Copyright © 1974, 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.

HIGH LEVEL INTEGRITY ASSURANCE IN RELATIONAL DATA BASE MANAGEMENT SYSTEMS

by

M. Stonebraker

Memorandum No. ERL-M473

16 August 1974

HIGH LEVEL INTEGRITY ASSURANCE IN RELATIONAL DATA BASE MANAGEMENT SYSTEMS

by

Michael Stonebraker

Memorandum No. ERL-M473

16 August 1974

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

HIGH LEVEL INTEGRITY ASSURANCE IN RELATIONAL DATA BASE MANAGEMENT SYSTEMS

by

Michael Stonebraker

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

ABSTRACT

Because the user interface in a relational 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 both from improper or malicious updates by a single process and from concurrent update of the data base by two or more processes. Basically, each interaction with the data base is immediately modified to one which is guaranteed to have no integrity violations of the first kind at the query language level. Potential violations of the second type are detected and resolved at the same level.

Research sponsored by the Naval Electronics Systems Command Contract N00039-71-C-0255 and Air Force Office of Scientific Research Contract F44620-71-C-0087.

I. INTRODUCTION

Integrity of stored data can be corrupted in at least two ways:

(1) By inadvertant, improper or malicious update by a process.

(2) By concurrent update of data items by two or more processes.

The first mechanism can result from access violations, i.e., an unauthorized user alters the data base in an unapproved way. In a recent paper [1] we indicated that user interactions with a data base could be efficiently modified into ones guaranteed to have no such access violations. However, data base 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 the data base that all salaries be nonnegative. Other possible constraints are that employees with a job classification of Assistant Professor must make between $12,000 and $16,000 and that department chairmen 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 [1].

A second way that integrity can be compromised, by concurrent update, is well known in operating systems and several solutions have been proposed including locks, semaphores and conditional critical sections [2]. However, as pointed out in [3], data bases present more difficult concurrency problems than typically addressed by the above mechanisms, both because of the large number of possible locks and the need to guarantee the truth of complex conditions. In this paper, we indicate that prevention of

integrity violations while allowing a high degree of concurrency can be accomplished efficiently and simply at the user language level. Moreover, our solution prevents deadlock [3,4,5] from ever occurring. Thus, not only does a process never need to be backed up but also no code to test for deadlocks need ever be included. Lastly, our solution has the property that the concurrency allowed is directly proportional to time invested in attempting to ascertain the truth of various conditions about interactions. Hence, CPU time can be traded in a natural way for concurrent use of secondary storage and each data base system can individually decide what investment to make.

The solution of both problems at the user language level should be contrasted with lower level solutions such as providing locks and data base procedure calls in the access paths to data [6,7,8] where they will be called repeatedly. In our scheme integrity checks are performed just once.

The observation is made in [9] and [10] that integrity constraints of the first kind should be predicates in a high level language. However, neither suggests an implementation scheme. The specification of our constraints of type 1 are very similar to these in [9] and [10]; however, we indicate reasonably efficient implementation algorithms. The suggestion that predicates could be used for integrity constraints of the second type is attributed to Jim Gray of IBM Corporation.

Both mechanisms are being implemented in a relational data base system [11] under development 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 briefly discussed in the next section.

II. QUEL

QUEL 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]. Since basic entities in QUEL are relations, we define them and indicate the sample relations which will be used in the examples in this paper.

Given sets $D_1,\ldots,D_n$ (not necessarily distinct) a relation $R(D_1,\ldots,D_n)$ is a subset of the Cartesian product $D_1 x D_2 x \ldots x D_n$. In other words, R is a collection of n-tuples $x = (x_1,\ldots,x_n)$ where $x_i \in D_i$ for $1 \leq i \leq n$. The sets $D_1,\ldots,D_n$ are called domains of R and R has degree n. The only restriction put on relations in QUEL is that they be normalized [17]. Hence, every domain must be simple i.e., it cannot have members which are themselves relations.

Clearly, R can be thought of as a table with elements of R appearing as rows and with columns labeled by domain names as illustrated by the following example.

	NAME	DEPT	SALARY	MANAGER	AGE
EMPLOYEE	Smith	toy	10,000	Jones	25
	Jones	toy	15,000	Johnson	32
	Adams	candy	12,000	Baker	36
	Johnson	toy	14,000	Harding	29
	Baker	admin	20,000	Harding	47
	Harding	admin	40,000	none	58

The above indicates 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.

Each column in a tabular representation for R can be thought of as a function mapping R into D_i. These functions will be called attributes. An attribute will not be separately designed but will be identified by the domain defining it.

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	B	10	1,000
	candy	1	5	2,000
	tire	1	16	1,500
	admin	4	10	0
	complaints	2	3	0

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

RANGE Relation Name (Symbol,...,Symbol):
Relation Name (Symbol,...,Symbol):
.
.
.
Relation Name (Symbol,...,Symbol)

The symbols declared in the RANGE statement are variables which will be used as arguments for attributes. These are called tuple variables. The purpose of the statement is to specify the relation over which each variable ranges.

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

```
⎧ RETRIEVE ⎫
⎪ REPLACE  ⎪
⎨ COMBINE  ⎬ Result Relation:  Target List:  Qualification
⎩ DELETE   ⎭
```

The following suggest valid QUEL statements. A complete description of the language is presented in Appendix 1.

Example 2.1

Find the salary of the employee Jones.

```
RANGE EMPLOYEE(X)
RETRIEVE W:X.SALARY:X.NAME = Jones
```

Here, X is a tuple variable ranging over the EMPLOYEE relation and all tuples in that relation are found which satisfy the qualification X.NAME = Jones. The salary attribute for all qualifying tuples is put in a workspace named W.

Example 2.2

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

```
RANGE EMPLOYEE(X) : TERMINAL(Y)
COMBINE EMPLOYEE:ALL(X,Y)::(Jackson, candy, 13000, Baker, 30)
```

Here, the result relation EMPLOYEE is formed by combining EMPLOYEE and the indicated tuple. TERMINAL is a reserved symbol indicating the user's terminal. The two colons before the tuple indicate that the qualification field for this statement is blank.

Example 2.3

Delete the information about employee Jackson

```
RANGE EMPLOYEE(X)
DELETE EMPLOYEE:X.ALL:X.NAME = Jackson
```

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

Example 2.4

Give a 10% raise to Jones

RANGE EMPLOYEE(X)
REPLACE EMPLOYEE:X.SALARY = 1.1*X.SALARY:X.NAME = Jones

Here, X.SALARY is to be replaced by 1.1*X.SALARY for those tuples in EMPLOYEE where X.NAME = Jones. Note that 1.1*X.SALARY is an A-function. Valid A-functions include those with any of the QUEL operators and any attributes.

Also, QUEL contains aggregation operators including COUNT, SUM, MAX, MIN, AVE and the set operator SET. An example of the use of aggregation follows.

Example 2.5

Replace the salary of all employees in the toy department by the average toy department salary.

RANGE EMPLOYEE(X)
REPLACE EMPLOYEE: X.SALARY = AVE(X. SALARY;;X.DEPT = toy): X.DEPT = toy

Here AVE is to be taken of the salary attribute for those tuples satisfying the qualification X.DEPT = toy.

Note that AVE(X.SALARY;;X.DEPT = toy) is scalar valued and consequently will be called an aggregate. 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 employees whose salary exceeds $10000.

RANGE EMPLOYEE(X)

RETRIEVE W: X.DEPT: AVE(X,SALARY;X.DEPT;X.SALARY>10000)>AVE(X.SALARY;;
X.SALARY>10000)

Here AVE(X.SALARY;X.DEPT;X.SALARY>10000) is an aggregate function and takes a value for each value of X.DEPT. This value is the aggregate AVE(X.SALARY;;X.SALARY>10000 $\wedge$X.DEPT = value). The qualification statement then is true for those departments for which this aggregate function exceeds AVE(X.SALARY;;X.SALARY>10000). Syntactically, note that an aggregate is simply an aggregate function with a null second argument.

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

A COMBINE statement is of the following general form.

RANGE $W_1(X_1)$: ...: $W_N(X_N)$
COMBINE W: D_1, ..., D_j(R): Q

Here X_1, ..., X_N are tuple variables over relations W_1, ..., W_N

R is a subset of $\{X_1, \ldots, X_N\}$.

D_1, ..., D_j are domain names in each relation specified by each tuple variable in R.

Q is a qualification statement in variables X_1, ..., X_N, i.e., $Q = Q(X_1, \ldots, X_N)$, or a subset thereof.

W is a relation not necessarily distinct from W_1, ...,W_N.

Note that D_1, ..., D_j may be replaced by the keyword "ALL" if all domains are meant.

Conceptually, the interpretation of a COMBINE statement is the following. For each tuple of the cartesian product $C = W_1 \times W_2 \times \ldots \times W_N$, Q is a qualification statement that is either true or false. For the sub-relation of C that satisfies Q, project on the domains D_1, ..., D_K for each tuple variable in R. Merge these relations and name the result W.

Of course, the actual processing of a COMBINE statement should streamline this process considerably. Our algorithms for processing COMBINE, DELETE and REPLACE statements are discussed in a subsequent section.

The general form of a DELETE statement is the following.

RANGE $W_1(X_1)$: ...: $W_N(X_N)$
DELETE W: $X.D_1$, ..., $X.D_K$: Q

Here X_1, ..., X_N are tuple variables ranging over relations W_1, ..., W_N.

W is a relation not necessarily distinct from W_1, ..., W_N.

X is a tuple variable ranging over W or W_i for some i.

D_1, ..., D_K are domains in the relation specified by X

Q is a qualification statement in variables X_1, ..., X_N, i.e.,

$$Q = Q(X_1, \ldots, X_N), \text{ or a subset thereof.}$$

Conceptually, Q specifies a qualification that is true or false for each tuple in $C = W_1 x \ldots x W_N$. For the subrelation of C not satisfying Q, project on domains D_1, ..., D_K of the relation specified by X. Name this relation W.

The general form of a REPLACE statement is:

RANGE $W_1(X_1)$: ...: $W_N(X_N)$
REPLACE W: $X.D_1 = \alpha_1$, ..., $X.D_j = \alpha_j$:Q

Here α_1, ..., α_j are valid QUEL A-functions

X, D_1, ... D_j, Q, W are as above.

The interpretation of this statement is the following. Find all tuples in W_1 x ... x W_N satisfying Q. For each such tuple calculate α_1, ... α_j and replace $X.D_1$, ... $X.D_j$ appropriately. Project the resulting relation on $X.D_1$, ... $X.D_j$ and name it W, if X does not range over W. Otherwise project on all domains of W.

Note that REPLACE statements have one potentially undesirable property which will be illustrated by example.

Example 2.7

Replace the salaries of all toy department employees by the salary of their manager

RANGE EMPLOYEE(X,Y)

REPLACE EMPLOYEE: X.SALARY = Y.SALARY: X.MANAGER = Y.NAME $\wedge$ X.DEPT = toy

Suppose, however, that there are two managers of the toy department. In this case, one desires to replace each toy department salary by two other salaries. The above algorithm handles this situation by creating two tuples for each employee, one with each salary. Obviously, this is undesirable. This problem arises when a REPLACE statement does not specify a function on the tuples of the relation indicated by X. Instead of restricting the syntax of a REPLACE statement to guarantee this functionality (for example, by restricting $\alpha_1, \ldots, \alpha_j$ to be functions only of X) we simply detect non functionality during processing and abort the command.

Note for DELETE and REPLACE statements the usual situation will be that X ranges over W. In this case an existing relation is to be modified. If X ranges over another relation W_i then some subset of the tuples in W_i are to be modified and copied into W. Similarly, the usual case for COMBINE statements will be that a tuple variable in R ranges over W. This situation is appropriate for insertions into existing relations.

It can be noted that DELETE and REPLACE statements can always be expressed equivalently as a sequence of one or more COMBINE and RETRIEVE commands. This redundancy is supported for the convenience of the users.

III. ENFORCEMENT OF INTEGRITY CONSTRAINTS OF THE FIRST TYPE

In this section we indicate the mechanism to enforce constraints concerning the ways a process may update a data base.

For each data base we allow <u>assertions</u> to be stored. Each assertion is (logically) a RANGE statement and a valid QUEL qualification expression in variables specified in the RANGE statement. This qualification expression 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 <u>true</u> 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.

<u>Example 3.1</u>

Employee salaries must be non-negative:

RANGE EMPLOYEE(X)
X.SALARY≥0

<u>Example 3.2</u>

Everybody except Harding must have a manager

RANGE EMPLOYEE(X)
X.MANAGER ≠ none ∨ X.NAME = Harding

<u>Example 3.3</u>

Everybody in the toy department must make at least $8000.

RANGE EMPLOYEE(X)
X.SALARY≥8000 ∨ X.DEPT ≠ toy

<u>Example 3.4</u>

Employees must earn less than 10 times the sales volume of their department if their department has a positive sales.

RANGE EMPLOYEE(X):DEPARTMENT(Y)
(X.SALARY<10*Y.SALES) ∨ (X.DEPT ≠ Y.DEPT) ∨ (Y.SALES = 0)

Example 3.5

No employee can make more than his manager

RANGE EMPLOYEE(X,Y)

X.SALARY<Y.SALARY ∨ (X.MANAGER ≠ Y.NAME)

There are two types of qualifications. Those not containing aggregates as above and those with aggregates as below.

Example 3.6

Harding must make more than twice the average employee salary.

RANGE EMPLOYEE(X)

X.NAME ≠ Harding ∨ X.SALARY>2*AVE(X.SALARY)

Example 3.7

Name must be a primary key

RANGE EMPLOYEE(X)

COUNT'(X.NAME) = COUNT(X.ID)

Here the ' indicates that duplicates are to be deleted before the enumeration. Also X.ID is a tuple ID which is guaranteed unique.

Example 3.8

FLOOR # is functionally dependent [18] on DEPT

RANGE DEPARTMENT(Y)

COUNT (Y.DEPT;;COUNT(Y.FLOOR#;Y.DEPT) ≠ 1) = 0

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

1. one-variable aggregate-free assertions as in examples 3.1 - 3.3.
2. multivariate aggregate-free assertions with only one tuple variable on the relation being updated as in example 3.4.

3. multivariate aggregate-free assertions with two or more tuple variables on the relation being updated as in example 3.5.

4. assertions involving aggregates as in examples 3.6 - 3.8.

We deal with each case individually in the next four sections. For all sections we deal with COMBINE, DELETE and REPLACE statements in their general form indicated previously in Section 2. One could argue that integrity constraints should be applied only when an existing relation is updated (i.e., when $W = W_i$ for some i) and not when a new relation is created (when $W \neq W_i$ for all i). The additional difficulty of allowing constraints in the second case is small so we see no reason to prohibit them.

IV. ENFORCEMENT OF ONE-VARIABLE AGGREGATE FREE ASSERTIONS

Intuitively, a one-variable aggregate-free constraint specifies an assertion which is true or false for each tuple of 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 whatever.

Hence a DELETE statement can be processed with no regard for such integrity constraints if X ranges over W (in which case tuples are to be deleted from an existing relation). Otherwise, a new relation is to be formed and the DELETE statement will be converted to an equivalent form:

RANGE $W_1(X_1)$, ..., $W_N(X_N)$
COMBINE W: D_1, ..., $D_K(X)$: $\neg Q$

The following algorithm can be applied to this and all other COMBINE statements.

Algorithm 1

a. Find all one-variable aggregate-free assertions with a RANGE statement of W(Y) for some tuple variable Y. Call the corresponding qualifications $Q_1(Y), \ldots, Q_h(Y)$

b. Replace Q, the qualification given in the COMBINE statement, by $Q \wedge Q^*$ where $Q^* = \bigwedge_{P \in R} \bigwedge_{\ell \in \{1,\ldots,h\}} Q_\ell(P)$ and $Q_\ell(P)$ is the qualification Q_ℓ with the tuple variable Y replaced by P. Here, $\bigwedge_{P \in R}(\ )$ means the logical AND of the terms resulting from P iterating over R.

The COMBINE statement can now be processed normally assured that W will satisfy all one-variable aggregate-free assertions after execution. The tuples which do not satisfy the conditions and hence are not added to W may be found as follows.

RANGE $W_1(X_1), \ldots, W_N(X_N)$
COMBINE TEMP: All (R): $Q \wedge \neg Q^*$

Appropriate action may now be taken for tuples in TEMP. Note that when a variable in R ranges over W, qualifications in that variable can be absent from Q* since W will assuredly satisfy the assertions before the update.

The following algorithm effectively deals with the common case of REPLACE statements where X ranges over W (in which case an existing relation is to be modified).

Algorithm 2

a. Find all assertions with a RANGE statement of W(Y) for some Y. Call the appropriate qualifications $Q_1, \ldots, Q_h$.

b. Replace Q by $Q \wedge Q^*$ where $Q^* = \bigwedge_{i=\{1,\ldots,h\}} Q_i(\alpha_1, \ldots, \alpha_j)$. Here $Q_i(\alpha_1, \ldots, \alpha_j)$ results from $Q_i(Y)$ by substituting X for Y and then α_ℓ for $X.D_\ell$, wherever $X.D_\ell$ appears in Q_i.

The tuples not altered because of the integrity constant can be found as follows

RANGE $W_1(X_1)$: ...: $W_N(X_N)$
RETRIEVE Z: X.All: $Q \wedge \neg Q^*$

In the much less common case that X does not range over W, the processing is more complex because tuples from another relation are moved into W and some are altered in the process. Algorithm 2 would insure that tuples which had values changed would satisfy the assertions. However, tuples not changed can not be checked in this fashion. An overly conservative (but simple) strategy which insures the assertions is to first execute

RANGE $W_1(X_1)$: ...: $W_N(X_N)$
COMBINE W: All(X)

and then execute the REPLACE statement with X ranging over W. This policy is overly cautious because the COMBINE statement may be modified to disallow tuples that would have subsequently been changed by the REPLACE statement to have valid values. In other words the end result of the REPLACE statement may satisfy the assertions yet the intermediate result, that of the COMBINE statement, may not.

To overcome this deficiency, algorithm 1 or 2 can be applied to each of the following statements to guarantee the assertions.

RANGE $W_1(X_1)$, ..., $W_N(X_N)$
COMBINE W: D_1, ..., $D_K(X)$: $\neg Q$
RETRIEVE TEMP: X.ALL, TL: Q
RANGE TEMP(Y): W(Z)
REPLACE TEMP: $Y.D_1 = \alpha_1$, ... $Y.D_K = \alpha_K$
COMBINE W: $D_1, \ldots, D_K(Y,Z)$

Here TL represents any additional attributes on which $\alpha_1, \ldots, \alpha_K$ depend.

Two examples illustrate these algorithms. Consider the enforcement of the constraint X.SALARY$\geq$0 as in Example 3.1

Suppose a user issues the following

RANGE EMPLOYEE(X) : T(Y)
COMBINE EMPLOYEE:All(X,Y): Y.NAME = Jones

Here Jones record is to be added to EMPLOYEE (assuming that it is in the relation T).

This statement will be automatically modified by algorithm 1 to

RANGE EMPLOYEE(X) : T(Y)
COMBINE EMPLOYEE: All (X,Y): Y.NAME = Jones $\wedge$ Y.SALARY$\geq$0

Hence Jones' record will be added only if he has a non-negative salary. Tuples disallowed by the integrity constraint can be found by

RANGE EMPLOYEE(X):T(Y)
COMBINE TEMP: All(Y): Y,NAME = Jones $\wedge\neg$ Y.SALARY$\geq$0.

In fact the following format issues both statements at once.

RANGE EMPLOYEE(X):T(Y)
COMBINE EMPLOYEE All(X,Y): X.NAME = Jones: ERRORS to TEMP

Suppose another user issues the statement

RANGE EMPLOYEE (X)
REPLACE EMPLOYEE:X.SALARY = X.SALARY - 500: X.NAME = Jones: ERRORS to TEMP

This will be expanded to

RANGE EMPLOYEE(X)
REPLACE EMPLOYEE: X. SALARY = X. SALARY - 500:
X.NAME = Jones $\wedge$X.SALARY - 500$\geq$0
RETRIEVE TEMP:X.All: X.NAME = Jones$\wedge\neg$ X.SALARY - 500$\geq$0

V. ENFORCEMENT OF MULTI-VARIABLE AGGREGATE FREE ASSERTIONS, I

Here, we consider the case that all assertions have two or more

tuple variables but only one ranging over W. In this case each tuple which is inserted or modified in W 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. Therefore DELETE's can be handled in the same way as previously.

The following algorithm must be applied to COMBINE statements.

Algorithm 3

a. Find all multivariate assertions which contain W(Y) for some Y. Let these qualifications be $Q_1, \ldots, Q_h$.

b. For the i-th qualification let Q_i be a qualification in variables $Y, U_1, \ldots, U_q$, i.e. $Q_i = Q_i(Y, U_1, \ldots, U_q)$

c. Replace Q by $Q \wedge [\bigwedge_{P \in R}$ COUNT $(U_1.ID, \ldots, U_q.ID; P.ID; Q_i(P)) =$ COUNT$(U_1.ID, \ldots, U_q.ID)]$.

d. Repeat c for each i

Here $Q_i(P)$ is Q_i with the variable indicated by P replacing Y wherever it appears. Again if X_i ranges over W for some i, it can be deleted from the variables considered in step c since W satisfies the assertion at the start of the update.

The algorithm for REPLACE statements in which X ranges over W is the following.

Algorithm 4

a. Find all multivariate assertions which contain W(Y) for some Y. Let these qualifications be $Q_1, \ldots, Q_h$

b. For the i-th qualification let Q_i have tuple variables $Y, U_1, \ldots, U_q$, i.e., $Q_i = Q_i(Y, U_1, \ldots, U_q)$

c. Replace Q by Q $\wedge$ COUNT(U_1.ID, ..., U_q.ID; X.ID; $Q_i(\alpha_1, \ldots \alpha_K,$ $U_1, \ldots, U_q)$) = COUNT (U_1.ID, ... U_q.ID)

d. Repeat c for each i

Here note that $Q_i(\alpha_1, \ldots, \alpha_K, U_1, \ldots, U_q)$ is Q_i with Y replaced by X and then X.D_j replaced by α_j wherever it appears.

The previous complications concerning REPLACE statements when X does not range over W are present in this situation. The same mechanism applied in Section IV must also be applied here.

The following examples illustrate these algorithms.

Suppose one wishes to enforce the constraint of example 3.4 that an employee must earn less than 10 times the sales volume in thousands of his department if sales is positive. The previous two examples will be restated to conform to this constraint.

```
RANGE EMPLOYEE(X) : T(Z)
COMBINE EMPLOYEE: All(X,Z): Z.NAME = Jones:
ERRORS to TEMP
```

becomes

```
RANGE EMPLOYEE(X) : T(Z) : DEPARTMENT(Y)
COMBINE EMPLOYEE:All(X,Z): Z.NAME = Jones ∧
   COUNT(Y.ID; Z.ID; (Z.SALARY < 10 * Y.SALARY) ∨ (Z.DEPT ≠ Y.DEPT) ∨
(Y. SALES = 0)) =  COUNT(Y.ID)
COMBINE TEMP: All(Z): Z.NAME = Jones ∧ COUNT(Y.ID; Z.ID;
(Z.SALARY < 10 * Y.SALARY) ∨ (X.DEPT ≠ Y.DEPT) ∨ (Y.SALES = 0)) ≠
COUNT(Y.ID)
```

The replace statement

```
RANGE EMPLOYEE(X)
REPLACE EMPLOYEE:X.SALARY = X.SALARY - 500:X.NAME = Jones
```

becomes

```
RANGE EMPLOYEE(X) : DEPARTMENT(Y)
REPLACE EMPLOYEE:X.SALARY = X.SALARY - 500:
     X.NAME = Jones ∧ COUNT(Y.ID; X.ID; (X.SALARY - 500 < 10 *
     Y.SALARY) ∨ (X.DEPT ≠ Y.DEPT) ∨ (Y.SALES = 0)) = COUNT(Y.ID)
```

VI. ENFORCEMENT OF MULTIVARIATE AGGREGATE FREE ASSERTIONS, II

We now consider the case of assertions, such as Example 3.5, which contain two or more tuple variables ranging over W. This situation differs from the cases considered above 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 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 EMPLOYEE(X), EMPLOYEE1(Y)
COMBINE EMPLOYEE: All(X,Y)
```

Moreover, suppose one wishes to enforce the constraint of Example 3.5, i.e., that each employee makes less than his manager. Lastly, suppose most or all of the employees in the relation EMPLOYEE1 violate this condition.

Now, suppose one inserts tuples from EMPLOYEE1 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 tuples are inserted will

affect which ones are not 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 previous sections.

It can easily be noted that DELETE's can be processed in the same manner as before. The integrity assurance algorithm for COMBINE's now follows.

Algorithm 5.

a. Find all multivariate assertions which have two or more tuple variables ranging over W. Let these qualifications be $Q_1 \ldots, Q_h$.

b. For the i-th qualification let Q_i have variables $Y_1 \ldots, Y_\ell$, $U_1, \ldots, U_k$ where Y_i ranges over W and U_i does not for all i.

c. Replace Q by Q $[_{\gamma_1 \in R \cdots \gamma_\ell \in R}$ COUNT $(\gamma_1.\mathrm{ID}, \ldots \gamma_\ell.\mathrm{ID};;$ $Q_i(\gamma_1, \ldots, \gamma_\ell, U_1, \ldots, U_k) =$ COUNT $(\gamma_1.\mathrm{ID}, \ldots, \gamma_\ell.\mathrm{ID})]$.

d. Repeat c for each i.

The reader can note if X_m ranges over W for some m that the COUNT term when $\gamma_i \neq X_m$ for all i can be eliminated since W satisfies the constraint before the update. Also note that when $\gamma_i = X_j$ for all i only one of the ℓ permutations need be included since the rest would be redundant. Also, when $\gamma_i = \gamma_m$ for some i,m, X_j can be assigned to γ_i and a new variable ranging over W_j must be assigned to γ_m in order that the constraint be correctly stated. Note finally that aggregates appear in this algorithm instead of the aggregate functions in algorithm 3. In this way the added qualification has either the value TRUE or FALSE and the update as a whole is allowed or disallowed as a result.

The reader can easily note the changes that must be made to create a working algorithm for REPLACE statements. We now indicate an example of the algorithm at work ensuring example 3.5.

The statement

RANGE EMPLOYEE(X) : T(Z)
COMBINE EMPLOYEE All(X,Z)

becomes

RANGE EMPLOYEE(X):T(Y,Z)
COMBINE EMPLOYEE All(X,Z):
COUNT(X.ID, Z.ID;;X.SALARY < Z.SALARY $\vee$ X.MANAGER $\neq$ Z.NAME) = COUNT(X.ID, Z.ID) $\wedge$ COUNT(Z.ID, X.ID ;;Z.SALARY < X.SALARY $\vee$ Z.MANAGER $\neq$ X.NAME) = COUNT(Z.ID, X.ID) $\wedge$ COUNT(Z.ID, Y.ID ;;Z.SALARY < Y.SALARY $\vee$ Z.MANAGER $\neq$ Y.NAME) = COUNT(Z.ID, Y.ID)

VII. CONSTRAINTS INVOLVING AGGREGATES

The reader will note that constraints involving aggregates have the same problem that occurred with the previous class of constraints: updates must be allowed or disallowed as a whole. Again, the reason is that the tuples which do not violate the constraints depend on the order in which they are changed or added. What is more perilous is that the previous mechanisms cannot be applied to allow or disallow the whole update.

For example, the assertion AVE(X.SALARY) $\leq$ 500 might be applied to the following update

RANGE EMPLOYEE(X):W(Y)
COMBINE EMPLOYEE: All(X.Y)

as follows

RANGE EMPLOYEE(X):W(Y)
COMBINE EMPLOYEE:All(X,Y): $\dfrac{\text{SUM(X.SALARY) + SUM(Y.SALARY)}}{\text{COUNT(X.ID) + COUNT(Y.ID)}} \leq 500$

In this fashion, the revised average salary would be computed and checked for the integrity constraint. Unfortunately, there may be tuples

in EMPLOYEE which are also in W. If so, the COMBINE statement will, of course, delete the duplicates. However, the appended qualification is, in effect, the integrity statement with the duplicates present. There is no easy way to express in QUEL the fact that the aggregate constraint is taken with duplicates deleted. Therefore the algorithm involving aggregates must simply be to try the update, test the resulting relation for the integrity constraints and undo the update if one is not satisfied.

VIII. EFFICIENCY CONSIDERATIONS

Efficiency considerations can only be discussed in view of the strategy employed to decompose QUEL interactions.

a. DELETE

If X ranges over W the following statement is issued

RANGE $W_1(X_1)$: ...: $W_N(X_N)$
RETRIEVE TEMP: X.All: Q

Tuples in TEMP are then deleted one by one from W by calls to the appropriate access method.

If X ranges over another relation, a DELETE is translated to a COMBINE statement as previously indicated.

b. COMBINE

For each tuple variable X_i in R the following statement is issued

RANGE $W_1(X_1)$: ...: $W_N(X_N)$
RETRIEVE $TEMP_i$: $X_i.D_1$, ..., $X_i.D_j$: Q

The set union of the resulting $TEMP_i$'s is taken and named W. If Q does not depend on X_ℓ, for some tuple variable X_ℓ in R, the procedure can be shortened by omitting the RETRIEVE for X_ℓ.

c. REPLACE

If X ranges over W, the statement

RANGE $W_1(X_1)$: ...: $W_N(X_N)$
RETRIEVE TEMP: X.All, TL: Q

is executed. Here TL stands for any attributes not in W on which $\alpha_1, \ldots, \alpha_j$ depend. For each tuple in TEMP the projection on the domains of W is deleted from W, a substitution for the α-functions performed, and a new typle inserted into W.

If X ranges over another relation, the sequence of statements indicated earlier is performed.

A RETRIEVE statement is processed by breaking it into a sequence of RETRIEVE statements each of which involves only a single tuple variable. As such it resembles the one used in [19]. These single variable queries involve only a single 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 redundant indices which can be used profitably to speed access are utilized.

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 result otherwise. Each such one-variable query is further qualified by one or more integrity qualifications. Such one-variable RETRIEVE's are usually at least as efficient to process as those without constraints. In fact the added clauses may be employed in speeding access. Hence the cost of integrity for one-variable aggregate-free controls should be negligible.

Unfortunately, that 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 for 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 can be avoided and any information available can be utilized to perform the update as efficiently as possible. Also note that schemes which append integrity at lower levels have considerable difficulty enforcing complex controls (such as those involving more than a single 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. CONCURRENCY CONTROL

The following example illustrates the problem that arises during concurrent update of a data base by two or more processes.

```
U1   RANGE EMPLOYEE(X)
     REPLACE EMPLOYEE: X.DEPT = toy: X.DEPT = shoe
U2   RANGE EMPLOYEE(Y)
     REPLACE EMPLOYEE: Y.DEPT = shoe: Y.DEPT = toy
```

Notice that if U1 precedes U2 then all employees end up in the shoe department, while if U2 precedes U1, all finish in the toy department. If they are processed concurrently some employees may end up in each department, and the particular results may not be repeatable. This

illustrates two unsafe updates. The notion of unsafe is defined formally as follows.

For a relation $W \subset D_1 \times \ldots \times D_n$ define $C(W)$ to be $D_1 \times D_2 \times \ldots \times D_n$. Any update, U, in the QUEL language can be thought of as a function

$U: C(W) \times C(W_1) \times \ldots \times C(W_n) \rightarrow C(W)$

Here W is the result relation and $W_1, \ldots, W_N$ are the relations specified in a RANGE statement.

Consider two such updates

$U: C(W) \times C(W_1) \times \ldots \times C(W_n) \rightarrow C(W)$
$V: C(S) \times C(S_1) \times \ldots \times C(S_m) \rightarrow C(S)$

Denote by J the product space $C(W) \times C(S) \times C(W_1) \times \ldots \times C(W_n) \times C(S_1) \times \ldots \times C(S_m)$ and uniquely extend U and V to U' and V' such that

$U',V': J \rightarrow J$

Lastly, let $\{^{U'}_{V'}\}$ (J) represent the (perhaps simultaneous) application of U' and V' to J. Now, two updates U and V will be said to be <u>safe</u> if

$\{^{U'}_{V'}\} (J) = U' \circ V' (J)$ or $V' \circ U' (J)$

Consequently, two updates are safe if their outcome is the same as would result from their sequential execution in either order. Note that other less restrictive notions of safety exist. One such notion is pursued in [11].

A date base management system that guarantees all updates are safe will be said to <u>preserve integrity for concurrent updates</u>. It is evident that safety can be assured by processing updates sequentially with no parallelism. However, performance can be improved by finding updates that are assuredly safe and processing them concurrently. There are at least two approaches that can be taken. First, a process may apply a lock or semaphore to any entity (data item, tuple, page, file) which it will alter or whose alteration by another process might compromise the

safety of the update. Using this approach update U1 must ensure that the DEPT column of the EMPLOYEE relation is locked and must set whatever locks are required to do this. Hence, U2 could not proceed concurrently since it must also lock the same column. An algorithm along these lines is suggested in [3]. This approach has two disadvantages. First, a set of perhaps millions of locks must be managed if locks are to be on small enough entities to allow considerable concurrency. Second, deadlock is possible and must be detected and one or more processes backed up, if necessary. The second approach is to implement concurrency control at the user language level. It will be seen that both of the deficiencies of the storage locking scheme can be avoided. In order to present our algorithm, we must indicate the actual sequence of functions that are performed. In the remainder of this section we consider only updates where $W = W_i$ for some i in which case an existing relation is being updated. Otherwise a new relation is being created and no concurrency can be allowed without compromising safety.

In INGRES such updates are processed by executing one or more RETRIEVE statements followed by a sequence of operations each of which inserts or deletes a tuple from W. Consider the result of these RETRIEVE statements for a given update, U, as $TEMP^U_1$, ..., $TEMP^U_K$ and denote these retrievals formally as

$$U_R: C(W) \times C(W_1) \times \ldots \times C(W_n) \rightarrow C(TEMP^U_1) \times \ldots \times C(TEMP^U_K)$$

Then, a sequence of operations U_1, ..., U_h is performed on W. Each involves the addition or deletion of a single tuple. Consider each as function

$$U_i: C(W) \times C(TEMP^U_1) \times \ldots \times C(TEMP^U_K) \rightarrow C(W)$$

Hence an update, U, is the sequence of functions

$$U_R, U_1, \ldots, U_h .$$

Similarly, a second update V is the sequence of functions

$V_R, V_1, \ldots, V_g$

$V_R: C(S) \times C(S_1) \times \ldots \times C(S_m) \rightarrow C(TEMP_1^V) \times \ldots \times C(TEMP_K^V)$

$V_i: C(S) \times C(TEMP_1^V) \times \ldots \times C(TEMP_K^V) \rightarrow C(S)$

Denote by J* the cartesian product $C(S) \times C(S_1) \times \ldots \times C(S_m) \times C(TEMP_1^V)$

$\times \ldots \times C(TEMP_K^V) \times C(W) \times C(W_1) \times \ldots \times C(W_n) \times C(TEMP_1^U) \times \ldots \times C(TEMP_K^U)$

Uniquely extend $U_R, U_1, \ldots, U_h, V_R, V_1, \ldots V_g$ to

$U'_R, U'_1, \ldots, U'_h, V'_R, V'_1, \ldots, V'_g: J^* \rightarrow J^*$

Assume only that the operating system guarantees that $U_1, \ldots, U_h$ and $V_1, \ldots, V_g$ are atomic operations. This requires that the access methods can do a read-modify-write sequence from a physical page of memory without interruption or that locks or semaphores can be applied to single physical pages for this interval. If so, two updates U and V will be safe if

1. $U'_R[V'_i V'_{i-1} \ldots V'_1(J^*)] = U'_R(J^*)$ for all i
2. $V'_R[U'_i U'_{i-1} \ldots U'_1(J^*)] = V'_R(J^*)$ for all i

The proof of this statement appears in Appendix 2. Note that the two conditions specify that no alteration of the data base by one update can affect the set of qualifying tuples for the other update. Stated differently no process can alter the data base so that tuples either enter or leave the set to be changed by the other process. Also, they do not allow two REPLACE statements to change the same tuple. An easily demonstrated consequence of 1 - 2 is that $U \circ V(J) = V \circ U(J)$. We now turn to conditions that guarantee 1 - 2.

For any update, U, (COMBINE, DELETE, REPLACE) denote by Q_u the qualification expression and by $L(Q_u)$ the set of attributes in Q_u. Let

T_u be the list of variables in the target list (for DELETEs the list given, for COMBINEs the list given duplicated for each tuple variable in R, and for RETRIEVEs those appearing in an A-function or being replaced by an A-function).

A. Two updates U,V (COMBINE, DELETE, REPLACE) are safe if

a_1. $T_u \wedge L(Q_v) = \emptyset$

a_2. $T_v \wedge L(Q_u) = \emptyset$

a_3. $Q_u \wedge Q_v = \emptyset$ or $Q_v \wedge \neg Q_u = Q_v$ or $Q_u \wedge \neg Q_v = Q_u$

Note that $Q_u \wedge Q_v$, Q_v, Q_u specify conditions true or false on J. Note also that certain cases always fail statement A such as a COMBINE and a REPLACE on the same relation and a COMBINE and a DELETE on the same relation.

Two other points should be made about statement A. First, there are two cases which fail statement A yet are automatically safe. They are two DELETE's with the same W and two COMBINE's with the same W. Secondly, safety only requires a_3 in other special cases. One such case is two DELETE's on different relations.

Other more elaborate conditions for safety are also possible. However they appear infeasible to check quickly. Hence, only statements A will be investigated in INGRES. The first two conditions in A are simple to check; however, a_3 may be exceedingly complex and its truth data dependent. Nevertheless, there are cases where a_3 is easy to demonstrate. These include the case where tuple variables in Q_u and Q_v do not range over the same relation and where Q_u and Q_v consist of clauses of the form attribute = value for which the attributes are the same but the values differ. We consider now our integrity algorithm.

Algorithm 6

For an arriving update U such that updates $V_1, \ldots, V_n$ are currently in progress

a. Attempt to demonstrate statement A for $(U,V_1), \ldots, (U,V_n)$

b. If a. is satisfied add U to the list of updates in progress; exit.

c. If a. fails add U to a wait list for each update for which statement A cannot be proved.

For an update V which finishes execution

a. Delete V from those updates in progress

b. Activate as arriving updates any elements only on V's wait list.

This algorithm has at least two potential disadvantages

1. Updates may wait a very long time

2. The algorithm blocks updates which cannot be shown to be safe and hence may be too conservative.

The following two modifications improve both situations.

1. Attach a count field to each update. Let the count field be decremented by one on each activation. When the count reaches 0 execute step d instead of step c of the previous algorithm

   d. If a. fails add U to a wait list for each update for which statement A cannot be proved. In addition add U to the list of those updates in progress.

Notice that the next time the update is activated it is guaranteed not to conflict with any updates in progress and can assuredly proceed. Hence, the count reflects the number of times an update can be forced to activate itself before being allowed to proceed. Note that the count can be

made larger for those updates which should get a lower priority or which are judged to tie up a large portion of the data base (for example, updates with several tuple variables).

The second improvement concerns updates for which a_3 cannot be shown because its truth is data dependent. Consider, for example, the following two updates

```
RANGE EMPLOYEE(X)
REPLACE EMPLOYEE: X.SALARY = 1.1 * X.SALARY:
                  X.AGE > 39
REPLACE EMPLOYEE: X.SALARY = .9 * X.SALARY:
                  X.AGE ≤ 40
```

These updates are unsafe only if some employee is 40 years old. There may be updates of this sort (especially complex ones) which one might wish to process in parallel on the good chance that they do not conflict. Modification 2 allows such a possibility.

2. Suppose a REPLACE, U, arrives and can be shown not to conflict with all updates in progress except V for which a_3 cannot be shown.

In this case add U to the update list and issue two RETRIEVE statements:

RANGE $W_1(X_1)$, ..., $W_N(X_N)$
RETRIEVE T1: T: Q_u
RETRIEVE T2: T: $Q_u \wedge \neg Q_v$

The first statement is simply the RETRIEVE portion of U (which would be done anyway). The second is similar except tuples satisfying Q_v are excluded. When both RETRIEVEs finish, T1 and T2 can be tested for equality. If so, U and V are safe and U can finish its update unimpaired. Otherwise U must enter the wait state until V finishes.

It should be noted that an update is never started until its safety with all other updates in progress is assured. This scheme avoids deadlock and therefore guarantees that an update need never be backed up to resolve a deadly embrace. This is in contrast to locking strategies applied at lower levels which suffer this drawback. Also only the target list, qualification list, and qualification statement for each update need be manipulated. This should be contrasted with the difficulty of managing storage locks.

X. EFFICIENCY

The efficiency of this scheme depends primarily on how "smart" the program ascertaining safety can be made. Certainly a tradeoff exists between the time required to execute a sophisticated theorem prover and the time saved by finding non obvious safe pairs of updates. However, it should be noted that statement A is trivially true for two updates of non primary key [18] attributes of tuples specified by a primary key. Hence, concurrency is allowed in common situations where it is possible.

Consider now updates involving more than one tuple variable. Since it may be difficult to prove safety for multivariable updates, there may be little parallelism in this situation. Fortunately, we expect this case to be rare. Hence, less than the most possible parallelism may be acceptable.

The overhead of modification 2 may not be as severe as one might expect since both RETRIEVE statements access many of the same tuples. In a virtual memory environment in which the statements are executed by

separate processes with shared relations, it may be that few extra page faults are caused by the addition of the second RETRIEVE statement. Consequently, the overhead may be much less than the factor of two one might otherwise suspect. This overhead may be unattractive for simple updates where the option is to wait a short time. However, for complicated updates, especially ones with more than one tuple variable which may take a long time, it may be reasonable.

XI. SUMMARY

The advantages of these integrity control schemes are briefly recapitulated here.

- 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 [3]).
- Little storage space is required to store integrity assertions and necessary information about interactions in progress.
- Only minimal synchronization need be done in the access methods to support concurrent updates.
- Deadlock is avoided so no code for checking this condition need be included.
- These algorithms involve small overhead at least in the simpler more common cases.

REFERENCES

1. 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.

2. Brinch Hansen, P., Operating Systems Principles, Prentice Hall, Englewood Cliffs, N.J., 1973.

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

4. Coffman, E. et al., "System Deadlocks," Computing Surveys, vol. 3, no. 2, June 1971.

5. Havender, J., "Avoiding Deadlocks in Multilasking Systems," IBM Systems Journal, vol. 7, no. 2, 1968.

6. CODASTL, "Data Description Language", Handbook #112, U.S. Department of Commerce, January, 1974.

7. Everest, G., "Concurrent Update Control and Data Base Integrity," 1974 IFIP Conference on Data Base Management Systems, Cargèse Corsica, April 1974.

8. Nolman, J., "Data Base Integrity as Provided for by a Particular Data Base Management System," Proc. 1974 IFIP Conference on Data Base Management Systems, Cargèse Corsica, April 1974.

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

10. Boyce, R. and Chamberlin, D., "Using a Structured English Query Language as a Data Definition Facility," IBM Research Laboratory, San Jose, Calif., RJ 1318.

11. MacDonald, N., Stonebraker, M. and Wong, E., "Preliminary Specification of INGRES", University of California, Electronics Research Laboratory, Memorandum #M435-436. April 1974.

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

13. Boyce, R., _et al_, "Specifying Queries as Relational Expressions: SQUARE", IBM Research, San Jose, Ca., RJ 1291.

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.F., "Relational Completeness of Data Base Sublanguages", Courant Computer Science Symposium _6_, May, 1972.

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.F., "A Relational Model of Data for Large Shared Data Banks", _CACM_, Vol. 13 #6, June, 1970.

18. Codd, E.F., "Normalized Data Base Structures: A Brief Tutorial", Proc. 1971 ACM-SIGFIDET Workshop on Data Description Access and Control, San Diego, Ca., November, 1971.

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.

APPENDIX 1 - Syntax of QUEL

For simplicity the language is described without allowed precedence altering parentheses and assumes standard precedence conventions.

INTERACTION SEQUENCE	=	RANGE STATEMENT INTERACTION SEQUENCE
	=	RETRIEVE STATEMENT INTERACTION SEQUENCE
	=	COMBINE STATEMENT INTERACTION SEQUENCE
	=	DELETE STATEMENT INTERACTION STATEMENT
	=	REPLACE STATEMENT INTERACTION STATEMENT
	=	DESIGNATE STATEMENT INTERACTION SEQUENCE
	=	ϕ
RANGE DECLARATION	=	<u>RANGE</u> RELATION-NAME (VARIABLE LIST) :...: RELATION-NAME (VARIABLE LIST)
VARIABLE LIST	=	VARIABLE,...,VARIABLE
DESIGNATE DECLARATION	=	<u>DESIGNATE</u> NAME = A-FUNCTION :...: NAME = A-FUNCTION
RETRIEVE STATEMENT	=	<u>RETRIEVE</u> : RESULT-NAME : TARGET LIST : QUALIFICATION
TARGET LIST	=	TARGET LIST, NAME

= TARGET LIST, A-FUNCTION

= ϕ

QUALIFICATION = ¬ QUALIFICATION

= QUALIFICATION V CLAUSE

= QUALIFICATION Λ CLAUSE

= CLAUSE

= ϕ

CLAUSE = A-FUNCTION * A-FUNCTION

= A-FUNCTION * A-FUNCTION * A-FUNCTION

= SET CLAUSE

* = = | > | ≥ | < | ≤

A-FUNCTION = ATTRIBUTE FUNCTION

= AGGREGATE FUNCTION

= A-FUNCTION $ A-FUNCTION

= LOG CONSTANT (A-FUNCTION)

= # A-FUNCTION

$ = + | - | * | / | **

= + | -

ATTRIBUTE FUNCTION = CONSTANT | ATTRIBUTE

AGGREGATE FUNCTION = $\sum^*$(SET FUNCTION)

= $\sum$(SET FUNCTION)

= $\sum^*$(ATTRIBUTE SEQUENCE;
ATTRIBUTE SEQUENCE: QUALIFICATION)

= $\sum$(A-FUNCTION; ATTRIBUTE SEQUENCE:
QUALIFICATION)

ATTRIBUTE SEQUENCE = ATTRIBUTE FUNCTION, ..., ATTRIBUTE FUNCTION

A-SEQUENCE = A-FUNCTION, ..., A-FUNCTION

ATTRIBUTE = VARIABLE.DOMAIN NAME

$\sum^*$ = COUNT | COUNT'

Σ	=	SUM | SUM' | AVG | AVG' | MAX | MIN
SET CLAUSE	=	SET FUNCTION @ SET FUNCTION
SET FUNCTION	=	SET(A-FUNCTION; ATTRIBUTE SEQUENCE; QUALIFICATION)
	=	SET FUNCTION & SET FUNCTION
@	=	$= \mid \subset \mid \subseteq \mid \supset \mid \supseteq$
&	=	$\wedge \mid \vee \mid \sim$
COMBINE STATEMENT	=	COMBINE RESULT NAME : TL : QUALIFICATION
TL	=	ATTRIBUTE SEQUENCE (VARIABLE LIST)
DELETE STATEMENT	=	DELETE RESULT NAME : ATTRIBUTE SEQUENCE : QUALIFICATION
REPLACE STATEMENT	=	REPLACE RESULT NAME : SUBSTITUTION : QUALIFICATION
SUBSTITUTION	=	ϕ
	=	SUBSTITUTION, ATTRIBUTE = A-FUNCTION

Terminal Symbols

KEY WORDS	RANGE, DESIGNATE, RETRIEVE COMBINE, DELETE, REPLACE
NAMES	
CONSTANTS	
VARIABLES	
LOGICAL CONNECTIVES	$\vee, \wedge, \neg$
COMPARISON OPERATORS	$=, \geq, >, \leq, <$
SPECIAL FUNCTIONS	LOG
ARITHMETIC OPERATORS	+, -, *, /, **
AGGREGATION OPERATORS	COUNT, COUNT', SUM, SUM', AVG, AVG', MAX, MIN
SET OPERATOR	SET

SET CONNECTIVES	$=, \supset, \supseteq, \subset, \subseteq, \vee, \wedge, \sim$
PUNCTUATION	: ; . ,
PARENTHESIS	( )

APPENDIX 2

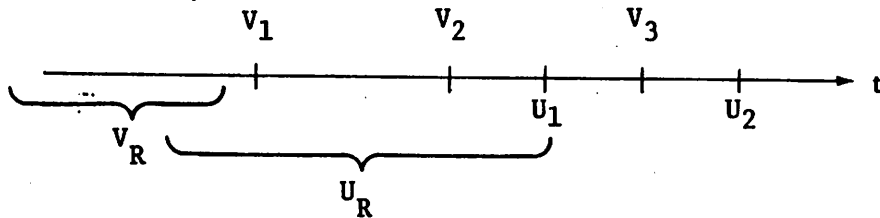
Two updates U and V are safe if

1. $U'_R[V'_i V'_{i-1} \ldots V'_1(J^*)] = U'_R(J^*)$ for all i

2. $V'_R[U'_i U'_{i-1} \ldots U'_1(J^*)] = V'_R(J^*)$ for all i

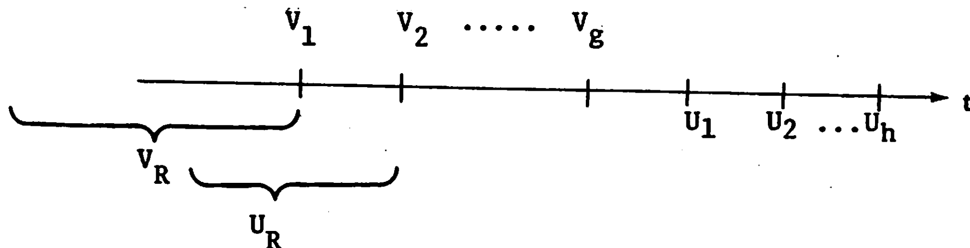
Proof

Consider the following execution sequence.

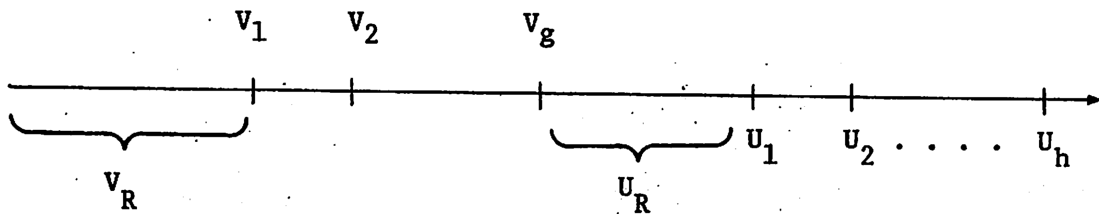


Here $V_1, \ldots, V_g$ and $U_1, \ldots, U_h$ are denoted by a fixed time since they are atomic. On the other hand V_R and U_R are denoted by the intervals over which they are accomplished.

Since $V'_1, \ldots, V'_g$ and $U'_1, \ldots, U'_h$ insert or delete tuples from a relation (or relations) and since neither set of updates is deleting a tuple inserted by the other as a result of 1. and 2., they are commutative, (i.e. $V'_i\ U'_j = U'_j\ V'_i$ for all i and j). Hence, the above diagram is equivalent to



As a result of 1, this is equivalent to



i.e. to $U' \cdot V'(J)$

The alternate diagram for which V_R overlaps U_i for some i is considered analogously.