SEQUEL: A STRUCTURED ENGLISH QUERY LANGUAGE

by

Donald D. Chamberlin
Raymond F. Boyce

IBM Research Laboratory
San Jose, California

ABSTRACT: In this paper we present the data manipulation facility for a structured English query language (SEQUEL) which can be used for accessing data in an integrated relational data base. Without resorting to the concepts of bound variables and quantifiers SEQUEL identifies a set of simple operations on tabular structures, which can be shown to be of equivalent power to the first order predicate calculus. A SEQUEL user is presented with a consistent set of keyword English templates which reflect how people use tables to obtain information. Moreover, the SEQUEL user is able to compose these basic templates in a structured manner in order to form more complex queries. SEQUEL is intended as a data base sublanguage for both the professional programmer and the more infrequent data base user.

Computing Reviews Categories: 3.5, 3.7, 4.2

Key Words and Phrases: Query Languages
Data Base Management Systems
Information Retrieval
Data Manipulation Languages

INTRODUCTION

As computer systems become more advanced we see a gradual evolution from procedural to declarative problem specification. There are two major reasons for this evolution. First, a means must be found to lower software costs among professional programmers. The costs of program creation, maintenance, and modification have been rising very rapidly. The concepts of structured programming (1,2) have been introduced in order to simplify programming and reduce the cost of software. Secondly, there is an increasing need to bring the non-professional user into effective communication with a formatted data base. Much of the success of the computer industry depends on developing a class of users other than trained computer specialists.

The work on the Structured English Query Language (SEQUEL), presented in this paper, is consistent with the trend to declarative problem specification. It attempts to identify the basic functions that are required by data base users and to develop a simple and consistent set of rules for applying these functions to data. These rules are intended to simplify programming for the professional and to make data base interaction available to a new class of users.

A brief discussion of this new class of users is in order here. There are some users whose interaction with a computer is so infrequent or unstructured that the user is unwilling to learn a query language. For these users, natural language or menu selection (3,4) seem to be the most viable alternatives. However, there is also a large class of users who, while they are not computer specialists, would be willing to learn to interact with a computer in a reasonably high-level, non-procedural query language. Examples of such users are accountants, engineers, architects, and urban planners. It is for this class of users that SEQUEL is intended. For this reason, SEQUEL emphasizes simple data structures and operations.

In a series of papers, E. F. Codd (5-9) has introduced the relational model of data, which appears to be the simplest possible general-purpose data structure, and which provides a maximum degree of data independence. In this paper we deal only with normalized relations, which can be viewed as tables of n columns and a varying number of rows, as illustrated in Figure 1.

EMP	NAME	SAL	MGR	DEPT
	SMITH	10000	JONES	TOY
	JONES	12000	ANDERSON	FURNITURE
	LEE	10000	THOMAS	APPLIANCES

Figure 1. Relation describing employees.

In addition to introducing the relational data structure, Codd has defined a language (9) which allows for the accessing or referencing of data

A Data Base Sublanguage Founded on the Relational Calculus

in relational format. This language and similar ones (COLARD (10), RIL (11)) are based on the first-order predicate calculus. Queries in these languages typically require:

1. The user to define extra variables which have as values rows or portions of rows of a relation, and

2. The user to state the query using Boolean expressions and universal and existential quantifiers.

Knuth (12) has shown that the majority of statements in FORTRAN are rather simple. We believe this is also true of queries to a data base. Previously, we have presented a data sublanguage called SQUARE (13, 14), which provides a simple and straightforward mechanism for referencing data in tables. SQUARE enables the user to describe data selection in terms of set-oriented table look-ups rather than in a row-at-a-time fashion. This makes possible the elimination of quantifiers and the elimination of explicit linking terms used to correlate information from several tables. Therefore, the SQUARE user does not need the mathematical sophistication of the predicate calculus to make relatively simple references to tables. However, it has been shown (14) that the SQUARE language is complete, i.e., any query expressible in the predicate calculus is expressible in SQUARE.

In this paper we attempt to summarize the important points of the SQUARE work and to report on subsequent work to develop a notation more familiar to untrained users than the concise mathematical notation of SQUARE. In developing this new syntax we have attempted to keep in mind the notions of top-down structured programming, the need for a linear notation, and the need for readable programs that are easy to maintain and modify. The resulting syntax can best be described as a block-structured English keyword syntax.

SUMMARY OF SQUARE specifying queries as relational expressions

In this section, the major features of the SQUARE query language will be reviewed by a series of examples. A more precise definition of these features can be found elsewhere (13). In the following sections, the query facilities of SEQUEL will be introduced and applied to the same set of examples, among others.

All examples of this and later sections are drawn from a data base describing the operation of a department store, and consisting of five tables:

EMP (NAME, DEPT, MGR, SAL, COMM)
SALES (DEPT, ITEM, VOL)
SUPPLY (SUPPLIER, ITEM, VOL)
LOC (DEPT, FLOOR)
CLASS (ITEM, TYPE)

The EMP table has a row for every store employee, giving his name, department, manager, salary and commission for the last year. The SALES table gives the volume (yearly count) in which each department sells each item. The SUPPLY table gives the volume (yearly count) in which each supplier company supplies various items to the store. The LOC table gives the floor on which each department is located, and the CLASS relation classifies the items sold into various types.

The simplest expression in SQUARE is called a mapping and is illustrated by Q1.

Q1. Find the names of employees in the toy department.

```
          EMP       ('TOY')
NAME      DEPT
```

A mapping consists of a table name (EMP), a domain (DEPT), a range (NAME), and an argument ('TOY'). The value of the mapping is the set of values of the range column of the named table whose associated values in the domain column match the argument. Mapping emulates the way in which people use tables. In this example, to find the names of employees in the toy department, a person might look down the DEPT column of the EMP table, finding 'TOY' entries and making a list of the corresponding NAME entries.

Either the domain or range of a mapping may involve more than one column, as illustrated by Q2. The result of Q2 is a list of (name, salary) pairs.

Q2. Find the names and salaries of employees who are in the toy department and whose manager is Anderson.

```
              EMP            ('TOY', 'ANDERSON')
NAME, SAL     DEPT, MGR
```

Mappings may be composed by applying one mapping to the result of another, as illustrated by Q3.

Q3. Find the items sold by departments on the second floor.

```
        SALES       °       LOC        ('2')
ITEM    DEPT        DEPT    FLOOR
```

Looking at this query procedurally, the LOC mapping produces a set of departments which serves as an argument to the SALES mapping; the result is the set of items sold by any department in the set. From a descriptive point of view, the query is written in a top-down left-to-right fashion much like the English expression. The user wants to receive ITEM(s) from SALES whenever the DEPT meets some criterion; the criterion is expressed as DEPT in LOC whenever FLOOR has the value '2'.

In general, the result of a mapping is a set of values or tuples of values. A set of values may be further processed by applying to it a mathematical function such as SUM, COUNT, AVG, MAX, or MIN:

Q4. Find the average salary of employees in the shoe department.

```
AVG (    EMP'       ('SHOE'))
     SAL       DEPT
```

The prime symbol on EMP denotes that duplicate salaries are not eliminated from the set before the AVG function is applied.

Sets produced by mappings may also be manipulated using the standard set operators union, intersection, and difference, as illustrated by Q5:

Q5. Find those items which are supplied by Levi and sold in the men's department.

```
      SUPPLY           ('LEVI') ∩       SALES       ('MEN')
ITEM          SUPPLIER              ITEM        DEPT
```

Example Q6 shows how the basic features of SQUARE can be nested to form complex queries:

Q6. Find the total volume of items of Type A sold by departments on the second floor.

```
SUM (    SALES           (     CLASS       ('A'),       LOC        ('2'))
     VOL        ITEM,DEPT  ITEM        TYPE         DEPT      FLOOR
```

BASIC FEATURES OF SEQUEL

The SEQUEL language is equivalent in power to SQUARE, but is intended for users who are more comfortable with an English-keyword format than with the terse mathematical notation of SQUARE. We will introduce the features of SEQUEL by reviewing the example queries of the previous section, and introducing some new examples.

As in SQUARE, the simplest SEQUEL expression is a mapping which specifies a table, a domain, a range, and an argument, as illustrated by Q1.

Q1. Find the names of employees in the toy department.

```
SELECT    NAME
FROM      EMP
WHERE     DEPT = 'TOY'
```

The mapping returns the entire set of names which qualify according to the test DEPT = 'TOY'.

SEQUEL presents the user with a consistent template for expression of simple queries. The user must specify the columns he wishes to SELECT, the table FROM which the query columns are to be chosen, and the conditions WHERE the rows are to be returned. The SELECT-FROM-WHERE block is the basic component of the language. In an interactive system this template might be presented to the user, who then fills in the blanks. For a simple mapping, the SELECT-FROM-WHERE block is similar to the approach taken by GIS (15) and IQF (16).

If the WHERE-clause of a mapping is omitted, the mapping returns a projection, or list of unique values of the selected column(s) taken from the entire table:

Q1.1. List all departments from the EMP table.

```
SELECT    DEPT
FROM      EMP
```

Similarly, if the SELECT clause and the word FROM are omitted, the mapping returns, in their entirety, all rows which qualify by the WHERE clause, as in Q1.2:

Q1.2. List the rows describing employees whose salary is greater than 8000.

```
EMP WHERE SAL > '8000'
```

As in SQUARE, either the domain or the range may involve more than one column, as in Q2:

Q2. Find the names and salaries of employees who are in the toy department and whose manager is Anderson.

```
SELECT    NAME, SAL
FROM      EMP
WHERE     DEPT = 'TOY'
AND       MGR = 'ANDERSON'
```

The highly explicit syntax of SEQUEL enables us to construct a logical test in the WHERE clause. For example, value-comparisons may be done using expressions of column-names, and using criteria other than equality. These points are illustrated by Q2.1.

Q2.1. Find names of employees who are either in the 'ADMIN' department or whose sum of salary and commission exceeds 10000.

```
SELECT    NAME
FROM      EMP
WHERE     DEPT = 'ADMIN'
OR        SAL + COMM > '10000'
```

As in SQUARE, a mapping may be applied to the results of an inner mapping, as in Q3.

Q3. Find the items sold by departments on the second floor.

```
SELECT    ITEM
FROM      SALES
WHERE     DEPT =
          SELECT    DEPT
          FROM      LOC
          WHERE     FLOOR = '2'
```

Once again it is important to notice the top-down structured programming influence in this example. In writing his basic SELECT-FROM-WHERE block, the user comes to the point where he wants to specify a matching criterion on the DEPT field of SALES. He may specify either a simple value match, such as DEPT = 'TOY', or an inner mapping as in Q3. If he needs to specify another mapping he merely fills out another SELECT-FROM-WHERE block. Again an interactive system can aid in this process. Thus the language reduces to a few basic building blocks and a set of simple rules for composing these blocks.

In SEQUEL, the matching criterion between the outer and inner mappings of a composed query is explicitly stated ('=' in Q3). We take advantage of this explicitness by allowing any of the relational operators =, ≠, >, >=, <, <= as a matching criterion. In general, the inner mapping generates a set of values. The outer mapping returns rows of the named table ('SALES' in Q3) whose domain-value ('DEPT' in Q3) compares by the matching criterion with any value generated by the inner mapping. Thus Q3 returns the items sold by any department on the second floor. By using the word ALL after the matching criterion, we can make the outer mapping return rows whose domain-value compares to all values generated by the inner mapping. Thus changing 'DEPT = ' to 'DEPT = ALL' in Q3 would return the items sold by all departments on the second floor. This illustrates what is called "conjunctive mapping" in SQUARE (13). Q3.1 is a further illustration.

Q3.1. Find employees whose salary is greater than that of any employee in the shoe department.

```
SELECT    NAME
FROM      EMP
WHERE     SAL > ALL
          SELECT    SAL
          FROM      EMP
          WHERE     DEPT = 'SHOE'
```

In general, any comparison operator (>, <, etc.) may be modified by the word 'ALL' on either or both sides in order to enable comparison of an item to a set or a set to a set. When a set is used in a comparison in the absence of the word 'ALL', the default meaning is 'some'.

As in SQUARE, we can apply a mathematical function to the result of a mapping by placing the function in the SELECT clause, as illustrated by Q4.

Q4. Find the average salary of employees in the shoe department.

```
SELECT    AVG (SAL)
FROM      EMP
WHERE     DEPT = 'SHOE'
```

In SEQUEL, the question of duplicates is resolved by the following defaults: if a mathematical function is applied to the results of a mapping, duplicates are preserved by the mapping; otherwise, duplicates are eliminated. These defaults can be explicitly overridden.

Q4 is an example of a general rule in SEQUEL that the SELECT clause may contain any arithmetic expression of column-names from the table being used. If mathematical functions appear in the expression, their argument is taken from the set of rows of the table which qualify by the WHERE clause. For example:

Q4.1. List each employee in the shoe department and his deviation from the average salary of the department.

```
SELECT    NAME, SAL - AVG (SAL)
FROM      EMP
WHERE     DEPT = 'SHOE'
```

Union and intersection operators may be used in SEQUEL in exactly the same way as in SQUARE, as illustrated by Q5.

Q5. Find those items which are supplied by Levi and sold in the men's department.

```
SELECT    ITEM
FROM      SUPPLY
WHERE     SUPPLIER = 'LEVI'
∩
SELECT    ITEM
FROM      SALES
WHERE     DEPT = 'MEN'
```

It is important to distinguish between the effects of using AND or OR within a mapping, and the effects of using union or intersection between mappings. In the first case, the clauses separated by AND/OR apply to the same row of the table; in the second case, the mappings are applied independently and the union/intersection is applied to the results. To illustrate this point, assume that the EMP table contains the following entries:

NAME	DEPT	MGR
JOHN	SHOE	BOB
FRED	SHOE	FRANK
FRED	TOY	BOB
BILL	TOY	FRANK

The the results of two example queries are given below:

```
Q5.1.  SELECT   NAME                Result:  JOHN
       FROM     EMP
       WHERE    DEPT = 'SHOE'
       AND      MGR = 'BOB'

Q5.2.  SELECT   NAME                Result:  JOHN, FRED
       FROM     EMP
       WHERE    DEPT = 'SHOE'
       ∩
       SELECT   NAME
       FROM     EMP
       WHERE    MGR = 'BOB'
```

Example Q6 is repeated here to show how SEQUEL features can be nested to form complex queries:

Q6. Find the total volume of items of type A sold by departments on the second floor.

```
SELECT    SUM (VOL)
FROM      SALES
WHERE     ITEM =
          SELECT    ITEM
          FROM      CLASS
          WHERE     TYPE = 'A'
AND       DEPT =
          SELECT    DEPT
          FROM      LOC
          WHERE     FLOOR = '2'
```

Note that in this query indentation is used to separate mappings and show the structure of the query. In practice it might be more convenient to make indentation optional and to terminate each mapping by a semicolon. Then in the above query there would be one semicolon after TYPE = 'A' and two semicolons after FLOOR = '2'.

ADDITIONAL CONCEPTS

The primary motivation for the SQUARE query language was to develop an easy means for accessing a relational data base. We believed that the

applied predicate calculus with its concepts of variables and quantifiers required too much sophistication for the ordinary user. SQUARE successfully avoids the concepts of bound variables and quantifiers, but still requires a free variable whenever it becomes necessary to correlate values from a specific row in a table with values from another row or rows. Typically in SQUARE such a query is of the following form:

free-variable : test

The free variable represents a row of a table. The result of the query is a set of values taken from the rows for which the test is true. A more complete description of free variables in SQUARE is provided elsewhere (13,14). Example Q7 illustrates the concept.

Q7. Find the names of managers who manage more than ten employees.

```
x       ∈ EMP : COUNT (     EMP      (x    ) ) > 10
  NAME                  NAME     MGR   NAME
```

Note that in Q7 the free variable x is used to correlate a manager's name with a group of rows representing his employees, so that this group may be counted. Experience has shown that this sort of "grouping" occurs quite frequently in queries. Accordingly, a way is provided in SEQUEL to divide the rows of a table into groups according to the values of one or more columns, in a way analogous to the concept of a "glump" in Information Algebra (17). An optional GROUP BY clause may be attached to any FROM clause in SEQUEL, with the effect that the rows of the table are considered to be in groups of matching column-values. For example, if a query contains the clause FROM EMP GROUP BY MGR, the rows of the EMP table are formed into groups of matching MGR for the purpose of this query. Within the scope of such a clause, there are certain restrictions on the column-references which may be made. The grouping column or columns (MGR in the above example) may be referred to because it has only one value per group. Mathematical functions on column-values may be used, with the implied rule that they take a group of column-values as their argument and return a single value for the group. For example, within the scope of the clause FROM EMP GROUP BY MGR, the function AVG (SAL) would return, for each manager, the average salary of his employees. Other functions such as SUM, COUNT, and MAX operate in similar ways. A column-name which is not part of the grouping criterion may not be referred to except as an argument to a function which returns a single value per group. Example Q7 is now repeated in SEQUEL to illustrate the GROUP BY feature.

Q7. Find the names of managers who manage more than ten employees.

```
SELECT    MGR
FROM      EMP GROUP BY MGR
WHERE     COUNT (NAME ) > 10
```

The grouping concept is further illustrated by example Q8.

Q8. List the departments and the sum of the employee salaries in each department.

```
SELECT    DEPT, SUM (SAL)
FROM      EMP GROUP BY DEPT
```

If needed, the user may construct a literal tuple or an entire table out of constants for use in a query. This is illustrated by Q9, which also illustrates the special function SET, which returns the set of values grouped together by the GROUP BY clause.

Q9. Find those departments which have employees named SMITH, JOHNSON, and HUGHES, all having JONES as their manager.

```
SELECT    DEPT
FROM      EMP
WHERE     SET (NAME, MGR) ⊇

          { < 'SMITH', 'JONES' >,
            < 'JOHNSON', 'JONES' >,
            < 'HUGHES', 'JONES' >}
```

There are a few remaining queries which require a free variable in SQUARE in order to resolve ambiguity in a column-reference. In most cases, the ambiguity can be resolved in SEQUEL by qualifying the column-name with its table-name. This is illustrated by Q10, which implements what Codd (7) would call a "join" between the SALES and SUPPLY tables on their ITEM columns:

Q10. List rows of SALES and SUPPLY concatenated together whenever their ITEM values match.

```
SALES, SUPPLY
WHERE SALES . ITEM = SUPPLY . ITEM
```

In those cases where a table-name is not sufficient to resolve the ambiguity (because the same table-name appears more than once in the query), SEQUEL allows an arbitrarily-chosen block label to be attached to a mapping and used to qualify column-references. This is illustrated by Q11.

Q11. Find names of employees whose salary is greater than their manager's salary.

```
B1:  SELECT    NAME
     FROM      EMP
     WHERE     SAL >
               SELECT    SAL
               FROM      EMP
               WHERE     NAME = B1.MGR
```

In this query, the outer mapping, labelled B1, returns the NAME value of all rows which meet the following test: the SAL value of the B1-row must be greater than the SAL value of the row whose NAME is the same as the MGR of the B1-row.

COMPARISON WITH PREDICATE CALCULUS BASED LANGUAGES

In this section we illustrate the differences in perception between queries expressed in languages (9-11) based on the first order predicate calculus and those expressed in SEQUEL. The calculus-based languages permit the description to be in terms of tests on individual rows of the relations in question.

Consider the expression of query Q6 in the calculus:

Q6. Find the total volume of items of type A sold by departments on the second floor.

$$\text{SUM } \{x[\text{VOL}] \in \text{SALES} :$$

$$(\exists y \in \text{CLASS})\ ((y[\text{ITEM}] = x[\text{ITEM}]) \wedge (y[\text{TYPE}] = \text{'A'}))$$

$$\wedge\ (\exists z \in \text{LOC})\ ((z[\text{DEPT}] = x[\text{DEPT}]) \wedge (z[\text{FLOOR}] = \text{'2'}))\}$$

The calculus programmer must be concerned with:

1. Setting up three variables x, y, and z to sequence through each table.

2. The notions of existential quantifiers and bound variables.

3. The explicit linking terms

   "y[ITEM] = x[ITEM]" and "z[DEPT] = x [DEPT]"

4. The actual matching criteria for membership in the output set.

We showed earlier, however, that this query could be expressed in SEQUEL simply by composing three mapping blocks. Of course, we do not suggest that really complex queries are simple to express in SEQUEL; rather we stress the difference between SEQUEL and more conventional approaches. This distinction in perception has been treated more fully in (13).

SUMMARY

This paper has presented the data manipulation facility (DMF) of a data sublanguage based on the relational model of data. This sublanguage has also been used as the basis of a data definition facility (DDF) (18). Together the DDF and DMF comprise a query facility oriented toward users who are not computer specialists. The simple block-structured English keyword syntax and simple operations on tables enable users to interact with the SEQUEL system with less training and sophistication than would be required in either a calculus-oriented system or a traditional procedural programming language. SEQUEL

users describe the relevant data to be accessed by set expressions rather than by row-at-a-time iteration. The resulting queries tend to be concise, clearly expressed, and easy to write, maintain, and modify. The formal syntax for SEQUEL queries is given in the Appendix.

ACKNOWLEDGEMENT

The authors wish to thank M. M. Astrahan, E. D. Carlson, E. F. Codd, P. L. Fehder, W. F. King, P. Reisner, and R. Williams for their interactions during the development of SEQUEL.

REFERENCES

(1) E. W. Dijkstra, "Structured Programming," Software Engineering Techniques, NATO Science Committee (ed. J. N. Burton and E. Randell), 1969, pp. 88-93.

(2) F. T. Baker, "System Quality Through Structured Programming," Proc. AFIPS 1972 FJCC, vol. 41, 1972, pp. 339-343.

(3) F. B. Thompson, P. Lockeman, B. R. Dostert, R. Deverill, "REL: A Rapidly Extensible Language System," Proc. 24th ACM National Conference, New York, N. Y., August 1969, pp. 399-417.

(4) E. F. Codd, "Seven Steps to Rendezvous with the Casual User," IBM Research Report RJ 1333 IBM Research Laboratory, San Jose, Calif., January 1974.

(5) E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Comm. ACM, vol. 13, no. 6 (June 1970) pp. 377-387.

(6) E. F. Codd, "Further Normalization of the Data Base Relational Model," Courant Computer Science Symposia, vol. 6, Data Base Systems, Prentice-Hall, New York, May 1971.

(7) E. F. Codd, "Relational Completeness of Data Base Sublanguages," Courant Computer Science Symposia, vol. 6, Data Base Systems, Prentice-Hall, New York, May 1971.

(8) E. F. Codd, "Normalized Data Base Structure -- A Brief Tutorial," Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif., November 1971.

(9) E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif., November 1971.

(10) G. Bracchi, A. Fedeli, and P. Paolini, "A Language for a Relational Data Base Management System," Proc. of the Sixth Annual Princeton Conf. on Info. Sci. and Systems, March 1972, pp. 84-92.

(11) P. L. Fehder, "The Representation-Independent Language", IBM Technical Report RJ 1121, IBM Research Laboratory, San Jose, Calif., November 1972.

(12) D. E. Knuth, "An Empirical Study of FORTRAN Programs," Software--Practice and Experience, Vol. 1, No. 2 (April 1971) pp. 105-133.

(13) R. F. Boyce, D. D. Chamberlin, W. F. King III, and M. M. Hammer, "Specifying Queries as Relational Expressions," Proceedings of ACM SIGPLAN/SIGFIDET Interface Meeting on Programming Languages and Information Retrieval, Gaithersburg, Md., November 1973.

(14) R. F. Boyce, D. D. Chamberlin, W. F. King III, and M. M. Hammer, "Specifying Queries as Relational Expressions: SQUARE," IBM Technical Report RJ 1291, IBM Research Laboratory, San Jose, Calif., October 1973.

(15) J. H. Bryant and P. Semple, Jr., "GIS and File Management," Proceedings of ACM National Conference, 1966, pp. 97-107.

(16) "Interactive Query Facility (IQF) for IMS/360," Publication No. GH20-1074, IBM Corp., White Plains, N.Y., (1971).

(17) CODASYL, "An Information Algebra," Comm. ACM, 5, 4 (April 1962) pp. 190-204.

(18) R. F. Boyce and D. D. Chamberlin, "Using a Structured English Query Language as a Data Definition Facility," IBM Technical Report RJ 1318, IBM Research Laboratory, San Jose, Calif., December 1973.

APPENDIX: SEQUEL QUERY SYNTAX

NOTE: [ ] denotes optional
| denotes alternative form

```
query :: = basic-query
         |  basic-query ∩ query
         |  basic-query ∪ query
         |  basic-query - query
         |  (query)

basic-query :: = [label:] sel-clause-list [where-clause];

sel-clause-list :: = sel-clause

                   |  sel-clause-list, sel-clause
```

```
sel-clause :: = [SELECT expr-list FROM] table-name
                [GROUP BY col-name-list]  [dup-code]

dup-code :: = DUPL | UNIQUE

expr-list :: = expr
             |   expr-list, expr

col-name-list :: = col-name
                 |   col-name-list, col-name

where-clause :: = WHERE boolean

boolean :: = predicate
           |   predicate AND boolean
           |   predicate OR boolean
           |   NOT boolean
           |   (boolean)

predicate :: = comparand comp-op comparand

comp-op :: = [ALL] rel-op [ALL]
           |   set-op

rel-op :: = | ≠ | < | ≤ | > | ≥

set-op :: = ⊃ | ⊇ | ⊂ | ⊆ | ∌ |∉

comparand :: = expr
             |   query

expr :: = atom
        |   expr + atom
        |   expr - atom
        |   expr x atom
        |   expr / atom
        |   (expr)
        |   literal

atom :: = col-name
        |   table-name . col-name
        |   label . col-name
        |   constant
        |   set-fn (col-name)

set-fn :: = SUM | COUNT | AVG | MAX | MIN | SET
```

```
literal :: = lit-table
           |   lit-tuple
           |   constant

lit-table :: {lit-tuple-list}

lit-tuple-list :: = lit-tuple
                  |   lit-tuple-list, lit-tuple

lit-tuple :: = <constant-list>

constant-list :: = constant
                  |   constant-list, constant

constant :: = 'string'
            |   'number'
            |   ϕ
```