D. D. Chamberlin
M. M. Astrahan
K. P. Eswaran
P. P. Griffiths
R. A. Lorie
J. W. Mehl
P. Reisner
B. W. Wade

SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control

Abstract: SEQUEL 2 is a relational data language that provides a consistent, English keyword-oriented set of facilities for query, data definition, data manipulation, and data control. SEQUEL 2 may be used either as a stand-alone interface for nonspecialists in data processing or as a data sublanguage embedded in a host programming language for use by application programmers and data base administrators. This paper describes SEQUEL 2 and the means by which it is coupled to a host language.

Introduction

Since the introduction of the relational model of data by Codd as a tool for general data base management [1], there have been proposed several relational data languages intended for inexperienced users. One such language is SEQUEL, the Structured English Query Language [2], which is based on English keywords and intended for use by nonspecialists in data processing as well as by professional programmers. Other well known languages with a similar orientation include QUEL [3], Query By Example [4], and SQUARE [5].

A series of human factors tests was conducted in which the query facilities of SEQUEL were taught to university undergraduates with and without programming experience [6, 7]. These tests isolated several features of the language that were sources of learning difficulty. As a result, several changes were made in the query facilities of the language.

In addition SEQUEL has been extended in several ways. A data manipulation facility has been added that permits insertion, deletion, and update of individual tuples or sets of tuples in a relational data base. A data definition facility allows definition of relations and of various alternative views of relations. A data control facility enables each user to authorize other users to access his data. The data control facility also provides for assertions about data integrity and for stored transactions triggered by various events. In addition, facilities have been added to SEQUEL to permit coupling with a high level host programming language, such as PL/I.

The result of these extensions and improvements is SEQUEL 2, which is described in this paper. SEQUEL 2 may be thought of as a language consisting of several "layers" of increasing complexity. The most casual user may learn only the simplest query features; more thoroughly trained users are provided more powerful features, including some facilities normally reserved for a data base administrator. All features are based on a consistent keyword-oriented syntax.

SEQUEL 2 is the main external interface to be supported by System R, an experimental relational data base management system now under development [8]. System R will make SEQUEL 2 available both as a stand-alone, display-oriented interface and as a data sublanguage embedded in PL/I.

SEQUEL 2 operates on relations in first (or higher) normal form, as described by Codd [1, 9]. The language is described here by a series of examples based on the data base of Fig. 1. The EMP relation describes a set of employees, giving the employee number, name, department number, job title, manager's employee number, salary, and commission for each employee. The DEPT relation gives the department number, name, and location of each department. The USAGE relation describes the parts used by the various departments. The SUPPLY relation describes the supplier companies from which the various parts may be obtained. SEQUEL 2 refers to relations by the more familiar term *table*. In this paper, the terms "relation" and "table" are used interchangeably.

The following sections introduce the facilities of SEQUEL 2 for query, data manipulation, data definition, data control, and host language coupling. Where necessary, reference is made to specific features of System R; however, SEQUEL 2 is adaptable with minor modifications to run on other relational systems. A complete BNF syntax for SEQUEL 2 is given in the Appendix. SEQUEL 2 accepts statements in free format; the arrangement of lines and indentation in the following examples is used for clarity only.

Query facilities

The most basic operation of the SEQUEL language, called a *mapping*, is illustrated by Q1 below. Mapping suggests that a known quantity (DNO = 50) is to be transformed into a desired quantity (NAME) by means of a relation (EMP). The SELECT clause lists the attributes to be returned—if the entire tuple is desired, one may write SELECT *. The WHERE clause may contain any collection of predicates that compare attributes of tuples to values (e.g., DNO = 50) or compare two attributes of a tuple with each other (e.g., SAL < COMM). The predicates may be connected by AND and OR, and parentheses may be used to establish precedence.

Q1. Find the names of employees in Dept. 50.

```
SELECT NAME
FROM   EMP
WHERE  DNO = 50
```

In general, a mapping returns a collection of values—the selected attributes of the tuples that satisfy the WHERE clause. Duplicate values are not eliminated from the returned set unless the user so requests by writing SELECT UNIQUE. We decided on this convention because the elimination of duplicate values is an expensive operation that we felt should not be provided by default. Q2 illustrates "projection" of the EMP relation on the DNO attribute.

Q2. List all the different department numbers in the EMP table.

```
SELECT UNIQUE DNO
FROM   EMP
```

A predicate in a WHERE clause may test an attribute for inclusion in a set, as illustrated by Q3, which also shows the syntax for representing a set of constants.

Q3. List names of employees in Depts. 25, 47, and 53.

```
SELECT NAME
FROM   EMP
WHERE  DNO IN (25,47,53)
```

```
EMP      | EMPNO | NAME | DNO | JOB | MGR | SAL | COMM |

DEPT     | DNO | DNAME | LOC |

USAGE    | DNO | PART |

SUPPLY   | SUPPLIER | PART |
```

Figure 1 Example data base.

It is possible to use the result of a mapping in the WHERE clause of another mapping. This operation, called *nested mapping*, is illustrated by Q4. The inner mapping returns the collection of DNO values of departments located in Evanston. The outer mapping then proceeds as though it were given a set of constants in place of the inner mapping. Mappings may be nested to any number of levels. The comparison operator = may be used in place of IN without changing the meaning of the query (this may seem more natural to a user when the inner mapping returns a single value).

Q4. Find names of employees who work for departments in Evanston.

```
SELECT NAME
FROM   EMP
WHERE  DNO IN
       SELECT DNO
       FROM   DEPT
       WHERE  LOC = 'EVANSTON'
```

SEQUEL 2 requires single quotation marks around all character-string constants, in order to distinguish them from attribute names (e.g., NAME = JOB and NAME = 'JOB' are both valid predicates but with different meanings). Quotation marks are optional on numeric constants (e.g., SAL = 10000 and SAL = '10000' are equivalent).

The result of a query is returned in system-determined order unless the user requests an ordering, as shown in Q5. The user may specify major and minor sorting attributes, and he may specify ascending or descending order. Ordering of character string attributes uses lexicographic order.

Q5. List the employee number, name, and salary of employees in Dept. 50, in order of employee number.

```
SELECT EMPNO,NAME,SAL
FROM   EMP
WHERE  DNO = 50
ORDER BY EMPNO
```

SEQUEL 2 provides several built-in functions that may be used in the SELECT clause, as illustrated in Q6. The functions provided include AVG, SUM, COUNT, MAX, and MIN. System R allows a user to add additional functions to the system by placing routines in a special function library.

Q6. Find the average salary of clerks.

```
SELECT AVG(SAL)
FROM   EMP
WHERE  JOB = 'CLERK'
```

The notation COUNT(*) denotes the count of tuples that satisfy the WHERE clause.

In general duplicates are not eliminated from the set of qualifying values before the built-in function is applied. However, the user may explicitly call for elimination of duplicates by placing the word UNIQUE inside the argument of the function, as shown in Q7.

Q7. How many different jobs are held by employees in Dept. 50?

```
SELECT COUNT(UNIQUE JOB)
FROM   EMP
WHERE  DNO = 50
```

In addition to simple attributes and built-in functions, a user may construct arithmetic expressions in the SELECT clause. All the following are valid expressions:

```
(AVG(SAL) / 52
AVG(SAL) + AVG(COMM)
AVG(SAL + COMM)
```

A relation may be partitioned into groups according to the values of some attribute and then a built-in function applied to each group. This type of query is illustrated by Q8. A GROUP BY clause is always used together with a built-in function. When a GROUP BY clause is used, each item in the SELECT clause must be a unique property of a group rather than of an individual tuple. For example, in Q8 each group of employees has a unique DNO and a unique average salary. If EMPNO were added to the SELECT clause of Q8, an error would result, since EMPNO is not a unique property of each group.

Q8. List all the departments and the average salary of each.

```
SELECT DNO,AVG(SAL)
FROM   EMP
GROUP BY DNO
```

A relation may be partitioned into groups and then a predicate or set of predicates applied to choose only certain of the groups and disqualify others. These group-qualifying predicates are always based on built-in functions and are placed in a special HAVING clause, as shown in Q9. A predicate in a HAVING clause may compare an aggregate property of the group to a constant (e.g., AVG(SAL) < 10000) or to another aggregate property of the same group (e.g., AVG(SAL) <= AVG (COMM).)

Q9. List those departments in which the average employee salary is less than 10000.

```
SELECT DNO
FROM   EMP
GROUP BY DNO
HAVING AVG(SAL) < 10000
```

When a query has both a WHERE clause and a HAVING clause, their precedence is as follows: First the WHERE clause is applied to qualify tuples; then the groups are formed; then the HAVING clause is applied to qualify groups, as shown by Q10.

Q10. List the departments that employ more than ten clerks.

```
SELECT DNO
FROM   EMP
WHERE  JOB = 'CLERK'
GROUP BY DNO
HAVING COUNT(*) > 10
```

A special built-in function called SET is provided that evaluates to the set of values for a particular attribute which is present in a given group. This set of attributes may then be compared with another set as part of the HAVING clause. In Q11, the inner mapping returns the set of all job titles in the EMP table (with duplicates). The outer mapping groups employees by department number and then chooses those groups whose set of job titles is equal to the set of all job titles. In this case, the = operator is used in the sense of comparison of two sets. Other set comparison operators are ¬=, [IS][NOT] IN, CONTAINS, and DOES NOT CONTAIN. All these set comparison operators eliminate duplicates from both of their operands before performing the comparison.

Q11. List the departments that have employees with every possible job title.

```
SELECT  DNO
FROM    EMP
GROUP BY DNO
HAVING  SET(JOB) =
        SELECT JOB
        FROM   EMP
```

The set-theoretic operators INTERSECT, UNION, and MINUS are also available in SEQUEL 2. They may be used to combine the results of two mappings, as shown in Q12. A query may contain several set theoretic operators, with parentheses used as needed to resolve ambiguities. Like the set comparison operators listed above, INTERSECT, UNION, and MINUS automatically eliminate duplicates from their operands before performing their function.

Q12. List the departments that have no employees.

```
SELECT DNO
FROM   DEPT
MINUS
SELECT DNO
FROM   EMP
```

A query may return values selected from more than one relation. An example is a *join* operation, as illustrated by Q13. The user may list several relations in the FROM clause. Conceptually, the Cartesian product of these relations is formed and then filtered by the predicates in the WHERE clause. (Of course, a well designed system would avoid actual formation of the Cartesian product but achieve the same effect by a more efficient means.) When more than one relation is named in the FROM clause, the user must be careful to properly qualify each attribute name in the SELECT and WHERE clauses (e.g., to distinguish EMP.DNO from DEPT.DNO). When an attribute name occurs in only one of the participating relations, it need not be qualified (e.g., Q13 could have specified simply SELECT NAME,LOC).

Q13. List the names of all employees and the locations of their departments.

```
SELECT EMP.NAME,DEPT.LOC
FROM   EMP,DEPT
WHERE  EMP.DNO = DEPT.DNO
```

In some types of queries it is necessary to join a relation with itself according to some criterion. This may be done by listing the relation name more than once in the FROM clause, as in Q14. In such a query, the user may invent an arbitrary label to be associated with each of the participating relations (in this example, X and Y were chosen as labels). The labels may then be used in place of the relation name for qualifying references in the SELECT and FROM clauses.

Q14. For each employee whose salary exceeds his manager's salary, list the employee's name and his manager's name.

```
SELECT X.NAME,Y.NAME
FROM   EMP X, EMP Y
WHERE  X.MGR = Y.EMPNO
AND    X.SAL > Y.SAL
```

The SEQUEL 2 language also permits a label to be used to qualify attribute names outside the mapping block in which the label is defined. For example, Q15 searches for those tuples X of the SUPPLY relation such that the set of parts supplied by the supplier in tuple X (computed by the first nested mapping) contains the set of parts used by Dept. 50 (computed by the second nested mapping).

Q15. List the suppliers that supply all the parts used by Dept. 50.

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

We designed SEQUEL 2 so that whenever a variable occurs outside the block in which it is defined, it may be brought inside the block (and often eliminated entirely) by means of GROUP BY and the special function SET. For example, Q16 is an equivalent restatement of Q15.

Q16. (Same as Q15.)

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

The notation for a set of constants was shown in Q3. A constant tuple is denoted as in the following example:

⟨'CLERK',50⟩

A set of constant tuples may be represented as follows:

(⟨'CLERK',50⟩, ⟨'CLERK',52⟩, ⟨'PROGRAMMER',52⟩)

The brackets ⟨⟩ may also be used to denote a subtuple of attributes selected from a tuple in the data base, as in Q17:

Q17. List the names of employees who have the same job and salary as Smith.

```
SELECT NAME
FROM   EMP
WHERE  ⟨JOB,SAL⟩ =
       SELECT JOB,SAL
       FROM   EMP
       WHERE  NAME = 'SMITH'
```

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

IF X THEN Y	Y = T	Y = F	Y = ?
X = T	T	F	?
X = F	T	T	T
X = ?	T	?	?

Figure 2 Truth tables for three-valued logic.

SEQUEL 2 allows for the existence of unknown, or null, values in the data base. The null value may be referred to by the keyword NULL. Null values are ignored in the computation of all built-in functions except COUNT (e.g., null salaries do not participate in computing AVG(SAL)).

In determining whether a given tuple satisfies the WHERE clause of a query, predicates that test attributes for which the tuple has a null value are assigned the unknown truth value (denoted ?). The truth value of the entire WHERE clause is then computed using three-valued logic to evaluate ANDS and ORS (see Fig. 2). The tuple is considered to satisfy the WHERE clause if the overall truth value of the clause is TRUE but not if the overall truth value is FALSE or ?. For example, a tuple having a DNO of 50 and a null salary would satisfy Q18 but not Q19.

Q18.
```
SELECT *
FROM   EMP
WHERE  DNO = 50
OR     SAL>15000
```

Q19.
```
SELECT *
FROM   EMP
WHERE  DNO = 50
AND    SAL>15000
```

An exception to the above rules is made in the case of predicates that search for null values explicitly, e.g., WHERE SAL = NULL. In these predicates, the null value is treated like any other value.

Data manipulation facilities

Data manipulation facilities enable a user to change values directly in the data base. These facilities fall into the categories of insertion, deletion, update, and assignment.

The insertion facility allows the user to insert a new tuple or set of tuples into a relation. Insertion of a single tuple is illustrated by M1. Attributes that are not specified by the insertion statement are given null values. If the tuple to be inserted has all its attributes present, in the correct order, the list of attribute names may be omitted.

M1. Insert a new employee named "Jones" with employee number 535 in Dept. 51, having other attributes null.

```
INSERT INTO EMP(EMPNO,NAME,DNO);
        <535,'JONES',51>
```

A SEQUEL insertion statement may also evaluate a query and insert the resulting set of tuples into some existing relation. Suppose the data base contains a relation called CANDIDATES, which has columns for employee number, name, department number, and salary. Then M2 could be used to select a set of values from the EMP relation and insert them into CANDIDATES.

M2. Add to the CANDIDATES table all those employees whose commission is greater than half their salary.

```
INSERT INTO CANDIDATES:

        SELECT EMPNO,NAME,DNO,SAL
        FROM   EMP
        WHERE  COMM > 0.5 * SAL
```

Deletion is a process of specifying tuples to be removed from the data base. The tuples are specified by means of a WHERE clause that is syntactically identical to the WHERE clause of a query, as shown in M3:

M3. Delete from EMP the employee with employee number 561.

```
DELETE EMP
WHERE  EMPNO = 561
```

It may sometimes be useful to invent a label for the tuples to be deleted and then to use the label to define the properties of the tuples. This is similar to the use of labels in the query facilities of the language. Deletion by use of a label is illustrated by M4:

M4. Delete all the departments having no employees from the DEPT table.

```
DELETE DEPT X
WHERE
        (SELECT COUNT(*)
        FROM   EMP
        WHERE  DNO = X.DNO) = 0
```

The update features of SEQUEL 2 are similar to those for deletion, except that an additional SET clause is used to specify the updates to be made to the selected tuples. New values for updated attributes may be stated as constants, as nested queries, or as expressions based on the original value of the attributes, as in M5. As in the case of deletion, a label may be placed in the UPDATE clause and used in nested queries in the SET clause or WHERE clause.

M5. Update the EMP table by giving a ten percent raise to all those employees whose employee number appears in the CANDIDATES table.

```
UPDATE EMP
SET    SAL = SAL*1.1
WHERE  EMPNO IN
       SELECT EMPNO
       FROM   CANDIDATES
```

Newly inserted or updated tuples are not checked for duplication of an existing tuple, since SEQUEL permits duplicate tuples to exist unless the user has specified otherwise. In the System R version of SEQUEL, duplicate tuples may be prohibited by means of a "unique image" (explained in the section on Data Definition).

An assignment statement creates a new relation in the data base and copies into it the result of a query. This new relation may then be queried, updated, or processed in the same way as any other relation. The assignment statement specifies the new relation name and column names. If the column names of the new relation are unambiguously determined by the SELECT clause of the query, they may be omitted. Assignment is illustrated by M6.

M6. Create a new relation called MANAGERS (with suitable column names), and place in it the employee number, name, department number, and salary of all employees who are managers.

```
ASSIGN TO MANAGERS
                  (EMPNO,NAME,DEPT,SALARY):
    SELECT EMPNO,NAME,DNO,SAL
    FROM   EMP
    WHERE  EMPNO IN
           SELECT MGR
           FROM   EMP
```

The new relation created by an assignment statement is independent of the relation(s) from which it was derived. After M6 is executed, adding a new manager to the EMP table does not affect the MANAGERS relation, and updating MANAGERS does not affect EMP.

Data definition facilities

Data definition facilities enable users to create and drop relations, define alternative views of relations, and specify the access aids (indexes, etc.) to be maintained on the data base. The data definition facilities of a language describe the data structures provided by the system on which the language runs. In this section we describe the data definition statements of SEQUEL 2 in the context of System R. Suitable modifications could be made to adapt the language to other relational systems.

Example D1 is a statement that creates a new relation (table) to be physically stored in the system. System R permits tables to be created and dropped dynamically. The user specifies the table name and the column names and data types. If null values are not to be permitted in a particular column (e.g., DNO in the example), the user may so state. The data types supported by System R are shown in the syntax in the Appendix.

D1. (This is the statement that might have been used to create the DEPT table.)

```
CREATE TABLE DEPT
    (DNO(CHAR(2),NONULL),
    DNAME(CHAR(12) VAR),
    LOC(CHAR(20) VAR) )
```

In SEQUEL 2, the name of a table may be qualified by the name of the user who created it, if necessary. For example, if users Smith and Jones each create a table named EMP, Smith can refer to his own table by EMP, or to Jones' table (if he is so authorized) by JONES.EMP. A user may also define a synonym, or alternate name, for a table, as shown in D2. This technique permits references to a table created by another user without repeating the creator's name with every reference.

D2. Define JEMP as a synonym for the EMP table created by Jones.

```
DEFINE SYNONYM JEMP AS JONES.EMP
```

The access aids supported by System R are called images and links [8]. An *image* is an index on one or more attributes of a relation, maintained in the form of a B-tree [10]. At most one image on a relation may have the *clustering* property, which means that tuples that are near each other in the ordering of that image are stored physically near each other in the data base. An image may also be declared *unique*, which means that the indexed attribute must be a key of its relation, i.e., no two tuples may have the same value for this attribute. A *link* is a set of pointers that connect the tuples of one relation to the tuples of another relation which match according to a certain attribute. A link may have the clustering property, in which case the system tries to maintain physical contiguity of the tuples on the link. Examples of statements to create images and links are shown in D3 and D4. Each image or link is given a name (I3 and L5 in the examples), which enables the user to refer to it (e.g., in a DROP statement).

D3. Create an image called I3 on the SAL attribute of the EMP table.

```
CREATE IMAGE I3 ON EMP(SAL)
```

D4. Create a link called L5 which connects rows of DEPT to those rows of EMP that match in the DNO attribute. Order the employees on the link by JOB and secondarily by SAL.

```
CREATE LINK L5
FROM DEPT(DNO)
TO EMP(DNO)
ORDER BY JOB,SAL
```

Although SEQUEL 2 allows the user to create and destroy structures such as images and links, it has no statements that use these structures directly. All queries and data manipulation statements in SEQUEL are stated in a nonprocedural way, which enables the system to choose the optimal image or other access path to execute the statement. Images and links contain no information that is not derivable from the actual data values in the tuples involved.

A very important aspect of data definition is the ability to define alternative views of stored data. In SEQUEL 2, the process of defining a view is very similar to the process of stating a query. This is because SEQUEL 2 has the property of *closure*: the result of any query on one or more tables is itself a table. Therefore, any query formulation may be used as a definition of a view. The DEFINE VIEW statement gives a name to the view and to its columns. (If the column names can be derived unambiguously from the query that defines the view, they may be omitted.) After definition of a view, the view may be used in the same ways that a stored table may be used: queries may be issued against it, other views may be defined in terms of it, and, subject to certain limitations [11], it may be updated. Unlike the assignment statement described earlier, a view is a dynamic window on the data base. Changes made to the underlying relations are visible via the view. In general, updates may be made via a view only if each tuple of the view is associated with exactly one tuple of a stored relation. This permits updates to tuples of the view to be implemented by updates to the corresponding stored tuples.

One important application for views is to permit a user to access only a certain part of a relation. For example, if a user is entitled to read only the employee number, name, and job of employees in Dept. 50, he might be given the view shown in D5.

D5. Define a view called D50 containing the employee number, name, and job of those employees in Dept. 50.

```
DEFINE VIEW D50 AS
    SELECT EMPNO,NAME,JOB
    FROM   EMP
    WHERE  DNO = 50
```

Views are also useful for providing statistical summaries of data. For example, a view based on query Q8 would provide the average salary of each department without disclosing any individual salary. A view may be defined using the keyword USER, which is always interpreted as the user-id of the current user. In this way, for example, we might define a view that allows each user to see only the employees in his own department.

A view need not be derived from a single underlying table. For example, D6 defines a view as a join of two tables, by means of a query similar to Q13. Example D6 also shows how a query may be issued against a view just as though it were a stored relation. In defining a join view such as the one in D6, users should beware of the "connection trap" described by Codd [1]. In terms of D6, the facts that employee X works for department Y and department Y is in location Z do not necessarily imply that employee X is in location Z.

D6. Define a view called PROGS consisting of the names and salaries of all programmers and the locations of their departments.

```
DEFINE VIEW PROGS
                (NAME,SALARY,HOMEBASE) AS
    SELECT EMP.NAME,EMP.SAL,DEPT,LOC
    FROM   EMP.DEPT
    WHERE  EMP.DNO = DEPT.DNO
    AND    EMP.JOB = 'PROGRAMMER'
```

Using the above view, find the average salary of programmers in Denver.

```
    SELECT AVG(SAL)
    FROM   PROGS
    WHERE  HOMEBASE = 'DENVER'
```

Occasionally it may be necessary to expand an existing table by adding a new column to it, e.g., to accommodate a new application. SEQUEL 2 allows columns to be added to the right side of existing tables by the EXPAND statement, which gives the name and data type of the new column. Existing tuples are considered to have null values in the new column until they are updated. Queries and views that were written in terms of the existing table are not affected by the expansion (except for those queries that SELECT * from the table).

D7. Add a new column called NEMPS, of integer type, to the table DEPT.

```
EXPAND TABLE DEPT
        ADD COLUMN NEMPS(INTEGER)
```

When they are no longer needed, tables, views, images, and links may be dropped from the system by a DROP command, as in D8.

D8. Drop the view D50.

```
DROP VIEW D50
```

A table or other object may be dropped only by the user who created it. For this reason, many installations will have one or more special user-ids which represent, not actual persons, but the "role" of data base administrator (DBA) for various portions of the data base. Thus the user id representing the role of DBA for a certain department may serve to create, control, and destroy the tables used in common by that department. Meanwhile, individual users may use their own user-ids to create tables for their own private use.

System R automatically maintains catalogs that describe all the tables, views, images, links, assertions, and triggers (see next section) that are known to the system. These catalogs are kept in the form of tables, which may be queried in the same way as any other table. Each catalog entry has space for a comment, which may be filled in by the creator of the relevant object, using the COMMENT statement:

D9. (Illustrates the use of comments.)

```
COMMENT ON THE VIEW D50:
     'LIMITED VIEW OF EMPLOYEES IN DEPT. 50'
```

Data control facilities

Data control facilities enable users to control access to their data by other users and to exercise control over the integrity of data values. Facilities are also provided to group several statements into a "transaction," and to back out updates that have been made to the data base.

Since SEQUEL permits any user to create new relations and views, it is the responsibility of each user to control access to the data objects he creates. When a user creates a relation or view, he is fully and solely authorized to perform actions upon it. (If the object is a view, his authorization is limited to the authorizations that he holds on the supporting relations.) A user may grant access to his relation or view to other users by means of the SEQUEL command GRANT. The following privileges may be granted:

- READ
- INSERT
- DELETE
- UPDATE (by column)
- EXPAND
- IMAGE (to define images on the relation)
- LINK (to create links on the relation)
- CONTROL (to make assertions or define triggers pertaining to the relation—see explanation below)

(An additional privilege called RUN, which applies to programs rather than to relations, is explained in the section on host language coupling.)

In addition, the grantor may permit the grantee to grant the listed privileges to other users by including the clause WITH GRANT OPTION. If a user does not have the grant option for a privilege, he may exercise that privilege, but he may not grant the privilege to other users.

C1. Give the following privileges on the EMP table to Smith and Anderson: Ability to read, to insert, to update the JOB and DNO columns and to grant these abilities to others.

```
GRANT READ,INSERT,UPDATE(JOB,DNO) ON EMP
    TO SMITH,ANDERSON WITH GRANT OPTION
```

The keyword PUBLIC may be used in place of the list of users to whom a privilege is to be granted, if it is to be granted to all users. The phrase ALL RIGHTS may be used in place of the list of privileges in a GRANT statement. If the list of privileges is omitted, the READ privilege is granted by default.

Once a privilege has been granted, it may be withdrawn through use of the REVOKE command. The named privileges are revoked from the grantee and from all users to whom he has granted them, unless the grantee has another, independent, source of the revoked privileges. Revocation of a privilege may have other ramifications as well. For example, if the READ privilege on the EMP relation is revoked from a user, all views defined on EMP by that user must be dropped. These issues are explored more fully in [12].

C2. Revoke from Anderson the power to update the EMP table.

```
REVOKE UPDATE ON EMP FROM ANDERSON
```

The SEQUEL language allows a user who has CONTROL privileges on a table to make *assertions* about the integrity of the data in that table [13]. An assertion is a SEQUEL predicate, which evaluates to TRUE or FALSE. When the ASSERT statement is issued, the system checks the current truth value of the predicate. If it is currently FALSE, the assertion is rejected. If it is TRUE, the system henceforth enforces the assertion against all further updates to the data base. Each assertion is given a name by the user who makes it. If an insertion, deletion, or update statement violates an assertion, the statement is rejected and a violation code is returned, together with the name(s) of the violated assertion(s).

The simplest kind of assertion pertains to a particular relation (identified by a phrase such as ON EMP) and is enforced for each tuple of the relation:

C3. Assert that all employee salaries are less than 50000.

```
ASSERT A1 ON EMP: SAL < 50000
```

C4. Assert that all clerks have a salary between 8000 and 15000.

```
ASSERT A2 ON EMP:
    IF JOB = 'CLERK' THEN
    SAL BETWEEN 8000 AND 15000
```

As in queries, a label may be associated with an assertion. The label may then be used in mapping blocks within the assertion, to state some property to be enforced for all tuples of the relation on which the assertion is based. The use of a label in an assertion is illustrated by C5.

C5. Assert that the NEMPS attribute of each row of the DEPT table is equal to the number of employees in the given department.

```
ASSERT A3 ON DEPT X: NEMPS =
        (SELECT COUNT(*)
        FROM EMP
        WHERE DNO = X,DNO)
```

Another class of assertions are those that make an overall statement about one or more relations, rather than about individual tuples of a relation. This type of assertion does not need an ON phrase, since the relation(s) involved are identified by the body of the assertion, as shown in C6.

C6. Assert that no row in the EMP table may have a DNO that is not present in the DEPT table.

```
ASSERT A4:
    (SELECT DNO FROM EMP)
      IS IN
    (SELECT DNO FROM DEPT)
```

The assertions presented so far illustrate conditions that must hold statically. Another class of assertions deals with transitions in the data base. This type of assertion must state the circumstances under which it is to be enforced: on insertion, deletion, or update of tuples in a certain relation. When the designated action is performed on a tuple of the given relation, the body of the assertion is checked to determine whether the transition is permissible. The transition may be described in terms of OLD and NEW values, which represent the attributes of the tuple before and after the transition. If a single SEQUEL statement updates many tuples, the assertion is checked for each tuple, and the entire statement is refused if any tuple violates the assertion.

C7. Assert that whenever an employee's salary is updated, the new salary must be at least as large as the old salary.

```
ASSERT A5 ON UPDATE OF EMP(SAL):
    NEW SAL >= OLD SAL
```

Transition assertions always apply to individual tuples rather than to aggregates of tuples (e.g., a transition assertion cannot be used to state that the new average salary of employees is greater than the old average salary).

The general rule for handling null values in the checking of an assertion is that the presence of null values should never cause an assertion to succeed if it would otherwise fail or to fail if it would otherwise succeed. This rule is implemented by the use of three-valued logic. The assertion predicate, which in general may be a Boolean condition, is evaluated using the truth tables in Fig. 2. The assertion is satisfied if the result is TRUE or ?, but it is violated if the result is FALSE. For example, an assertion that no salary exceeds 20000 is not violated by the presence of a null salary. An exception to this rule is made in the case of assertions that mention null values explicitly, e.g., EMPNO ¬= NULL. In these assertions the null value is treated like any other value.

SEQUEL allows several statements to be grouped into a transaction by placing them between the statements BEGIN TRANSACTION and END TRANSACTION. Integrity assertions are normally suspended within a transaction. At the end of the transaction, all relevant assertions are checked, and, if any are violated, the entire transaction is backed out. This permits updates to be made that cause the data base to pass through a temporary inconsistent state. For example, if a new employee is hired, updates to EMP and DEPT might be made, causing the data base to momentarily violate assertion A3 in example C5. However, at the end of the transaction, assertion A3 is satisfied.

The user who makes an assertion may optionally declare that the checking of his assertion is never to be delayed. If he includes the word IMMEDIATE in the assertion prefix, the assertion is always enforced at the completion of each SEQUEL statement. Since transition assertions (i.e., those that compare OLD and NEW values) operate on a tuple-by-tuple basis, they are always enforced immediately rather than at the end of a transaction.

In System R, placing several statements inside a transaction has the additional significance of declaring that the system should execute these statements as an atomic act without permitting interference (e.g., updates to relevant data) by other users during the transaction. Insofar as possible, System R attempts to protect each user from any awareness of other concurrent users. Therefore SEQUEL does not require a user to issue locking statements or statements of intent to update. The setting and clearing of locks, and the detection and resolution of deadlocks, are left up to the underlying system. These issues, and their implementation in System R, are discussed more fully in [8].

At any time within a transaction, a user may declare a "save point" by issuing the statement SAVE ⟨save-point-name⟩. Many save points may be declared within a single transaction. At any time, a user may back out all the changes that he has made to the data base since a particular save point by issuing the statement RESTORE ⟨save-point-name⟩. If the save point name is omitted from the RESTORE statement, the user is backed out to the beginning of the current transaction. No RESTORE statement may back out to a save point earlier than the beginning of the current transaction. A RESTORE statement has no effect on updates that have been made by other users.

In order to help maintain the integrity of the data base, a user may define a *trigger* to be executed upon occurrence of a specified action: READ, INSERTION, DELETION, or UPDATE of a tuple in some particular table. The body of the trigger, which consists of one or more SEQUEL statements, is executed immediately after the designated action is performed by any user. If a SEQUEL statement performs an action (e.g., an update) on many tuples and this action invokes a trigger, the trigger is executed repeatedly, once after the update of each tuple. Triggers are always executed immediately and may not be delayed until the end of a transaction. The body of the trigger may use the words OLD and NEW to refer to the previous and the updated values of the tuple, as shown in C8.

C8. (This trigger automatically updates the relevant NEMPS entries in the DEPT table whenever the DNO of an employee is updated.)

```
DEFINE TRIGGER T1
ON UPDATE OF EMP(DNO):
    (UPDATE DEPT
    SET       NEMPS = NEMPS + 1
    WHERE     DNO = NEW EMP.DNO;

    UPDATE DEPT
    SET       NEMPS = NEMPS − 1
    WHERE     DNO = OLD EMP.DNO)
```

A label may be attached to the tuple that invokes a trigger, and the label may be used in the body of the trigger. Also, a statement in the body of a trigger may contain an IF clause, which makes its execution contingent on some condition, as shown in C9.

C9. When an employee is deleted from EMP, if there are no remaining employees in his department, delete the corresponding DEPT record.

```
DEFINE TRIGGER T2
ON DELETION OF EMP X:
    (IF (SELECT COUNT(*)
        FROM   EMP
        WHERE DNO = X.DNO) = 0
    THEN DELETE DEPT
        WHERE      DNO = X.DNO)
```

It is possible that a single update statement may invoke several triggers and several assertions. In this case, the triggers are executed first, in system-determined order. Since triggers are executed on a tuple-by-tuple basis, a single SEQUEL statement may cause several triggers to be executed after the update of each tuple. If execution of a trigger invokes other triggers, these (second-level) triggers are executed before continuing with the original set of triggers. The definer of a trigger is responsible for ensuring that it does not take some action that results in invoking itself in an infinite loop. Finally, when the original statement and all its invoked triggers, of all levels, have been executed, the set of relevant assertions are checked. If any assertion fails, the statement and all its triggers (or the current transaction and all its triggers) are backed out.

Transition assertions (i.e., those that compare the OLD and NEW values of a tuple) are always applied, like triggers, on a tuple-by-tuple basis. A transition assertion compares the value of the tuple before it is updated with its value after it has been updated and all triggers invoked by the updating of that tuple have been executed.

A user with CONTROL authorization may drop an assertion or trigger by means of a DROP command, e.g.,

```
DROP TRIGGER T2
```

Triggers are considered to be a part of the transaction that invokes them (i.e., if the transaction is backed out, all its triggers are backed out also). However, triggers and assertions are authorized in the context of their definer rather than in the context of their invoker. Typically the definer of an assertion or trigger is a highly authorized user. He may, for example, define a trigger that, in execution, updates a table that is not visible to the user whose action invoked the trigger.

In System R assertions and triggers may be placed on stored relations but not on derived views. This policy avoids the difficult problem of finding, on update of a tuple, the set of views that are affected and that should have their assertions or triggers invoked. We feel that a user who is sophisticated enough to be placing an assertion or trigger into the system should be knowledgeable enough to frame the assertion or trigger in terms of a stored relation.

A final comment on the use of triggers is in order here. The creator of a relation may wish to ensure that, whenever the relation is updated, a complex series of associated actions takes place. Rather than using a SEQUEL trigger, the creator may wish to write a host language program, as illustrated in the next section, that updates the relation according to some input parameters and also performs the associated actions. The creator may then grant to other users, not generalized UPDATE rights on the relation, but only the right to execute the program.

Host language coupling

SEQUEL 2 is designed to be used both as a stand-alone language for interactive users and as a data sublanguage embedded in a host programming language. The features in the preceding sections have been described from the point of view of the interactive user. This section describes the additional language features that are provided for the purpose of host language coupling. Where necessary, we refer to the specific details of the System R implementation.

System R permits SEQUEL statements to appear as statements in a PL/I program. The SEQUEL statements are discovered by a precompiler, which replaces them with valid PL/I calls to a run time module that performs the desired function.

For a query, a means must be provided to deliver the result to the host program. Therefore, when a query appears in a program, it may have an INTO clause containing a list of host program variables that serve as targets for the selected attributes, as shown in P1.

P1. Return employee no. 507's job in program variable X and his salary in program variable Y.

```
SELECT JOB,SAL
INTO   X,Y
FROM   EMP
WHERE  EMPNO = 507;
```

In addition to program variables in the INTO clause, program variables may appear anywhere a constant may be used, in a query or any other type of SEQUEL statement. Wherever a program variable appears, the programmer may optionally specify a pair of variable names, separated by a colon. The first variable of the pair is called the value holder, and the second, which must have data type BIN FIXED, is called the null indicator. A zero in the null indicator denotes an actual value in the value holder; a negative number in the null indicator denotes the null value. These features are illustrated by P2.

P2. Return in program variable X1 the salary of the employee whose employee number matches program variable Z1. (Variables X2 and Z2 serve as null indicators.)

```
SELECT SAL
INTO   X1:X2
FROM   EMP
WHERE  EMPNO = Z1:Z2;
```

Whenever a declared program variable name duplicates the name of a column in the data base (e.g., WHERE SAL = COMM if COMM is a declared program variable), the system assumes the reference is to the program variable rather than to the column.

The above examples return only one tuple to the host program (if the query evaluates to more than one tuple, only the first is returned). More often, however, the programmer may wish to identify a set of tuples and process them one after another. For this purpose we introduced the concept of a "cursor." A cursor is a symbolic name that a programmer may associate with a query and use to retrieve the query result, one tuple at a time, as shown in P3.

P3. Read a department number from the terminal, then fetch and display the names of all employees in the given department. (The statements marked by an asterisk in the margin are intercepted by the SEQUEL precompiler. All other statements are standard PL/I. The asterisks are not part of the program.)

```
    P3: PROC OPTIONS(MAIN);
        DCL X CHAR(50), Y CHAR(2);
*       LET C1 BE
*           SELECT NAME INTO X
*           FROM EMP WHERE DNO = Y;
        DISPLAY('DNO?') REPLY(Y);
*       OPEN C1;
        DO WHILE (CODE = OK);
*           FETCH C1;
            DISPLAY(X);
        END;
*       CLOSE C1;
    END P3;
```

In this example the declaration of X and Y is standard PL/I. The statement LET C1 BE ⟨query⟩ is like a declaration to the data base system, which associates the cursor name C1 with the given query. The OPEN C1 statement binds the value of the input variable Y and prepares to deliver tuples according to the query. Each execution of FETCH C1 delivers a new tuple into the program variable(s) specified with the query. CODE is a special variable in which the data base system places a result code after each data base call. (We do not give a complete treatment of result codes here.) The CLOSE C1 statement informs the system that no further fetches will be issued on the query currently associated with C1. If C1 is re-opened, the input variable Y is re-bound to a possibly different value, and a new set of tuples may be fetched.

Whenever a cursor is open, it maintains a position in the set of tuples on which it is defined (called the "active set"). If each tuple of the active set is associated uniquely with a tuple of a relation (i.e., the query includes a key of the relation), the cursor is said to be "updateable." This means that the current position of the cursor may be referred to in an UPDATE or DELETE statement to denote the tuple to be updated or deleted. For example, suppose we wish to modify the program in P3 so that it gives a raise in salary to each employee in the indicated department. This can be done by replacing the DO loop of P3 with the following:

```
  DO WHILE (CODE = OK);
*     FETCH C1;
      /* Compute the new salary for this
         employee in program variable Z */
*     UPDATE EMP
*         SET SAL = Z
*         WHERE CURRENT OF C1;
  END;
```

If the phrase WHERE CURRENT OF ⟨cursor-name⟩ is used in an update or deletion statement, it may not be mixed with other selection predicates. A cursor reference may not be used in an INSERT statement, since relations are unordered objects and hence insertion via a cursor is undefined.

Another type of cursor reference may be used to select tuples by comparing their values with those of the current tuple of a cursor, as shown in P4.

P4. Suppose C2 is positioned on a DEPT tuple. Define C5 to be the set of EMP tuples whose DNO matches the DEPT tuple of C2.

```
LET C5 BE
    SELECT * FROM EMP
    WHERE DNO = DNO OF CURSOR C2 ON DEPT;
```

Note that CURRENT OF C2 denotes the actual tuple on which C2 is positioned, but DNO = DNO OF CURSOR C2 searches for another set of tuples by value matching. The phrase ON DEPT informs System R that Cursor C2 is positioned on a DEPT tuple. This fact may be useful in access path selection (e.g., there may be a link from DEPT to EMP by matching DNO). Neither of the two types of cursor reference changes the position of the cursor.

SEQUEL 2 provides a special facility called EXECUTE that enables a host program to support interactive users. Suppose that the program, at run time, reads from a terminal a SEQUEL statement to be executed. It can call for the execution of this statement as shown in P5:

P5. Call for System R to execute the SEQUEL statement in program variable QSTRING.

```
EXECUTE QSTRING;
```

If an interactive user wishes to execute a query, the host program supporting the user must have some means of fetching the query result and displaying it. This is difficult because the program cannot know in advance the number of fields in the query result or their data types. The method used combines a cursor name with an EXECUTE statement, as shown in P6.

P6. Read a query into QSTRING and fetch and display the query result. (Asterisks denote SEQUEL; remainder is PL/I.)

```
  DISPLAY('ENTER QUERY') REPLY(QSTRING);
* LET C1 BE EXECUTE QSTRING;
* DESCRIBE C1 INTO ⟨pointer-1⟩;
  /* Format a buffer to hold one tuple and
     set pointer-2 to point to it—see
     explanation below. */
* OPEN C1;
  DO WHILE (CODE=OK);
*     FETCH C1 INTO ⟨pointer-2⟩;
      /* Display the tuple */
  END;
* CLOSE C1;
```

In this program the statement LET C1 BE EXECUTE QSTRING associates the name C1 with the query that is present at run time in QSTRING. The DESCRIBE C1 statement returns a description of the number of fields and data types of the result into the array indicated by ⟨pointer-1⟩. This enables the program to allocate a buffer space to hold each field of a result tuple. The program then constructs an array of pointers to these field-buffers and sets ⟨pointer-2⟩ to point to the array. The OPEN C1 statement prepares the system for delivery of the first tuple. Each FETCH C1 statement then delivers one tuple into the indicated buffers. After the statement CLOSE C1 has been executed, a new query may be read into QSTRING. Subsequent execution of the statements DESCRIBE C1, OPEN C1, and FETCH C1 would then refer to the new query.

In writing a program to support an interactive user, the programmer must distinguish between query statements (e.g., SELECT* FROM EMP) and other statements that yield no result (e.g., DELECT EMP WHERE EMPNO = 505). This is done by examining the first word of the input string. All queries begin with the word SELECT and are handled by means of

```
LET C1 BE EXECUTE QSTRING;
```

All statements not beginning with SELECT are handled by means of

```
EXECUTE QSTRING;
```

To allow a program to save a query result after examining it, the syntax of an assignment statement is extended to permit assignment of the active set of a cursor to a new relation, as shown in P7.

P7. Assign the active set of cursor C5 to a new relation named EXEMPT having columns EMPNO,NAME, and JOB.

```
ASSIGN TO EXEMPT(EMPNO,NAME,JOB): CURSOR C5;
```

A PL/I program with embedded SEQUEL statements is presented to the System R precompiler. All SEQUEL statements are discovered and replaced by calls to a run time module that executes the statement. Where possible, the SEQUEL statement is parsed and an optimal access path is chosen for it at precompilation time. When this is not possible, as in the case of the EXECUTE feature, the parsing and access path selection is done at run time.

Authorization for a program to perform various actions is checked, at precompilation time, against the privileges of the user who is compiling the program. If the user possesses the necessary privileges, he receives the RUN privilege on the program. He can then run the compiled program without further checking of authorization. If the author of the program possesses the GRANT option for all the privileges invoked by the program, he receives the RUN privilege with the GRANT option. This enables him to grant to other users the ability to run his compiled program. Thus a user may have the RUN privilege on a program that updates salaries, even though he is not authorized to update salaries in any way except by invoking the program.

An exception to the above authorization rules is made in the case of the EXECUTE statement. An EXECUTE statement should not be authorized in the context of the program author, because the author cannot predict what action will be called for at run time. The action taken by an EXECUTE statement is always authorized at run time, in the context of the user who is running the program.

Summary

We have described the SEQUEL 2 data sublanguage, which provides a consistent, English keyword-oriented syntax for query as well as for data definition, manipulation, and control. Features of SEQUEL 2 range from simple query facilities easily learned by nonspecialists in data processing to complex facilities intended for professional programmers and data base administrators. We have shown how SEQUEL 2 may be embedded in a PL/I program and how such a program may be written to support SEQUEL 2 as a stand-alone interface for interactive users. SEQUEL 2 is the main external interface of the System R experimental data base system. We wish to emphasize that System R and SEQUEL 2 are parts of a research project in data base management and are not planned as IBM products.

Appendix: SEQUEL 2 syntax

Here we present a simplified version of the BNF syntax definition for SEQUEL 2. Emphasis has been placed on readability rather than rigor, and therefore a few minor ambiguities have been permitted. Some features that are highly specific to System R, relating to bulk loading and management of physical segments, have been omitted. Also, this syntax permits the generation of some statements that are not semantically meaningful. A more complete but less readable production syntax is currently in use in the System R project.

In this notation, square brackets [ ] indicate optional constructs. The sections that describe query, dml-, ddl-, and control-statements are useable either from a host language or as a stand-alone query language. The last section, which describes cursor operations, is intended for use with a host language only.

```
statement ::= query
            | dml-statement
            | ddl-statement
            | control-statement
            | cursor-statement
dml-statement ::= assignment
                | insertion
                | deletion
                | update
query ::= query-expr [ ORDER BY ord-spec-list ]
assignment ::= ASSIGN TO receiver : query-expr
             | ASSIGN TO receiver : CURSOR
                                        cursor-name
receiver ::= table-name [ ( field-name-list ) ]
field-name-list ::= field-name
                  | field-name-list , field-name
insertion ::= INSERT INTO receiver : insert-spec
insert-spec ::= query-expr
              | lit-tuple
deletion ::= DELETE table-name [ var-name ]
                                  [ where-clause ]
update ::= UPDATE table-name [ var-name ]
             SET set-clause-list [ where-clause ]
where-clause ::= WHERE boolean
               | WHERE CURRENT OF cursor-name
set-clause-list ::= set-clause
                  | set-clause-list , set-clause
```

```
set-clause ::= field-name = expr
             | field-name = ( query-block )
query-expr ::= query-block
             | query-expr set-op query-block
             | ( query-expr )
set-op ::= INTERSECT | UNION | MINUS
query-block ::= select-clause [ INTO target-list ]
                FROM from-list
                [ WHERE boolean ]
                [ GROUP BY field-spec-list
                [ HAVING boolean ] ]
select-clause ::= SELECT [ UNIQUE ] sel-expr-list
                | SELECT [ UNIQUE ] *
sel-expr-list ::= sel-expr
                | sel-expr-list , sel-expr
sel-expr ::= expr
           | var-name . *
           | table-name . *
target-list ::= host-location
              | target-list , host-location
from-list ::= table-name [ var-name ]
            | from-list , table-name [ var-name ]
field-spec-list ::= field-spec
                  | field-spec-list , field-spec
ord-spec-list ::= field-spec [ direction ]
                | ord-spec-list , field-spec [direction]
direction ::= ASC | DESC
boolean ::= boolean-term
          | boolean OR boolean-term
boolean-term ::= boolean-factor
               | boolean-term AND boolean-factor
boolean-factor ::= [ NOT ] boolean-primary
boolean-primary ::= predicate
                  | ( boolean )
predicate ::= expr comparison expr
            | expr BETWEEN expr AND expr
            | expr comparison table-spec
            | ⟨ field-spec-list ⟩ = table-spec
            | ⟨ field-spec-list ⟩ [IS ] [ NOT ]
                                        IN table-spec
            | IF predicate THEN predicate
            | SET ( field-spec-list ) comparison
                  table-spec
            | SET ( field-spec-list ) comparison
                  SET ( field-spec-list )
            | table-spec comparison table-spec
table-spec ::= query-block
             | ( query-expr )
             | literal
expr ::= arith-term
       | expr add-op arith-term
arith-term ::= arith-factor
             | arith-term mult-op arith-factor
arith-factor ::= [ add-op ] primary

primary ::= [ OLD | NEW ] field-spec
          | set-fn ( [ UNIQUE ] expr )
          | COUNT ( * )
          | constant
          | ( expr )
field-spec ::= field-name
             | table-name . field-name
             | var-name . field-name
comparison ::= comp-op
             | CONTAINS
             | DOES NOT CONTAIN
             | [ IS ] IN
             | [ IS ] NOT IN
comp-op ::=  = | ¬= | > | >= | < | <=
add-op ::=  + | -
mult-op ::=  * | /
set-fn ::= AVG | MAX | MIN | SUM | COUNT |
                                         identifier
literal ::= ( lit-tuple-list )
          | lit-tuple
          | ( entry-list )
          | constant
lit-tuple-list ::= lit-tuple
                 | lit-tuple-list , lit-tuple
lit-tuple ::= ⟨ entry-list ⟩
entry-list ::= entry
             | entry-list , entry
entry ::= [ constant ]
constant ::= quoted-string
           | number
           | host-location
           | NULL
           | USER
           | DATE
           | field-name OF CURSOR cursor-name
                 ON table-name
table-name ::= name
image-name ::= name
link-name ::= name
asrt-name ::= name
trig-name ::= name
name ::= [ creator . ] identifier
creator ::= identifier
user-name ::= identifier
field-name ::= identifier
var-name ::= identifier
cursor-name ::= identifier
pointer ::= identifier
save-point-name ::= identifier
host-location ::= identifier [ : identifier ]
integer ::= number

ddl-statement ::= create-table
                | expand-table
                | create-image
                | create-link
                | define-view
                | define-synonym
                | drop
                | comment
create-table ::= CREATE TABLE table-name
                                  ( field-defn-list )
field-defn-list ::= field-defn
                | field-defn-list , field-defn
field-defn ::= field-name (type [ , NONULL ] )
type ::= CHAR ( integer ) [ VAR ]
       | INTEGER
       | SMALLINT
       | DECIMAL ( integer , [ integer ] )
       | FLOAT
expand-table ::= EXPAND TABLE table-name ADD
                         COLUMN field-defn
create-image ::= CREATE [ image-mod-list ] IMAGE
                                       image-name
                 ON table-name (ord-spec-list )
image-mod-list ::= image-mod
                 | image-mod-list image-mod
image-mod ::= UNIQUE
            | CLUSTERING
create-link ::= CREATE [ CLUSTERING ] LINK
                                        link-name
                FROM table-name ( field-name-list )
                TO table-name ( field-name-list )
                [ ORDER BY ord-spec-list ]
define-view ::= DEFINE VIEW table-name
                [ ( field-name-list ) ] AS query
define-synonym ::= DEFINE-SYNONYM identifier AS
                                       table-name
drop ::= DROP system-entity name
comment ::= COMMENT ON system-entity name :
                                    quoted-string
          | COMMENT ON COLUMN table-name .
                                       field-name
                : quoted-string
system-entity ::= TABLE | VIEW | ASSERTION
                | TRIGGER | IMAGE | LINK
control-statement ::= asrt-statement
                    | define-trigger
                    | grant
                    | revoke
                    | begin-trans
                    | end-trans
                    | save
                    | restore
asrt-statement ::= ASSERT asrt-name [ IMMEDIATE ]
                   [ ON asrt-condition ] : boolean
asrt-condition ::= action-list
                 | table-name [ var-name ]
```

```
action-list ::= action
              | action-list , action
action ::= INSERTION OF table-name [ var-name ]
         | DELETION OF table-name [ var-name ]
         | UPDATE OF table-name [ var-name ]
              [ ( field-name-list ) ]
define-trigger ::= DEFINE TRIGGER trig-name
                   ON trig-condition :
                                 ( statement-list )
trig-condition ::= action
                 | READ OF table-name [ var-name ]
statement-list ::= cond-statement
                 | statement-list ; cond-statement
cond-statement ::= statement
                 | IF boolean THEN statement
grant ::= GRANT [ auth ] table-name TO user-list
              [ WITH GRANT OPTION ]
auth ::= ALL RIGHTS ON
       | operation-list ON
       | ALL BUT operation-list ON
user-list ::= user-name
            | user-list , user-name
            | PUBLIC
operation-list ::= operation
                 | operation-list , operation
operation ::= READ
            | INSERT
            | DELETE
            | UPDATE [ ( field-name-list ) ]
            | EXPAND
            | IMAGE
            | LINK
            | CONTROL
            | RUN
revoke ::= REVOKE [ operation-list ON ] table-name
                FROM user-list
begin-trans ::= BEGIN TRANSACTION
end-trans ::= END TRANSACTION
save ::= SAVE save-point-name
restore ::= RESTORE [ save-point-name ]

cursor-statement ::= let
                   | open
                   | fetch
                   | close
                   | describe
                   | execute
let ::= LET cursor-name BE query
      | LET cursor-name BE execute
open ::= OPEN cursor-name
fetch ::= FETCH cursor-name [ INTO pointer ]
close ::= CLOSE cursor-name
describe ::= DESCRIBE cursor-name INTO pointer
execute ::= EXECUTE host-location
```

Cited and general references

1. E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM* **13,** 377 (June 1970).
2. D. D. Chamberlin and R. F. Boyce, "SEQUEL: A Structured English Query Language," *Proc. ACM-SIGFIDET Workshop*, Ann Arbor, MI, May 1974.
3. G. D. Held, M. R. Stonebraker, and E. Wong, "INGRES: A Relational Data Base System," *Proc. AFIPS National Computer Conference*, Anaheim, CA, May 1975.
4. M. M. Zloof, "Query By Example," *Proc. AFIPS National Computer Conference*, Anaheim, CA, May 1975.
5. R. F. Boyce, D. D. Chamberlin, W. F. King, and M. M. Hammer, "Specifying Queries as Relational Expressions: The SQUARE Data Sublanguage," *Commun. ACM* **18,** 621 (November 1975).
6. P. Reisner, R. F. Boyce, and D. D. Chamberlin, "Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL," *Proc. AFIPS National Computer Conference*, Anaheim, CA, May 1975.
7. P. Reisner, "Use of Psychological Experimentation as an Aid to Development of a Query Language," *Research Report RJ 1707*, IBM Research Laboratory, San Jose, CA, January 1976.
8. M. M. Astrahan, et al., "System R: A Relational Approach to Data Base Management," *ACM Trans. on Data Base Systems* **1,** 97 (June 1976).
9. E. F. Codd, "Normalized Data Base Structure: A Brief Tutorial," *Proc. ACM-SIGFIDET Workshop*, San Diego, CA, November 1971.
10. R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indices," *Proc. ACM-SIGFIDET Workshop*, Houston, TX, November 1970.
11. D. D. Chamberlin, J. N. Gray, and I. L. Traiger, "Views, Authorization, and Locking in a Relational Data Base System," *Proc. AFIPS National Computer Conference*, Anaheim, CA, May 1975.
12. P. P. Griffiths and B. W. Wade, "An Authorization Mechanism for a Relational Data Base System," *Proc. ACM SIGMOD Conference*, Washington, DC, June 1976.
13. K. P. Eswaran and D. D. Chamberlin, "Functional Specifications of a Subsystem for Data Base Integrity," *Proc. International Conf. on Very Large Data Bases*, Framingham, MA, September 1975.
14. J. Mylopoulos, S. A. Schuster, and D. Tsichritzis, "A Multi-level Relational System," *Proc AFIPS National Computer Conference*, Anaheim, CA, May 1975.

Received May 25, 1976; revised July 6, 1976

The authors are located at the IBM Research Laboratory, 5600 Cottle Rd., San Jose, CA 95193.