A UNIFORM NOTATION FOR EXPRESSING QUERIES

by

C. J. Prenner

Memorandum No. UCB/ERL M77/60

8 September 1977

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A Uniform Notation for Expressing Queries

by

Charles J. Prenner

Department of Electrical Engineering and Computer Sciences

University of California at Berkeley

Abstract

A query language which is suitable for use with a relational database system is discussed. The language is developed as an extension of ideas found in programming languages as well as the relational calculus. The development is a step towards the integration of programming languages, query languages and database systems. The language offers a simple logical model of the way in which the query might be processed in order to simplify query construction. However, the model does not constrain the actual processing method used by the database system. The language has a single construct, the relational expression, which appears in either a "short" or "long" form. The language provides a uniform notation for expressing queries in that any retrieval request can be expressed by suitable nesting of this single construct.

This research was supported in part by U.S. Army Research Office Grant DAAG-29-76-G-0245, and by the National Science Foundation under an NSF Participation Fellowship at IBM San Jose Research Laboratory.

1. Introduction

Most query languages developed in conjunction with relational data base systems were designed independently of any programming language. However, it was soon realized that the query languages could not stand alone, and thus, these languages have been loosely coupled [1] [2] to existing programming languages in order to allow free movement of data from the programming language to the data base, and conversely. In considering the query language as part of the programming language one finds that the notation and orientation of the two are quite different. Many programming languages are procedural, with powerful facilities for data definition, iteration and list processing. On the other hand, most query languages are non-procedural and use such terms as universal and existential quantification, composition, join, and so on. This difference can be confusing for a programmer who must switch back and forth between the two.

In this paper we will consider relations and query languages from a programming language viewpoint. Such a consideration is a step towards the integration of programming languages and query languages. In section two we consider some difficulties with existing query languages. In section three we introduce some notation and discuss the correspondence between relations and sequences of records. This provides the background for sections four through six in which we develop a new query language, UNEQ (Uniform Notation for Expressing Queries), based upon the familiar concept of iteration. In section seven we indicate why we believe that UNEQ provides a uniform notation and consider possible objections to the language.

2. Current Query Languages

Virtually all query languages for relational data base systems, such as SEQUEL[11], QUEL[2], SQUARE[13], and DEDUCE[14], are non-procedural, i.e. the approach is to describe what information is desired as opposed to describing how it is to be obtained. Since the query itself imposes no constraints on the processing order the data base system is free to choose the best method available to process the query, given the information it has about the size of relations, access paths, and so on. However, except for the simplest of queries, the non-procedural approach can make it difficult to actually construct a query. Alternatively, it can be difficult to understand what computation is being expressed by an existing query. This can be true for programmers because they are accustomed to specifying computations procedurally, as well as non-programmers who typically possess less computational skills than the former group. An alternative approach, which we will consider in this paper, is to allow a query to be written in terms of how it <u>might</u> be processed in such a fashion that would still allow the data base system to choose how <u>it is</u> to be processed.

An additional difficulty with existing query languages is that they often do not adhere to customary practices in programming language design. Unconventional aspects of query languages include the following:

- Unusual name scoping rules are sometimes used.
- Some keywords suffer from "semantic overload", i.e. they have different meanings in different contexts.
- The meaning of a query can be changed radically by the inclusion of a single delimeter.
- An excessive number of variables are often required to express the query.
- It is often difficult to determine in what order the parts of the query should be read in order to understand the query as a whole
- The constructs are not designed to nest easily with one another.

To illustrate some of these problems, we will consider queries written in a number of different query languages. In section seven we will reconsider these queries and contrast them with their counterparts in UNEQ. Throughout the paper we will use the following relations taken from [11].

EMP(EMPNO, NAME, DNO, JOB, MGR, SAL, COM)

DEPT(DNO, DNAME, LOC)

USAGE(DNO, PART)

SUPPLY(SUPPLIER, PART)

The following query is written in ALPHA[11].

Q-A : Find the names of employees who work in Detroit.

```
Q-A:        RANGE E EMP
            RANGE D DEPT
            GET W E.NAME   ∃D(D.DNO=E.DNO
                              AND D.LOC="DETROIT")
```

In ALPHA, one declares the range of each variable, e.g. E ranges over EMP. The output is to go into a named relation (W) and the relation is to contain a single column consisting of a subset of the NAME column of E (E.NAME). The NAMEs chosen are those E.NAMEs

for which there exists some D where D.DNO=E.DNO and D.LOC="DETROIT."

In this notation it is hard to see that it is the employees that are being considered in turn (as opposed to departments). In addition, the specification of qualifying tuples by requiring the existence of some D for which the department number matches E's department number and D's location is Detroit is substantially less direct than simply asking (for each employee E) if Detroit is the location of E's department.

The next query, taken from [11], is written in SEQUEL.

Q-B: List the suppliers which supply all parts used by department 50.

Q-B:

```
SELECT UNIQUE SUPPLIER
FROM SUPPLY SUP
WHERE
  (SELECT PART
   FROM SUPPLY
   WHERE SUPPLIER = SUP.SUPPLIER)
CONTAINS
   (SELECT PART
    FROM USAGE
    WHERE DNO=50)
```

In SEQUEL one indicates that a distinct (UNIQUE) subset of the SUPPLIER column of SUPPLY is desired. The variable SUP is to range over the tuples of SUPPLY and the SUPPLIER component is output for all tuples satisfying the WHERE clause. Here one considers all of the tuples of the SUPPLY relation, and duplicate suppliers are removed during the processing of the query. Note that if a comma was inserted between SUPPLY and SUP then it would indicate that the join of two relations (SUPPLY and SUP) is desired.

The concept of considering each supplier in turn can be expressed in SEQUEL using additional syntax, viz

```
SELECT SUPPLIER
FROM SUPPLY
GROUP BY SUPPLIER
        HAVING SET (PART) CONTAINS
        SELECT PART
        FROM USAGE
        WHERE DNO=50
```

Here, the GROUP BY specifies that each of the distinct suppliers is to be considered, and the HAVING clause states that only those suppliers should be considered for which the set of parts

(SET(PART)) contains all parts for department 50. Thus, SEQUEL has two unrelated syntactic forms for expressing the same kind of computation.

The following query is written in QUEL[2].

Q-C: For each department, find the average salary of those employees whose salaries are greater than the average salary of their department.

```
Q-C:        RANGE OF E IS EMP
            RANGE OF F IS EMP
            RETRIEVE(COMP=
                    AVG(E.SAL BY E.DNO
                            WHERE
                    E.SAL>AVG(F.SAL WHERE F.DNO=E.DNO)),
                        DEPT=E.DNO)
```

As in ALPHA, QUEL requires declaration of range variables (E and F). The order of declaration is irrelevant. RETRIEVE specifies that a two column relation is to be constructed with columns COMP and DEPT. The "BY E.DNO" indicates that the output relation is to contain the distinct department numbers from EMP (as the DEPT component since E.DNO is repeated there) and the COMP component, for each department number, is to contain the average of all of the saleries of E from the department where the salaries used are those which are greater than the average salary (F.SAL) of all employees in the deparment.

As mentioned in [2], this query is "hard to understand even if the semantics of aggregation are known". Part of the problem is that the "BY E.DNO" controls the interaction and therefore it has global meaning. Yet it appears syntactically the same as E.SALARY which has only local meaning. Furthermore, "BY" is equivalent to "=". Thus, the second WHERE clause could be written "F.DNO BY E.DNO" causing further confusion. "BY" is also used in other contexts for updating relations.

3. Programming Language and Data Base Concepts

Before we consider UNEO itself, we must first introduce some notation. Most modern programming languages, such as PASCAL [3], ECL [4], and ALGOL/68 [5] allow the user to define data types for records, i.e. compound objects consisting of a fixed number of objects of specified types. These sub-objects may be accessed by selector names which are given in the definition of the type. For example,

```
DEPTREC = RECORD
            DNO:INT,
            DNAME:STRING,
            LOC:STRING
          END
```

defines the type DEPTREC as consisting of 3 objects. In addition, some of these languages allow the definition of variable-length sequences, i.e. one-dimensional arrays of objects where all objects are of one particular type and individual constitutent objects may be selected by integer index. For example,

```
DEPTROW = ARRAY [1:] DEPTREC
```

defines the type DEPTROW as a one-dimensional array of objects of type DEPTREC. The sequences are considered to be variable length in the following sense: any DEPTROW object has a fixed length but different DEPTROW objects may have different lengths. To construct an object of type DEPTREC or DEPTROW, one can use a constructor function (whose name is identical to the type name), where arguments to the function are objects of appropriate type. For example,

```
DEPTROW( DEPTREC(3, "TOY", "DETROIT"),
         DEPTREC(4, "CANDY", "DENVER"),
         DEPTREC(7, "PENCIL", "BOSTON"))
```

constructs a DEPTROW consisting of three DEPTRECs. Note that DEPTROW() creates a DEPTROW of lengh zero. If D is an object of type DEPTROW, then D[2].DNAME gives the department name of the second record in the sequence, LENGTH(D) gives the number of records in the sequence, and

```
CONCAT(D, DEPTREC(9,"PAPER","PORTLAND"))
```

creates a new DEPTROW which is identical to D except that it has one additional record at the end of the sequence.

There is an obvious correspondence between a relation and a sequence of records. A record definition can be viewed as defining the data type for the tuples of the relation and a sequence of such records is the type definition for the relation as a whole. Of course, the problem with this correspondence is that relations are intended to be unordered while sequences are inherently ordered. We make the correspondence because programmers are accustomed to processing sequences of records. Thus, it serves as a starting point for the construction of a query language from a programming language viewpoint. An additional problem is that relations do not contain duplicates. At least one relational system, SYSTEM R [1], has removed this restriction. Here, we choose to follow SYSTEM R and consider relations as bags [15], i.e unordered collections of objects which may contain duplicates, as in artificial intelligence languages.

4. Queries from a Programming Language Viewpoint

Although query languages also contain facilities for updating and deleting relations, in this paper we will only be concerned with retrieval requests, i.e. requests to list information (e.g., on a terminal) in the form of a table, or requests to construct a new relation from existing relations.

One of the simplest types of query is one in which a subset of the columns of a relation are listed for all tuples which satisfy some predicate. For example,

Q1: Find the names of employees in department 10.

Viewing the relations defined in section 1 as sequences of records, the following program fragment will yield this information.

```
FOR I <- 1 : LENGTH(EMP) DO
   IF EMP[I].DNO=10
         THEN PRINTLINE(EMP[I].NAME)
```

A slight variation on this query is one in which the information obtained is itself formed into a relation.

Q2: Create a table of the names and jobs of employees whose salaries are over 50,000.

```
EREC = RECORD
        NAME:STRING,
        JOB:STRING
       END

EROW = ARRAY [1:] EREC

BEGIN
  RESULT<-EROW();
  FOR I<- 1:LENGTH(EMP) DO
    IF EMP[I].SAL>50000 THEN
      RESULT <- CONCAT(RESULT,
                       EROW(ERECORD(EMP[I].NAME,
                                    EMP[I].JOB)));
  RESULT
END
```

Here, the relation is constructed by repeated concatenations to the variable RESULT. RESULT is initialized to a null sequence so that CONCAT will operate correctly the first time it is called. The last statement of the block is simply "RESULT," which becomes the value of the block as a whole. If no records satisfy the predicate of the FOR loop then a null relation will be returned.

The program fragment for Q2 can be rewritten to avoid repeated reference to EMP[I] within the body of the FOR loop.

```
BEGIN
  RESULT<-EROW();
  FOR I<- 1: LENGTH(EMP) DO
  BEGIN
    DECLARE X:EMPREC; X <- EMP[I] ;
    IF X.SAL>50000 THEN
        RESULT<--CONCAT(RESULT,
                        EROW(ERECORD(X.NAME,X.JOB))
  END;
  RESULT
END
```

Here the body of the loop is a block. Each time the block is entered a local variable X is declared and initialized to EMP[I].

In the evolution of high-level programming languages, new linguistic forms are introduced to replace commonly occurring program fragments when it is recognized that such forms will add clarity, conciseness and structure to the resulting code. The idea, as expressed by Perlis [7], is to suppress what is constant and display what is variable. Considering the two queries above, we can see that the data type definitions, the order of the sequence entries, the RESULT variable and the successive concatenations can all be suppressed since all such queries will require data types, processing of all records, and either the listing or construction of a new relation. Thus, we introduce a new linguistic form, the FOR EACH expression, which captures and suppresses all of the above and displays only that which is variable, i.e. the predicate applied to each record and the set of columns to be output. The following two FOR EACH expressions are semantically equivalent to the corresponding program fragments above.

```
Q1: FOR EACH X:EMP
      X.DNO = 10 => X.NAME

Q2: [FOR EACH X:EMP
      X.SAL>50000 => X.NAME, X.JOB]
```

Here, X acts like the variable declared in the inner block above; it is bound to each record of EMP in turn. For each binding of X, the infix operator => evaluates its left argument. If the value is TRUE, then the list of components on the right of the arrow are output to some output device. If the FOR EACH (perhaps abbreviated as FE) is surrounded with square brackets, then the components are output as a tuple of a new relation. In this latter case the FOR EACH expresses the creation of a relation, i.e. a relational expression.

The general form of a relational expression is:

[FE X:REL {predicate on X} => {selectors on X}]

This form of a relational expression is similar to one described in [12] for PASCAL. However, further suppression is possible. The variable X may be omitted and the record selectors used by themselves. The short form of a relational expression is:

[{selectors}|REL:{predicate}]

For example, in short form Q2 is as follows

[NAME, JOB|EMP: SAL>50000]

which may be read "Construct a relation consisting of the name and job components from the EMP relation, for those tuples whose salary component is greater than 50000.

If all components of qualifying tuples are desired, then either the keyword ALL may be used or the selectors and the vertical bar may simply be dropped. For example, both

[ALL|EMP: JOB = "PROGRAMMER"]

and

[EMP:JOB = "PROGRAMMER"]

construct relations concerning only those tuples from EMP for employees who are programmers.

If the predicate is TRUE, i.e. all tuples satisfy the predicate, then the colon and TRUE may be dropped. Thus,

[NAME|EMP]

constructs a one column relation consisting of all employees names from the EMP relation.

Combining both of the above defaults we have

[ALL|REL:TRUE] = [REL] = REL

Most relational systems provide a set of built-in functions that may be applied to relations. For example, the number of tuples in the relation is obtained by applying the function COUNT to the relation

COUNT[EMP]

COUNT is analagous to the LENGTH function defined on sequences. However, COUNT may be applied to any relational expression, e.g.

COUNT[EMP:JOB = "PROGRAMMER"]

returns the number of employees who are programmers.

Some functions are defined for single column relations only, e.g.

```
AVG[SAL¦EMP]
```

gives the average salary of all employees. Functions such as MAX, MIN and SUM are also defined.

Since the result of applying one of these functions is a scalar (non-relational) value, the function application may be embedded in the predicate of a relational expression. However, the use of embedded expressions could result in name conflicts because record selectors appearing in the short form are unqualified. To avoid such conflicts, unqualified selectors are restricted to have meaning only within the current expression (delimited by square brackets), and are not accessible within embedded relational expressions. For example,

Q3: What is the average salary of all employees who earn more than the average salary for all employees?

```
Q3:        AVG[SAL¦EMP: SAL > AVG[SAL¦EMP]]
```

The first use of SAL stands for the desired column. The second use is to get qualifying tuples, as in [EMP:SAL>1000]. The third use is in the embedded relational expression. Since the square brackets delimit the scope of meaning for the terms of the expression, this third use of SAL is distinct from the other two. This is best seen by considering the same query stated in "long form."

```
AVG[FE X:EMP
        X.SAL>AVG[FE Y:EMP TRUE => Y.SAL]
                => X.SAL]
```

Here, the three uses of SAL may be clearer, but the query itself may be harder to read because of the focus on the two variables which are not actually required.

In the above query, AVG[SAL¦EMP] appears inside the predicate of the relational expression. Since the value of AVG[SAL¦EMP] is constant it need only be computed once. However, it is possible to construct relational expressions in which embedded expressions must be recomputed each time through the loop. Consider

Q4: List employees who earn more than the average salary for their jobs.

```
Q4:     FE E:EMP
          E.SAL > AVG[SAL¦EMP:JOB = E.JOB] => E.NAME
```

Here, since E.JOB may have a different value each time through the loop, the inner expression may have to be evaluated many times.

We close this section with a discussion of user defined functions on relations. This facility is desirable since it is unlikely that the built-in functions provided (AVG, MAX, MIN, SUM, etc.) will be sufficient for all users. Although it is possible to define functions which take entire relations as arguments, this approach is undesirable for two reasons. First, since all functions of this form have to iterate through the tuples of the relation, it is better to make this iteration implicit in the notation. Second, in order to pass the relation as an argument it would be necessary to materialize the relation in its entirity. Instead, we view such functions as co-routines [10], where the function is repeatedly resumed to process each of the tuples in turn. In addition, it will be necessary to perform some initialization before any tuples are passed to the co-routine, and final computations after all tuples have been processed in order to compute the result. The general form for such functions is as follows

```
FUNCTION F(X1:T1,...,XN:TN): TRESULT
  BEGIN
        {declare locals}
        ON ENTRY DO {entry block}
        ON EXIT RETURN {exit block}
        {body of F using RESUME}
  END
```

Here, F would be applied to a relational expression which produces an n column relation, where Ti is the type for the ith column. The {entry block} is executed before any tuples are passed to F. When each tuple is passed to F, Xi is bound to the value for the ith column (1<i<n). Whenever F is reentered, it is restarted from the last resumption point. Finally, the exit block is executed after all tuples have been processed in order to obtain the value of the function (which could be a tuple). For example, AVG could be defined as follows.

```
FUNCTION AVG(X:INT):REAL
  BEGIN
    DECL  COUNT,SUM:INT;
    ON ENTRY DO
        BEGIN COUNT <- 0 ; SUM <- 0 END;
    ON EXIT RETURN BEGIN SUM/COUNT END;

    WHILE TRUE DO
        COUNT<-COUNT+1;
        SUM <- SUM+X;
        RESUME
    END
```

```
END;
```

In this context it is reasonable to pass expression values to such functions, e.g.

```
AVG[SAL+COM¦EMP]
```

Here, the sum of the two columns is computed and is passed to AVG.

User defined functions can pose problems with respect to data base security because of the potential for infinite loops. These problems can be avoided if the functions are shown to be well behaved through the use of existing program verifcation techniques.

5. Nested Iterations.

Up to this point, the range of a FOR EACH variable has always been a constant relation. A relational expression could be used instead. For example,

```
[FE X:[EMP:JOB = "PROGRAMMER"]
        X.SAL>1000 => X.NAME]
```

Here X iterates over only those tuples for programmers. The result is the names of programmers who earn over 10000. Of course, this query could have been expressed as an iteration over the entire EMP relation.

```
[FE X:EMP
   X.JOB = "PROGRAMMER" AND X.SAL>10000 => X.NAME]
```

The utility of iterating over relational expressions becomes clear when we consider the issue of duplicate tuples. As mentioned earlier, the result of a relational expression can contain duplicate elements, e.g. [SAL¦EMP]. It is sometimes desirable to have the result of the expression contain no duplicate elements. This is specified in UNEQ by using two vertical bars in the "short" form of the expression, or by including the keyword UNIQUE in the long form, e.g.

```
[SAL¦¦EMP]
```

or

```
[FE X:EMP  TRUE => UNIQUE X.SAL].
```

Thus, COUNT[SAL¦EMP] simply produces the count of the number of tuples in EMP but COUNT[SAL¦¦EMP] gives the count of the number

of different salaries of EMP. [DNO¦¦EMP:JOB="PROGRAMMER"] gives a list of the different departments that employ programmers.

A relational expression which contains no duplicates is useful as the range of a FOR EACH variable whenever one wishes to perform some computation for each of the different values of some column. Consider

Q5: List departments and the average salary for employees in the departments.

```
Q5:          FE D:[DNO¦¦EMP]
               TRUE => D, AVG[SAL¦EMP:DNO=D]
```

Here, D iterates over the different departments of EMP. Since [DNO¦¦EMP] is a single column relation we allow D to be used without a selector (as opposed to D.DNO). Also, we allow an expression to appear as part of the output list.

Some queries may be written as nested iterations. For example,

Q6: Find employees who earn more than their managers.

Returning to our programming language notation for a moment, a programmer might write

```
FOR E<-1:LENGTH(EMP) DO
  FOR M<-1:LENGTH(EMP) DO
    IF EMP[E].MGR=EMP[M].EMPNO AND
        EMP[E].SAL>EMP[M].SAL
          THEN PRINTLINE(E.NAME)
```

Here, E iterates over the employees and the body of the FOR loop is itself a FOR loop which iterates over the employees until it finds E's manager and then determines if E's salary is greater than his manager's salary. This nested iteration translates into UNEQ as a nested FOR EACH,

```
FE E:EMP
  FE M:EMP
    E.MGR = M.EMPNO AND E.SAL > M.SAL => E.NAME
```

Here, the inner FE is executed once for each binding of E.

In programming, it is often the case that the range of an inner loop depends upon the value of the iteration variable from the outer loop, e.g.

```
FOR I <- 1:N DO
  FOR J<- 1:I DO
```

BEGIN ... END

In UNEQ a similar effect is achieved when the range of the inner FE is a relational expression which uses the outer variable. Consider

Q7: List department numbers and employees who earn more than the average salary for their departments.

```
Q7:          FE D:[DNO||EMP]
               FE E:[EMP:DNO=D]
                     E.SAL > AVG[SAL|EMP:DNO=D] => D,E.NAME
```

Here, we iterate over distinct departments and for each department, we consider all employees within that department. Q7 could also be written as

```
FE D:[DNO||EMP]
  FE E:EMP
    E.DNO=D AND E.SAL>AVG[SAL|EMP:DNO=D] => D, E.NAME
```

or as

```
FE D:[DNO||EMP]
  FE E:[EMP:DNO=D AND E.SAL>AVG[SAL|EMP:DNO=D]
        TRUE => D,E.NAME
```

Choice between the three is a matter of taste. The author prefers the first version in that it is a natural expression of "for each department and for each employee within that department...".

It is sometimes desirable to declare a so-called temporary relation in order to improve the clarity of the query. In UNEQ this may be achieved by associating a name with a relational expression just before the predicate portion of the query.

Q8: List departments and their average salaries and commissions.

```
Q8:    FE D:[DNO||EMP]
         LET E BE [SAL,COM|EMP:DNO=D]
               TRUE=>D,AVG[SAL|E], AVG[COM|E]
```

We close this section with a query in which a relational expression is included in the list of objects to be printed on the output device.

Q9: List all parts and the cities where the parts are used.

```
Q9:          FE P:[PART||USAGE]
               FE D:[DNO|USAGE:PART=P]
                 TRUE=> UNIQUE P, [LOC|DEPT:DNO=D]
```

When a relational expression is used in this context it must evaluate to a single column, single tuple relation, in which case the single scalar value contained in the relation is output.

6. Other Features

In this section we consider some additional aspects of the query language, including set operations on relations, queries which produce scalar values, and an aditional linquistic form which may be used as an alternative to FOR EACH in some situations.

So far, the predicate of a FOR EACH may only be a boolean expression in terms of scalar objects. A relational expression may only appear in such an expression if it is prefixed with a function, i.e. if a scalar quantity is to be computed. However, it is often necessary to express predicates in terms of relationships among relational expressions themselves. Towards this end, we allow the following set operations on records and relations to appear in predicates.

$$\{\cup, \cap, -\}: \text{REL} \times \text{REL} \rightarrow \text{REL}$$

$$\{\subseteq, =, \langle\rangle\}: \text{REL} \times \text{REL} \rightarrow \text{BOOL}$$

$$\{\text{IN}\}: \text{RECORD} \times \text{REL} \rightarrow \text{BOOL}$$

All of these operations have their obvious definitions from set theory. We adopt the convention that "[]" represents the null relation. With respect to duplicate tuples, multiple instances are removed from operands before the operations are applied.

Queries 10,11,12 compute information about suppliers using set operations. Because these queries appear to be complex, we will step slowly through the construction of the first one. The others should then be obvious.

Q10: List names of suppliers who supply at least one part supplied by ACME.

First, we see that we wish to consider each of the distinct suppliers, and to list those that satisfy some predicate. Thus we have

```
FE S:[SUPPLIER¦¦SUPPLY]
   {predicate on S} => S
```

We need an expression to capture "parts supplied by ACME," i.e. [PART¦SUPPLY: SUPPLIER = "ACME"] and since we wish to compare this set with parts for the particular supplier we need [PART¦SUPPLY:SUPPLIER = S]. "At least one" means that the two expressions have at least one element in common, i.e. their intersection is non empty, this yields

```
Q10:    FE  S:[SUPPLIER¦¦SUPPLY]
              [PART¦SUPPLY:SUPPLIER = S] ∩
                   [PART¦SUPPLY:SUPPLIER = "ACME"] <> [] => S
```

Q11 and Q12 are similar except that different predicates are required.

Q11: List names of suppliers who do not supply part P2.

```
Q11:    FE S:[SUPPLIER¦¦SUPPLY]
           NOT("P2" IN [PART¦SUPPLY:SUPPLIER=S]) => S
```

Q12: List suppliers who supply all parts.

```
Q12:    FE  S:[SUPPLIER¦¦SUPPLY]
           [PART¦SUPPLY] = [PART¦SUPPLY: SUPPLIER = S] => S
```

Set operations may also be used to combine relational expressions in order to produce a new relation to be listed on an output device.

Q13: List all parts supplied by ACME or ACE.

```
[PART¦SUPPLY:SUPPLIER = "ACME"] U [PART¦SUPPLY:SUPPLIER = "ACE"]
```

Some queries require either "yes" or "no" answers or a scalar value to be computed. To allow for this, any expression that can appear in a predicate may simply stand alone:

Q14: How many programmers are in the organization.

```
Q14:            COUNT[EMP: JOB="PROGRAMMER"]
```

Q15: Are there more programmers than clerks in the organization?

```
Q15:    COUNT[EMP: JOB="PROGRAMMER"] > COUNT[EMP: JOB="CLERK"]
```

The final query for the section requires some explanation.

Q16: Do all departments use parts supplied by ACME?

This is equivalent to asking if the number of departments in the organization is equal to the number of departments that use parts supplied by ACME? Thus, we start with

```
COUNT[DNO¦¦USAGE] = COUNT [?]
```

where "?" stands for the departments that use parts supplied by ACME. To determine this set we must consider each department and see if any of the parts used by the department are supplied by ACME, i.e.

```
FE D:[DNO¦¦USAGE]
  FE P:[PART¦USAGE:DNO=D]
        "ACME" IN [SUPPLIER¦SUPPLY: PART=P] => D
```

However, if more than one part used by a given department is supplied by ACME, then D will appear more than once. Thus we must indicate that we wish to have no duplicates in the output.

```
Q16:    COUNT[DNO¦¦USAGE]=
          COUNT[  FE:[DNO¦¦USAGE]
                    FE P:[PART¦USAGE:DNO=D]
                        "ACME" IN [SUPPLIER¦SUPPLY:PART=P]
                                => UNIQUE D]
```

Although this query is correct, it is slightly unnatural because for a given D we would really like to stop the inner loop as soon as some part supplied by ACME is found, i.e. to stop when the existence of such a part is determined. We introduce the notation FOR SOME (FS) which gives the same result as the version above. However, "UNIQUE" is not required since the inner loop terminates (for a given value of D) as soon as some P is found.

```
Q16:    COUNT[DNO¦¦USAGE]=
          COUNT[ FE D:[DNO¦¦USAGE]
                 FS P:[PART¦USAGE:DNO=D]
                   "ACME" IN [SUPPLIER¦SUPPLY:PART=P]
                      => D]
```

Since "FOR SOME" may be considered to mean "FOR 1" it is tempting to allow any integer value to appear instead of SOME, e.g. FOR 5, which in the above query would determine departments that use exactly five parts supplied by ACME. We do not include this facility since we believe that while the concept of existence (FOR SOME) is natural for the former query, the latter query is best handled using existing mechanisms, e.g.

```
COUNT [DNO¦¦USAGE]=
 COUNT [ FE D:[DNO¦¦USAGE]
           COUNT [ FE P:[PART¦USAGE:DNO=D]
                     "ACME" IN [SUPPLIER¦SUPPLY:PART=P]
                          => P]   = 5
                 => D]
```

7. A Uniform Notation

In the introduction we mentioned that UNEQ provides a uniform notation for expressing queries. A query is expressed as a relational expression (perhaps with the outermost square brackets removed to force listing on an output device), where a relational expression is a combination of relational expressions using the

set operations, or a single FOR EACH expression (in either short or long forms), where a FOR EACH consists of a (possibly nested) iteration over either a relation or a relational expression, with a qualifying predicate used to determine the elements to be output into the computed relation. The predicate is itself an expression, defined on scaler or relational expressions. Thus, the relational expression is a unifying construct whose recursive definition allows any query to be expressed.

The advantages of using a single construct to express queries are fairly obvious. First, there is less syntax and semantics to learn. Aside from the basic concept of a FOR EACH, all of the expression mechanisms are similar to those found in many high-level (or very high level) programming languages.

The second, and perhaps most important, advantage is that since there is a single notation, there is only one approach used in writing a query. One first chooses the relation (or expression) over which one wishes to iterate. The next step is to determine the information to be output. Finally, one constructs the predicate which determines the values of the iteration variable that qualify. With this method one can readily translate a query from English into UNEQ. The major imperative of the sentence determines the range of the FOR EACH and the information to be output. The various subordinate clauses of the query are translated into relational expressions used in the predicate.

Let us reconsider the queries discussed in section 2.

Q-A: Find the names of employees who work in Detroit.

Q-A:
```
FE E:EMP
  "DETROIT" = [LOC¦DEPT: DNO=E.DNO] => E.NAME
```

Q-B: List the suppliers which supply all parts used by department 50.

Q-B:
```
FE S:[SUPPLIER¦¦SUPPPLY]
  [PART¦USAGE: DNO = 50] ⊆ [PART¦SUPPLY: SUPPLIER = S]
        => S]
```

Q-C: For each department, find the average salary of those employees whose salaries are greater than the average salary of their department.

Q-C:
```
FE D:[DNO¦¦EMP]
  TRUE => D,
          AVG[SAL¦EMP : DNO = D AND
                        SAL>AVG[SAL¦EMP:DNO=D]]
```

When written in UNEQ, each query may be read back in a form which is similar to the way in which the query is stated. Consider the following restatements of the queries which have been indented to reflect their counterparts in UNEQ.

```
Q-A:    For each employee,
          if Detroit is the location of that employee's dept.
               then list the employee's name.


Q-B:    For each distinct supplier,
          if the set of parts supplied by dept. 50 is equal to
             the set of parts supplied by this supplier
                  then list the supplier.

Q-C:    For each distinct department,
             list the department and
                  the average salary of employees
                    who are in the department and whose
                   .salaries exceed the average salary for the dept.
```

The major objections to the language style of UNEQ are that, first, by giving the user a model in which to express how the query might be processed, the data base system may be constrained to process the query in this way. We maintain, however, that allowing the user to use such a model makes writing the queries easier than in an environment in which all notions of processing have been removed. The latter is reminiscent of verification systems in which the loop invariant is substantially harder to write than the loop itself.

Second, the processing objection may itself be erroneous in that the UNEQ language processor may be able to "deconstrain" the query. For example, in Q6, the two FOR EACHs may be interchanged by UNEQ without changing the meaning of the expression. In Q7, we demonstrated that by transforming the predicate the inner FE could be changed to iterate over the entire relation. Having transformed the predicate in this way, the nested FEs may be interchanged

Finally, it may be argued that the iteration model and the concise notation may be difficult for non-programming users to understand. We believe that for more complex queries the model gives the user a basis for understanding the processing and thus much of the magic is removed. Although a keyword notation could be developed, we believe that considerable power is derived from the notation, especially when relational expressions are used in predicates. These considerations are the proper domain of a human factors study and we plan to conduct such a study in the future using an implementation of UNEQ developed as as an alternative query langugage for the INGRES system.

ACKNOWLEDGEMENTS

Bruce Englar, Pat Griffiths, Randy Katz, Larry Rowe, and Mike Stonebraker all read earlier versions of this manuscript. I wish to thank them for their many valuable comments.

REFERENCES

1. Astrahan, M.M et. al., "System R: A Relational Approach to Database Management", ACM Transactions on Database Systems, Vol 1. No. 2, June 1976, pp. 97-137.

2. Allman, E. and Stonebraker, M., "Embedding a Relational Data Sublanguage in a General Purpose Programming Language", SIGPLAN Notices, Special Issue, March 1976, pp. 25-35.

3. Jensen, K. and Wirth, N., "PASCAL Users Manual and Report", (2nd ed.), Springer-Verlag, 1975.

4. Wegbreit, B., "The ECL Programming System", FJCC 1971, pp. 253-262.

5. VanWijngaarden, A. (editor), "Report on the Algorithmic Language ALGOL 68", Numerishe Mathematik, 14 (1969).

6. Held, G. D., Stonebraker, M., and Wong, E., "INGRES - A Relational Database Management System", Proceedings 1975 NCC, AFIPS Press, 1975.

7. Perlis, A. J., "The Synthesis of Algorithmic Systems", Proc. of 21st National Conference, ACM (1966), Thompson Book Co, Washington, D.C., pp. 1-6.

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

9. Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM Vol. 13, No. 6 (June 1970), pp. 377-387.

10. Conway, M.E., "Design of a Separable Transition-Diagram Compiler", CACM Vol. 6, No. 7, (July 1963), pp. 396-408.

11. Chamberlin, D. et. al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", IBM Journal of Research and Development, 20, 6, November 1976, pp. 560-575.

12. Schmidt, J. W., "Some High-level Language Constructs for Data of Type Relation", Proceedings 1977 ACM-SIGMOD Conference on the Management of Data, Toronto, Canada, August 1977.

13. Boyce, R. F., et. al., "Specifying Queries as Relational Expressions: SQUARE", CACM, Vol. 18 No. 11,(November 1975), pp.621-628.

14. Chang, C. L., "DEDUCE --- A Deductive Query Language for Relational Data Bases", Pattern Recognition and Artificial Intelligence, Academic Press 1976.

15. Rulifson, J. F., et.al., "QA4: A Language for Writing Problem Solving Programs", Proceedings IFIP Congress, 1968.