}\tilde{}(\Delta R_2), \text{join\_pred}\tilde{})$,
  $(\tilde{}\text{tid}_1 = \text{tid}_1\tilde{})$ AND $(\tilde{}\text{tid}_2 = \text{tid}_2\tilde{})$]

We can now express $\texttt{IncrJoin}([R_1, R_2, \Delta R_1, \Delta R_2]$, join_pred) as
 $\texttt{LeftIncrJoin}(\Delta R_1, R_2, \text{join\_pred}) \cup$
 $\texttt{RightIncrJoin}(R_1, \Delta R_2, \text{join\_pred}) \cup$
 $\texttt{Both}(\Delta R_1, \Delta R_2, \text{join\_pred})$

3.2 Transforming Expressions Inside Changes

The Chain Rule is an identity that is used to create an incremental form of an algebraic expression (denoted

Expr) from incremental forms of its constituent operators. Each application of the chain rule moves the outermost operator of Expr outside **Changes**. Where Expr is a single function F, recall that **Changes**(F, L) = **IncrF**(L). Applying the chain rule repeatedly and using this base case, the optimizer removes all appearances of the special operator **Changes**. We present the identity only for the case where the root of Expr is a unary operator (denoted F), so that Expr can be written as a composition of functions, denoted F ° G. Let [RList, ΔRList] denote the substitution list L.

Chain Rule:
Changes (*F* ° *G*; [RList, ΔRList]) =
    IncrF (G(RList), **Changes**(G; [RList, ΔRList]))

The chain rule is proved in the appendix. There are three benefits to using the chain rule to open up the **Changes** expression. 1) **IncrF** now appears explicitly, and it may have an efficient implementation. 2) **IncrF** now needs to be executed only if **Changes**(G; [RList, ΔRList]) is non-empty. 3) It may be easier to derive optimizing transformations (e.g., Selections before Outerjoins) using the incremental forms that replace the **Changes** operator.

We will illustrate this with an example. Consider relation `Sensor_Data[tid, Temperature, Pressure, other_attributes]`. Let `Filter` denote a user-defined function that removes outliers and smoothes the data in a relation with schema `[tid, Temperature, Pressure]`. And let $\texttt{Proj}_{TP}$ denote the operator $\texttt{Project}_{[\text{tid, Temperature, Pressure}]}$

Example 3: `Change To Filtered Temperature/ Pressure`

`Event: Update to Sensor_Data`
`Condition`:
Changes(`Filter` ° $\texttt{Proj}_{TP}$;
    [Sensor_Data, ΔSensor_Data])

If we apply the chain rule to the outermost operator (i.e., `Filter`), we get the Condition:
Incr_Filter[
    $\texttt{Proj}_{TP}$(Sensor_Data), **Changes**($\texttt{Proj}_{TP}$;
    [Sensor_Data, ΔSensor_Data]]

We now replace the inner projection by its incremental form, to obtain
`IncrFilter(`
    $\texttt{Proj}_{TP}$(Sensor_Data), $\texttt{IncrProject}_{TP}$
    (Sensor_Data, ΔSensor_Data]

$\texttt{IncrProject}_{TP}$ does not need the base relation Sensor_Data as an input. The result is the signal graph of Figure 3.

The chain rule has permitted two major improvements in the execution strategy. First, `IncrFilter` need not be evaluated if output of $\texttt{IncrProj}_{TP}$ is empty. Second, it exposed **Changes**($\texttt{Proj}_{TP}$; [...]) so that it could be replaced by $\texttt{IncrProj}_{TP}$.

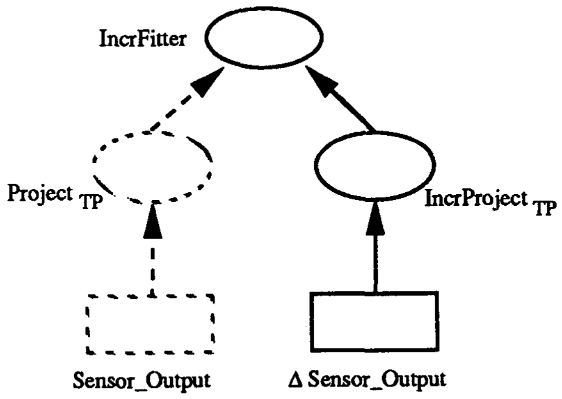

Figure 3: Optimized Signal Graph of Filtered Sensor Data

3.3 Compilation and Evaluation of Situations

Once a final signal graph has been obtained, it is necessary to determine its execution strategy. A good execution strategy will exploit the expected small size of signal relations and relations derived from them.

A leaf of the signal graph that corresponds to an event's signal relation is called *active*. Any node or edge that is on a directed path from an active node is also called active. Other nodes and edges (i.e., those defined strictly over stored relations) are called *passive*.

The passive/active distinction leads to the following simple guidelines for execution strategy:

- For many multi-input operators (e.g., **Join**, **Changes**), when the active input is ∅, the result is ∅.
- Complete evaluation of passive subgraphs is often unnecessary.
- Active nodes often have much smaller relations than passive nodes. Furthermore, they are often com-

puted directly from a signal and hence require no disk accesses. Therefore nested-loops is often a good implementation for a join operator. It requires accessing only those of the inner-relation's tuples that match a tuple from the outer relation (assuming that an index is available). For instance, in Example 1.2, the `Link_Technology` relation may contain thousands of microwave links, but only the tuple for link2 is required for the join.

There are two architectural approaches to finding execution strategies. The first approach is simply to use the heuristics mentioned above, plus perhaps some other simple ones (e.g., `Select` before `Join`, cheap selection predicates before expensive ones). This approach might be most appropriate if one did not have access to an appropriate query optimizer.

An alternative approach is to have a DBMS query optimizer generate the strategies. The signal graph can be regarded as a query to a database that consists of database plus signal relations. Given statistics on signal relations (i.e. their expected small size and lack of indexes), the optimizer will presumably select nested-loops implementations. A further benefit is that the DBMS will supply implementations of relational operators (selection, projection, etc.) and an interpreter for the strategies that are produced; otherwise we need to implement these capabilities as part of the situation evaluator.

But there are also serious drawbacks to using a conventional DBMS — software integration and efficiency. Software integration may be difficult or impossible with a non-extensible DBMS. One must find a way to add operators that manipulate ΔRelations, and to accept operator trees as input (rather than SQL). Efficiency is a problem if signals that originate in main memory need to be stored on disk to appear in DBMS queries. Also, existing optimizers may not do a very good job of exploiting a node value of $\oslash$, an occurrence that may be more common in situation evaluation than for ordinary queries.

4. Summary

4.1 Related Work

Work related to condition monitoring is found spread over several traditional problem areas of the design of database management systems. Such areas include integrity constraint enforcement, support of views, triggers and alerters, rule management, and production systems in general.

Recent contributions include [STON87, DARN87, DITT86]. [STON87] triggers condition evaluation upon database operations, and for a pre-defined set of other events (e.g., date). Conditions are expressed in QUEL, and therefore can be evaluated by the regular query processor. [DITT86] separates events from actions and triggers are defined explicitly using an event, action pair. Our work separates events, conditions, and actions explicitly and the emphasis is on optimizations and algebraic structure. [DARN87] describes a commercial implementation (Sybase). Events are limited to inserts, deletes, and modifies, described by a structure resembling a ΔRelation. Our approach generally differs in that we have provided special constructs to make it easier to express situations involving database changes, and to optimize them. Data from user-defined events may be referenced in conditions.

An algorithm to incrementally compute the changes on materialized views defined by `Select`, `Project`, and `Join` is described in [BLAK86]. However, that paper does not address the issues of representation for database changes (i.e. ΔRelations), changes to general expressions, and optimization of the expressions that compute the changes. [KOEN81] transforms programs containing database update operations so that they will maintain derived data. The transformations require recompiling the program; also, Insert, Delete, and Modify are not given a uniform treatment.

4.2 Conclusions

In this paper we have presented key concepts of a situation evaluator for HiPAC. The main contributions of this paper are:

- a unified treatment of database updates and nondatabase events. This allows the situation evaluator to be implemented in any system that provides the desired event detectors.
- ΔRelations as a compact representation for all database updates. Even transition constraints requiring values from different database states can be naturally expressed using **Changes** and Δrelations.
- an algebra for computations about database changes. The algebra includes efficient implementations of incremental forms of familiar operators and accommodates user-defined operators. Previous work generally provides a separate form for insertions, deletions, and modifications, but one transaction can include all three.

- the Chain Rule as a key transformation in optimizing changes to views. With the exception of [KOEN81], most previous work is limited to Select/Project/Join expressions; we make improvements even with user-defined operators.

Promising areas for future work include: materialized data for condition monitoring; rules to maintain derived data; dispatching strategies for accumulating update events before evaluation; a generalized chain rule for n-ary functions; and simultaneous evaluation of multiple conditions. Larger problems include transposing situation evaluation techniques to distributed and object-oriented systems.

5. Acknowledgements

The authors appreciate the helpful suggestions of the anonymous referees. The authors would also like to thank Dr. Alex P. Buchmann for his comments on earlier versions of this paper.

6. References

[BLAK86] J. A. Blakeley, P. Larson, and F. W. Tompa., Efficiently Updating Materialized Views, *ACM SIGMOD Conf.*, pp. 61-71, Washington, D.C. (May 1986).

[CHAK89] Chakravarthy, S., et al., HiPAC :A Research Project in Active, Time-Constrained Database Management, Final Project Report, XAIT, 1989.

[DARN87] M. Darnovsky, J. Bowman, "TRANSACT-SQL USER'S GUIDE," Document 3231-2.1, Sybase Inc., 1987.

[DAYA88a] Dayal, U., et al., The HiPAC Project: Combining Active Databases and Timing Constraints, *ACM SIGMOD Record*, March 1988.

[DAYA88b] U. Dayal, Active Database Management Systems, *Conf. of Data and Knowledge Bases*, Jerusalem, 1988.

[DITT86] K. R. Dittrich, A. M. Kotz, J. A. Mulle, "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases," *SIGMOD Record 15*, No. 3, 1986, pp. 22-36.

[KOEN81] S. Koenig and R. Paige., A Transformational Framework for the Automatic Control of Derived Data., In *VLDB Conf.*, pp. 306-318, Cannes, (1981).

[STON87] M. Stonebraker, M. Hanson, S. Potamianos, "A Rule manager for Relational Database Systems," Technical Report, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, 1987.

Appendix

Proof of IncrSelect identity: To shorten the formulas, we will use σ to denote $\mathtt{Select}_{pred}$ and σR to denote $\mathtt{Select}_{pred}$(R). From the definition of **Changes** we have:
Changes(σR:[R,ΔR]) $\equiv$
Δ**difference**[σR, σ(Compose(R, ΔR))]. We need to show that this equals the expression given for IncrSel.
From the definition of **Compose**, this equals:
Δ**difference**(σR,
σ[(R -**Removals**(ΔR))$\cup$**Additions**(ΔR)])
Substituting the definition of Δ**difference**, we obtain:
$\mathtt{Outerjoin}_{tid}$(
(σR - σ[(R -**Removals**(ΔR)) $\cup$ **Additions**(ΔR)]),

(σ[(R -**Removals**(ΔR)) $\cup$ **Additions**(ΔR)] - σ(R)))
Now by the distributive law of selection over difference and union, we pull out each selection, to obtain:
$\mathtt{Outerjoin}_{tid}$(
σ(R - [R -**Removals**(ΔR) $\cup$ **Additions**(ΔR)]),
σ([R -**Removals**(ΔR) $\cup$ **Additions**(ΔR)] - R))
This simplifies to the formula for **IncrSel**(ΔR, pred):
Outerjoin(σ(**Removals**(ΔR)), σ(**Additions**(ΔR)))

Proof of Chain Rule: We need to show:
Changes (F $\circ$ G; [RList, ΔRList]) =
IncrF(G(RList), **Changes**(G; [RList, ΔRList])).
Let RHS denote the right hand side of the equality to be demonstrated. Substituting the definition of **IncrF**:
RHS =
Changes(F; [G(RList), **Changes**(G; [RList, ΔRList])])
Replace the "inner" **Changes** by its definition, to get:
Changes(F; [G(RList),
Δ**difference**(G(RList),
G(**Compose**(RList,ΔRList)))]).

Now replace the "outer" Changes by its definition:

```
RHS = Δdifference(F(G(RList)),
        F(Compose[G(RList),
          Δdifference(G[RList],
            G[Compose(RList, ΔRList)])]))
```

We now apply Lemma 1B with G(RList) in the role of R1 and G[Compose(RList, ΔRList)] in the role of R2:

```
RHS = Δdifference[F(G(RList)),
        F(G[Compose(RList, ΔRList)])].
```

This expression is just Changes(F ° G; [RList, ΔRList]).