

# ENFORCING INCLUSION DEPENDENCIES AND REFERENTIAL INTEGRITY

Marco A. Casanova, Luiz Tucheran

Rio Scientific Center - IBM Brasil  
P.O. Box 4624 - Rio de Janeiro - Brasil

Antonio L. Furtado

Pontificia Universidade Católica do Rio de Janeiro  
R. Marquês de S. Vicente, 225 - Rio de Janeiro - Brasil

## ABSTRACT

The general architecture of a monitor that enforces inclusion dependencies and referential integrity is described. The monitor traces the operations a user submits in a session and can either modify an operation or propagate it, depending on additional information the database designer provided at design time. Propagation is implemented by executing new operations when the session terminates, using summary data collected during normal processing.

## 1. INTRODUCTION

When the database designer specifies a conceptual schema, he may include a set of integrity constraints to capture when a database state correctly reflects the real world. A database state is consistent when it satisfies all integrity constraints. Therefore, any operation modifying the database must preserve consistency, that is, map consistent states into consistent states.

An important feature of a database system would then be to automatically enforce constraints. Such feature would completely free users from worrying about consistency preservation and protect the information stored from incorrect operations. We can devise two basic strategies to accomplish this goal, depending on when constraints are taken into consideration. The difference between the two strategies is essentially between compilation and interpretation. The system may incorporate a *constraint enforcement pre-compiler*

that accepts a user's program and produces a new program that has new tests and operations, depending on the constraints of the schema. The new program would have the same basic behavior as the old program, but it would preserve consistency of the database. Such strategy would then help produce correct pre-defined update programs. In a second scenario, the system may have a *constraint enforcement monitor*, acting as a front-end to the DBMS, that controls (interprets) streams of operations guaranteeing that the final database state is consistent. This second strategy would allow users to submit on-line streams of operations leaving to the monitor the problem of consistency preservation.

However, it is very difficult to find an optimized enforcement strategy due to the intrinsic complexity of general integrity constraints. Therefore, it is reasonable to concentrate on small, but significant classes of constraints that can be enforced efficiently.

This paper contributes to the investigation on the automatic enforcement of constraints by describing the general architecture of a monitor that enforces inclusion dependencies and a variation of these dependencies that expresses referential integrity. Depending on additional information the database designer provides at design time, the monitor can reject operations or execute new operations. The monitor optimizes the process even when the stream consists of several different operations submitted in any order, but it assumes that the underlying DBMS guarantees keys, type checking and absence of null values.

In general, the motivation for the paper lies in the unquestionable importance of inclusion dependencies for conceptual modelling and in that their enforcement offers an ample margin for optimization at many different levels, not yet fully explored.

To enforce constraints, the monitor uses two basic strategies: it can either modify the qualification of DML statements, an idea first introduced in [St], or propagate the operation using summary data collected during the session by a technique similar to finite differencing. Fast automatic maintenance of derived data by finite differencing, applied to the enforcement of

general constraints, is discussed in [KP,Pa]. A special case of this technique is discussed in [BBC]. The database designer must inform, for each inclusion dependency, which strategy the monitor must use. This idea and a general overview of referential integrity is contained in [Da]. New storage structures to speed up testing referential integrity are discussed in [BL]. Techniques to optimize the enforcement of general constraints are discussed, for example, in [BB,HI,HS,LB,QS]. A discussion on constraint compiling, using theorem proving techniques, is contained in [HMN]. The implementation of alterers is discussed in [BC] and delayed integrity checking in [La]. A discussion on integrity checking for deductive databases can be found in [ASM,Li]. Finally, the newer relational DBMS prototypes, such as POSTGRES [SAH] and STARBURST [LMP], offer interesting facilities to implement integrity checking.

The paper is organized as follows. Section 2 introduces the notation and basic definitions used throughout the paper. Section 3 informally discusses important points such as the basic strategies for enforcing inclusion dependencies, simple optimizations that can be done at execution time and problems related to the processing of triggers. Section 4 describes the monitor. Finally, section 5 contains the conclusions.

## 2. PRELIMINARIES

We assume that the reader has some familiarity with the relational model and, thus, we just recall here a few basic concepts and notation.

A *relational schema* is a pair  $S = (R, C)$  where  $R$  is a set of relation schemes and  $C$  is a set of integrity constraints. A *relation scheme* is a statement of the form  $R[A_1, \dots, A_n]$  where  $R$  is the *name* and  $A_1, \dots, A_n$  is the *list of attributes* of the scheme. A *database state*  $d$  over  $S$  is a function that assigns an  $n$ -ary relation to each scheme in  $S$  with  $n$  attributes. The state  $d$  is *consistent* iff it satisfies all constraints of  $S$ .

The classes of *integrity constraints* considered in this paper will be keys, statements indicating the type of an attribute and whether it admits null values, inclusion dependencies and references. We assume that the reader is familiar with the first two classes and leave them undefined.

We require that the conceptual schema specify, for each relation scheme, a unique key, called the *primary key* of the scheme, the type of each attribute of the scheme and whether it admits null values or not. By assumption, no attribute participating in the key admits null values.

We now define inclusion dependencies and references. Let  $R[A_1, \dots, A_m]$  and  $S[B_1, \dots, B_n]$  be relation schemes (not necessarily distinct) of a relational schema  $S$ ,  $X$  be a sequence of  $k$  distinct members of  $A_1, \dots, A_m$  and  $Y$  be a sequence of  $k$  distinct members of  $B_1, \dots, B_n$ . Let  $d$  be a database state of  $S$  that assigns the relations  $r$  and  $s$  to  $R$  and  $S$ .

We call the statement  $S[Y] \subset R[X]$  an *inclusion dependency* (IND for short) [CFP]. The state  $d$  *satisfies*  $S[Y] \subset R[X]$  iff  $s[Y]$  is a subset of  $r[X]$ .

Assume now that  $X$  is a key of  $R$ . We call the statement  $S[Y] \rightarrow R[X]$  a *reference* (REF for short) and say that  $Y$  is a *foreign key* of  $S$ . The state  $d$  *satisfies*  $S[Y] \rightarrow R[X]$  iff  $s[Y] - \{\lambda\} \subset r[X]$ . In words, the projection of  $s$  on  $Y$ , excluding all tuples with a foreign key composed of just null values, must be a subset of the projection of  $r$  on  $X$ . Therefore, a REF is not an IND, unless we allow more general relational expressions on the left-hand side of INDs.

Given either an IND of the form  $S[Y] \subset R[X]$  or a REF of the form  $S[Y] \rightarrow R[X]$ , we say that a tuple  $u$  in  $s$  *references* a tuple  $t$  in  $r$  iff  $u[Y] = t[X]$ .

We will adopt an SQL-like notation to describe operations that depart from the SQL standard [SQL] in many points. In particular, we will use two non-standard statements:

$V := \langle \text{select statement} \rangle$   
meaning "store in  $V$  the result of the select statement"

insert into  $R$  tuples  $\{t_1, \dots, t_n\}$   
meaning "insert into  $R$  tuples  $t_1, \dots, t_n$ ".

Finally, we assume throughout the paper that updates cannot modify key values.

## 3. PROBLEMS WITH THE MONITORING OF OPERATIONS

### 3.1 Blocking versus Propagation of Operations

One of the first problems one must face when designing an automatic constraint enforcement subsystem is that the declaration of a constraint says nothing about how to preserve it. Consider, for example, the reference  $S[Y] \rightarrow R[K]$ . If a deletion from  $R$  violates the reference, one can block the deletion, propagate the deletion by deleting tuples from  $S$  or propagate the deletion by setting to null the  $Y$ -value of tuples in  $S$ . The mere declaration of the reference does not indicate

which choice the constraint enforcement subsystem should take.

To solve this first problem, one may prioritize the relation schemes and dictate that operations can propagate only from the higher priority schemes to the lower priority ones [La]. But this strategy does not distinguish between the two propagation options above, for example. A second alternative, which we adopt following [Da], is to require that the database designer explicitly declare which option he wants.

Considering that insertions to  $R$  and deletions from  $S$  cannot violate an IND of the form  $S[Y] \subset R[X]$ , the options are:

- **block deletion**, meaning “do not delete or update the  $X$ -value of a tuple in  $R$  if it is the last tuple referenced by some tuple in  $S$ ”;
- **block insertion**, meaning “do not insert, or update the  $Y$ -value of a tuple in  $S$  if it will not reference any tuple in  $R$ ”;
- **propagate deletion**, meaning “propagate the deletion, or update of the  $X$ -value of a tuple in  $R$  by deleting those tuples in  $S$  that no longer reference any tuple in  $R$ ”;
- **propagate insertion**, meaning “propagate the insertion or the update of a tuple  $u$  in  $S$  by creating a tuple  $t$  in  $R$  such that  $u[Y] = t[X]$ , if  $u$  now references no tuple in  $R$ ”. In this case, the monitor will prompt the user to supply the other attribute values of  $t$ .

The options for a REF of the form  $S[Y] \rightarrow R[K]$  differ from those just described because  $K$  is a key of  $R$  and, hence, by assumption an update to  $R$  never modifies  $K$ -values, and because a tuple in  $S$  may have a null  $Y$ -value. Therefore, in addition to those listed above, REFs have the following option:

- **propagate by nullifying**, meaning “propagate the deletion of a tuple in  $R$  by nullifying the  $Y$ -value of those tuples in  $S$  that referenced it”.

Note that the database designer must specify a valid option for deletions from  $R$  and a valid option for insertions into  $S$ , thus generating four different possibilities for INDs and six for REFs.

Finally, we will use the term *trigger* to refer to an operation automatically executed to implement a propagation, and the term *firing* to refer to the act of invoking a trigger.

### 3.2 A First Look at the Problem of Monitoring Operations

The monitor described in section 4 tries to reduce the cost of enforcing constraints by: (i) rolling back the session as early as possible and when there is no other alternative; (ii) never directly testing if a state is consistent. Intuitively, the monitoring strategy adopted will be as follows:

1. for each operation  $O$  the user submits:
  - a. for each constraint  $C$  with a block option for  $O$ , modify  $O$  to preserve  $C$ ;
  - b. for each constraint  $C$  with a propagate option for  $O$ , collect the values necessary to propagate  $O$ ;
  - c. execute  $O$ ;
2. when the user signals that he has terminated the session, execute all triggers resulting from propagated operations, using the values collected during the processing.

The monitor will use in-core data structures similar to differential files to efficiently and correctly propagate operations and speed up certain tests needed for blocking operations.

The rest of this subsection contains a preliminary analysis of the problem of monitoring operations that will be refined and revised in the next subsections. The analysis will be based on examples that cover all block and propagate options for a REF (since the treatment of INDs is more complex than that of REFs).

Let  $R[A,B]$  and  $S[A,B]$  be two relation schemes, with key  $A$  and subjected to  $S[B] \rightarrow R[A]$ . Consider the deletion:

DR1. delete from  $R$  where  $Q$

Suppose initially that the database designer selected the option **block deletion**.

We first observe that this option does not require rejecting an operation  $O$  if just some of the tuples  $O$  deletes or modifies cause a consistency violation. It leaves open the possibility of modifying  $O$  to reject deleting or modifying just those tuples that would cause problems (we shall see in section 3.4 that this approach does not work correctly for triggers, though). The monitor will adopt this second alternative, implemented by query modification [St] and optimized when possible.

In the example, the monitor will modify DR1 to:

```
DR2. delete from R
      where Q
      and not exists
        (select * from S
         where S.B = R.A)
```

which reads "delete from **R** those tuples satisfying **Q** and which are not referenced by any tuple in **S**". Note that, since **A** is a key of **R**, each tuple in **S** references exactly one tuple in **R**. This is not true about INDS.

Consider now the option **propagate deletion** from **R**.

The monitor will implement the **propagate deletion** option for REFs by keeping a set  $V$  containing the key values of the tuples deleted by an operation and by deleting those tuples in **S** whose foreign key is in  $V$ . (Keeping such sets is also important to solve problems to be discussed in the next two subsections).

The monitored execution of DR1 for the propagate deletion option will be equivalent to:

```
DR3. V := select R.A from R where Q
      delete from R where Q
      ...
DS3. delete from S where S.B in V
```

Note that the monitor must obviously retrieve the sets of values necessary to fire triggers before actually processing the operation since otherwise it would lose the needed values. Also note that  $V$  may contain values not referenced by tuples in **S**, thus increasing the cost of DS3. However, we consider that the cost of filtering out such tuples does not compensate since it would require replacing DR3 by:

```
DR4. V := select S.B from S, R
      where R.A = S.B and Q
      delete from R where Q
```

which, unlike DR3, involves a join between **S** and **R**.

Finally, note that if the DBMS offered a deletion-and-retrieve command, then we could simplify DR3 to:

```
DR5. delete from R
      retrieve R.A into V
      where Q
```

The statement DR5 deletes from **R** all tuples satisfying **Q** and, at the same time, saves their **A**-values in  $V$ .

The option **propagate by nullifying** raises problems of its own, specially when two foreign keys overlap.

Since, except for this point, it is fairly similar to the previous case, we move to the options involving insertions into **S**.

The option **block insertion** can be treated by query modification without problems. For example, given the insertion:

```
IS6. insert into S values (k',k)
```

the monitor will proceed as follows. If the attribute value  $k$  is null, then the monitor executes IS6 unchanged, otherwise it executes:

```
IS7. Z := select * from R
      where R.A = k
      if Z ≠ ∅
      then
        insert into S values (k',k)
```

Finally, the option **propagate insertion** requires prompting the user to supply the missing attribute values for the tuple to be inserted into **R**.

### 3.3 Monitoring Multiple User Operations

The discussion in section 3.2 must be revised, first of all, because the user may himself create several operations that together preserve consistency, thus making it unnecessary to modify or propagate operations in certain cases.

For example, consider again the two relation schemes **R[A,B]** and **S[A,B]**, with key **A** and subjected to **S[B]→R[A]**. Assume that the options selected are **block insertion** into **S** and **propagate deletion** from **R**.

We first illustrate how the modification of an operation may become unnecessary. Using a strategy similar to that described in section 3.2, if the user submits a sequence of insertions of the form:

```
IR1. insert into R values (k,b)
IS1. insert into S values (k',k)
```

the monitor will process IR1 without modification, but it will unnecessarily change IS1 to:

```
IS2. Z := select * from R
      where R.A = k
      if Z ≠ ∅
      then
        insert into S values (k',k)
```

Indeed, the qualification of IS2 is trivially satisfied in view of IR1.

It is very difficult to completely avoid unnecessary tests because we would have to take into account not only insertions, but also complex updates. Therefore, we adopted a compromise solution. The monitor will maintain in core the set  $W$  of all foreign key values that it knows to be in the database because of the operations it has already processed. It will then use such set to speed up the acceptance test for insertions.

In the case of the current example, the monitor will then produce the following stream of operations when processing IR1 and IS1:

```
IR3.  W := {k};
      insert into R values (k,b)
IS3.  if k occurs in W
      then
        insert into S values (k',k)
      else
        begin
          Z := select * from R
              where R.A = k
          if Z ≠ ∅
          then
            insert into S values (k',k)
          end
```

An identical problem occurs with deletions, which the monitor will minimize by keeping the set of all foreign key values that it knows not to be in the database because of the operations it has already processed.

Modifications may also be wrongly applied if operations are submitted in the wrong order, as would be the case if IS1 were submitted before IR1. The monitor will not avoid this problem, however, since it will always modify an operation before processing it, for each constraint with a block option for that type of operation.

We now turn to operation propagation, which creates a different source of problems. Indeed, if the monitor fires triggers immediately after an operation, it may be anticipating a corrective action that will become unnecessary or even wrong due to an operation that the user will still submit.

For example, suppose that the user intends to submit the following sequence of operations:

```
DR4.  delete from R where R.A = k
US4.  update S set S.B = null
      where S.B = k
DR4'. delete from R where R.A = k'
```

Intuitively, the database designer may have decided that deletions from  $R$  propagate to deletions from  $S$

when he specified the **propagate deletion** option. But the user, for this particular session, decided that the deletion of a tuple with key  $k$  from  $R$  propagates by nullifying the foreign key values of the appropriate tuples of  $S$ .

Firing triggers right after operations would imply executing statement DS4 below before processing US4:

```
DS4.  delete from S where S.B = k
```

But DS4 wrongly deletes all tuples US4 will process.

To avoid such problems, the monitor will: (i) postpone firing triggers until the user signals that he has terminated the session; (ii) maintain the set of values needed to fire triggers using a technique similar to that used to implement differential files [Pa].

In the current example, the monitor will then produce the following stream of operations ( $V$  is the set of key values of the tuples deleted from  $R$ ):

```
DR5.  V := {k}
      delete from R where R.A = k
US5.  update S set S.B = null
      where S.B = k
DR5'.  V := V ∪ {k'}
      delete from R where R.A = k'
/* session ends - fire triggers */
DS5.  if V ≠ ∅
      then
        delete from S where S.B in V
```

In this particular case, DS5 will delete only those tuples in  $S$  whose foreign key value is  $k'$  since, after US5, no tuple whose foreign key value is  $k$  remains in  $S$ . Note that we could, again in this particular case, easily deduce that  $k$  can be removed from  $V$  after executing US5.

### 3.4 Monitoring Triggers

On a first approximation, the monitor may treat a trigger  $O$  as if it were part of the stream of operations the user submitted. In particular, and this is very important, the monitor must check: (i) if  $O$  may violate a constraint  $C$  and apply the appropriate block or propagate option; (ii) if  $O$  requires changing one of the sets of values kept to fire further triggers (such as  $V$  in the examples in sections 3.2 and 3.3). This is done exactly as for user operations, except for the differences discussed in what follows.

First, unlike a user operation, the monitor cannot modify a trigger as otherwise the final state may be inconsistent. Therefore, if any block option applies to a trigger, the monitor must perform a test to decide if the trigger can run unchanged. If not, the monitor has to abort and rollback the complete session or to return to the user for corrective action.

For example, consider the three relation schemes  $R[A,B]$ ,  $S[A,B]$  and  $T[A,B]$ , with key  $A$  and subjected to  $S[B] \rightarrow R[A]$  and  $T[B] \rightarrow S[A]$ . Suppose that the options are **propagate deletion** from  $R$  and **block deletion** from  $S$ .

Let DR1 be a deletion of the form:

```
DR1. delete from R where Q
```

Then, the monitor will proceed as follows:

```
DR2. V := select R.A from R
      where Q
      delete from R where Q
/* session ends - fire triggers */
```

```
DS2. W :=
      select * from S
      where S.B in V
      and exists
      (select * from T
       where S.A = T.B)
if W = {}
then
  delete from S where S.B in V
else
  rollback
```

In DS2, the monitor first tests if the deletion of any tuple from  $S$  needed to propagate DR2 violates consistency. If not, the monitor will execute the deletion from  $S$ , otherwise it will abort execution. This course of action is necessary as otherwise the propagation from DR2 would be executed only partially thus possibly not fully restoring consistency with respect to  $S[B] \rightarrow R[A]$ .

The last problem we illustrate is trigger interference. Assume the same scenario as in the previous example, but suppose that the options are **propagate deletion** from  $R$  and **propagate insertion** into  $S$ .

Consider the sequence of operations:

```
DR3. delete from R where R.A = k
IS3. insert into S values (k',k)
```

During normal processing, the monitor executes:

```
DR4. V := {k}
      delete from R where R.A = k
IS4. W := {k}
      insert into S values (k',k)
```

When the session terminates, the monitor will then fire the triggers in some order, for instance:

```
DS4. delete from S where S.B in V
IR4. for each x in W do
begin
  Z := select * from R
      where R.A = x
  if Z = {}
  then
    begin
      ask the user for
        the B-value y of the tuple
        to be inserted with key x
      insert into R values (x,y)
    end
  end
```

But the two triggers interfere with each other. If, as written above, the monitor executes DS4 before IR4, the tuple  $(k', k)$  inserted by IS4 is deleted by DS4, making the firing of IR4 no longer necessary.

On the other hand, if the monitor executes IR4 before DS4, then DS4 need not execute at all because  $R$  will again have a tuple with key value  $k$ .

The monitor will resolve interference by having triggers modify the values kept to fire further triggers. It will also give preference to insertion triggers to avoid firing deletion triggers unnecessarily.

Therefore, the monitor will execute (modified) triggers in the following order:

```
IR5. for each x in W do
begin
  ask the user for
    the B-value y of the tuple
    to be inserted with key x
  insert into R values (x,y)
end
V := V - W
DS5. delete from S where S.B in V
```

In this specific example, we have  $V = W = \{k\}$  just before the execution of the triggers. Thus, the firing of DS5 becomes vacuous, as desired, since  $V$  will be

come empty just before the execution of this statement.

Finally, we observe that the strategy of keeping the sets of values required to process triggers also copes, without change, with the recursive firing of triggers.

## 4. DESCRIPTION OF THE MONITOR

### 4.1 The Basic Data Structure of the Monitor

The monitor will process **block** options by modifying an operation, if necessary, as soon as it is submitted. To process **propagate** options, the monitor will maintain, during normal processing, summary information and, when the session terminates, it will process all triggers required to restore consistency as if they were user operations, with the differences pointed out in section 3.4.

The summary information will take the form of a list `fire` whose entries will be quadruples with the format  $(t, R, X, V)$  where  $t$  is either `d` (for deleted) or `i` (for inserted),  $R$  is a relation name,  $X$  is a list of attributes of  $R$  and  $V$  is a set of  $X$ -values. Briefly, the entries in `fire` will serve the following purposes:

- keep information needed to fire triggers;
- avoid trigger interference;
- solve the problem of recursive propagation of triggers;
- speed up testing if an operation can run unmodified.

The rest of this section details how the monitor maintains the list `fire`.

Let  $S = (R, C)$  be the relational schema in question. For each constraint  $C$  of  $S$ , if  $C$  is an IND of the form  $S[Y] \subset R[X]$  or a REF of the form  $S[Y] \rightarrow R[X]$ , the monitor will maintain the following entries in `fire` (the two cases are not mutually exclusive):

**Case 1:**  $C$  has one of the options - **propagate deletion** or **propagate by nullifying** from  $R$ , or **block insertion** into  $S$ .

Let  $r_f$  indicate the value of  $R$  in a state  $f$  and  $s_f$  the value of  $S$  in  $f$ .

The monitor will maintain an entry in `fire` of the form  $(d, R, X, V)$  in such a way that the following assertion is an invariant:

$v \in V$  iff there is a state  $d$  previous to the current state  $c$  such that there is  $t_d$  in  $r_d$  with  $t_d[X] = v$  and there is no  $t_c$  in  $r_c$  with  $t_c[X] = v$ .

It is then possible to prove that:

**Lemma 1:** Let  $f$  be any state during the processing. Then:

for any  $u$  in  $s_f$ ,  $u[Y]$  is in  $V$  iff there is no  $t$  in  $r_f$  such that  $u[Y] = t[X]$ .

Thus, in particular, any tuple  $u$  in  $s_f$  such that  $u[Y] \in V$  must be deleted or have its  $Y$ -value nullified, depending on the option chosen.

We now describe how the monitor maintains  $V$  to satisfy the above assertion, considering each possible operation over  $R$ . Intuitively, the monitor must include in  $V$  a value  $v$  iff  $v$  was in the projection of  $R$  on  $X$  before, but not after, the operation is executed (in all situations below, the reader must remember that, when  $C$  is an IND of the form  $S[Y] \subset R[X]$ ,  $X$  is not a key of  $R$  and, hence, there may be more than one tuple in  $R$  with the same  $X$ -value).

*Case A:* delete from  $R$  where  $Q$

```

/*
   select X-values deleted from R[X]
*/
V' :=
select R.X from R
where Q
and not exists
  (select * from R R'
   where R.X = R'.X
   and not Q[R'/R])
/*
   add them to V
*/
V := V U V'

```

where the notation  $Q[R'/R]$  indicates  $Q$  with all occurrences of  $R$  replaced by  $R'$ .

*Case B:* insert into  $R$  tuples  $\{t_1, \dots, t_n\}$

```

/*
   remove inserted X-values from V
*/
V := V - \{t_1[X], \dots, t_n[X]\}

```

Case C: update R set P where Q

```

/*
  select X-values deleted from R[X]
*/
V' :=
  select R.X from R
  where Q
  and not exists
    (select * from R R'
     where R.X = R'.X
     and not Q[R'/R])
/*
  select X-values inserted into R[X]
*/
V'' := select P_X from R where Q
/*
  add X-values deleted and
  remove those inserted
*/
V := (V U V') - V''

```

where  $P_X$  indicates the projection on  $X$  of the result of applying the changes described by  $P$  on each tuple.

The qualifications enclosed within a box in cases A and C become unnecessary if the following condition holds:

(\*)  $(\forall t \in R)(\forall u \in R)$   
 $((Q[t] \wedge t[X] = u[X]) \rightarrow Q[u])$

that is, when for any two tuples in  $R$  with the same  $X$ -value, if one satisfies  $Q$  so does the other. Two simple and sufficient conditions for (\*) are:

- (\*.1)  $X$  functionally determines all other attributes used in  $Q$ ;
- (\*.2)  $Q$  is a condition involving only attributes in  $X$ .

Note that, when  $C$  is a REF of the form  $S[Y] \rightarrow R[X]$ , condition (\*.1) is satisfied since  $X$  is a key of  $R$ . Therefore, the qualifications within boxes above may all be dropped.

Case 2:  $C$  has one of the options **block deletion** from  $R$  or **propagate insertion** into  $S$ .

The monitor will maintain an entry in  $W$  of the form  $(i, S, Y, W)$  in such a way that the following assertion is an invariant:

$w \in W$  iff there is a state  $d$  previous to the current state  $c$  such that there is no  $u_d$  in  $S_d$  with  $u_d[Y] = w$  and there is  $u_c$  in  $S_c$  with  $u_c[Y] = w$ .

In this case, it is possible to prove that:

**Lemma 2:** Let  $f$  be any state during the processing. Then:

for any  $u$  in  $S_f$ , if there is no  $t$  in  $r_f$  such that  $u[Y] = t[X]$  then  $u[Y]$  is in  $W$ .

Since the lemma holds in just one direction, to propagate insertions into  $S$ , the monitor has to synthesize a trigger that tests if tuples need at all be inserted into  $R$ .

The maintenance of  $W$  depends only on the operations over  $S$  to satisfy the above assertion. Intuitively, the monitor must include in  $W$  a value  $w$  iff  $w$  is in the projection of  $S$  on  $Y$  after the operation is executed, but it was not there before (the reader must again remember that there may be more than one tuple in  $S$  with the same  $Y$ -value):

Case A: delete from  $S$  where  $Q$

```

/*
  select Y-values deleted from S[Y]
*/
W' :=
  select S.Y from S
  where Q
  and not exists
    (select * from S S'
     where S'.Y = S.Y
     and not Q[S'/S])
/*
  remove them from W
*/
W := W - W'

```

Case B: insert into S tuples  $\{u_1, \dots, u_n\}$

```

/*
  select Y-values already in S[Y]
*/
W' :=
  select S.Y from S where S.Y in
    {u1[Y], ..., un[Y]}
/*
  add Y-values inserted,
  except those already in S[Y]
*/
W := W ∪ ((u1[Y], ..., un[Y]) - W')

```

Case C: update S set P where Q

```

/*
  select Y-values deleted from S[Y]
*/
W' :=
  select S.Y from S
  where Q
  and not exists
    (select * from S S'
     where S'.Y = S.Y
     and not Q[S'/S])
/*
  select Y-values inserted into S[Y],
  except those already there
*/
W'' :=
  select PY from S
  where Q
  and not exists
    (select * from S S'
     where S'.Y = PY
     and not Q[S'/S])
/*
  add new Y-values and
  remove those deleted
  but not re-inserted
*/
W := (W - W') ∪ W''

```

All conditions enclosed within a box above again become unnecessary when a condition similar to that introduced in the previous case holds.

## 4.2 Synthesis of Triggers and Modification of Operations

For the sake of clarity, the exposition considers separately each possible block/propagate alternative for INDs and REFs, omitting repetitive details as much as possible.

**Case 1:** Consider an IND of the form  $S[Y] \subset R[X]$ .

**Case 1.1: propagate deletion from R and block insertion into S.**

Recall from section 4.1 that the monitor will maintain an entry in  $V$  of the form  $(d, R, X, V)$ . Then, in view of Lemma 1, to propagate deletions from R, the monitor must synthesize a trigger of the form:

DS. delete from S where S.Y in  $V$

Now, to block insertions into S, the monitor has to modify both insertions and updates as follows.

*Case A:* let IS be a multiple insertion statement of the form:

IS. insert into S tuples  $\{u_1, \dots, u_n\}$

If IS is an operation submitted by the user, the monitor will modify IS to:

```

IS1. Z :=
  select R.X from R
  where R.X in {u1[Y], ..., un[Y]}
IS1. for each ui in {u1, ..., un}
  such that ui[Y] is not in Z do
  insert into S values ui
  end

```

But if IS is a trigger, the monitor will first perform the following acceptance test:

```

TS1. Z :=
  select R.X from R
  where R.X in (u1[Y], ..., un[Y])

```

If Z is not equal to  $\{u_1[Y], \dots, u_n[Y]\}$ , then some tuple to be inserted by IS will not reference any tuple in R and, hence, the monitor must reject IS and rollback the whole session.

The monitor can also use the entry  $(d, R, X, V)$  to speed up rejecting IS in both cases. Indeed, if  $u_i[Y] \in V$  holds, then by definition of  $V$  there is no

tuple  $t$  in the current value of  $R$  such that  $t[X] = u_i[Y]$ .

Case B: let UP be an update statement of the form:

UP. update S set P where Q

If UP is an operation submitted by the user that changes Y-values, the monitor will modify it to:

UP1. update S set P  
 where Q  
 and exists  
 (select \* from R  
 where R.X = P<sub>Y</sub>)

But if UP is a trigger, the monitor performs acceptance tests whose qualification is similar to that added above.

**Case 1.2: propagate insertion into S and block deletion from R.**

Recall from section 4.1 that the monitor will maintain an entry in fire of the form  $(i, S, Y, W)$ . Therefore, to propagate insertions into S, since Lemma 2 holds in just one direction, the monitor has to synthesize a trigger that tests if tuples need at all be inserted into R:

IR.  $W' := \text{select } R.X \text{ from } R$   
       where R.X in W  
 $W := W - W'$   
 for each  $w \in W$  do  
 begin  
   construct a tuple  $t$   
   with X-value equal to  $w$  and  
   with the other attribute values  
   supplied by the user  
   insert into R values  $t$   
 end

Note that, if the monitor also maintains an entry in fire of the form  $(d, R, X, V)$  then, by definition of V, it could speed up IR as follows (Z and Z' are just temporary variables):

IR'.  $Z := W \cap V$   
 $Z' := W - V$   
 $W' := \text{select } R.X \text{ from } R$   
       where R.X in Z'  
 $W := W' \cup Z$   
 for each  $w \in W$  do  
 ...

Now, to block deletions from R, the monitor has to modify both deletions and updates as follows:

Case A: let DR be a deletion statement of the form:

DR. delete from R where Q

The monitor will proceed as follows.

If DR is an operation submitted by the user, the monitor will modify DR to:

DR1. delete from R  
 where Q and  
 not (exists  
   (select \* from S  
   where S.Y = R.X)

and  
 not exists  
 (select \* from R R'  
   where R'.X = R.X  
   and not Q[R'/R]))

If DR is a trigger, the monitor will perform the following acceptance test:

TR1.  $Z :=$   
 select S.Y from R, S  
 where Q  
 and S.Y = R.X

and  
 not exists  
 (select \* from R R'  
   where R'.X = R.X  
   and not Q[R'/R]))

**Note:** as in section 4.1, the qualifications enclosed within boxes may be dropped if condition (\*) holds.

If Z is not empty, then some tuple DR deletes is the last one referenced by some tuple in S and, hence, the monitor must reject DR and rollback the whole session.

In a particular situation, the monitor can use the entry  $(i, S, Y, W)$  to speed up rejecting DR in both cases. Indeed, suppose that the qualification Q is equivalent to the disjunction "R.X=c<sub>1</sub> or ... or R.X=c<sub>n</sub>". Then, if  $c_i \in W$ , for some i, DR must be rejected because, by definition of W, there is a tuple u in the current value of S such that  $u[Y] = c_i$ .

Case B: let UP be an update statement of the form:

UP. update R set P where Q

If UP is a user operation and it changes X-values, the monitor modifies it as follows:

```
DR1. update R set P
      where Q and
      not (exists
            (select * from S
              where S.Y = R.X)
            and
            not exists
            (select * from R R'
              where
                (R.X = R'.X
                 and not Q[R'/R])
                or (R.X = P_X[R'/R]
                 and Q[R'/R])))
```

But if UP is a trigger, the monitor performs acceptance tests similar to those already discussed.

**Case 1.3: block deletion** from R and **block insertion** into S.

The monitor will maintain entries in *fire* of the form  $(d, R, X, V)$  and  $(i, S, Y, W)$  just to speed up rejection tests as in cases 1.1 and 1.2.

**Case 1.4: propagate deletion** from R and **propagate insertion** into S.

The monitor will maintain entries in *fire* of the form  $(d, R, X, V)$  and  $(i, S, Y, W)$  to process propagations as in cases 1.1 and 1.2.

**Case 2:** Consider a REF of the form  $S[Y] \rightarrow R[K]$

The treatment of REFs is similar to that of INDs, but considerably simpler because K is the key of R, which implies that:

- no two tuples in R have the same K-value;
- updates do not affect, by assumption, K-values.

However, we must also take into account the fact that tuples in S may have null Y-values. We refer the reader to the full paper for the details [CFT].

## 5. CONCLUSIONS

A monitor that enforces INDs and REFs for single operation transactions has already been implemented [FCT]. The monitor is coupled with a design helper that automatically maps an entity-relationship schema into a relational schema and that incorporates optimization features at the design level. We began to extend the monitor to control streams consisting of multiple operations, along the lines of this paper, and also to enhance the design helper to cope with other optimization strategies at the design level.

The monitoring strategy can be enhanced along many lines. First, the strategy may be locally improved in many points, such as ordering acceptance tests based on their estimated cost.

Second, the strategy can be further elaborated to cope with more sophisticated options. We may introduce **immediate propagation** options that force the firing of triggers immediately after operations, as an alternative to the deferred propagation options we defined in section 3.1. We may also create different block/propagation options for different classes of users. Finally, we may introduce **modify** options that explicitly indicate how to modify certain types of operations.

The monitor can obviously be extended to cope with other classes of constraints. Naturally the qualification modification algorithms and the synthesis of triggers would have to be reworked. But the maintenance and the general idea behind the basic data structure, *fire*, might possibly remain the same.

## ACKNOWLEDGEMENT

We thank Prof. Claudia B. Medeiros for her careful reading of a preliminary version of the manuscript.

## REFERENCES

- [ASM] P. Asirelli, M. de Santis and M. Martelli, "Integrity constraints in logic databases", *J. Logic Programming* 2:3 (Oct. 1985), 221-232.
- [BB] P.A. Bernstein and B.T. Blaustein, "Fast method for testing quantified relational calculus assertions", *Proc. SIGMOD Int. Conf. on Management of Data*, Orlando, Florida (June 1982), 39-50.
- [BBC] P.A. Bernstein, B.T. Blaustein and E.M. Clarke, "Fast maintenance of semantic integrity assertions using redundant aggregate data", *Proc. of 6th Int. Conf. on Very Large Data Bases*, Montreal, Canada (Oct. 1980), 126-136.
- [BC] O.P. Buneman and E.K. Clemons, "Efficiently monitoring relational databases", *ACM TODS* 4:3 (Sept. 1979), 368-382.
- [BL] M. Bever and R. Lorie, "An enhanced referential integrity scheme supporting complex objects", *IBM Research Report*, RJ-5585 (1987).
- [CFP] M.A. Casanova, R. Fagin and C.H. Papadimitriou, "Inclusion dependencies and their interaction with functional dependencies", *J. Computer and System Sciences* 28:1 (Feb. 1984), 29-59.
- [CFT] M.A. Casanova, A.L. Furtado and L. Tucheran, "Enforcing Inclusion Dependencies and Referential Integrity", *Technical Report CCR052*, Rio Scientific Center, IBM Brazil (Feb. 1988).
- [Da] C.J. Date, "Referential integrity", *Proc. of 7th Int. Conf. on Very Large Data Bases*, Cannes, France (Sept. 1981), 2-12.
- [FCT] A.L. Furtado, M.A. Casanova and L. Tucheran, "The CHRIS consultant", *Proc. 6th Int. Conf. on Entity-Relationship Approach* (Nov. 1987), 479-486.
- [HI] A. Hsu and T. Imielinski, "Integrity checking for multiple updates", *Proc. Int. Conf. on Management of Data*, Austin, Texas (May 1985), 152-168.
- [HMN] L.J. Henschen, W.W. McCune and S.A. Naqvi, "Compiling constraint-checking programs from first-order formulas", *Advances in Data Base Theory*, vol. 2, H. Gallaire, J. Minker and J.M. Nicolas, Eds., Plenum (1984), 145-169.
- [HS] M. Hammer and S. Sarin, "Efficient monitoring of database assertions", *Proc. Int. Conf. on Management of Data*, Austin, Texas (May 1978), 38-49.
- [KP] S. Koenig and R. Paige, "A transformational framework for the automatic control of derived data", *Proc. of 7th Int. Conf. on Very Large Data Bases*, Cannes, France (Sept. 1981), 306-318.
- [La] G.M.E. Lafue, "Semantic integrity dependencies and delayed integrity checking", *Proc. of 8th Int. Conf. on Very Large Data Bases*, Mexico (Sept. 1982), 292-299.
- [LB] L. Lilien and B. Bhargava, "A scheme for batch verification of integrity assertions in a database systems", *IEEE Trans. on Software Engineering* 10:6 (Nov. 1984).
- [Li] T. Ling, "Integrity constraint checking in deductive databases using the Prolog not-predicate", *Data & Knowledge Engineering* 2:2 (June 1987), 145-168.
- [LMP] B. Lindsay, J. McPherson and H. Pirahesh, "A data management extension architecture", *Proc. Int. Conf. on Management of Data*, San Francisco, CA (May 1987), 220-226.
- [Pa] R. Paige, "Applications of finite differencing to database integrity control and query/transaction optimization", *Advances in Data Base Theory*, vol. 2, H. Gallaire, J. Minker and J.M. Nicolas, Eds., Plenum (1984), 171-209.
- [QS] X. Qian and D.R. Smith, "Integrity constraint reformulation for efficient validation", *Proc. of 13th Int. Conf. on Very Large Data Bases*, Brighton, UK (Sept. 1987), 417-425.
- [SAH] M. Stonebraker, J. Anton and E. Hanson, "Extending a database system with procedures", *ACM TODS* 12:3 (Sept. 1987), 350-376.
- [SQL] ANSI, *Database Language SQL*, New York, NY, X3.135-1986 (1986).
- [St] M. Stonebraker, "Implementation of integrity constraints and views by query modification", *Proc. Int. Conf. on Management of Data*, San Jose, CA (May 1975), 65-78.