Copyright © 1975, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

IMPLEMENTATION OF INTEGRITY CONSTRAINTS AND VIEWS BY QUERY MODIFICATION

by

Michael Stonebraker

Memorandum No. ERL-M514

17 March 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

IMPLEMENTATION OF INTEGRITY CONSTRAINTS AND VIEWS BY QUERY MODIFICATION

Michael Stonebraker

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

Because the user interface in a relatonal data base management system may be decoupled from the storage representation of data, novel, powerful and efficient integrity control schemes are possible. This paper indicates the mechanism being implemented in one relational system to prevent integrity violations which can result from improper updates by a process. Basically each interaction with the data is immediately modified at the query language level to one guaranteed to have no integrity violations. Also, a similar modification technique is indicated to support the use of "views," i.e. relations which are not physically present in the data base but are defined in terms of ones that are.

I INTRODUCTION

Integrity of stored data can be corrupted in at least two ways: 1) By concurrent update by two or more processes; 2) by inadvertant, improper or malicious update by a process.

The first mechanism is a well known operating system problem [1] which is addressed in [2,3] in the context of a relational data base system. In this paper we shall focus only on the second mechanism.

These corruptions can result from access violations, i.e. an unauthorized user updates the data base in an unapproved way. In a recent paper [4] we indicated that user interactions with a data base could be efficiently modified into ones guaranteed to have no access violations. However, integrity can also be destroyed by inadvertant update by an authorized user. For example, a data base containing salaries of employees might be inadvertantly updated to give some employee a negative salary. Such an update would violate a constraint which might be put on a data base that all salaries be non negative. Other possible constraints are that employees with a job classification of Assistant Professor must make between $12,000 and $16,000 and that department chairman must be full professors. In this paper we will show that a wide variety of integrity constraints can be effectively guaranteed using the same interaction modification technique indicated in [4].

We also show that support for "views" (i.e. virtual relations which are not actually present in the data base) can be handled effectively in the identical manner.

The solution of these problems at the user language level should be contrasted with lower level solutions (such as providing data base procedure calls in the access paths to data [5,6,7] where they will be called repeatedly).

The observation is made in [8] and [9] that integrity constraints should be predicates in a high level language. However, neither suggests an implementation scheme. The specification of our integrity constraints are very similar to those in [8] and [9]; however, we indicate reasonably efficient implementation algorithms. The suggestion is made in [9] that views can also be stated in a high level language. Again, our contribution is the indication of an implementation algorithm.

These mechanisms are being implemented in a relational data base system [10,11] under development (and now mostly operational) at Berkeley. This system, INGRES, must be briefly described to indicate the setting for the algorithms to be presented. Of particular relevance is the query language, QUEL, which will be discussed in the next section.

II QUEL

QUEL (QUEry Language) has points in common with Data Language/ALPHA [12], SQUARE [13] and SEQUEL [14] in that it is a complete [15] query language which frees the programmer from concern for how data structures are implemented and what algorithms are operating on stored data. As such it facilitates a considerable degree of data independence [16]. We assume that the reader is familiar with standard relational terminology [17] and now indicate the relations which will be used in the examples of this paper.

EMPLOYEE

NAME	DEPT	SALARY	MANAGER	AGE
Smith	toy	10000	Jones	25
Jones	toy	10000	Johnson	32
Adams	candy	12000	Baker	36
Johnson	toy	14000	Harding	29
Baker	admin	20000	Harding	47
Harding	admin	40000	none	58

Indicated above is an EMPLOYEE relation with domains NAME, DEPT, SALARY, MANAGER and AGE. Each employee has a manager (except for Harding who is presumably the company president), a salary, an age and is in a department.

The second relation utilized will be a DEPARTMENT relation as follows. Here, each department is on a floor, has a certain number of employees and has a sales volume in thousands of dollars.

	DEPT	FLOOR#	#EMP	SALES
DEPARTMENT	toy	8	10	1,000
	candy	1	5	2,000
	tire	1	16	1,500
	admin	4	10	0

A QUEL interaction includes at least one RANGE statement of the form;

```
RANGE OF variable-list IS relation-name
```

The symbols declared in the range statement are variables which will be used as arguments for tuples. These are called TUPLE VARIABLES. The purpose of this statement is to specify the relation over which each variable ranges.

Moreover, an interaction includes one or more statements of the form:

```
COMMAND Result-name (Target-list)
        WHERE Qualification
```

Here, COMMAND is either RETRIEVE, APPEND, REPLACE, or DELETE. For RETRIEVE and APPEND, Result-name is the name of the relation which qualifying tuples will be retrieved into or appended to. For REPLACE and DELETE, Result-name is the name of a tuple variable which, through the qualification, identifies tuples to be modified or deleted. The Target-list is a list of the form

```
Result-domain = Function, ...
```

Here, the Result-domain's are domain names in the result relation which are to be assigned the value of the corresponding function.

The following suggest valid QUEL interactions. A complete description of the language is presented in [10] and [18].

Example 2.1. Find the birth date of employee Jones.

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO W(BDATE = 1975 - E.AGE)
WHERE E.NAME = 'Jones'
```

Here, E is a tuple variable which ranges over the EMPLOYEE relation and all tuples in that relation are found which satisfy the qualification E.NAME = 'Jones'. The result of the query is a new relation, W, which has a single domain, BDATE, that has been calculated for each qualifying tuple. If the result relation is omitted, qualifying tuples are printed on the user's terminal. Also in the target list, the 'Result-domain =' may be omitted if the function is a simple domain (i.e. NAME = E.NAME may be written as E.NAME - see example 2.6).

Example 2.2. Insert the tuple (Jackson,candy,13000,Baker,30) into EMPLOYEE.

```
APPEND TO EMPLOYEE(NAME = 'Jackson', DEPT = 'candy', SALARY = 13000, MGR = 'Baker', AGE = 30)
```

Here, the result relation EMPLOYEE is formed by adding the indicated tuple to the EMPLOYEE relation.

Example 2.3. Delete the information about employee Jackson.

```
RANGE OF E IS EMPLOYEE
DELETE E WHERE E.NAME = 'Jackson'
```

Here, the tuples corresponding to all employees named Jackson are deleted from EMPLOYEE.

Example 2.4. Give a 10 percent raise to Jones.

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY 1.1 * E.SALARY)
WHERE E.NAME = 'Jones'
```

Here, E.SALARY is to be replaced by 1.1*E.SALARY for those tuples in EMPLOYEE where E.NAME = 'Jones'. (Note that the keywords IS and BY may be used interchangeably with '=' in any QUEL statement.)

Also, QUEL contains aggregation operators including COUNT, COUNT', SUM, SUM', AVG, AVG', MAX, MIN, and the set operators, SET and SET'. Two examples of the use of aggregation follow.

Example 2.5. Replace the salary of all toy department empolyees by the average toy department salary.

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY AVG'(E.SALARY WHERE E.DEPT = 'toy')) WHERE E.DEPT = 'toy'
```

here, AVG' is to be taken of the salary domain for those tuples satisfying the qualification'E.DEPT = 'toy.' Note that AVG'(E.SALARY WHERE E.DEPT = 'toy') is scalar valued and consequently will be called an AGGREGATE. For the example chosen this aggregate has the value (1/3)*(10000+10000+14000) which equals 11,333. It is sometimes useful to allow aggregates to be taken in such a way that duplicates tuples are not included. Non primed aggregates (SET, AVG, COUNT, and SUM) perform this function. For example, AVG(E.SALARY WHERE E.DEPT = 'toy') has a value 12,000.

More general aggregations are possible as suggested by the following example.

Example 2.6. Find those departments whose average salary exceeds the company wide average salary, both averages to be taken only for those employees whose salary exceeds $10000.

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO HIGHPAY(E.DEPT)
WHERE AVG'(E.SALARY BY E.DEPT WHERE E.SALARY > 10000) >
      AVG'(E.SALARY WHERE E.SALARY>10000)
```

Here, AVG'(E.SALARY BY E.DEPT WHERE E.SALARY>10000) is an AGGREGATE FUNCTION and takes a value for each of E.DEPT. This value is the aggregate AVG'(E.SALARY WHERE E.SALARY>10000 AND E.DEPT = value) as indicated below.

E.DEPT	AVG'(E.SALARY BY E.DEPT WHERE E.SALARY>10000)
toy	14000
candy	12000
admin	30000

The qualification expression for the statement is then true for departments for which this aggregate function exceeds the aggregate AVG'(E.SALARY WHERE E.SALARY>10000). The later is simply the scalar 21,500. Hence, admin is the only qualifying department.

As with aggregates, aggregate functions can have duplicates deleted with an unprimed operator.

In the sequel there will be several integrity control algorithms applied to APPEND, DELETE and REPLACE statements. Consequently, we indicate their general form and interpretation at this time.

An APPEND statement is of the following general form:

```
RANGE OF X1 IS R1
RANGE OF X2 IS R2
        .
        .
        .
RANGE OF XN IS RN
APPEND R(D1=f1,...,Dr=fr)      WHERE Q
```

Here, X1,...,XN are tuple variables over relations R1,...,RN

R is the name of the result relation and may or may not be Ri for some i.

D1,...,Dr are the names of ALL domains in R. They may be defaulted as indicated earlier.

f1,...,fr are valid QUEL alpha-functions. For a discussion of alpha-functions, see [10].

Q is a qualification statement in variables X1,...,XN, i.e. Q = Q(X1,...,XN), or a subset thereof.

Conceptually, the interpretation of an APPEND statement is the following:

1) Issue the statement

```
RANGE OF X1 is R1
RANGE OF X2 IS R2
            .
            .
            .
RANGE OF XN IS RN
RETRIEVE INTO TEMP (D1=f1,...,Dr=fr) WHERE Q
```

An error results if TEMP and R do not have identical domains. Also, if Q is absent and f1,...,fr are simply the domains from a given relation, then step 1) would be a copy operation. Hence, it need not be done in this case.

2) Perform a set union of TEMP and R into R

The general form of a DELETE statement is the following:

```
RANGE OF X1 IS R1
RANGE OF X2 IS R2
            .
            .
            .
RANGE OF XN IS RN
DELETE X1 WHERE Q
```

The interpretation is the following:

1) Issue the statement

```
RANGE OF X1 IS R1
RANGE OF X2 IS R2
            .
            .
            .
RANGE OF XN IS RN
RETRIEVE INTO TEMP(X1.ALL) WHERE Q
```

Here, ALL is a keyword indicating all domains.

2) Perform the relative complement of R1 and TEMP into R1

The general form of a REPLACE statement is the following:

```
RANGE OF X1 IS R1
RANGE OF X2 IS R2
            .
            .
            .
RANGE OF XN IS RN
REPLACE X1(D1=f1,...,Ds=fs)  WHERE Q
```

Here, D1,...,Ds are a subset of the domains in R1.

The interpretation is the following:

1) Issue the statement

```
RANGE OF X1 IS R1
RANGE OF X2 IS R2
            .
            .
            .
RANGE OF XN IS RN
RETRIEVE INTO TEMP(X1.TID,f1,...,fs) WHERE Q
```

Here, TID is a tuple identifier which is guaranteed to be unique to a tuple.

2) For each tuple in TEMP, obtain the tuple in R1 identified by TID, substitute f1,...,fs for D1,...,Ds and replace the tuple. Should there be more than one tuple in the TEMP with a given TID, the update is

NON-FUNCTIONAL and is aborted. This problem is discussed further in [3].

A RETRIEVE statement is processed by breaking it into a sequence of RETRIEVE statements each of which involves only a single tuple variable. This decomposition is discussed in [10] and a similar one in [19]. These single variable queries involve only one relation and can be directly executed (in the worst case by a sequential scan of the relation tuple by tuple). Often the relation will be stored in such a way that a complete scan is not needed. Also, secondary indices which can profitably be used to speed access are utilized.

The actual processing of update commands follows the general flavor indicated above. However, where possible, the creation of TEMP is avoided and steps 1 and 2 performed at once.

III INTEGRITY CONSTRAINTS

For each data base we allow ASSERTIONS to be stored. Each assertion is (logically) a RANGE statement and a valid QUEL qualification in variables specified in the RANGE statement. This qualification is true or false for each tuple in the Cartesian product of the relations specified by variables in the RANGE statement. In the next four sections we indicate algorithms which guarantee that the qualification is TRUE for all tuples in the product space after each update. The general mechanism is to modify each user interaction so that updates which violate an assertion are disallowed. In the remainder of this section we indicate examples of possible assertions.

Example 3.1. Employee salaries must be positive

```
RANGE OF E IS EMPLOYEE
INTEGRITY E.SALARY>0
```

Example 3.2. Everyone in the toy department must make more than $8000

```
RANGE OF E IS EMPLOYEE
INTEGRITY E.SALARY>8000 OR E.DEPT ≠ 'toy'
```

Example 3.3. Employees must earn less than ten times the sales volume of their department if their department has a positive sales

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPARTMENT
INTEGRITY E.SALARY < 10*D.SALES
            OR E.DEPT ≠ D.DEPT
            OR D.SALES = 0
```

Example 3.4. No employee can make more than his manager

```
RANGE OF E,M IS EMPLOYEE
INTEGRITY E.SALARY ≤ M.SALARY
            OR E.MANAGER ≠ M.NAME
```

Example 3.5. Harding must make more than twice the average employee salary

```
RANGE OF E IS EMPLOYEE
INTEGRITY E.NAME ≠ 'Harding'
            OR E.SALARY > 2*AVG' (E.SALARY)
```

Example 3.6. Name must be a primary key

```
RANGE OF E IS EMPLOYEE
INTEGRITY COUNT(E.TID) = COUNT(E.NAME)
```

Example 3.7. Floor# is functionally dependent [20] on DEPT

```
RANGE OF D IS DEPT
INTEGRITY COUNT(D.FLOOR# BY D.DEPT) = 1
```

There will be four algorithms of increasing complexity (and cost) for dealing with:

1) one variable aggregate-free assertions as in Examples 3.1-3.2.

2) multivariate aggregate-free assertions with only one tuple variable on the relation being updated as in Example 3.3.

3) multivariate aggregate-free assertions with two or more tuple variables on the relation being updated as in Example 3.4.

4) assertions involving aggregates as in Examples 3.5-3.7.

We deal with each case individually in the next four sections. For all sections we deal with APPEND, DELETE and REPLACE in their general form indicated in Section 2. Hence, notation introduced in Section 2 will be used throughout.

IV ENFORCEMENT OF ONE VARIABLE AGGREGATE-FREE ASSERTIONS

Intuitively, a one variable aggregate free assertion specifies a condition which is true or false for each tuple in a relation. Hence, integrity assurance reduces to checking each tuple that is inserted or modified to ensure the truth of the assertion. Tuples may, of course, be deleted with no checking whatsoever. As a result, a DELETE statement can be processed with no regard for such integrity constraints.

The following algorithm must be applied to APPEND statements.

ALGORITHM 1

a) Find all one variable aggregate-free assertions with RANGE OF Y IS R for some tuple variable Y. Call the corresponding qualifications Q1(Y),...,Qh(Y).

b) Replace Q, the given qualification, by Q AND Q* where

```
Q* = Q1(f1,...,fr)
     AND Q2(f1,...,fr)
     AND
       .
       .
       .
     AND Qh(f1,...,fr)
```

Here, Qj(f1,...,fr) results from Qj(Y) by replacing Y.Dk by fk whenever Y.Dk appears in Qj.

The APPEND statement can be processed normally assured that R will satisfy all one variable aggregate free assertions after execution. The result of the RETRIEVE portion of the algorithm to execute APPEND statements will contain only tuples which satisfy the constraints; R, of course, satisfies the constraints; hence, the set union of R and this result will also.

The tuples which are not APPENDED to R because of an integrity violation can be easily found as follows:

```
RANGE OF X1 IS R1
RANGE OF X2 IS R2
              .
              .
              .
RANGE OF XN IS RN
RETRIEVE INTO MISTAKE(D1 = f1,...,Dr = fr) WHERE Q AND NOT Q*
```

The algorithm for REPLACE statements differs only slightly from the algorithm above.

Algorithm 2

a) same as step a) for Algorithm 1

b) Replace Q by Q AND Q* for which Qj(f1,...,fs) is formed by replacing Y by X1 then by replacing X1.Dk by fk wherever X1.Dk appears in Qj.

The tuples which cannot be altered are found by:

```
RANGE OF X1 IS R1
RANGE OF X2 IS R2
              .
              .
              .
RANGE OF XN IS RN
RETRIEVE INTO MISTAKE(X1.ALL) WHERE Q AND NOT Q*
```

Two examples illustrate these algorithms at work. Here, we enforce the constraint on positive salaries given in Example 3.1.

Example 4.1. Insert the tuple (Jackson, candy, 13000, Baker, 30) into EMPLOYEE

```
APPEND TO EMPLOYEE(NAME = 'Jackson', DEPT = 'candy',
          SALARY = 13000, MGR = 'Baker', AGE = 30)
```

This is exactly Example 2.2 and becomes after application of the algorithm:

```
APPEND TO EMPLOYEE(NAME = 'Jackson', DEPT = 'candy',
          SALARY = 13000, MGR = 'Baker', AGE = 30)
              WHERE 13000>0
```

Hence, Jackson's tuple will be added only if he has a positive salary. Tuples disallowed by the integrity constraint would be found by:

```
RETRIEVE INTO MISTAKE(NAME = 'Jackson', DEPT = 'candy',
   SALARY = 13000, MGR = 'Baker', AGE = 30)
       WHERE NOT 13000>0
```

In fact, the following format issues both statements at once:

```
APPEND TO EMPLOYEE(NAME = 'Jackson', DEPT = 'candy',
          SALARY = 13000, MGR = 'Baker', AGE = 30)
ERRORS TO MISTAKE
```

Example 4.2. Give a 500 dollar paycut to Jones

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = E.SALARY-500) WHERE E.NAME = 'Jones'
ERRORS TO MISTAKE
```

Upon modification, this becomes:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = E.SALARY - 500) WHERE E.NAME = 'Jones'
          AND E.SALARY-500 > 0

RETRIEVE INTO MISTAKE(E.ALL) WHERE E.NAME = 'Jones'
          AND NOT E.SALARY-500 > 0
```

V ENFORCEMENT OF MULTIVARIATE AGGREGATE-FREE ASSERTIONS, I

Here, we consider the case that all assertions have two or more tuple variables but only one ranging over the relation being updated. In this case each tuple which is inserted or modified will add or change many tuples in the product space for which the assertion must be guaranteed. As a result the algorithms in this section are more complex than previously. Note, however, that tuples may still be deleted from a relation with no checking.

The following algorithm must be applied to APPEND statements:

Algorithm 3

a) Find all multivariable assertions which contain RANGE OF Y IS R for some Y. Let these qualifications be Q1,...,Qh.

If f1,...,fr are simple domains with only a single tuple variable, say Xm, then do b), otherwise do b1) and b2).

b) Append Q* to the users qualification where Q*=Q1* AND ... AND Qh* and where Qi* is found as follows. Let Qi be qualification in variables Y, U1,...,Uq, i.e. Qi = Qi(Y,U1,...,Uq). Then Qi* is:

```
COUNT(U1.TID,...,Uq.TID BY Xm.TID
          WHERE Qi(Xm,U1,...,Uq))
          =
COUNT(U1.TID,...,Uq.TID)
```

b1) Do step 1 of the algorithm to process APPEND statements therefy creating TEMP (as noted in Section 2).
b2) Issue the interaction

```
RANGE OF Z IS TEMP
APPEND INTO R(Z.ALL) WHERE Q*
```

Here, Q* is the qualification Q1* AND ... AND Qh* where Qi* is:

```
COUNT(U1.TID,...,Uq.TID BY Z.TID
    WHERE Qi(Z, U1,...,Uq))
          =
COUNT(U1.TID,...,Uq.TID)
```

The algorithm for REPLACE statements is the following.

Algorithm 4

a) Find all multivariate assertions which contain RANGE OF Y IS R for some Y. Let these qualification be Q1,...,Qh.

b) For the i-th qualification, let Qi have tuple variables Y, U1,...,Uq, i.e. Qi = Qi(Y, U1,...,Uq).

c) Replace Q, the given qualification, by Q and Q1* AND ... AND Qh* where Qi* is:

```
COUNT(U1.TID,...,Uq.TID BY X1.TID
     WHERE Qi(f1,...,fs, U1,...,Uq))
           =
COUNT(U1.TID,...,Uq.TID)
```

Here, note that Qi(f1,...,fs, U1,...,Uq) is Qi with Y replaced by X1 then X1.Dk replaced by fk wherever it appears.

The following examples illustrates these algorithms.

Suppose one wants to enforce the constraint of Example 3.3 that an employee must earn less than 10 times the sales volume of his department if sales volume is positive.

Suppose the employee tuple for Jackson is in a relation W and is to be added to EMPLOYEE as follows.

Example 5.1.

```
RANGE OF Y IS W
APPEND TO EMPLOYEE(Y.ALL) WHERE Y.NAME = 'Jackson'
```

Upon modification this becomes:

```
RANGE OF Y IS W
RANGE OF D IS DEPARTMENT
APPEND TO EMPLOYEE(Y.ALL) WHERE Y.NAME = 'Jackson'

      AND    COUNT(D.TID BY Y.TID
             WHERE Y.SALARY < 10*D.SALES
               OR Y.DEPT ≠ D.DEPT
               OR D.SALES = 0)
                    =
             COUNT(D.TID BY Y.TID)
```

The algorithm applied to Example 4.2 now follows.

Example 5.2. Give a 500 dollar paycut to Jones

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = E.SALARY-500) WHERE E.NAME = 'Jones'
```

Upon modification this becomes:

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPARTMENT
REPLACE E(SALARY = E.SALARY-500) WHERE E.NAME = 'Jones'

      AND   COUNT(D.TID BY E.TID
            WHERE E.SALARY-500 < 10*D.SALES
              OR E.DEPT ≠ D.DEPT
              OR D.SALES = 0
                   =
            COUNT(D.TID BY E.TID)
```

VI ENFORCEMENT OF MULTIVARIATE AGGREGATE-FREE ASSERTIONS, II

We now consider the case of assertions such as Example 3.4 which contain two or more tuple variables ranging over the relation to be updated. This situation differs from the cases considered previously in

the following respect. In effect, integrity control was exercised by examining each tuple to be updated, allowing updates for those tuples satisfying the assertions and denying them otherwise. Unfortunately, updates subject to the assertions considered in this section must be allowed or disallowed as a whole, and decisions cannot be made incrementally. The following example illustrates the problem which arises.

Consider the combination of two relations on employees (which might happen if two companies merge) i.e.

```
RANGE OF N IS NEWEMP
APPEND TO EMPLOYEE(N.ALL)
```

Moreover, suppose one wants to enforce the constraint of Example 3.4, i.e. that each employee must make less than his manager. Lastly, suppose most of all of the employees in the relation NEWEMP violate this condition.

Now, suppose one inserts tuples from NEWEMP into EMPLOYEE in an order such that each employee is inserted before his manager. Each employee who is not a manager can be inserted without a violation while each manager will not be allowed. On the other hand, if managers are inserted first, at least one will satisfy the constraints while all non managers will fail. Hence, the order in which the tuples are inserted will affect which ones are in violation of the constraints. Since ordering of tuples in a relation should not affect the outcome of any operation, one must treat an update subject to this form of integrity constraint as an entity and allow or disallow the whole procedure. Consequently, the algorithms are somewhat different than those in the previous sections.

It can again be noted that DELETES can be processed with no checking; the integrity assurance algorithm for APPENDS now follows.

Algorithm 5

a) Find all multivariate assertions which have two or more tuple variables ranging over R. Let these qualifications be Q1,...,Qh.

b) Do step one of the algorithm to process APPEND statements, thereby creating TEMP

c) Issue the interaction:

```
RANGE OF Z IS TEMP
APPEND TO R(Z.ALL) WHERE Q*
```

Here, Q* = Q1* AND ... AND Qh* and where Qi* is found as follows. Let Qi have tuple variables Y1,...,Ym, U1,...,Uq where Yj ranges over R and Uj does not for all j.

Then Qi* is the following:

```
 AND  . . .  AND     COUNT(V1.TID,...,Vm.TID, U1.TID,...,Uq.TID
V1 in        Vm in
{R,TEMP}     {R,TEMP}          WHERE Qi(V1,...,Vm, U1,...,Uq))

                                  =

                        COUNT(V1.TID,...,Vm.TID, U1.TID,...,Uq.TID)
```

The reader should note several points concerning this algorithm:

1) the COUNT term when Vi=R for all i can be eliminated since R satisfies the constraint before the update.

2) when Vi=TEMP for all i, only one of the m permutations need be included since the rest would be redundant.

3) new tuple variables are required in the second APPEND statement because of the introduction of Q*.

4) aggregates appear in this algorithm instead of the aggregate functions in algorithms 3 and 4. In this way Q* has either the value TRUE or FALSE and the update as a whole is either allowed or disallowed as a result.

The reader can also note the changes which must be made to create a working algorithm for REPLACE statements. We now indicate an example of the algorithm at work ensuring Example 3.4.

Example 6.1. Add Jackson to EMPLOYEE

```
RANGE OF Y IS W
APPEND TO EMPLOYEE(Y.ALL) WHERE Y.NAME = 'Jackson'
```

Upon modification this becomes:

```
        RANGE OF Y IS W
        RETRIEVE INTO TEMP(Y.ALL) WHERE Y.NAME = 'Jackson'

        RANGE OF T, T2 IS TEMP
        RANGE OF E IS EMPLOYEE
        APPEND TO EMPLOYEE(T.ALL) WHERE

        COUNT(T.TID,T2.TID WHERE
             T.SALARY < T2.SALARY
               OR T.MANAGER ≠ T2.NAME)
          -
        COUNT(T.TID, T2.TID)

AND     COUNT(T.TID, E.TID WHERE
             T.SALARY < E.SALARY
               OR T.MANAGER ≠ E.NAME)
          -
        COUNT(T.TID, E.TID)

AND     COUNT(E.TID, T.TID WHERE
             E.SALARY < T.SALARY
               OR E.MANAGER ≠ T.NAME)
          -
        COUNT(E.TID,T.TID)
```

VII CONSTRAINTS INVOLVING AGGREGATES

The reader can note that constraints involving aggregates have the same problem that occurred with the previous class of constraints, namely updates must be allowed or disallowed as a whole. Again the reason is that the tuples which violate the constraints depend on the order in which they are changed or added. There is, however, a more serious problem.

For example, the assertion AVG'(X.SALARY) < 14000 might be applied to the following update.

```
        RANGE OF Y IS W
        APPEND INTO EMPLOYEE(Y.ALL)
```

as follows

```
        RANGE OF Y IS W
        RANGE OF E IS EMPLOYEE
        APPEND INTO EMPLOYEE(Y.ALL) WHERE
                     SUM'(E.SALARY) +SUM'(Y.SALARY)
                     ------------------------------  < 14000
                     COUNT(E.TID) + COUNT(Y.TID)
```

In this fashion the revised average salary would be computed and checked for the integrity constraint during the update. Unfortunately, there may be tuples in W which are also in EMPLOYEE.

If so, the APPEND statement will, of course, delete the duplicate tuples when it performs a set union of EMPLOYEE and W. However, the added qualification is, in effect, the integrity statement with the duplicates present. There is no way in QUEL to express the fact that the constraint should be taken with duplicates deleted.

Therefore, the algorithm for constraints involving aggregates must be to try the update, test the resulting relation for the integrity constraint and then undo the update if one is not satisfied.

VIII EFFICIENCY CONSIDERATIONS

The addition of single variable aggregate-free integrity constraints will usually result in the same decomposition to a sequence of one-variable queries that would have resulted otherwise. Each such one variable query is further qualified by one or more integrity qualifications. Such one variable queries are usually at least as efficient to process as those without constraints. In fact, the added qualification may be employable to speed access. Hence, the cost of integrity for one variable aggregate-free assertions should be negligible.

Unfortunately, this is not the case for the other forms of constraints. All involve testing for equality, pairs of aggregates or aggregate functions. These operations are usually very costly. Consequently, the user may enforce more complex controls but only at considerable cost.

Note that our algorithms generally have the effect of testing constraints for only small subrelations on each update. Of course, this is to be preferred to examining the whole relation each time.

Also, if controls are desired at each update, we believe the proper approach is to append them at as high a level as possible. In this way checks in the access paths are avoided, and any information available can be utilized to perform the update as efficiently as possible. Note also that schemes which append integrity controls at lower levels have considerable difficulty enforcing complex controls (such as those involving more than a single tuple variable).

Lastly, note that the power of RETRIEVE statements can also be used to ascertain the truth of integrity constraints. Thus, users who do not wish to pay the price of checking each update may less frequently make their own checks and take appropriate action.

IX SUPPORT FOR VIEWS

"Views" or virtual relations are relations which do not physically exist in the data base but may be definable in terms of ones which do exist.

One such view might be the relation EMPTOY with domains NAME, SALARY, AGE, and DEPT defined as follows:

```
RANGE OF E IS EMPLOYEE
DEFINE EMPTOY(NAME=E.NAME,SALARY=E.SALARY,AGE=E,AGE,DEPT=E.DEPT)
        WHERE E.DEPT = 'toy'
```

Note again that defaults could have been used for the names of the domains in EMPTOY.

In INGRES any user is allowed to define views for his own use. Moreover, the data base administrator can define views which apply to others. The syntax of a DEFINE statement is identical to a RETRIEVE statement and is parametrically:

```
RANGE OF X1 IS R1
RANGE OF X2 IS R2
            .
            .
            .
RANGE OF XN IS RN
DEFINE VIEWNAME(D1=f1,...,Dj=fj)  WHERE Q
```

Note that the RANGE statement of a view can involve a relation which is itself a view. Views are supported for two reasons.

1) user convenience as a 'MACRO' facility

2) the stored relations may change over time and views allow previous relations to be defined in terms of current ones. Hence, programs written for previous versions of the data base can continue to be supported (with certain restrictions to be discussed presently).

Our view algorithm now follows.

Algorithm 6

For each tuple variable V specified in a user interaction which ranges over a view defined by a target list Tv and qualification Qv:

1) Delete V from the RANGE statement of the user interaction and add all tuple variables in the view definition (modifying them to have unique names if necessary).

2) If VIEWNAME is the relation in a DELETE statement from which tuples will be deleted, then replace V by Xm in the Result-name portion of the interaction if f1,...,fj have only a single tuple variable, say Xm. Otherwise, abort the command.

Similarly, if VIEWNAME is the relation to which tuples are to be added by an APPEND statement and if f1,...,fj involve only a single tuple variable, say Xm, then replace VIEWNAME by Rm, the relation over which Xm ranges, as the result relation. If f1,...,fj involve more than one tuple variable, abort the interaction in this situation.

Lastly, in REPLACE statements for which V indicates that VIEWNAME is to be modified, then append V to each domain name to the left of an equals sign in the target list; do step 6 of the algorithm; then factor out the tuple variable again. If more than one tuple variable results to the left of an equals sign then abort the command.

3) Abort all REPLACE statements which update a domain appearing in Qv. In this case several problems are

present. One is the possibility of updating a tuple in such a way that it is deleted from the view (for example by updating DEPT in the previous example). This anomoly should not be allowed.

4) Append

```
AND Qv
```

to the user's qualification.

5) Append Qv to the qualificaiton portion of any aggregate or aggregate function which contains V unless V appears as part of the BY argument in the qualification portion of the user's interaction. In this case the variable is not local to the aggregate in question. Hence, step 4 appropriately conditions such functions.

6) Replace each domain V.Dj in the user interaction with fj from the target list of the view definition.

The following points should be carefully noted concerning this algorithm:

1) The algorithm translates interactions on views into interactions on real relations.

2) The resulting interaction may be syntactically illegal. For example, APPEND and DELETE statements may be so modified that TEMP created in step 1 of the execution algorithm (see Section 2) does not have the correct domains. Such an interaction will be aborted automatically.

3) In step 2 of the view algorithm an abort occurs because a SINGLE interaction on a view must be translated into MORE THAN ONE interaction on real relations. Such a translation is, in general, impossible as noted in [21].

The following examples illustrate the algorithm at work.

Example 9.1 Give Jones a 10 percent raise

```
RANGE OF Y IS EMPTOY
REPLACE Y(SALARY = 1.1*Y.SALARY)  WHERE Y.NAME = 'Jones'
```

Upon application of algorithm 6 this becomes:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = 1.1*E.SALARY)
      WHERE      E.NAME = 'Jones' AND E.DEPT = 'toy'
```

Example 9.2

The following statements define a second view of EMPLOYEE.

```
RANGE OF E IS EMPLOYEE
DEFINE EMPOTHER(NAME=E.NAME, PROGRESS=E.SALARY/E.AGE)
```

The update

```
RANGE OF O IS EMPOTHER
REPLACE O(PROGRESS=1.1*O.PROGRESS)
```

becomes

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY/AGE =1.1*E.SALARY/E.AGE)
```

which is syntactically illegal and is aborted.

Example 9.3

The following statements define a view involving both EMPLOYEE and DEPARTMENT.

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPARTMENT
DEFINE COMBEMP(NAME=E,NAME, SALARY=E.SALARY, FLOOR#=D.FLOOR#)
     WHERE E.DEPT=D.DEPT
```

The update

```
RANGE OF C IS COMBEMP
REPLACE C(FLOOR#=3) WHERE C.NAME = 'Jones'
```

becomes

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPARTMENT
REPLACE D(FLOOR#=3)
      WHERE E.NAME = 'Jones' AND E.DEPT=D.DEPT
```

which is processed correctly.

The following table summarizes the actions of the view algorithm.

	(A) The view is a restriction[15] of an existing relation	(B) f1,...,fj involve a single tuple variable but (A) is not satisfied	(C) f1,...,fj involve more than a single relation
	(Example 8.1)	(Example 8.2)	(Example 8.3)
REPLACE (tuples in the view)	correct (unless disallowed by 3)	correct (unless disallowed by 3 or syntactically illegal)	correct (unless disallowed by 2 or 3 or syntactically illegal)
APPEND (to the view)	correct	syntactically illegal (point 2)	disallowed by 2
DELETE (from the view)	correct	syntactically illegal (point 2)	disallowed by 2
RETRIEVE (and all other updates)	correct	correct	correct

The notion of "correct" is the following . If the view were materalized (by replacing the DEFINE with a RETRIEVE and issueing the query) and the interaction were processed; the result would be the same as would be obtained by applying the view algorithm to the interaction, then processing the interaction which results and materalizing the view.

Note that this algorithm processes all RETRIEVES and many REPLACES correctly; APPENDS and DELETES are usually disallowed. Moreover, it is easy to implement and involves the same sort of processing needed for protection and integrity constraints. Note lastly that more elaborate algorithms are possible which will handle more cases than algorithm 6. In fact, supporting APPENDS and DELETES for certain views in the second column above does not appear difficult. However, the difficulty of handling most other disallowed cases appears very great, and in some cases there may be no possible procedure.

X SUMMARY

The advantages of these integrity control and view support algorithms are briefly recapitulated here.

1) In both cases control is placed at the source language level. As such, access control, integrity checks and support for views can all be accomplished at once. Also, at this level the algorithms are conceptually simple and easy to implement. This should be contrasted with lower level schemes (such as [2,5]).

2) Little storage space is required to store integrity assertions and view definitions.

3) These algorithms involve small overhead at least in the simpler (and presumably more common) cases.

ACKNOWLEDGEMENT

Research sponsored by the National Science Foundation Grant GK-43024X and the Naval Electronic Systems Command Contract N00039-75-C-0034.

REFERENCES

[1] Brinch Hansen, P., Operating Systems Principles, Prentice Hall, Englewood Cliffs, N.J., 1973.

[2] Chamberlin, D., et al. "A Deadlock-Free Scheme for Resource Locking in a Data Base Environment," IBM Research Laboratory, San Jose, Ca., March 1974.

[3] Stonebraker, M., "High Level Integrity Assurance in Relational Data Base Systems," University of California, Electronics Research Laboratory, Memorandum M473, August 1974.

[4] Stonebraker, M. and Wong, E., "Access Control in a Relational Data Base Management System by Query Modification," Proc. 1974 ACM National Conference, San Diego, Ca., Nov. 1974.

[5] Committee on Data Systems Languages, "Data Description Language," U.S. Dept. of Commerce, National Bureau of Standards, Handbook #112, January, 1974.

[6] Hoffman, L., "The Formulary Model for Flexible Privacy and Access Control," Proc. 1971 Fall Joint Computer Conference, Las Vegas, Nev., November 1971.

[7] Fossum, B., "Data Base Integrity as Provided for by a Particular Data Base Management System," Proc. 1974 IFIP Conference on Data Base Management Systems, Cargese, Corsica, April 1974.

[8] Florentin, J., "Consistency Auditing of Data Bases," The Computer Journal, vol. 17, no. 1, February 1974.

[9] Chamberlin, D. and Boyce, R., "Using a Structured English Query Language as a Data Definition Facility," IBM Research Report RJ 1318, San Jose, Ca. December 1973.

[10] Held, G., et al., "INGRES - A Relational Data Base System," Proc. 1975 National Computer Conference, Anaheim, Ca., May 1975. (to appear).

[11] Held, G. and Stonebraker, M., "Storage Structures and Access Methods in the Relational Data Base Management System, INGRES," Proc. ACM-PACIFIC-75 San Francisco, Ca., April 1975.

[12] Codd, E., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca., November 1971.

[13] Boyce, R. et al., "Specifying Queries as Relational Expressions: SQUARE," Proc. ACM SIGPLAN-SIGIR Interface Meeting, Gaithersberg, Md., November 1973.

[14] Chamberlin, D. and Boyce, R., "SEQUEL: A Structured English Query Language," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[15] Codd, E., "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposium, May 1971.

[16] Stonebraker, M., "A Functional View of Data Independence," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich, May 1974.

[17] Codd, E., "A Relatonal Model of Data for Large Shared Data Banks," CACM, 13 6 (June 1970).

[18] McDonald, N. and Stonebraker, M., "CUPID - A Friendly Query Language," Proc. ACM-PACIFIC-75, San Francisco, Ca., April 1975.

[19] Rothnie, J., "An Approach to Implementing a Relational Data Base Management System," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[20] Codd, E., "Normalized Data Base Structure: A Brief Tutorial," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca. November 1971.

[21] Codd, E., "Recent Investigations in Relational Data Base Systems," Information processing '74, North Holland, 1974.