

A MONITOR ENFORCING REFERENTIAL INTEGRITY

Marco A. Casanova
Antonio L. Furtado*
Luiz Tucheran

Rio Scientific Center
IBM Brasil
Estrada da Canoa, 3520
22.610, Rio de Janeiro, RJ

ABSTRACT

The design decisions behind a monitor that enforces referential integrity are 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 behaviour 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

* On leave from the Pontificia Universidade Católica do Rio de Janeiro

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 basic design decisions behind a monitor that enforces 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.

Although presented in the context of a monitor, most of the strategies described in the paper can also be adapted and directly incorporated into application programs or even into constraint enforcement pre-compilers. In addition, they may help indicate which new features a DBMS should have.

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 constraint, 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 alerters 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 the basic strategies for enforcing referential integrity, simple optimizations

that can be done at execution time and problems related to the processing of triggers. Finally, section 4 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.

We adopt the usual notation for relations, tuples and relational algebra expressions. In particular, we use λ to denote either a null value or a tuple of null values of arbitrary length.

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, 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, all attributes participating in the key do not admit null values.

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 .

Assume 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 . We also say that a tuple u in s *references* a tuple t in r iff $u[Y] = t[X]$.

The problem of enforcing referential integrity is the problem of guaranteeing that the database state is consistent with the references of the relational schema.

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

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

3. MONITORING OPERATIONS

This section describes the major problems related to the enforcement of REFs and illustrates the solutions that can be adopted by a monitor.

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 a REF of the form $S[Y] \rightarrow R[X]$, the options are:

- **block deletion**, meaning "do not delete a tuple in R if it is 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 of a tuple in R by deleting those tuples in S that referenced it";

- **propagate insertion**, meaning “propagate the insertion of a tuple u in S with a non-null Y -value by creating a tuple t in R such that $u[Y] = t[K]$, if u references no tuple in R ”. In this case, the monitor will prompt the user to supply the other attribute values of t .
- **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”.

We use the term **propagate** instead of **cascade** as in [Da].

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

A naive strategy for monitoring user operations would be as follows:

1. Execute all user operations as they are submitted;
2. For each constraint C , test if the final state satisfies C ; if not, then rollback.

Such strategy requires two expensive operations: rolling back the complete session and testing if a state satisfies a constraint.

The optimized monitor (henceforth called the monitor) 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 monitor will proceed 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 efficient 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.

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 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 .

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

We first remark that the naive monitor would, after executing all user operations, recheck each tuple u in S and delete u if it no longer references a tuple in R . Note that this implementation completely ignores the set of tuples deleted or modified by O and the assumption that the sequence of operations began execution on a consistent state.

In the running example, the naive monitor would then execute the following statement after DR1:

```

DS2. delete from S
      where not exists (select * from R
                       where S.B = R.A)

```

Note that DS2 contains a join between S and R and no restrictions and is thus potentially expensive.

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

```

The statements DR3 and DS3 will in general be less expensive than DS2 depending on the selectivity of Q.

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** does not raise new problems, so 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 constant 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.

This concludes our preliminary analysis of the problem of monitoring operations. The next subsections show that blocking and, especially, propagation in fact require much more elaborated mechanisms.

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 that operation modifications 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.

Completely avoiding such problems is very difficult because not only insertions, but complex updates may also affect relations. 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
        Z := select * from R where R.A = k
        if Z ≠ ∅
        then insert into S values (k',k)

```

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.

A possible solution to the ordering problem would be to defer execution of all operations the user submits until the session terminates. We discarded this possibility because it implies that all queries the user formulates during a session will not reflect the result of previously submitted operations. An alternative would be to let the user inform the monitor that a subsequence of the operations he is submitting can be reordered, if necessary, and that he will not submit queries in between. However, a simpler solution would be to define at database design time a fixed and correct update order for the relation schemes and force users to follow it.

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** propagate 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 exists (select * from R
                  where S.B = R.A and R.A = 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 U {k'}
      delete from R where R.A = k'
      /* user ends session - 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.

However, as already remarked in section 3.2, we believe that, in general, the cost of eliminating extra values exceeds the cost of firing triggers based on sets that may contain more values than necessary. Moreover, the monitor will guarantee that these extra values never cause unnecessary deletions or updates, as in the previous example.

3.4 Monitoring Triggers

The previous two sections essentially discussed how to process user operations. This subsection will then concentrate on the monitoring of 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.  $V := \text{select } R.A \text{ from } R \text{ where } Q$ 
      delete from R where Q
      /* session terminates - fire triggers */
DS2.  $W := \text{select } * \text{ from } S$ 
      where S.B in  $V$ 
      and exists (select * from T
                  where S.A = T.B)

if  $W = \emptyset$ 
  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
      Z := select * from R where R.A = x
      if Z =  $\emptyset$ 
      then
        ask the user for
          the B-value y of the tuple to be inserted with key x
        insert into R values (x,y)

```

But the two triggers interfere with each other. If, as written above, the monitor executes DS4 before IR4, the tuple (k', k) inserted by IR4 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
      ask the user for
        the B-value y of the tuple to be inserted with key x
      insert into R values (x,y)
      V := V - W
DS5. delete from S where S.B in V

```

Note that the maintenance of V is rather simple in this case.

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. CONCLUSIONS

A monitor that enforces 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 will have to be reworked. But the maintenance and the general idea behind fire need not be at all changed to be useful for the enforcement of other classes of constraints.

Finally, we could let users participate much more actively in the enforcement of constraints. Once the monitor discovers that an operation will violate a constraint, it may so inform the user, exhibiting the constraint and the data that causes the violation. It may also suggest a modification for the operation or some corrective action, depending on the option the database designer defined. However, the user is free to suspend the operation and do some preliminary corrective operation, which will in turn suffer the same processing. The process of changing the database will then become very similar to aided plan generation [FM,Wa]. Indeed, the monitor will assume the status of a plan generation helper with very limited (but very efficient) deductive capability represented by the enforcement strategies for the block/propagate options.

REFERENCES

- [ASM] P. Asirelli, M. de Santis and M. Martelli, Integrity constraints in logic databases, *The Journal of Logic Programming*, 2, 3, October 1985 (pp. 221-232).
- [BBC] P.A. Bernstein, B.T. Blaustein and E.M. Clarke, Fast maintenance of semantic integrity assertions using redundant aggregate data, *Proc. of 6th International Conference on Very Large Data Bases*, Montreal, Canada, October 1980 (pp. 126-136).
- [BC] O.P. Buneman and E.K. Clemons, Efficiently monitoring relational databases, *ACM TODS*, 4, 3, September 1979 (pp. 368-382).
- [BL] M. Bever and R. Lorie, An enhanced referential integrity scheme supporting complex objects, *IBM Research Report*, RJ-5585 (1987).

- [Da] C.J. Date, Referential integrity, Proc. of 7th International Conference on Very Large Data Bases, Cannes, France, September 1981 (pp. 2-12).
- [Es] K.P. Eswaran, Specification, implementation and interaction of a trigger subsystem in an integrated data base system, IBM Technical Report RJ-1820, August 1976.
- [FCT] A.L. Furtado, M.A. Casanova and L. Tucheran, The CHRIS consultant, Proc. 6th International Conference on Entity-Relationship Approach, November 1987 (pp. 479-486).
- [FM] A.L. Furtado, C.M.O. Moura, Expert Helpers to data-based information systems, Expert Database Systems, Proc. First International Workshop (L. Kerschberg, Ed.) Benjamin/Cummings Publishing Co., Menlo Park, CA, 1986 (pp. 581-596).
- [HI] A. Hsu and T. Imielinski, Integrity checking for multiple updates, Proc. International Conference on Management of Data, Austin, Texas, May 1985 (pp. 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, (Eds. H. Gallaire, J. Minker and J.M. Nicolas), Plenum, 1984 (pp. 145-169).
- [HS] M. Hammer and S. Sarin, Efficient monitoring of database assertions, Proc. International Conference on Management of Data, Austin, Texas, May 1978 (pp. 38-49).
- [JK] D.S. Johnson and A. Klug, Testing containment of conjunctive queries under functional and inclusion dependencies, Proc. First ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, Los Angeles, 1982 (pp. 164-169).
- [KP] S. Koenig and R. Paige, A transformational framework for the automatic control of derived data, Proc. of 7th International Conference on Very Large Data Bases, Cannes, France, September 1981 (pp. 306-318).
- [La] G. M.E. Lafue, Semantic integrity dependencies and delayed integrity checking, Proc. of 8th International Conference on Very Large Data Bases, Mexico, September 1982 (pp. 292-299).
- [LB] L. Lilien and B. Bhargava, A scheme for batch verification of integrity assertions in a database systems, IEEE Transactions on Software Engineering, 10, 6, November 1984
- [Li] T. Ling, Integrity constraint checking in deductive databases using the Prolog not-predicate, Data & Knowledge Engineering, 2, 2, June 1987 (pp. 145-168).
- [LMP] B. Lindsay, J. McPherson and H. Pirahesh, A data management extension architecture, Proc. International Conference on Management of Data, San Francisco, CA, May 1987 (pp. 220-226).

- [Pa] R. Paige, Applications of finite differencing to database integrity control and query/transaction optimization, *Advances in Data Base Theory*, vol. 2, (Eds. H. Gallaire, J. Minker and J.M. Nicolas), Plenum, 1984 (pp. 171-209).
- [QS] X. Qian and D.R. Smith, Integrity constraint reformulation for efficient validation, *Proc. of 13th International Conference on Very Large Data Bases*, Brighton, UK, September, 1987 (pp. 417-425).
- [SAH] M. Stonebraker, J. Anton and E. Hanson, Extending a database system with procedures, *ACM TODS*, 12, 3, September 1987 (pp. 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. International Conference on Management of Data*, San Jose, CA, May 1975 (pp. 65-78).
- [Wa] D.H.D. Warren, *WARPLAN: a system for generating plans*, memo 76, University of Edinburgh (1974).