OBSERVATIONS ON DATA MANIPULATION LANGUAGES AND THEIR EMBEDDING IN GENERAL PURPOSE PROGRAMMING LANGUAGES

by

M. Stonebraker and L. A. Rowe

Memorandum No. UCB/ERL M77/53

29 July 1977

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

OBSERVATIONS ON DATA MANIPULATION LANGUAGES AND THEIR EMBEDDING IN GENERAL PURPOSE PROGRAMMING LANGUAGES

Michael Stonebraker and Lawrence A. Rowe

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

ABSTRACT

Many data base query languages, both stand-alone and coupled to a general purpose programming language, have been proposed: A number of issues that various designs have addressed in different ways are treated in this paper. These issues include the specification of performance options, side effects, implicitness, the handling of types and the time of binding. In all cases, the emphasis is on a comparative analysis, rather than on an exhaustive survey of proposals. Several general observations on language design for data base access are also made.

1. INTRODUCTION

In recent years many data manipulation languages have been proposed. This paper attempts to explore issues that these languages face and to indicate the range of possible solutions. In this discussion, which occurs in Section 2, no attempt is made to exhaustively list languages, rather, we contain our remarks to a smaller set of representative cases. Moreover, we specifically exclude low-level, record-at-a-time languages, commonly called "access methods" (e.g., VSAM [KEEH74], ISAM [IBM 66], etc.). In addition, natural language interfaces, such as Rendezvous [CODD74] and TORUS [MYLO75], are also excluded.

In Section 3, the problems associated with coupling data manipulation languages to general purpose programming languages are discussed. Again, the discussion treats representative cases and we make no effort to be complete.

In the last section, some general observations are made on language design as it relates to data base systems.

2. PROBLEMS OF DATA MANIPULATION LANGUAGES

The examples in this section are drawn from the following languages:

- CODASYL Data Manipulation Language [CODA71, CODA74],
- Data Language Alpha [CODD71],
- Data Language/1 (IMS Data Base Management System [IBM 75]),
- Link and Selector Language (OMEGA Data Base Management System [TSIC76b]),
- QUEL (INGRES Data Base Management System [HELD75, STON76]),
- Query by Example [ZLOO75],
- Relational Algebra [CODD72, PECH75, SMIT75],
- SEQUEL 2 (System R Data Base Management System [ASTR76, CHAM76]).

In all cases we use the language supported to mean the data base system when appropriate. For example, we refer to features of Data Language/1 (DL/1) which might more appropriately be called features of IMS.

In the following subsections we discuss issues related to data definition, performance specification, side effects, tuple variables, procedural versus nonprocedural specification, and implicit specification.

2.1. Relation to Data Definition

All data manipulation languages (DML) have an associated data definition language (DDL) to describe the data on which actions are performed. The DDL acts as a specification mechanism for the data in a data base and cannot be avoided. Many DML's which have been proposed but not implemented (e.g., Data Language/Alpha and Relational Algebra) would have to be augmented with a DDL.

Research sponsored by the Army Research Office Grant DAAG29-76-G-0245.

For most relational languages, the DDL is a "create relation" command. Syntactically, it can be expressed as:

```
CREATE NAME(DOMAIN_1=format,DOMAIN_2=format,...)
```

The data definition for a relation must specify a relation name, column names, and the data type for each column. For network and hierarchical languages, the DDL contains additional information about record inter-relationships.

Because there are two languages, constructs can be moved from the DML to the DDL, and vice-versa. This is illustrated by examining three possibilites: (1) minimal information in the DDL, (2) linking information in the DDL, and (3) maximal information in the DDL.

2.1.1. Minimal Information in the DDL (e.g., QUEL and SEQUEL 2).

In this case, only a create relation function exists in the DDL. For example, one can specify two relations as follows:

```
CREATE EMPLOYEE(NAME=C20,SALARY=F4,DEPT=C10,MANAGER=C20)[1]
CREATE DEPT(DNAME=C10,SALES_VOL=I4,FLOOR=I2)
```

Any association between the EMPLOYEE and DEPT relations must appear in the data manipulation language. Thus, to express the query "find the names of all employees on the first floor" in SEQUEL 2 one requires

```
SELECT NAME
FROM   EMPLOYEE
WHERE  DEPT IN
       SELECT   DNAME
       FROM     DEPT
       WHERE    FLOOR = 1
```

Notice that linking information (i.e., that DEPT in the EMPLOYEE relation must match DNAME in the DEPT relation) appears in the data manipulation language.

2.1.2. Link Information in the DDL (e.g., CODASYL, DL/1, and LSL)

In this case, linking information appears in the DDL as opposed to the DML as shown above. In IMS for example, linking information is specified in a hierarchy definition. In the CODASYL proposal, it is specified by the definition of sets. Lastly, in the Link and Selector Language (LSL) it is specified by the existence of explicitly defined LINKS.

This case is illustrated by an example from LSL. Besides creating the two relations via DDL commands, one can create a LINK as follows:

```
DEFINE LINK IS_IN
FROM EMPLOYEE
TO DEPT
BY EMPLOYEE.DEPT=DEPT.DNAME
```

As a result of the LINK IS_IN, our example query is written

```
GIVE NAME FROM EMPLOYEE
LINKED BY IS_IN
WHERE FLOOR=1
```

The linking information has been moved to the DDL. Consequently, there is more information in the DDL and less in the DML.

2.1.3. Maximal Information in the DDL

In this case, all information is moved into the DDL. There is no language that proposes doing this; hence, a hypothetical example is given to illustrate our point. Of the languages considered, LSL comes closest to this point of view. The DDL specification would be:

---

[1] In the type specification, C specifies a string of characters, I an integer, and F a floating point number of the indicated width.

```
CREATE  EMPLOYEE( . . . )
CREATE  DEPT( . . . )
DEFINE  FIRST_FLOOR AS
        SELECT  NAME
        FROM    EMPLOYEE
        WHERE   DEPT IN
                SELECT  DNAME
                FROM    DEPT
                WHERE   FLOOR = 1
```

and our example query would be invoked by

```
RUN FIRST_FLOOR
```

The entire manipulation facility has been moved into the DDL and only predefined transactions are invoked in the DML. It might be noted that views [CHAM75, STON75a] allow SEQUEL 2 and QUEL to be augmented with similar facilities.

The point to be stressed is that information can be moved back and forth between a data definition language and a data manipulation language. Furthermore, all implemented systems separate functions between the DML and DDL. It is debatable how much information should go in each language. However, any discussion of the relative simplicity or cleanliness of a DML must include the DDL because any DML can be made arbitrarily simple by making its associated DDL more complicated.

2.2. Performance Information

All implemented systems known to the authors include a facility to specify performance information. For relational systems, this information includes: (1) the storage structure used to physically hold the relation, (2) auxilary access paths that might exist (e.g., secondary indices [SCHK75] and LINKS [TSIC75]), and (3) blocking factors for secondary storage, buffering strategies, and hash algorithms. For the CODASYL proposal, this information includes: (1) the LOCATION MODE for each record type, (2) how records are to be allocated to files (AREA attribute), and (3) clustering effects between records. For IMS, this information includes: (1) an access method choice (e.g., HDAM, HSAM, etc.) and (2) secondary data sets that should be maintained.

This information can be specified implicitly by use patterns or explicitly by a knowledgable user (typically a data base administrator). Three possibilities are discussed in the following paragraphs.

No implemented system known to us uses only implicitly specified performance information, although it has been widely discussed in the literature [HAMM76b]. The basic idea is to automatically adapt the implementation to changing use requirements by physical restructuring. Similar ideas are being discussed in programming language circles with respect to core resident data structures. This is an interesting research area that warrants further work.

QUEL and SEQUEL 2 support a mixed strategy where minor performance information is gathered implicitly, while major performance information is specified explicitly. For example, hash functions, blocking factors, and the like are automatically handled by the system. Secondary indices and clustering decisions, on the other hand, must be specified explicitly by some user. The specification of this information is separated from the DDL and can be changed without impacting any data manipulation programs.

Finally, all performance information can be specified explicitly in the DDL. Both CODASYL and IMS adopt this strategy, mixing the specification of performance information with data definition. For these systems, it is often impossible to change performance information without impacting data manipulation programs.

There is a definite trend toward implicit specification of performance information. Until research on automatic restructuring provides the algorithms and knowledge representations needed to achieve an acceptable level of performance, explicit specification will be necessary in commercial systems. Such performance information should not be visible to the end user and should be separated from the DDL and DML.

2.3. Side Effects

Several data manipulation languages have side effects. A side effect is an implicit change in the execution environment as a result of a DML command. One example of a side effect is shown in the following IMS example. Using the employee and department example described above, a stylized definition of a hierarchy for DL/1 is

```
DEFINE  RECORD  DEPT
        FIELD   DNAME
        FIELD   SALES_VOL
        FIELD   FLOOR
DEFINE  SUBORDINATE  RECORD  EMPLOYEE
        FIELD   NAME
        FIELD   SALARY
        FIELD   MANAGER
```

Notice that the linking information (the DEPT field in the EMPLOYEE record) need not be present because it is implicit in the hierarchy definition. Each employee record is stored physically subordinate to the record describing his department. In stylized DL/1, the shoe department is deleted by

```
GET (DNAME='SHOE')
DELETE
```

However, it has the added side effect of removing all employees who work in the shoe department. DL/1 commands do not have to be given explicitly to delete the employee records.

The argument given in favor of side effects is that they allow the number of data manipulation commands to be reduced by having some actions be effects of others. Moreover, execution efficiencies may be realized. The argument against side effects is that they lead to data base transactions that are hard to understand and debug. Using side effects is considered poor programming practice because the desired action is not stated explicitly which often leads to misunderstanding the action.

Data manipulation languages are divided on this issue. Side effects are present in CODASYL (mandatory and automatic set membership, set selection criteria, and data base procedures), DL/1 (removing a record automatically removes all descendents), and SEQUEL 2 (data base procedures, which are called "triggers" [ESWA76]). QUEL, Data Language Alpha, and Relational Algebra do not have side effects.

2.4. Presence of Tuple Variables

This issue is illustrated by examples from SEQUEL 2 and QUEL. In Section 2.1.1, a query in SEQUEL 2 was given that finds all employees who work on the first floor. Another example is to find all employees who earn more than their managers, which can be written:

```
SELECT  NAME
FROM    E IN EMPLOYEE
WHERE   MANAGER IN
        SELECT  NAME
        FROM    EMPLOYEE
        WHERE   SALARY < E.SALARY
```

The query in Section 2.1.1 makes no explicit reference to a specific row in the EMPLOYEE or DEPT relation. However, the second query requires an explicit iteration variable (E in the example). This variable iterates over all rows of EMPLOYEE, at any point indicating a specific one. As such, it is often called a "tuple variable" [CODD71]. SEQUEL 2 includes tuple variables only where necessary.

Now consider these same two queries written in QUEL. The first floor employees query is

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPT
RETRIEVE (E.NAME) WHERE
        E.DEPT = D.DNAME AND D.FLOOR = 1
```

while the employees earning more than their managers is

```
RANGE OF E IS EMPLOYEE
RANGE OF M IS EMPLOYEE
RETRIEVE (E.NAME) WHERE
        E.MANAGER = M.MANAGER
        AND E.SALARY > M.SALARY
```

Tuple variables are present for every domain name referenced in a query. No attempt is made to avoid them.

There are three possibilities for dealing with tuple variables: (1) they are never explicitly present (e.g., Relational Algebra), (2) they are present only when needed (e.g., SEQUEL 2 and LSL), and (3) they are always present (Data Language Alpha and QUEL). Note that DL/1 and CODASYL do not have a notion of iterators because they are

record-at-a-time languages.

The argument for the elimination of tuple variables is that the resulting DML is simpler for easy queries and a casual user (who may only need elementary features) can avoid learning a confusing concept. The argument for the presence of tuple variables is that the DML can have a semantically clean and consistent interpretation. Moreover, while easier queries may be harder to write, difficult queries are easier (and easier to optimize, see the next section). Lastly, most casual users will be using a "customized end user facility," so arguments against tuple variables do not hold because sophisticated applications programmers will be the only ones learning the language.

We believe that the human factors of data manipulation languages [LOCH77, MCDO75a, REIS75, REIS76] are not well enough understood to authoritatively address this issue. Moreover, any arguments not based on human factors experiments are simply untested expressions of belief on the part of the protagonists. Speculation about human factors responses is no substitute for actual experiments. We hope to see more such experiments in the future.

2.5. Procedural versus Nonprocedural Languages

In the previous section, examples were given in both SEQUEL 2 and QUEL. SEQUEL 2 is a procedural language because a processing order is implied by the query specification (from the inside block out, as noted in [ASTR75]). QUEL, on the other hand, does not induce a preferred processing order. Examples of procedural languages are: SEQUEL 2, LSL, and Relational Algebra. Examples of nonprocedural languages are: Data Language Alpha, QUEL, CUPID [MCDO75b], and Query by Example.

Nonprocedural languages are amenable to elaborate optimization [WONG76]. Complex transformations for efficiency can also be preformed on procedural languages (e.g., see [PECH75, SMIT75]), but the ordering makes optimization more tedious.

The controversy between procedural and nonprocedural specification is not new to computer science. Other debatable points between these approaches are the ease of writing down a desired query and the ease of understanding a query written by someone else. The argument may well be moot (as happened, for example, in artificial intelligence and data abstraction specification) because it likely depends on the background and experience of the individual user.

2.6. Implicitness

Looking again at the examples discussed in Section 2.4, notice that SEQUEL 2 implicitly binds domain names to a relation by scoping rules. In effect, all domains indicated in a particular SEQUEL 2 block are associated with the relation indicated in the "FROM RELATION_NAME" clause (unless a tuple variable indicates otherwise). Thus, a degree of implicitness is present in SEQUEL 2 in its treatment of tuple variables. Defaulting them is done by an implicit convention. This implicitness was not present in the QUEL examples.

Besides tuple variables, other features can be made implicit. For example, Query by Example makes boolean operators implicit as shown in the following two examples. To find the names of all employees who work for Jones or Smith, one writes

EMPLOYEE	NAME	AGE	SALARY	MANAGER	DEPT
	p.example			Jones	
	p.example			Smith	

On the other hand, to find the names of all employees who work for Jones and Smith, one writes

EMPLOYEE	NAME	AGE	SALARY	MANAGER	DEPT
	p.example			Jones	
				Smith	

The fact that there are two rows beneath MANAGER indicate a boolean connective. The difference between "and" and "or" is indicated by repeating the "p.example" (and) or by leaving the name field blank (or). Moreover, Query by Example makes equality comparisons between domains the default case unless overridden by a different operator.

These examples indicate some of the things that can be made implicit. All data manipulation languages make a commitment as to what should be implicit and what should be explicit.

3. COUPLING A PROGRAMMING LANGUAGE TO A DATA BASE SYSTEM

Three approaches to providing data base access from a programming language are:

1. defining subroutines that execute data base requests when called (e.g., IMS-PL/1 [IBM75, TSIC76a]),
2. embedding data base constructs into an existing language and using a preprocessor to translate these constructs into run-time calls on the data base system (e.g., DBTG-COBOL, EQUEL [ALLM76], and SEQUEL 2 [CHAM76]), or
3. designing a new programming language in which data base facilities are integrated into the language environment.

Previous work has focused on the first two approaches, as indicated by the references cited, although more recently several groups have begun work on new languages [PREN77, WASS77, WELL76].

Regardless of which approach is used many fundamental design issues must be resolved. Using sample programs written in four representative languages (DBTG-COBOL, EQUEL, IMS-PL/1, and SEQUEL 2), we discuss possible solutions to some of these issues. The sample programs are presented in the first subsection. Following this, the design issues are discussed. Finally, some ideas suggested for a new language are briefly described. Readers familar with the languages may want to skip the first subsection.

3.1. Sample Programs

The example problem is to write a program that executes a procedure for each employee working in the shoe department. How each program accomplishes the task is explained briefly, so that an understanding of the languages is not required. As much of the DDL description as possible is suppressed so that the programs can be kept to a managable length. Readers interested in the complete details are referred to the appropriate source material referenced previously.

Figure 1 shows a data structure diagram for the data base used in the DBTG and IMS programs. The relations defined in Section 2.1.1 are used for the EQUEL and SEQUEL 2 programs.

3.1.1. DBTG-COBOL

Figure 2 shows a fragment of a COBOL program that performs the desired computation[2]. In the FILE SECTION of the DATA DIVISION, the data base subschema is expanded (for this example the subschema is named EMPLOYEE-DATA-BASE). The DECLARATIVES section of the PROCEDURE DIVISION specifies the error handling routines to be invoked by the data base system when an abnormal situation arises (code "04021" indicates end-of-set). Following the DECLARATIVES section, the actual program begins.

---

[2] The program is written using DML constructs from the 1974 CODASYL report. See [CODA74, TAYL76].

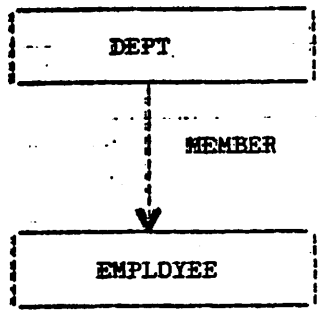

Figure 1: A Data Base Structure

```
IDENTIFICATION DIVISION.

ENVIRONMENT DIVISION.

DATA DIVISION.
   FILE SECTION.
     DB EMPLOYEE-DATA-BASE

   WORKING STORAGE SECTION.
     77 END-OF-SET PIC 9(5) "04021"

PROCEDURE DIVISION.
   DECLARATIVES.

     EXPECTED ERROR SECTION.
      USE FOR DATABASE-EXCEPTION ON "04021"
      EXPECTED ERROR HANDLING.
          EXIT.

     UNEXPECTED ERROR SECTION.
      USE FOR DATABASE-EXCEPTION ON OTHER.
      UNEXPECTED ERROR HANDLING.

              <PROCESS ERROR>

   END DECLARATIVES.

   INITIALIZATION.
      READY DBAREA.
      FIND DEPT USING "SHOE".
      IF MEMBER IS EMPTY
         GO TO FINISH.
      FIND FIRST EMPLOYEE IN MEMBER.

   NEXT-EMP.
      GET EMPLOYEE.

              <PROCESS RECORD>

      FIND NEXT EMPLOYEE IN MEMBER.
      IF DATABASE-STATUS <> END-OF-SET
         GO TO NEXT-EMP.

   FINISH.
      FINISH DBAREA.
      STOP RUN.
```

Figure 2: DBTG-COBOL Program

The desired action is accomplished by opening the data base (READY DBAREA), finding the appropriate DEPT record, and then iterating through the EMPLOYEE records in the DEPT's MEMBER set. We assume that the DEPT records are stored using a hash function on department name (LOCATION MODE IS CALC).

The GET EMPLOYEE statement moves the current EMPLOYEE record from secondary storage (or perhaps system buffers) to the work space of the program. This work space has been declared in the subschema. In fact, storage for each distinct record defined in the subschema is allocated.

3.1.2. EQUEL

An EQUEL program to perform the desired computation is shown in Figure 3. EQUEL is an embedding of QUEL into the systems programming language C [RITC73]. Lines beginning with "##" are translated by a preprocessor into procedure calls to the data base system.

First, program environment variables are declared and the data base system is initiated. The range statement binds the tuple variable to the employee relation. The retrieve statement executes the iteration body (delimited by "{" and "}", the begin-end symbols in C) once for each record satisfying the qualification. Values from the named fields in the retrieved record (i.e., name, salary, and manager) are copied into the program variables (i.e., namev, salaryv, managrv) before each iteration. If the data type of the field in the record does not match that of the program variable, the value is

```
##      char    namev[21];
##      int     salaryv;
##      char    managrv[21];

prog();
{
    /* Initiate INGRES with the appropriate data base. */

##      ingres . EmployeeDataBase

##      range of e is employee

##      retrieve (namev=e.name,salaryv=e.salary,
##                managrv=e.manager)
##      where e.dept ='SHOE'
##      {

            <process record>


##      }

}  /* end of program */
```

Figure 3: EQUEL Program

translated to the type of the destination variable (e.g., integer to real).

3.1.3. IMS-PL/1

Figure 4 shows a PL/1 program with DL/1 subroutine calls to solve the example problem. IMS is intiated which calls the procedure DL1TPL1 (DL/1 to PL/1), passing to it the base address of a shared communication area (QUERY_PCB in the program). The declarations for the procedure include: (1) a templated describing the fields in the shared communication area, (2) buffers for each record accessed, (3) character strings describing the search qualifications, and (4) local data. Data base retrievals are requested by calling PL1TDL1 (PL/1 to DL/1) with arguments describing the query.

The data base semantics of this procedure are similar to those for the DBTG-COBOL program (get DEPT record and iterate through the subordinate EMPLOYEE records). The first statement in the program gets the unique record described in the search argument

```
DL1TPL1:PROCEDURE(QUERY_PCB) OPTIONS(MAIN);

        DECLARE QUERY_PCB POINTER;

            /* DECLARE COMMUNICATION BUFFER */

        DECLARE
        1   PCB BASED(QUERY_PCB),
            2  DBNAME    CHAR(8),
            2  SEGLEVEL  CHAR(2),
            2  STATUS    CHAR(2),
            2  PROCOPT   CHAR(4),
            2  RESERVED  FIXED BINARY(31,0),
            2  SEGNAMFB  CHAR(8),
            2  KEYLENFB  FIXED BINARY(31,0),
            2  NOSENSEG  FIXED BINARY(31,0),
            2  KEYFB     CHAR(28);

            /* DECLARE RECORD BUFFERS */

        DECLARE DEPT_IO_AREA(16),
        1  DEPT DEFINED DEPT_IO_AREA,
            2  DNAME     CHAR(10),
            2  SALES_VOL FIXED BINARY(31,0),
            2  FLOOR     FIXED BINARY(15,0);

        DECLARE EMPLOYEE_IO_AREA(46),
        1   EMPLOYEE  DEFINED  EMPLOYEE_IO_AREA,
            2  NAME      CHAR(20),
            2  SALARY    FLOAT BINARY(31,0),
            2  MANAGER   CHAR(20);

            /* SET UP SEARCH ARGUMENTS */

        DECLARE  /* 'DEPT(DNAME=SHOE)' */
        1   DEPT_SA  STATIC  UNALIGNED,
            2  SEG_NAME         CHAR(5) INIT('DEPT '),
            2  LEFT_PAREN       CHAR(1) INIT('('),
            2  FIELD_NAME       CHAR(8) INIT('DNAME'),
            2  COND_OP          CHAR(2) INIT('='),
            2  SEARCH_VALUE     CHAR(4) INIT('SHOE'),
            2  RIGHT_PAREN      CHAR(1) INIT(')');

        DECLARE  /* 'EMPLOYEE' */
        1   EMPLOYEE_SA  STATIC  UNALIGNED,
            2  SEG_NAME         CHAR(9) INIT('EMPLOYEE');

            /* DECLARE SOME USEFUL VARIABLES */

        DECLARE
            SUCCESSFUL  CHAR(2) INIT('  '),
            GU          CHAR(4) INIT('GU  '),
            GNP         CHAR(4) INIT('GNP ');

            /* MAIN PROCEDURE */

        CALL PL1TDL1(4,GU,QUERY_PCB,DEPT_IO_AREA,DEPT_SA);
        IF PCB.STATUS<>SUCCESSFUL THEN RETURN;
        CALL PL1TDL1(4,GN,QUERY_PCB,EMPLOYEE_IO_AREA,EMPLOYEE_SA);
        DO WHILE (PCB.STATUS=SUCCESSFUL);

            <PROCESS RECORD>

            CALL PL1TDL1(4,GNP,QUERY_PCB,EMPLOYEE_IO_AREA,EMPLOYEE_SA);
        END;
END DL1TPL1;
```

Figure 4: IMS-PL/1 Program

(DEPT_SA). Following this, subordinate records are fetched and processed until the data base status (STATUS in the shared communication buffer) indicates that there are no more records.

3.1.4. SEQUEL 2

Figure 5 shows a PL/1 program with embedded SEQUEL 2 commands to solve the problem. Asterisk ("*") plays the same role as "##" in EQUEL. First, program variables are declared, followed by the specification of the query (the query is not executed by the LET statement, only specified). The query is bound to a cursor variable (C1 in the example) which is used for all later references to the retrieval. To execute the query, it is opened (at which time the retrieval is started), individual records are fetched, and values bound to the program variables (those specified after a colon in the LET statement). After all records are processed (signaled by the global variable CODE), the

```
PROG:PROCEDURE OPTIONS(MAIN);
     DECLARE NAMEV   CHAR(20),
             SALARYV FLOAT BINARY(31,0),
             MANAGRV CHAR(20);

*    LET C1 BE
*            SELECT NAME:NAMEV,SALARY:SALARYV,
*                   MANAGER:MANAGRV
*            FROM EMPLOYEE
*            WHERE  DEPT = 'SHOE';

*    OPEN C1;
     DO WHILE (CODE = OK);
*            FETCH C1;        /* GET NEXT RECORD */

             <PROCESS RECORD>

     END;
*    CLOSE C1;
END PROG;
```

Figure 5: SEQUEL 2 Program

query is closed.

3.2. Design Issues

In the following paragraphs we discuss issues relating to: (1) the interface between the data base system and language environment, (2) data types and conversion, (3) the language constructs provided, (4) side effects, and (5) the tradeoff between efficiency and flexibility.

3.2.1. Data Base System - Language Environment Interface

A program passes requests to the data base system and receives data and control information in return. The typical mechanism for communication is to have a shared workspace. The issue is how much about this workspace must an applications programmer know and how is it protected from inadvertant modification.

We believe that a programmer should know as little as possible about the workspace. The more a programmer knows about it, the more likely he is to take advantage of it. For example, a programmer may intentionally modify currency pointers normally set only by a DBTG system unless specifically prohibited. This can lead to some efficiencies. However, program bugs may unintentionally write into the workspace causing strange errors. In practice, such bugs can be hard to identify and to isolate.

To use IMS-PL/1 a programmer must declare some variables in the workspace. In the other three languages the preprocessor can do storage allocation for the workspace. Thus, avoiding a preprocessor has the cost that more programming detail must be handled by the programmer.

DBTG-COBOL and IMS-PL/1 put data used by the data base system (e.g., currency pointers in DBTG) into the program address space. This means that only one copy of the data base system is needed. The disadvantage of this approach is that the data in the program

space is not protected from errors. The issue reduces to one of efficiency versus reliability.

3.2.2. Data Types and Translation

Obviously, the set of primitive types (e.g., character strings, integers, and reals) provided in the data base system and the language environment should be the same. If this is not true, an obvious conversion problem is present. For example, if data of type "date" is treated as a string in a program, illegal dates may arise (e.g., Janarch 50, -10).

Another problem is whether data of one type in the data base can be bound to a program variable declared to be another type. If so, who controls the conversion from one internal representation to another (e.g., packed decimal to integer). Does the system implicitly convert between all types or does an application programmer or a data base administrator (DBA) specify explicitly what conversions should be done?

Conversion is a complex issue because it influences efficiency, program reliability, and program complexity. Providing the user, either programmer or DBA, with explicit control over conversions provides more protection against unintended conversions. However, requiring explicit specification of conversions means that simple programs involving obvious conversions (e.g. integer to real) are more complicated. Implicit conversions, on the other hand, can degrade performance as well as allow programs to access data in ways that may not have been intended by the data base owner.

The issue of type conversion has been argued at length in the programming language community. The current trend seems to be toward strong typing[3] with limited implicit conversions with the programmer having explicit control over other conversions. In our example languages, DBTG-COBOL comes closest to this viewpoint. There is a uniform set of data types with explicit control of conversions provided to the manager of the subschema, presumably a DBA. IMS has no notion of types, so a program may do anything. EQUEL does implicit conversions between all types. SEQUEL 2 does the same but does not support all types found in PL/1.

A disadvantage of the type systems in EQUEL and SEQUEL 2 is that records, as used in the data base system, are not provided as composite structures in the language environment. For example, values from a data base record must be individually copied into program variables. This means that records can not be passed as arguments to procedures. In addition, because relations are not treated as composite structures in the language, they may not be operands to many otherwise meaningful operations.

3.2.3. Language Control Structures

The third issue relates to how hard it is to read and write programs. The questions are what control structures are provided for accessing data base records and what facilities are provided for signaling data base errors? Our bias is that concise, easy-to-understand language constructs should be provided for commonly used control structures, such as iterations, and that an exception handling facility (with reasonable default error routines) should be provided for handling unexpected errors.

In DBTG-COBOL and IMS-PL/1 iterations must be coded by explicitly fetching records and testing a return status. As can be seen by the sample programs, the DBTG and IMS programs are more confusing than the others (IMS-PL/1 is further complicated by the fact that query specifications are buried in declared structures). Part of the reason for this confusion is that both programs involve the EMPLOYEE and DEPT records, while the EQUEL and SEQUEL 2 programs happen to involve only the EMPLOYEE record. Nevertheless, a query involving both relations would be only slightly more complicated. The essential feature making the queries understandable is the iteration control structure. Notice that SEQUEL 2 requires the same explicit coding of iterations as DBTG and IMS, but an iteration involving a query is easier to understand because a separate iterative specification of the query is made in the cursor definition statement.

The availability of iterative control structures is a language issue, not a data base issue. As discussed elsewhere [MICH76, STON75b, TAYL74], iteration constructs can be provided by adding them to the preprocessor or by using a macro facility. Higher level constructs make programming easier, sometimes at the expense of execution efficiency. However, we do not believe that to be the case here.

---

[3] Note that conversion is not an issue when the language is weakly typed because a variable takes the type of its current value.

As for error handling, DBTG and EQUEL have clean exception handling facilities. IMS-PL/1 and SEQUEL 2 depend on the user to test explicitly a return code.

3.2.4. Side Effects

The issue of side effects arises in language couplings too (see Section 2.3). Basically, the arguments are the same, efficiency versus understandability. However, understandability is more critical here because programs often are used for many years. They are modified periodically to meet new requirements or to fix bugs. The use of side effects makes modifying a program harder. This is because they cause actions to happen which are not explicitly stated. Thus, it is harder to determine what a program is doing. Also, small changes frequently cause unforseen complications.

Of course, programs which are to be executed for years are precisely the ones where small efficiencies lead to big savings. DBTG-COBOL and IMS-PL/1 make use of side effects.

3.2.5. Efficiency and Flexibilty

The final design issue concerns the tradeoff between efficiency and flexibility. Both DL/1 and EQUEL interpret data base requests, while DBTG and SEQUEL 2 execute compiled requests. Interpretation is much less efficient in CPU time than compilation.

On the other hand, interpretation means more flexibility because binding time is delayed. In EQUEL for example, relation and domain names can be changed at run-time. In DBTG-COBOL names are fixed at the time the application program is compiled. Dynamic changes are necessary if one wants to write a report generator or general terminal interaction program because field names and record names (relation names in the relational model) are parameters to the program.

In the first version of SEQUEL, relation and domain names were fixed at compile-time. SEQUEL 2 has an execute command that allows a string argument to be interpreted as a query request. By changing the string, any part of the query may be varied. This is a useful facility, but using it is quite complicated (storage for individual values returned by the data base must be allocated explicitly by the program at run-time). Furthermore, to change a program so that a relation is a variable instead of a constant requires a substantial change to the program text.

3.3. New Languages

Figure 6 shows a program that solves our sample problem in a proposed language being developed in part by the second author. The declarations define EMP to be a constant bound to the EMPLOYEE relation, and x to be a pointer to a record in that relation. The retrieval is specified using a general "for construct" where an iteration variable ranges over all records in the relation. The body of the for-statement is executed once for each record satisfying the qualification.

An important feature of this language is that the language and data base system support the same set of types. The issue of type conversion is less of a problem then because it only takes place if requested by the programmer. The language uses strong type-checking (i.e., variables have declared types and only values of the right type can be assigned to them) to insure value integrity (not this is only a small part of the entire data base integrity problem). Relations and records in relations are structures defined in the language so particular instances can be assigned to and passed as arguments to procedures.

A data abstraction facility is provided so that abstractions involving data bases can be maintained and used [HAMM76a].

Relation and field names can vary dynamically. For example, in the sample program EMP is declared to be a constant. To change it to a variable, the declaration of EMP is changed to

```
EMP: relation of EmployeeRec
```

Now EMP may be assigned a relation (a variable or constant entity of type relation having records of type EmployeeRec). Suppose that S was a string (perhaps read from a terminal) containing the name of a relation; to assign that relation to EMP one writes

```
EMP := relation(S)
```

where relation is a predefined function that converts a string to a relation. Because

```
procedure Prog;
  begin
    declare EmployeeRec:type = record
                                 name:char(20),
                                 salary:real,
                                 dept:char(10),
                                 manager:char(20)
                               end,
            x: ptr EmployeeRec,
            EMP = relation('EMPLOYEE');

    for x in EMP st x.dept='SHOE' do

            <process record>;

    end;

  end; {Prog}
```

Figure 6: A Proposed Language

the language uses strong type-checking, the type of the relation named in the string S must be checked at run-time. Notice that no change is required to the iteration statement.

An idea used in ECL [WEGB72] will be used in compiling the language. When enough parts of a construct are fixed (e.g., the relation name is constant) code as efficient as possible will be compiled. Otherwise, less efficient code will be compiled. This means that the tradeoff between efficiency and flexibility can be controlled by the applications programmer. Furthermore, it means that only those programs that need more flexibility must pay for it in terms of efficiency. Of course, in those cases where code is compiled to access a specific relation, programs will have to be recompiled if the definition of that relation changes.

4. GENERAL OBSERVATIONS

In this section, general observations are made on the design and use of query languages and programming language - data base system couplings.

The first observation is that the concept of "completeness" is not very useful when applied to query languages. By definition, a language is complete if it has the expressive power of a first order predicate calculus [CODD71].

There are two problems with this concept. First, what is the size of the language construct that is complete. It can be one DML command, a collection of DML commands, or a collection of DML commands in an arbitrary host language. Notice, for example, that assembly language as a DML is complete (as are most programming languages) if the third notion of size is used. Moreover, a language can be complete if multiple commands are allowed yet not be complete when the unit is a single command. Most authors prove completeness for a collection of commands not for a single command. Consequently, whenever the term "complete" is used, a reader should understand which case is meant.

The second problem with "completeness" is that it measures the expressive power of a language against an arbitrary binary standard (complete or not complete). This is a coarse measure which may not calibrate languages in a meaningful way. Perhaps, for example, arithmetic and aggregate facilities are more important than completeness. It is left as an open question to decide what is a reasonable standard by which to measure the expressive power of a data manipulation language.

The second observation on language design is that human factors studies may be more important at this time than the design of "new" query languages. As previously stated, designing a new query language and then arguing that it is better than some other languages without careful experimentation does not prove anything.

The third observation relates to data definition, data manipulation, and performance information. Each is important and all data base systems make decisions as to how this information is distributed among the different specifications to which the system has access. In particular, performance information, while important, should not be intertwined with data manipulation or data definition specifications. Queries should execute even in the absence of performance information, although perhaps much less

efficiently.

The fourth observation has to do with the implicitness of a specification, either in a query language or a programming language. Highlighting the variability in a language construct while suppressing the constant is a good rule in language design. It is possible to go too far towards implicitness as well as not far enough.

The last observation is that the development of a good programming language for interacting with a data base necessitates the design of a new programming language or the freedom to make substantive changes to the host language. The difficult aspects of providing access to a data base are how data is used in the two environments, how processing is divided between the environments, and the special needs imposed by the environments themselves. If the programming language and data base system are fixed these problems are more difficult to solve. Because new languages meet resistance in the commercial community (sometimes for good reasons), it may be a long time before data base oriented programming languages are widely used. Nevertheless, at this time we feel the development of these languages should be encouraged.

5. REFERENCES

ALLM76 Allman, E., Stonebraker, M., and Held, G., "Embedding a Relational Data Sublanguage in a General Purpose Programming Language," Proc. Conference on Data: Abstraction, Definition, and Structure, FDT, vol 8, no 2, March 1976.

ASTR75 Astrahan, M. M. and Chamberlin, D. D., "Implementation of a Structured English Query Language," Communications ACM, vol 18, no 10, October 1975.

ASTR76 Astrahan, M., et. al., "System R: A Relational Approach to Data," TODS, vol 1, no 2, June 1976.

CHAM75 Chamberlin, D.D., Gray, J. N., and Traiger, I. L., "Views, Authorization and Locking in a Relational Data Base System," Proc. 1975 National Computer Conference, Anaheim, CA, May 1975.

CHAM76 Chamberlin, D.D., et. al. "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control," IBM Journal of Research and Development, vol 20, no 6, November 1976.

CODA71 Committee on Data Systems Languages, "CODASYL Data Base Task Group Report," available from ACM, New York, NY, 1971.

CODA74 Committee on Data Systems Languages, "CODASYL Data Description Language Journal of Development," NBS Handbook #112, U.S. Department of Commerce, January 1974.

CODD71 Codd, E.F., "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, CA, November 1971.

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

CODD74 Codd, E.F., "Seven Steps to Rendezvous with the Casual User," Proc. IFIP TC-2 Working Conference on Data Definition, Cargese, Corsica, October 1974.

ESWA76 Eswaren, K., "Specification and Implementation of a Trigger Subsystem in an Integrated Data Base System," RJ1820, IBM Research, San Jose, CA.

HAMM76a Hammer, M., "Data Abstractions for Data Bases," Proc. Conference on Data: Abstraction, Definition, and Structure, FDT, vol 8, no 2, March 1976.

HAMM76b Hammer, M. and Chan, A., "Index Selection in a Self-Adaptive Data Base Management System," Proc. 1976 ACM-SIGMOD Conference on Management of Data, Washington, D.C., June 1976.

HELD75 Held, G.D. and Stonebraker, M. and Wong, E., "INGRES - A Relational Data Base Management System," Proc. 1975 National Computer Conference, Anaheim, CA, May 1975.

IBM66 IBM, "OS ISAM Logic," GY28-6618.

IBM75 IBM, Information Management System Virtual Storage (IMS/VS) Publications 1975:
General information manual, GH20-1260-3.
System application design guide, SH20-9025-2.
Application programming reference manual, SH20-9026-2.
System programming reference manual, SH20-9027-2.
Operator's reference manual, SH20-9028-1.
Utilities reference manual, SH20-9029-2.
Messages and codes reference manual, SH20-9030-2.

KEEH74 Keehn, D.G. and Lacy, J. O., "VSAM Data Set Design Parameters," IBM Systems Journal, vol 13, no 3, March 1974.

LOCH77 Lochovsky, F. and Tsichritzis, D., "Human Factors Considerations in DBMS Selection," Computer Systems Research Group, University of Toronto,CSRG-78, February, 1977.

MCDO75a McDonald, N., "CUPID: A Graphics Oriented Facility for Support of Non-programmer Interactions with a Data Base," Ph.D. Thesis, Dept. of Electrical Engineering and Computer Sciences, U.C. Berkeley, 1975.

MCDO75b McDonald, N. and Stonebraker, M.R., "Cupid -- The Friendly Query Language," Proc. ACM-Pacific-75 Conference, San Francisco, CA, April 1975.

MICH76 Michaels, A.S., Mittman, B., and Carlson, C.R., "Relational and CODASYL Approaches," Computing Surveys, vol 8, no 1, March 1976.

MYLO75 Mylorpulis, J., et. al., "TORUS:: A Natural Language Understanding System for Data Management," Proc. 4th International Joint Conference in AI, Tbilisi, USSR, September, 1975.

PECH75 Pecherer, R.M., "Efficient Evaluation of Expressions in a Relational Algebra," Proc. ACM-Pacific-75 Conference, San Francisco, CA, April 1975.

PREN77 Prenner, C.J. and Rowe, L.A., "A Data Base Oriented Programming Language," ERL Memorandum, Electronic Research Laboratory, U.C. Berkeley, July 1977.

REIS75 Reisner, P., Boyce, R.F. and Chamberlin, D.D., "Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL," Proc. 1975 National Computer Conference, Anaheim, CA, May 1975.

REIS76 Reisner, P., "Use of Psychological Experimentation as an Aid to Development of a Query Language," RJ1707, IBM Research, San Jose, CA, January 1976.

RITC73 Ritchie, D.M., "C Reference Manual," Unpublished Memorandum, Bell Telephone Laboratories, 1973.

SCHK75 Schkolnick, M., "The Optimal Selection of Secondary Indices for Files," Proc. 1975 ACM-SIGMOD Workshop, Washington, D.C., May 1975.

SMIT75 Smith, J.M. and Chang, P. Y. T., "Optimizing the Performance of a Relational Data Base Interface," Communications ACM, vol 18, no 10, October, 1975.

STON75a Stonebraker, M.R., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Workshop on Management of Data, San Jose, CA, May 1975.

STON75b Stonebraker, M.R. and Held, G., "Networks, Hierarchies, and Relations in Data Base Management Systems," ERL Memorandum No ERL-M504, Electronic Research Laboratory, U.C. Berkeley, May 1975.

STON76 Stonebraker, M., et. al., "The Design and Implementation of INGRES," TODS, vol 1, no 3, September 1976.

TAYL74 Taylor, R.W., Data Administration and the DBTG Report," Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access, and Control, Ann Arbor, Mich., May 1974.

TAYL76 Taylor, R.W. and Frank, R.L., "CODASYL Data-Base Management Systems," Computing Surveys, vol 8, no 1, March 1976.

TSIC75 Tsichritzis, D. "A Network Framework for a Relational Implementation," in Data Base Description, Noth Holland Publishing Company, 1975.

TSIC76 Tsichritzis, D.C. and Lochovsky, F.H., "Hierarchical Data-Base Management: A Survey," Computing Surveys, vol 8, no 1, March 1976.

TSIC76 Tsichritzis, D., "LSL: A Link and Selector Language" Proc. 1976 ACM-SIGMOD Conference on Management of Data, Washington, D.C., June 1976.

WASS77 Wasserman, A.I., "Reliable Interactive Software and Programming Language Design," Technical Report 20, Medical Information Science, University of California, San Francisco, April 1976 (revised January 1977).

WELL76 Wells, M.B. and Cornwall, F.L., "A Data Type Encapsulation Scheme Utilizing Base Language Operators," Proc. Conference on Data: Abstraction, Definition, and Structure, FDT, vol 8, no 2, March 1976.

WEGB72 Wegbreit, B., "The ECL Programming System," Proc. 1972 National Computer Conference, Las Vegas, NV, December 1972.

WONG76 Wong, E. and Youssefi, K., "Decomposition -- A Strategy for Query Processing," TODS, vol 1, no 3, September 1976.

ZLOO75 Zloof, M. M., "Query by Example," Proc. 1975 National Computer Conference, Anaheim, CA, May 1975.