EMBEDDING A RELATIONAL DATA SUBLANGUAGE IN A GENERAL PURPOSE PROGRAMMING LANGUAGE

by

ERIC ALLMAN AND MICHAEL STONEBRAKER
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA, BERKELEY, CA.

and

GERALD HELD
TANDEM COMPUTERS, INC.
CUPERTINO, CA.

ABSTRACT

This paper describes EQUEL, a programming language which embeds the relational data sublanguage QUEL into the general purpose programming language "C". Both QUEL and EQUEL are operational parts of the INGRES relational data base management system at Berkeley. Also briefly described are two operational subsystems written in this combined language. Lastly some of the language oriented shortcomings that have been observed in QUEL and EQUEL are discussed.

I. INTRODUCTION

The advantages of a relational model for data base management systems have been eloquently detailed in the literature, [CODD70, CODD74, DATE74] and hardly require further elaboration. In an attempt to fully understand the advantages and problems of relational data base management systems, a research and development effort was begun in 1973 to construct a full scale relational data base management system. The result of this project is INGRES, a relational data base system which has been implemented on a PDP-11 based hardware configuration at Berkeley.

This paper is concerned with programming language issues in a data base environment. A high level non-procedural language, QUEL, for data base query and update has been implemented in INGRES. Although QUEL alone provides the flexibility for most data management requirements, there are many applications which require a customized user interface in place of the QUEL language. For this as well as other reasons, it is often useful to have the flexibility of a general purpose programming language in addition to the data base facilities of QUEL. To this end, a new language, EQUEL (Embedded QUEL), has been implemented which consists of QUEL embedded in the general purpose programming language "C" [RITC74a]. C, an ALGOL-like language, was chosen since it was used both to implement UNIX [RITC74], the operating system on which INGRES runs, and to implement INGRES itself. However, the discussion in this paper applies to QUEL embedded in almost any general purpose programming language.

We will begin by briefly describing QUEL in Section 2. Then in Section 3 we indicate the form of EQUEL and decisions made in its design. We also indicate the nature of two applications written using this language. Lastly in Section 4 we indicate some language problems associated with QUEL and EQUEL. These include:

a) dynamic schemas
b) no recursion
c) types and type checking
d) syntax, especially in nested aggregation

II. QUEL

QUEL (QUEry Language) has points in common with Data Language/ALPHA [CODD71], SQUARE [BOYC73] and SEQUEL [BOYC74] in that it is a complete [CODD72] 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 [STON74].

The QUEL examples in this section all concern the following relation.

	NAME	DEPT	SALARY	MANAGER	AGE
	Smith	toy	10000	Jones	25
EMPLOYEE	Jones	toy	15000	Johnson	32
	Adams	candy	12000	Baker	36
	Johnson	toy	14000	Harding	29
	Baker	admin	20000	Harding	47
	Harding	admin	40000	none	58

Indicated here 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.

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 one of 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 [HELD75a].

Example 2.1 Find the birth year 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 returned to the calling process. If this process is the terminal monitor, it in turn prints them on the user's terminal. Other front-end processes may do what they please with such tuples. Also, in the Target-list, the 'Result-domain =' may be omitted if Function is of the form Variable.Attribute (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 modified by adding the indicated tuple to the relation. If less than all domains are specified, the remainder default to zero for numeric fields and null for character strings.

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, SUM, MAX, MIN, and AVG. Two examples of the use of aggregation follow.

Example 2.5 Replace the salary of all toy department employees 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 attribute 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. 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 value of E.DEPT. This value is the aggregate AVG(E.SALARY WHERE E.SALARY>10000 AND E.DEPT = value). 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).

In addition to the above QUEL commands to manipulate relations, INGRES also supports a variety of utility commands including ones to:

a) bulk copy data into and out of INGRES relations from or to UNIX files
b) create and destroy relations
c) add and delete integrity constraints [STON75]
d) add and delete secondary indices [HELD75].
e) change the access method used to store a given relation.

For a complete description of the currently operational INGRES commands the reader is referred to [ZOOK75].

III. EQUEL

In this section we describe the EQUEL language, indicate how it operates in a data base environment, and then describe two applications written in EQUEL.

In the design of EQUEL, the following goals were set:

1) The new language must have the full capabilities of both C and QUEL.
2) The C-program should have the capability for processing each tuple individually which satisfies the qualification in a QUEL RETRIEVE statement. (this is the "piped" return facility described in Data Language/ALPHA [CODD71]).
3) The implementation should make as much use as possible of the existing C and QUEL language processors. (The implementation cost of EQUEL should be small).

With these goals in mind, EQUEL was defined as follows:

1) Any C language statement is a valid EQUEL statement.
2) Any QUEL statement (or INGRES utility command) is a valid EQUEL statement as long as it is prefixed by two number signs ("##").
3) C-program variables may be used in QUEL statements in place of relation names, domain names, target list elements, or domain values. The declaration statements of C-variables used in this manner must also be prefixed by double number signs.
4) RETRIEVE statements without a result relation have the form

```
##      RETRIEVE (Target-list)
##                [WHERE Qualification]
##               {
                 C-Code
##               }
```

which results in the C-Code being executed once for each qualifying tuple. This C-Code may not contain other QUEL statements.

Two short examples illustrate EQUEL syntax.

Example 3.1 The following section of code implements a small front-end to INGRES which performs only one query. It reads in the name of an employee and prints out the employee's salary in a suitable format. It continues to do this as long as there are more names to be read in. The functions READ and PRINT are assumed to have the obvious meaning.

```
main()
{
## char ENAME[20];
## int SAL;
while (READ(ENAME))
        {
##      RANGE OF X IS EMP
##      RETRIEVE (SAL = X.SALARY)
##      WHERE X.NAME = ENAME
##              {
                PRINT("The salary of ",ENAME," is ",SAL);
##              }
        }
}
```

In this example the C-variable ENAME acts as an domain value in the QUEL statement and for each qualifying tuple, the C-variable SAL is set to the appropriate value. Then the PRINT statement is executed. (note: in C "{" and "}" are equivalent to BEGIN and END in ALGOL).

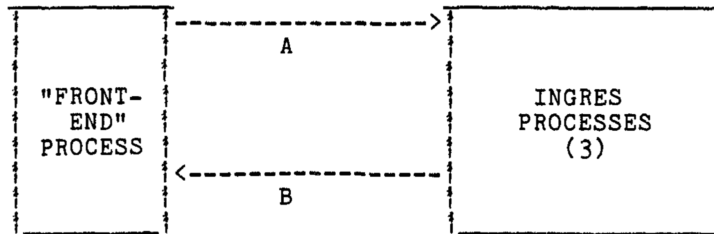
Example 3.2 Read in a relation name and two domain names. Then for each of a collection of values which the second domain is to assume, do some processing on all values which the first domain assumes. (We assume the functions READ and PROCESS exist and have the obvious meanings.) A more elaborate version of this program could serve as a simple report generator.

```
## int VALUE;
## char RELNAME[13], DOMNAME[13], DOMVAL[80];
## char DOMNAME_2[13];
READ(RELNAME);
READ(DOMNAME);
READ(DOMNAME_2);
## RANGE OF X IS RELNAME
while (READ(DOMVAL))
        {
##      RETRIEVE (VALUE = X.DOMNAME)
##          WHERE X.DOMNAME_2 = DOMVAL
##              {
                PROCESS(VALUE);
##              }
        }
```

Two important differences between this example and the previous one should be noted. First, in the RANGE statement of this example, the C-variable, RELNAME, is used in place of a specific relation name. This means that the tuple variable, X, can not be associated with a specific relation until execution time (when the READ statement is executed). Similarly, the domains to be used in the qualification DOMNAME and DOMNAME_2 can not be determined until execution time. This execution time binding is discussed further in section IV.

In order to implement EQUEL, a translator (precompiler) was written which converts an EQUEL program into a valid C-program with QUEL statements converted to appropriate C-code and calls to INGRES data base interface routines. The resulting C-program is then compiled by the normal C-compiler producing an executable module. To explain how an EQUEL program is executed, we must first describe a few of the details of the INGRES data base run time environment.

In the UNIX operating system, a "process" is an instance of a program in execution. Processes execute independently of each other, however, they can communicate through data paths called "pipes". Figure 1 shows the INGRES process structure, where INGRES is actually 3 processes with internal communication (pipes not shown) and a "front-end" process which passes data base requests through pipe A and receives results back through pipe B. Normally the front-end process is a simple interactive terminal monitor which allows the user to formulate queries in QUEL and submit them to INGRES for processing. However, when an EQUEL program is run, the executable module produced by the C-compiler is used as the front-end process. During execution of the front-end program, data base requests (QUEL statements in the EQUEL program) are passed through pipe A and processed by INGRES. If tuples must be returned for tuple at a time processing, then they are returned through pipe B. A condition code is also returned through pipe B to indicate success or the type of error encountered. (Actually pipe B is two interprocess communication pipes, one for data and one for termination conditions; however, for simplicity of exposition they will not be distinguished.)



The INGRES Process Structure

Figure 1

Consequently, the EQUEL translator must perform the following five functions:

1) insert system calls to create at run time the process structure shown in Figure 1.

2) note C-variable declarations prefaced by ## as legal for inclusion in INGRES commands.

3) process other lines prefaced by ##. These are parsed to isolate C-variables. In addition, C-statements are inserted to write the QUEL line down pipe A in character format, modified so that values are substituted for any C-variables.

4) insert C-statements to read pipe B for completion information and call the procedure IIerror. The user may define IIerror himself or have EQUEL include a standard version which prints the error message (for abnormal terminations) and continues.

5) If data is to be returned through pipe B (by a RETRIEVE with no result relation specified), EQUEL must also:

   a) insert C-statements to read pipe B for a tuple formatted as a type/value pair.

   b) insert C-statements to substitute values into C-variables declared in the target list. If necessary, values are converted to the types of the declared C-variables.

   c) insert C-statements to pass control to the C-block following the RETRIEVE.

   d) insert C-statements following the block to return to step a) if there are more tuples.

There are currently two applications written using EQUEL. One is a geo-data system called GEO-QUEL described in [GO75]. Basically it augments QUEL with display oriented features. The designers estimated a factor of 50 code reduction was achieved using EQUEL over what would be required in C alone. The second application is an event monitoring and reporting system. It was estimated that EQUEL afforded a thirty to one savings in programming effort over straight "C", due largely to the simplicity of coding in EQUEL vs. "C". Such factors should be realized in a wide variety of applications that are largely data management (e.g. computer aided design, computer aided instruction etc.). It is envisioned that a wide variety of user tailored "front-ends" will eventually be implemented in this fashion.

IV. PROBLEMS WITH SUBLANGUAGES AND EMBEDDED SUBLANGUAGES

We note that EQUEL bears some resemblance in features to other very high level languages, e.g., VERS and SETL. However, it was implemented primarily as a data base management system which involved the following design decisions.

a) Relations are implemented in the file system via a variety of access methods and not as core arrays.

b) Protection is being handled at the user language level [STON74a] and in no way involves the UNIX protection system for files.
c) Integrity control schemes [STON75] are supported.
d) For protection reasons as well as because of address space limitations, the system runs as multiple processes.

The following problems are present in QUEL or in EQUEL. Some result from the above considerations.

1) Dynamic Schemas.
The CODASYL DBTG report proposes a schema (in essence a description of the data base) which is static and does not vary with time. In INGRES a user is allowed to execute a RETRIEVE INTO statement which creates a new relation, and hence alters the relational schema. This is analogous in general purpose programming languages to allowing a user to create space for variables at run time (such as is done in SIMULA). However, in programming languages such variables are local to an invocation of the program and space disappears when the program terminates.

In INGRES, relations created during execution do not disappear when the program terminates (since the programmer may wish to use such a relation again). In fact, INGRES keeps relations for a period of time specified by a data base administrator and provides a SAVE command should the user wish that they be kept longer.

This causes several problems. If the relational schema is altered at precompile time at least four dilemmas occur:

a) It may be impossible to do the alteration because the relation to be created at run time may depend on other relations (created by other programs) which do not yet exist in the schema (because the other programs have not yet been precompiled).
b) There may be name conflicts; i.e., a user may be required not to use the same name for a relation in different programs.
c) DESTROY relname is a legal command, and in this case the schema cannot be altered until run-time.
d) It may be impossible to alter the schema because changes could easily depend on run-time values.

For instance, EQUEL supports the statements

```
READ(A);
## RETRIEVE INTO A (    )
```

which creates a new relation that is not known until the C-variable, A, is set during execution of the READ statement.

If the schema is not altered until run time, the following problem is evident: QUEL statements cannot be parsed until run time since the legality of a command cannot be determined until then. (e.g., RANGE OF E IS W gives a run time error if W does not exist. This cannot be known at precompile time.)

Moreover, one can write programs such that the legality of an INGRES command cannot even be determined at the start of program execution. For example, in the program

```
READ(A);
if (A > 5)
        {
##      CREATE W (   )
        }
         .
         .
##      RANGE OF E IS W
         .
         .
         .
```

the legality of the RANGE command cannot be determined until it is executed.

Parsing during execution (as is done in INGRES) has an obvious run time cost. For example, the following program will result in parsing the RETRIEVE statement 1000 times:

```
    for (i = 0; i < 1000; i = i + 1)
        {
    ##  RETRIEVE ...
        }
```

One approach to avoiding multiple parsing could be to name the retrieve command in a definition and then call it with parameters in a manner similar to the definition and invocation of MACRO's. The following syntax could, for example, be supported.

```
## DEFINE MYSTATEMENT(RELATION, DOMAIN, QUAL)
## char RELATION[20], DOMAIN[20], QUAL[20]
## int SAL
## RANGE OF E IS RELATION
## RETRIEVE (SAL = E.SALARY)
## WHERE E.DOMAIN = QUAL
## END MYSTATEMENT

    .
    .

## MYSTATEMENT(MYRELATION, MYDOMAIN, MYQUAL)
```

In certain situations this might save multiple parsing. This could happen if the definition was sent down pipe A, parsed, and stored by INGRES in parsed form. Invocation would then simply involve parameter validation. However, when the parameter list contains a relation name (as above), reparsing the statement at each call is unavoidable. The added complexity of distinguishing the above cases was judged not worth the effort by the authors. Restricting the allowable definitions to ensure no multiple parsing was considered unacceptable. Also, if definitions are not local to the invocation of a program (as might be desirable), some of the problems mentioned earlier reappear. As a result, this approach was not followed.

The relational schema in INGRES can also be considered to include

a) integrity constraints on relations
b) access control statements for relations
c) view definitions

Algorithms to support (a)-(c) are given in [STON74a, STON75] and basically involve making modifications to a QUEL statement at the source language level. If the precompiler makes these modifications, the same sorts of problems discussed earlier in this section appear. (For example, an integrity constraint can be changed between precompile time and execution time. Also dilemma (d) above indicates that one cannot necessarily know at precompile time what integrity constraints to enforce.) Again, if modifications are done at execution time (as in INGRES), the run time overhead must be tolerated.

Other than substantially restricting the allowable operations, (and as a result restricting the generality of the "front-ends" which are possible) the authors see no solution to this parse-at-execution problem. One attempt at restriction to support precompile-time-parsing is to forbid all commands which can change the data base schema. These include RETRIEVE INTO, CREATE, and DESTROY. Even this is not foolproof unless one forbids these same commands when the interactive terminal monitor is the front end process.

2) Recursion
There is no recursion in INGRES itself. (i.e. no INGRES commands are implemented by invoking other INGRES commands. Secondary indices could easily be treated this way if this feature existed). Also, no recursion is allowed by the precompiler (i.e. the C-block executed for each tuple cannot contain an INGRES command.) There are, however, many data base applications in which recursive algorithms for data access are very natural. A common example is the bill of materials (or parts explosion) problem. Here, the data base contains tuples describing every sub-assembly and part in a product. The problem is to create a list of all parts which are components of the major assembly, a sub-assembly, a

sub-sub-assembly and so on to an unknown depth. An equivalent example in the employee relation would be to "find all employees who work directly or indirectly for Harding".

Although recursion is allowed in C and many other programming languages, it appears difficult to implement in programming systems which span more than one process. Tuples are returned to a C-program through a pipe. Allowing a second QUEL command to be executed before tuples from the first command are cleared from the pipe will result in additional information at the end of the pipe which is not available until all tuples from the first command are read from the pipe.

One solution would appear to be to fork a second collection of INGRES processes with their own pipes (i.e., call INGRES recursively). Since recursion could be fairly deep, the available pipes could be soon exhausted. Moreover, pipes are much more expensive to create than the execution of a subroutine call.

Since recursive calls to the data base system do not seem feasible, what is necessary is to include within INGRES the capability for recursive data access. The data sublanguage must be extended to have a primitive operation such as RETRIEVE* in the following query:

```
/*find all employees who work for Harding
  (directly or indirectly)              */
RANGE OF E IS EMPLOYEE
RANGE OF A IS ANSWER
RETRIEVE* INTO ANSWER (NAME = E.NAME)
        WHERE E.MANAGER = "Harding"
        OR    E.MANAGER = A.NAME
```

Here, RETRIEVE* must be defined so as to re-process the query until the size of the result relation stops growing. No current relational system has this capability and it is suggested as a important problem for relational language designers to cope with.

3) Types and Type Checking
Conversion of types between the data base system and the C-program acting as the front-end presents a number of problems. Since the precompiler cannot know what types various domains will be at run time (for the same reasons as outlined in (1) above), one of two approaches must be used.

The first approach is to insist that the type of a variable that is used in the C-program match the type of the value that is returned from INGRES. Otherwise, an error could be generated. This is clearly unacceptable, as it removes desirable data independence. For example, it should make no difference to the C-program if INGRES stores a domain as a one or a two byte integer even though the C-program prefers to consider it as a two byte integer.

The second approach (which is followed by the precompiler) is to perform run time type conversion between all types of C-variables and all INGRES domain types.

In the current implementation the collection of types is not extensible in INGRES. Similarly, C does not support extensible types. Hence, the precompiler can include all necessary conversion routines. However, languages with extensible types have obvious appeal and the same statement holds for data base systems. Should such systems appear, they would be required to cope with the dilemma illustrated by the following example.

Suppose the data base system supported the domain type COLOR = {RED, BLUE, GREEN, YELLOW}. Suppose further that a user of the host language defined a type COLOR1 = {BLUE, VIOLET, GREEN, RED, BLACK}. Obviously the conversion from one to the other is complex and requires a complete description of both COLOR and COLOR1. It is not clear how the precompiler can easily be made aware of both descriptions.

Moreover, consider the case where the data base system requires user supplied conversion routines (for example for the domain types DOLLARS and PESOS). This routine must be available to the precompiler if it is required to convert one to the other (or something else to either one). Moreover, if the user wishes to define a new data type (say FRANCS), he must know the type of the stored domain which he will be converting from in order to supply the appropriate routine. Data independence is unavoidably sacrificed or the user must supply a complete collection of conversion routines.

4) Syntax

There are various syntactical problems in QUEL. One concerns aggregation in the language and involves the scope of tuple variables. It will be illustrated by the examples to follow.

Example 4.1 Find the average salary of those employees who make more than the average company salary.

```
RANGE OF E IS EMPLOYEE
RANGE OF F IS EMPLOYEE
RETRIEVE  (COMP= AVG(E.SALARY WHERE E.SALARY >
           AVG(F.SALARY))
```

It should be noted that the scope of each tuple variable is local to the aggregate involved. However, when a BY clause is present the scope of the variable used must be global as the following example suggests.

Example 4.2 For each department find the average salary of those employees who make more than the average salary of their department.

```
RANGE OF E IS EMPLOYEE
RANGE OF F IS EMPLOYEE
RETRIEVE (COMP = AVG(E.SALARY BY E.DEPT
                     WHERE E.SALARY
                           >
                           AVG(F.SALARY WHERE F.DEPT=E.DEPT)),
          DEPT = E.DEPT)
```

This command has several objectionable features.

a) It is difficult to understand even if the semantics of aggregation are known.

b) In the first AVG, E is a local variable when used anywhere but in the BY clause. Hence, the first E.DEPT is global to the interaction, and is the same variable as the third E.DEPT. On the other hand, E.SALARY is local to the aggregation. This mixing of scopes is objectionable; however, it cannot be solved simply by introducing a new dummy variable. The following two statements are not semantically equivalent:

```
RANGE OF E IS EMPLOYEE
RETRIEVE (COMP = AVG(E.SALARY BY E.DEPT),
    DEPT = E.DEPT)

RANGE OF E IS EMPLOYEE
RANGE OF E1 IS EMPLOYEE
RETRIEVE (COMP = AVG(E1.SALARY BY E.DEPT),
    DEPT = E.DEPT)
```

For the example relation considered the first answer is:

COMP	DEPT
13000	toy
12000	candy
30000	admin

while the second answer is:

COMP	DEPT
18500	toy
18500	candy
18500	admin

This result is caused by the second expression being evaluated on the cross product of E and E1.

Several fixes have been considered but rejected for one reason or another.

c) the second E.DEPT which appears is not local to the second aggregate because it appears in an outer aggregation.

A clean resolution of these objectionable features is clearly desirable. The option of restricting aggregates to be simple enough so the problem can go away is not very attractive. (For example no nesting, no multivariable aggregations, and delete the tuple variable everywhere but in the BY clause is one solution.)

V. SUMMARY

In this paper we have described how the relational language QUEL has been embedded in the programming language C and have discussed problems with data sublanguages and embedded data sublanguages. Most of these problems are inherent to a data base environment and are suggested as important topics for research.

REFERENCES

[BOYC73] Boyce, R., et al, "Specifying Queries as Relational Expressions: SQUARE", IBM Research, San Jose, Ca., RJ 1291.

[BOYC74] 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.

[CODD70] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," CACM, 13 (1970), pp. 377-387.

[CODD71] 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, Ca., November 1971.

[CODD72] Codd, E. F., "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposium 6, May 1972.

[CODD74] 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.

[DATE74] Date, C. J. and E. F. Codd, "The Relational and Network Approach: Comparison of the Application Programming Interfaces," Proc. of the 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

[GO75] Go, A., M. Stonebraker, and C. Williams, "An Approach to Implementing a Geo-Data System," Proceedings of the Workshop on Data Bases for Interactive Design, Waterloo, Ontario, Canada, September 1975.

[HELD75] 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.

[HELD75a] Held, G., M. Stonebraker, and E Wong, INGRES -- A Relational Data Base System, Proceedings of the 1975 National Computer Conference, Anaheim, Ca., May 1975.

[JOHN74] Johnson, S. C., YACC -- Yet Another Compiler-Compiler, Bell Telephone Laboratory, Murray Hill, N.J.

[RITC74] Ritchie, D. and K. Thompson, "The UNIX Time-Sharing System," CACM, 17 (1974), pp 365-375.

[RITC74a] Ritchie, D., "C Reference Manual", Bell Telephone Laboratories, Murray Hill, N.J., 1974.

[STON74] Stonebraker, M., "A Functional View of Data Independence," Proc. 1974 ACM-SIGFIDET Workshop on Data Description Access and Control, Ann Arbor, Mich., May 1974.

[STON74a] Stonebraker, M., and E. Wong, "Access Control in a Relational Data Base System by Query Modification", Proc. 1974 ACM National Conference, San Diego, Ca., November 1974.

[STON75] Stonebraker, M., Implementation of Views and Integrity Constraints in Relational Data Base Systems by Query Modification, Proc. 1975 SIGMOD Workshop on Management of Data, San Jose, Ca., May 1975.

[ZOOK75] Zook, W., et al, INGRES Reference Manual, University of California, Electronics Research Laboratory, Memorandum ERL-M519, April 1975.