INGRES—A relational data base system*

by G. D. HELD, M. R. STONEBRAKER and E. WONG
University of California
Berkeley, California

INTRODUCTION

INGRES (Interactive Graphics and Retrieval System) is a relational data base and graphics system which is being implemented on a PDP-11/40 based hardware configuration at Berkeley. INGRES runs as a normal user job on top of the UNIX[1] operating system developed at the Bell Telephone Laboratories. The only significant modification to UNIX that INGRES requires is a substantial increase in the maximum file size allowed. This change was implemented by the UNIX designers. The implementation of INGRES is primarily programmed in "C", a high level language in which UNIX itself is written. Parsing is done with the assistance of YACC,[2] a compiler-compiler available on UNIX.

The advantages of a relational model for data base management systems have been eloquently detailed in the literature,[3,4,5] and hardly require further elaboration. In choosing the relational model, we were particularly motivated by (a) the high degree of data independence that such a model affords, and (b) the possibility of providing a high level and entirely procedure free facility for data definition, retrieval, update, access control, support of views, and integrity verification.

In this paper we shall describe most of the principal components of INGRES. These include: the query language QUEL, an algorithm for processing interactions based on the principle of "decomposition," the access methods supported, and an approach to access control, views, and integrity preservation via query modification. Not described in this paper are: control of concurrent updates, the graphics facilities, the system utility commands, and a more "friendly" graphics oriented user interface. These topics are presented respectively in References 6, 7, 8 and 9.

BASIC CONCEPTS AND DEFINITIONS

Let $D1$, $D2$, . . . , Dn be nonempty sets, not necessarily distinct. A subset R of the product $D1 \times D2 \times \cdots \times Dn$ is called a *relation*, and Di are called the *domains* of R. Let r be an element of R, then r is an n-tuple $(r1, \ldots, rn)$ where ri belongs to Di. It is convenient to introduce the notation $r[Di]=ri$ in terms of which the projection of R on Di is defined as $R[Di]=\{r[Di]: r \text{ in } R\}$ where it is understood that any duplicate values are eliminated. If one visualizes R as a table with its elements appearing as rows, then $R[Di]$ is just the column corresponding to Di.

We have found it convenient to distinguish the projection $R[D]$ from the domain D itself, i.e., to distinguish a column from the set of its possible values. To do this, we have introduced the term *attribute* to stand for $R[D]$. An attribute can be viewed as a function on R taking values in D and its alternative notation $\{r[Di]: r \text{ in } R\}$ makes this clear. This point of view is important in understanding the syntax of QUEL.

QUEL: A RELATIONAL QUERY LANGUAGE

QUEL is a calculus based language. Though closely modeled after the data-sublanguage ALPHA of Codd,[10] it has some significant differences. Among these are the following:

(a) Rather than relying on a host language for arithmetical operations, QUEL is closed under such operations.
(b) QUEL is free of all quantifiers.
(c) Aggregation operations such as SUM and MAX are treated with much greater generality.

An initial version of QUEL became operational in October, 1974 which used punctuation as delimiters.[7] The system implementors found this unnatural and the delimiters were changed to keywords. Since the parsing of QUEL is done by YACC, this modification was easily implemented. The designers of SEQUEL[11] are to be credited with emphasizing the desirability of keywords in relational languages. Our experience has confirmed that keywords are nearly universally preferred over alternative delimiters.

Each query of QUEL contains one or more Range-Statements and one or more Retrieve-Statements. We shall use { } to denote "one or more" and [ ] to denote "zero or more". With these conventions the form of a query in QUEL can be expressed as

Query
= {Range-Statement} {Retrieve-Statement}

* Research sponsored by the National Science Foundation Grant GK-43024x, U.S. Army Research Office—Durham Contract DAHCO4-74-G0087, the Naval Electronic Systems Command Contract N00039-71-C-0255, and a Grant from the Sloan Foundation.

Range-Statement

=RANGE of {Variable} IS Relation

Retrieve-Statement

=RETRIEVE INTO Result-name (Target-List)
WHERE Qualification

Target-List = {Result-Domain = Function}

The goal of a query is to create a new relation for each Retrieve-Statement. The relation so created is named by the "Result-Name" clause and the domains in that relation are named by the "Result-Domain" names given in the Target-List. In the frequent case where the Function is simply Variable. Domain-Name, the Result-Domain name may be omitted and is then taken to be the same as the Domain-Name in the Function. Also, if the "Result-Name" is TERMINAL then the result of the query is displayed on the user's terminal. To create the desired relation, first consider the product of the ranges of all variables which appear in the Target-List and the Qualification of the Retrieve-Statement. Each term in the Target-List is a function and the Qualification is a truth function, i.e., a function with values true or false, on the product space. The desired relation is created by evaluating the Target-List on the subset of the product space for which the Qualification is true, and eliminating duplicate tuples.

Example: CITY(CNAME,STATE,POPULATION,AREA)
"Find the population density of all cities in California with population greater than 50k"

```
RANGE OF C IS CITY
RETRIEVE INTO W(C.CNAME,
              DENSITY=C.POPULATION/C.AREA)
              WHERE C.STATE='California'
              AND C.POPULATION>50K
```

(note the default used for CNAME=C.CNAME and that the result of the query is a relation W(CNAME,DENSITY)).

It is clear from the above discussion that the basic quantities used in QUEL are functions on products of relations. The allowed functions can be exceedingly complex and fall into three categories: (a) Functions resulting from arithmetical combinations of attributes. (b) Set valued functions such as "the set of cities for each state". (c) Aggregate functions obtained by aggregating set functions, e.g., "total population of the cities of each state." The precise definition of the allowed classes of functions will be given recursively as follows: Consider a nested sequence of sublanguages of QUEL

QUEL0, QUEL1, . . . , QUELn, . . .

Let Ci denote the class of all functions and Qi the class of all qualifications allowed in QUELi. We first define $C0$ and $Q0$.

$C0$

(a) Any constant is in $C0$.
(b) Any attribute is in $C0$.
(c) If f and g are in $C0$ then $f+g, f-g, f*g, f/g, f**g$ and $\log(f)g$ are in $C0$.

(Note: The functions being combined need not have identical arguments. The resulting function is a function of the union of the variables.)

$Q0$

(a) An atomic formula in $Q0$ has the form $f(comp)g$, where *comp* is any of the comparison operators: $<, \leq, =, \neq$, and f and g are in $C0$.
(b) $Q0$ consists of all sentences made up of atomic formulas connected by the Boolean connectives: NOT, AND, OR.

Comment: A function in $C0$ will be referred to as an *attribute-function.* The value of an attribute function for a tuple depends only on the data contained in that tuple. This is not true for functions in Ci for $i>0$. A similar comment applies to $Q0$ as well.

We now proceed to define QUELn recursively. Suppose $X=(X1, X2, \ldots, Xm)$ are the declared tuple variables with range $R=R1\times R2\times\cdots\times Rm$. Let X.f and X.qual be respectively a function and a qualification allowed in QUEL $(n-1)$. We define SET(X.f WHERE X.qual) as the set of f-values obtained by evaluating f on the subset of R for which X.qual is true, ie.,

SET(X.f WHERE X.qual)

= {X.f: X is in R AND X.qual = true}

Example:

X	CNAME	STATE	POPU
r1	SF	CAL	1M
r2	NYC	NY	6M
r3	CHI	ILL	4M
r4	LA	CAL	3M

SET(X.POPU WHERE X.STATE=CAL) = {1M, 3M}

SET(X.POPU WHERE X.POPU>3M) = {4M, 6M}

Comment: By definition a set contains no duplicate values. However, it is useful to define SET' as the collection obtained by retaining duplicates, for example,

SET'(X.STATE WHERE X.POPU<4M) = {CAL, CAL}

The aggregation operators COUNT, SUM, AVG, MAX, MIN have an obvious meaning when they operate on sets. If AGG is any of these operators, we shall adopt the notation

AGG(X.f WHERE X.qual)

=AGG(SET(X.f WHERE X.qual))

and AGG' will denote AGG(SET').

We shall refer to quantities of the form AGG(X.f WHERE

X.qual) as *aggregates*. An aggregate depends on the data contained in the range R but does not vary as X varies. The appearance of X merely serves to indicate the range. In this way, it acts as a dummy variable not unlike that in a definite integral. To put it more precisely, denote a function in Cn by $F(X, R)$ to indicate the fact that in general it depends on both the tuple X and on R overall. Then, we can say that constants depend on neither X nor R, functions in $C0$ depend on X but not on R, aggregates depend on R but not X.

Now suppose that f and g are in $C(n-1)$ and qual is in $Q(n-1)$. Define

SET(X.f BY X.g WHERE X.qual)

as a set valued function of X such that it is constant on any set of X for which g is constant and on such a set it is given by

SET(X.f BY X.g WHERE X.qual)(X.g=alpha)
=SET(X.f WHERE (X.g=alpha) AND X.qual)

Example:

X	X.STATE	SET(X.CNAME BY X.STATE WHERE X.POPU<5M)
r1	CAL	{SF, LA}
r2	NY	empty
r3	ILL	{CHI}
r4	CAL	{SF, LA}

The notation AGG(X.f BY X.g WHERE X.qual) is now self-explanatory, and so are the notations SET′(X.f BY X.g WHERE X.qual) and AGG′(X.f BY X.g WHERE X.qual).

Example:

X	X.STATE	MAX(X.POPU BY X. STATE WHERE X.POPU<5M)
r1	CAL	3M
r2	NY	0
r3	ILL	4M
r4	CAL	3M

Note that AGG(X.f BY X.g WHERE X.qual), unlike aggregates, is a function of both X and R and will be called an *aggregate-function*. It is a function of X through X.g and only through X.g. Thus, the three appearances of X play a mixture of roles. This is an objectionable syntactic feature, which however cannot be repaired by using a dummy variable for the first and last term. AGG(X′.f BY X.g WHERE X′.qual) involves aggregation on the product of the ranges of X' and X and means something quite different from AGG(X.f BY X.g WHERE X.qual). Several possible solutions have been considered but rejected for one reason or another. In particular, if X is restricted to be a single variable, then its presence can be suppressed in the first and third term. For the time being we have chosen not to impose such a restriction.

Set functions of the form SET(X.f BY X.g WHERE X.qual) can be combined by union, intersection, and relative complement. We can define the class of set functions allowed in QUELn as follows:

Sn

(a) *SO* contains all constant sets.
(b) *Sn* includes $S(n-1)$.
(c) If f and g are in $C(n-1)$ and qual is in $Q(n-1)$ then
SET(X.f BY X.g WHERE X.qual)
and SET(X.f WHERE X.qual)
are in *Sn* as are SET′(X.f BY X.g WHERE X.qual)
and SET′(X.f WHERE X.qual)
(d) *Sn* is closed under union, intersection and relative complement.

The classes Cn and Qn can now be defined as follows:

Cn

(a) Cn includes $C(n-1)$
(b) If s is in *Sn* then AGG(s) and AGG′(s) are in Cn.
(c) If f and g are in $C(n-1)$ and qual in $Q(n-1)$ then
AGG(X.f BY X.g WHERE X.qual)
and AGG′(X.f BY X.g WHERE X.qual) are in Cn.
(d) If f and g are in Cn, then $f+g, f-g, f*g, f/g, f**g$ and $\log(f)g$ are in Cn.

Qn

(a) Qn contains $Q(n-1)$
(b) If f and g are in Cn, then $f(\text{comp})g$ is in Qn, where comp is any of the operators $<, \leq, =, \neq$.
(c) If u and v are in *Sn* then $u(\text{set-comp})v$ is in *Sn* where set-comp is any of the set-comparison operators: inclusion, strict inclusion, equality, and inequality.
(d) If s is in *Sn* and alpha is a value, then (alpha belongs to s) is in Qn.
(e) Qn is closed under Boolean combinations.

Example:

SUPPLY(S#,P#,PRICE)

Query: Find those suppliers whose price for every part that he supplies is greater than the average price for that part.

```
RANGE OF S IS SUPPLY
RETRIEVE INTO W(S.S#)
WHERE COUNT(S.P# BY S.S# WHERE S.PRICE
                    >
                AVG′(S.PRICE BY S.P#))
=COUNT(S.P# BY S.S#)
```

Comments:

(a) It is clear that the Qualification of the Retrieve-Statement is in $Q2$.
(b) Instead of using COUNT, we could also have used the operator SET. In terms of processing efficiency, COUNT is preferrable.

UPDATE COMMANDS

In addition to RETRIEVE, QUEL permits three commands; REPLACE, DELETE, and APPEND, which are update operations. The syntax of the update statements is nearly identical to that of queries. Range statements have the same form and interpretation. The update statements have the same basic form as Retrieve-Statements:

```
Command Result-Name(Target-List) WHERE
  Qualification
```

For the APPEND Command, "Result-Name" must be the name of some existing relation, onto which qualifying tuples will be appended.
For the REPLACE (and DELETE) Command, "Result-Name" must be a tuple variable which, through the qualification, indentifies the tuples to be modified (or deleted).
The Target-List must contain explicitly (or by default) the existing Domain-Names for the relation being changed (the DELETE Command has no Target-List).
A few examples will indicate the usage of the Update-Commands. All of the examples will use the following relations and RANGE statements

```
EMP(NAME,SAL,BDATE,MGR)
NEWEMP(NAME,AGE,SALARY)
RANGE OF E IS EMP
RANGE OF N IS NEWEMP
```

Example:

All new employees over 35 are to work for SMITH

```
APPEND TO EMP(N.NAME,SAL=N.SALARY,
  BDATE=1975-N.AGE,MGR='SMITH') WHERE
  N.AGE>35
```

Example:

Give all employees a ten percent raise who work for Jones

```
REPLACE E(SAL=1.1*E.SALARY) WHERE
  MGR='JONES'
```

Note: The keywords BY and IS may be used interchangeably with '=' anywhere in INGRES to improve readability.

Example:

Remove all employees who are in the EMP relation from the NEWEMP relation.

```
DELETE N WHERE N.NAME=E.NAME
```

DECOMPOSITION

QUEL is obviously a very high level language and the success of system implementation depends critically on how the commands are processed. Palermo[12] and Rothnie[13] have considered the problem of query-processing in a relational data base system and have offered interesting suggestions on its solution. Neither, however, has provided an algorithm which is adequately general to process QUEL in its present form. Our approach is both more general and simpler. For the time being, we have opted for a uniform algorithm to deal with all queries rather than special strategies for special situations. How the algorithm can be tuned for each query is a problem that must be addressed at a later date.

Our overall strategy can be simply stated. Rather than compiling QUEL into a lower level language, we shall decompose an arbitrary QUEL-query into a series of one-variable QUEL-1 queries, at which point most of the difficult problems will have disappeared. Thus, for QUEL the "optimization" which is necessary for all high level languages lies nearly entirely in the decomposition. How "optimization" can be achieved is a problem still very far from a satisfactory solution. In the first version of our implementation, nothing more than some common sense rules of thumb are being used in selecting alternative paths at each stage of the decomposition.

Decomposition of QUEL queries is made difficult by an inconsistency in the language, viz., set-valued functions are allowed in the Qualification but not in the Target List. The role of set functions in the Qualification is to take the place of the universal quantifier. They are not allowed in the Target List because the result would be a relation with set-valued elements, i.e., an unnormalized relation. For example, the query

```
RANGE OF X IS CITY
RETRIEVE INTO W(X.STATE,CITIES-OF-STATE
 =SET(X.CNAME BY X.STATE))
```

is an illegal QUEL query because it would result in an unnormalized relation

	STATE	CITIES-OF-STATE
W	CAL	{SF, LA}
	ILL	{CHI}
	NY	{NYC}

The only way of processing set functions which is consistent with the restrictions of QUEL is to substitute specific values one at a time for the conditioning function (i.e., the middle argument). For example, SET(X.CNAME BY X.STATE) becomes successively SET(X.CNAME WHERE X.STATE =CAL), SET(X.CNAME WHERE X.STATE=ILL), etc. This strategy, when carried out on a multivariable query, can result in a combinatorial explosion in complexity. At the present time the use of set functions in QUEL statements is permitted but discouraged. Fortunately, a query involving set functions can often be replaced by one without them. The following example illustrates this point.

Query: which states have only cities with population less than 4M?

```
RANGE OF X IS CITY
RETRIEVE INTO W(X.STATE)
  WHERE SET(X.CNAME BY X.STATE
  WHERE X.POPU<4M)
  =SET(X.CNAME BY X.STATE)
```

The Qualification in the above query can be replaced by

```
COUNT(X.CNAME BY X.STATE WHERE
  X.POPU<4M) =COUNT(X.CNAME BY X.STATE)
```

For the remainder of this section we shall outline a general decomposition algorithm for those queries in QUEL which do not contain set functions. Sets, however, will be allowed. The overall strategy has two parts: (a) A QUELn query will be replaced by a series of QUEL$(n-1)$ queries and one-variable QUEL1 queries. (b) A multivariable QUEL0 query will be decomposed into a series of one-variable QUEL0 queries. Thus, repeated applications of the algorithm will decompose any QUELn query into a series of one-variable queries in QUEL1 or QUEL0.

(a) QUELn→QUEL$(n-1)$

Consider a query involving one or more tuple variables $X=(X1, X2, \ldots, Xm)$ with range $R=R1\times R2\times\cdots\times Rm$. Let its Qualification be denoted by $Q(X)$. Suppose the query contains (either in its Target List or in its Qualification) an aggregate function AGG(X.f BY X.g WHERE X.qual) where f and g belong to $C(n-1)$ and qual belongs to $Q(n-1)$.

(i) RANGE OF Y IS R

```
RETRIEVE INTO TEMPO(A=Y.f,B=Y.g)
WHERE Y.qual
```

Comment: The purpose of this statement is twofold: to convert the multivariable range R into a single relation TEMPO and to delineate the tuples which satisfy qual.

(ii) RANGE OF Y1 IS TEMPO

```
RETRIEVE INTO TEMP1
(Y1.B,C=AGG(Y1.A BY Y1.B))
```

Comment: Thus far, we have created the portion of AGG(X.f BY X.g WHERE X.qual), for X.g belong to SET(Z.g WHERE Z.qual).

(iii) RANGE OF Y IS R

```
RANGE OF Z IS TEMPO
RETRIEVE INTO TEMP2(B=Y.g,C=AGG(φ))
WHERE Y.g does not belong to SET(Z.B)
```

Comment: AGG(ϕ) means AGG operating on an empty set. We shall adopt the following convention:

$$\begin{aligned}
\text{SUM}(\phi) &= 0 = \text{SUM}'(\phi)\\
\text{COUNT}(\phi) &= 0 = \text{COUNT}'(\phi)\\
\text{AVG}(\phi) &= \text{undefined} = \text{AVG}'(\phi)\\
&\quad \text{(error if occurs)}\\
\text{MAX}(\phi) &= -\infty \text{ (i.e., the smallest possible value for that domain)}\\
\text{MIN}(\phi) &= \infty \text{ (i.e., the largest possible value for that domain)}
\end{aligned}$$

(iv) RANGE OF Z2 IS TEMP2

```
APPEND TO TEMP1(Z2.B,Z2.C)
```

Comment: We have finally created the desired aggregate function

(v) In the original query we add RANGE OF Z IS TEMP1, add the clause "AND (X.g=Z.B)" to the Qualification and substitute Z.C. for AGG(X.f BY X.g WHERE X.qual).

Comment: (iii) and (iv) are omitted if either X.g or X.qual is absent in the aggregate function.

(b) Multivariable QUEL0→One-Variable QUEL0

Suppose that the Qualification is expanded in a conjunctive normal form so that it consists of clauses connected by AND with each clause containing atomic formulas or their negation connected by OR.

(0) Stop if already one-variable.

(i) For each variable, say $X1$ with range $R1$, collect all the attributes which depend on $X1$ and all the clauses in the Qualification which depend only on $X1$. Say $D1, D2, \ldots, Dk$ are the attributes, and the clauses put together yield $Q1(X1)$. Issue the query.

```
RANGE OF X1 IS R1
RETRIEVE INTO R1'(X1.D1,X1.D2,. . . ,
X1.Dk) WHERE Q1(X1)
```

(ii) Replace the range $R1$ of $X1$ in the original query by $R1'$

Comment: The purpose of (i) and (ii) is to limit the range of each variable in the original query to as small a relation as possible by projecting and by enforcing the part of the Qualification which operates only on this variable.

(iii) Take the variable with the fewest tuples in its range and substitute in turn the actual values of its tuples. This reduces the number of variables by 1. After each substitution, repeat (0), (i), (ii) and (iii).

Comment: Step (iii) will be referred to as *tuple-substitution* and represents the most time consuming

step in the overall algorithm. The choice of which variable to substitute for is critical. Our criterion of choosing the variable with the fewest values is by no means optimal in general.

This concludes the discussion of decomposition of queries. Update statements are transformed into queries followed (perhaps) by a sequence of insertions and deletions of tuples.[6]

ACCESS METHODS

As a result of the steps in the preceding section a sequence of one-variable QUEL0 and QUEL1 queries are generated. These queries can be executed directly (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 required. Also secondary indices can be declared and are used if possible to limit the number of tuples examined. Currently there are five modes of relation storage and more can be added easily by implementing a common set of access calls (get next tuple, get unique tuple with equality on offered key, find starting point of a scan, etc.). These conventions are further discussed in Reference 14. Currently, a relation owner can decide both the relation storage and what secondary indices (if any) to construct. Soon both decisions will be done automatically by the system.

The current access methods are

(1) Unsorted Tables. This access method is supported for ease of entering relations into the system and is used for temporary relations (workspaces).
(2) Hashed Tables.[15,16] These are used when interactions almost always specify equality on a given domain or set of domains. A division algorithm is used, bucket-size is the page size used by UNIX and overflows are handled by chaining.
(3) Order Preserving Address Computation with a variable number of parameters. This access method is useful when scans over portions of a relation must be performed and the order preserving nature of the function can be used to limit the scan. The number of parameters is data dependent and ranges from none to the number of data pages used by the relation. With no parameters it resembles the computed functions of Rothnie[17] and Rivest.[18] With a maximum number of parameters it resembles VSAM.[19] This function and the choice of the number of parameters is discussed in Reference 14.
(4) & (5) Compressed Access. Access methods 2) and 3) can optionally (and transparently to higher level software) apply a compression scheme to each data page. Currently only front compression is used but the scheme will become more sophisticated in the future. These two access methods are useful when a large space saving will result in decreased I/O activity. Of course, the price paid is coding and decoding tuples at each access.

QUERY MODIFICATION

A high-level and nonprocedural language has benefits beyond its power and ease of use. For our purposes a major benefit will be the possibility of solving a number of system problems in a unified way, viz., by query modification. Here "query" is to be interpreted broadly to include the update commands as well. The specific problem areas to be addressed are: access control, integrity verification, and the support of "views." A suggestion along similar lines was made by Boyce and Chamberlin.[20] Since the details of our approach to these problems are being reported elsewhere,[21,22] we shall confine ourselves to a brief account of the basic ideas here.

Access control

Problem: We wish to provide a means whereby the access of each user to the database can be selectively restricted.

Solution: We define a pseudo QUEL command called RESTRICT which has a syntax nearly identical to that of RETRIEVE. The access control for each user is specified by a set of RESTRICT statements. For example, suppose EMPLOYEE(NAME,DEPT,SALARY,MANAGER) is a relation, and the restriction on SMITH is given by

```
RANGE OF E IS EMPLOYEE
RESTRICT ACCESS FOR 'SMITH' TO EMPLOYEE
  WHERE E.NAME='SMITH' OR E.MANAGER=
  'SMITH'
```

The interpretation here is that SMITH can retrieve only the data on himself and on anyone whom he manages. Different restrictions may be in force for the update commands, for example,

```
RANGE OF E IS EMPLOYEE
RESTRICT UPDATE FOR 'SMITH' TO EMPLOYEE
```

Execution: The Qualification of every RESTRICT statement for a given user is appended to the Qualification of every one of his interactions by conjunction (i.e., AND). For example, if SMITH issues the query

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO W(E.SALARY)
  WHERE E.NAME='JONES'
```

it is automatically executed as

```
RETRIEVE INTO W(E.SALARY)
  WHERE E.NAME='JONES'
  AND (E.NAME='SMITH' OR
  E.MANAGER='SMITH')
```

Comment: Queries involving aggregation are naturally

more difficult to handle, but nevertheless can be handled in a similar way.[21]

Integrity assurance

Problem: We want to provide a means for maintaining certain constraints or consistency conditions on the data in the face of updates. For example, suppose we require that the data satisfy the constraints: "no employee can earn more than 21K" which may well be violated by the update operation "give everyone earning less than 20K a 10 percent raise".

Solution: Introduce a QUEL command, called INTEGRITY which expresses the constraint in the form of a QUEL qualification, e.g.,

```
RANGE OF F IS EMPLOYEE
INTEGRITY CONSTRAINT IS E.SALARY≤21K
```

Execution: Each update command which potentially violates the integrity constraint will be filtered by the INTEGRITY statement. This is done by appending qualification from the INTEGRITY Statement. The algorithms become somewhat complicated when the INTEGRITY Statement involves more than one tuple variable or aggregation.[6] Here, we confine ourselves to an illustrative example. Suppose that a command to "give everyone earning less than 20K a 10 percent raise" is issued. In QUEL this takes the form

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY 1.1*E.SALARY)
  WHERE E.SALARY<20K
```

This statement upon modification becomes:

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY 1.1*E.SALARY)
  WHERE E.SALARY<20K
  AND 1.1*E.SALARY≤21K
```

Views

Problem: We wish to accommodate queries referencing relations which do not exist but can be derived from existing relations. Due to reorganization of the database, existing programs may have obsolete "views" that need to be supported.

Solution: We introduce a QUEL command called DEFINE, which relates a "view" of the database to reality. For example: Suppose that at one time the database contained a relation EMPLOYEE(E#,NAME,DEPT,SALARY,SPOUSE,#CHILD). After reorganization, this relation becomes two relations. JOB(E#,DEPT,SAL) and FAMILY(E#,NAME,SPOUSE,#CHILD). The DEFINE statement is given by

```
RANGE OF J IS JOB
RANGE OF F IS FAMILY
DEFINE EMPLOYEE(J.E#,J.DEPT,SALARY=
  J.SAL,F.NAME,F.SPOUSE,F.#CHILD)
```

Execution: Any interaction referencing a relation which is the "Result" of a DEFINE statement is translated into an interaction referencing the relations declared in the RANGE of the DEFINE statement. For example, consider the statement

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

Upon modification, it becomes

```
RANGE OF J IS JOB
RANGE OF F IS FAMILY
REPLACE J(SAL BY 1.1*J.SAL)
  WHERE F.NAME='JONES'
  AND F.E#=J.E#
```

CONCLUSION

The first, and a very primitive, version of our system is now implemented. At the time of writing of this paper only QUEL0 queries are permitted. A complete, but relatively unclever, implementation for QUEL, is expected by April, 1975. Access control is the first of the query-modification features to be implemented, and is now working. Other query-modification features and strategies for greater efficiency in processing are on the more distant horizon.

ACKNOWLEDGMENT

Implementation of INGRES is being undertaken by Jim Ford, Peter Kreps, Nancy McDonald, Carol Williams, Karel Youssefi and Bill Zook. Their dedication and resourcefulness never cease to amaze us.

REFERENCES

1. Ritchie, E. and K. Thompson, "The UNIX Time-Sharing System," *CACM, 17*, 1974, pp. 365-375.
2. Johnson, S. C., *YACC—Yet Another Compiler-Compiler*, Bell Telephone Laboratory, Murray Hill, N.J.
3. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," *CACM, 13* (1970), pp. 377-387.
4. Codd, E. F. and C. J. Date, "Interactive Support for Non-Programmers: the Relational and Network Approaches," *Proc. of the 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control*, Ann Arbor, Mich., May 1974.
5. Date, C. J. and E. F. Codd, "The Relational and Network Approach: Comparison of the Application Programming Inter-

faces," *Proc. of the 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control*, Ann Arbor, Mich., May 1974.
6. Stonebraker, M., *High Level Integrity Assurance in Relational Data Base Management Systems*, University of California, Electronics Research Laboratory, Memorandum ERL-M473, August 1974.
7. McDonald, N., M. Stonebraker, M. and E. Wong, *Preliminary Design of INGRES*, University of California, Electronics Research Laboratory, Memorandum ERL-435-436, April, 1974.
8. Stonebraker, M. and E. Wong, *INGRES—A Relational Data Base System*, University of California, Electronics Research Laboratory, Memorandum ERL-M477, November, 1974.
9. McDonald, N. and M. Stonebraker, *CUPID—The Friendly Query Language*, University of California, Electronics Research Laboratory, Memorandum ERL-M483, December, 1974.
10. Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," *Proc. of the 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*, San Diego, Calif., November 1971.
11. Boyce, R. and D. Chamberlin, "SEQUEL—A structured English query language," *Proc. of the 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control*, Ann Arbor, Michigan, May, 1974.
12. Palermo, E. P., "A Data Base Search Problem," *Proc. 4th International Symposium on Computers and Information Science*, Miami Beach, December 1972.
13. Rothnie, J. B., "A—Approach to Implementing a Relational Data Management Data Management System," *Proc. of the 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control*, Ann Arbor, Mich., May 1974.
14. Held, G. and M. Stonebraker, "Access Methods in the Relational Data Base Management System—INGRES," *Proceedings of ACM-PACIFIC-75*, San Francisco, Ca., April, 1975.
15. Morris, R., "Scatter Storage Techniques," *CACM, 11*, 1968, pp. 35-38.
16. Lum, V., "General Performance Analysis of Key-to-Address Transformation Methods Using an Abstract File Concept," *CACM, 16*, (1973,) pp. 603-612.
17. Rothnie, J. B., Jr. and T. Lozano, "Attribute Based File Organization in a Paged Memory Environment," *CACM, 17*, 1974, pp. 63-69.
18. Rivest, R., *Analysis of Associative Retrieval Algorithms*, Institute de Recherche d'Informatique et d'Automatique (IRIA), Rapport de Recherche No. 54, February, 1974.
19. Keehn, D. and J. Lacy, "VSAM Data Set Design Parameters," *IBM Systems Journal*, Vol. 13, No. 3, 1974.
20. Boyce, R. F. and D. D. Chamberlin, *"Using a Structured English Query Language as a Data Definition Facility,"* IBM Research Report RJ1318, December, 1973.
21. Stonebraker, M. and E. Wong, "Access Control in a Relational Data Base Management System by Query Modification," *Proc. 1974 ACM National Conference*, San Diego, Calif., November 1974.
22. Stonebraker, M., *Implementation of Views and Integrity Constraints in Relational Data Base Systems by Query Modification*, to be published.