Information Retrieval and Language Processing
C.A. Montgomery Editor

Specifying Queries as Relational Expressions: The SQUARE Data Sublanguage

Raymond F. Boyce, Donald D. Chamberlin, and W. Frank King III
IBM Research Laboratory, San Jose
and
Michael M. Hammer
Massachusettes Institute of Technology

This paper presents a data sublanguage called SQUARE, intended for use in ad hoc, interactive problem solving by non-computer specialists. SQUARE is based on the relational model of data, and is shown to be relationally complete; however, it avoids the quantifiers and bound variables required by languages based on the relational calculus. Facilities for query, insertion, deletion, and update on tabular data bases are described. A syntax is given, and suggestions are made for alternative syntaxes, including a syntax based on English key words for users with limited mathematical background.

Key Words and Phrases: database, data sublanguages, relations, query languages, casual user, relational data model, tabular data, interactive problem solving, nonprocedural languages, relational completeness

CR Categories: 3.50, 3.70, 4.20

Authors' addresses: R.F. Boyce, D. Chamberlin, and W.F. King III, IBM Research Laboratory, Monterey and Cottle Roads, San Jose, CA 95193; M.M. Hammer, MIT, Cambridge, MA 02139.

1. Introduction

In a series of papers [7–10], Codd has introduced the relational model of data and discussed its advantages in terms of simplicity, symmetry, data independence, and semantic completeness.

Given sets $S_1, S_2, \ldots, S_n$ (not necessarily distinct), $R(S_1, S_2, \ldots, S_n)$ is a relation of degree n on these n sets if it is a set of n-tuples each of whose elements has its first component from S_1, its second component from S_2, etc. In other words, $R(S_1, S_2, \ldots, S_n)$ is a subset of the Cartesian product $S_1 \times S_2 \times \ldots \times S_n$. In this paper we will deal only with *normalized* relations [10]. A relation is normalized if each of its domains is simple, i.e. no domain is itself a relation.

A normalized relation can be viewed as a table of n columns and a varying number of rows, for example:

EMP:

NAME	SALARY	MANAGER	DEPARTMENT
SMITH	10,000	JONES	TOY
JONES	12,000	DAHL	FURNITURE
LEE	10,000	THOMAS	APPLIANCE

A normalized relation has the following properties:
1. Column homogeneity—in any particular column all items are of the same type.
2. All rows of the table are distinct.
3. The ordering of the rows is immaterial.
4. If distinct names are given to the columns the ordering of the columns is immaterial.

In addition to introducing the relational data structure, Codd [8] has defined two language models, called relational calculus and relational algebra, which are intended to serve as bases for database sublanguages.

The relational calculus is an applied predicate calculus [17]. Data sublanguages based on the relational calculus require the user to invent a variable to represent a tuple of a relation, and to state a predicate which defines those tuples which are of interest in a particular query. Relational calculus-based languages rely on the universal and existential quantifiers in the formulation of queries. Relations and quantifier-based languages have a background of use in inferential question-answering systems such as the RAND Relational Data File [13, 15]. In this context, Kuhns [14] has proposed an "extensional universal quantifier." Codd [9] has proposed a noninferential data sublanguage called ALPHA, based directly on the relational calculus. ALPHA

permits the user, by means of variables and quantifiers, to state a query in a nonprocedural way, thus providing data independence and allowing the database system to optimize execution of the query. Other languages with the same basic orientation include COLARD [3] and RIL [11]. A more recent calculus-based language which avoids the use of quantifiers is QUEL [19].

Codd's relational algebra, on the other hand, allows the user to state a query by means of a procedure based on high level operators such as join, projection, and restriction, which take whole relations as their operands. Early work in developing algebras for information retrieval includes the "information algebra" of Codasyl [6] and the "set-theoretic data structure" of Childs [5]. Implemented systems whose orientation is similar to that of Codd's relational algebra include MACAIMS [12], RDMS [18], and IS/1 [16]. Algebra-based data sublanguages have the advantage that they avoid the use of quantifiers in expressing queries. However, as Codd has pointed out [8], algebraic languages require the user to express a query in the form of a procedure, or sequence of operations, and hence require a certain "programming" orientation on the part of the user, while at the same time limiting the opportunities for system-directed optimization of the query.

This paper presents a data sublanguage called SQUARE, based on the relational model of data but different in orientation from both the relational algebra and the relational calculus. SQUARE is a nonprocedural language having the same advantages for data independence and query optimization as the relational calculus; however, SQUARE does not involve the use of quantifiers or bound variables, and hence does not require a high degree of mathematical sophistication on the part of the user. SQUARE is intended for use by the *nonprogramming professional*: a user such as an urban planner, accountant, or sociologist, who has a need for interaction with a large database but does not wish to learn a conventional computer programming language. SQUARE enables the user to express queries in terms of the natural primitive functions used by people to find information in tables, such as looking up a value in a column and finding the associated value in another column. Thus, most semantically simple queries are expressed very simply and concisely in SQUARE. At the same time, SQUARE has been shown to be "relationally complete" in that its power is equivalent to that of the relational algebra and relational calculus.

Section 2 introduces the data manipulation facilities of SQUARE, a subset of which has been presented in Boyce et al. [1]. Section 3 explores in greater depth the differences in perception of queries between SQUARE and languages based on the relational calculus. Section 4 briefly considers the question of alternative syntaxes for SQUARE. Section 5 contains the proof of relational completeness. An informal syntax for SQUARE is given in the Appendix.

In this paper we do not deal with the question of a data description language for relations. The use of a SQUARE-like language for describing multiple views of data and for controlling data integrity and authorization is discussed elsewhere [2].

2. Data Manipulation Facilities

As we introduce the facilities of SQUARE, we will illustrate them by examples. The examples of this section are drawn from a database describing the operation of a department store, as follows:

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

The EMP relation has a row for every store employee, giving his name, salary, manager, and department. The SALES relation gives the volume (yearly count) in which each department sells each item. The SUPPLY relation gives the volume (yearly count) in which each department obtains various items from its various supplier companies. We assume that the SALES and SUPPLY relations have no zero-volume entries (e.g. if the Toy Department does not sell dresses, there is no 'TOY, DRESS, 0' entry in the SALES relation). The LOC relation gives the floor on which each department is located, and the CLASS relation classifies the items sold into various types.

We now proceed to describe the syntax of a relational expression, i.e. an expression which evaluates to a relation. The simplest form of relational expression is called a mapping, and is illustrated by Q1.

Q1. Find the names of employees in the Toy Department.

$$\underset{\text{NAME}}{\overset{\text{EMP}}{\underset{\text{DEPT}}{}}}(\text{'TOY'})$$

A mapping consists of a relation name (EMP), a domain name (DEPT), a range name (NAME), and an argument ('TOY'). The value of the mapping is the set of values in the range column of the named relation whose associated values in the domain column match the argument. The mapping in Q1 evaluates to a unary relation (a list of names). 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 relation, finding 'TOY' entries and making a list of the corresponding NAME entries.

In terms of the relational calculus the notion of mapping can be defined as:

$$\underset{B}{\overset{R}{\underset{A}{}}}(S) \equiv \{r[B] : r \in R \wedge r[A] = S\}$$

The argument of a mapping may be either a single value (e.g. 'TOY') or a set of values. If the argument is a set, the mapping returns all those range values whose

corresponding domain values match *any* element of the argument. Formally, if the argument S is a set of individual values s_i,

$$\underset{B\ A}{R}(S) = \bigcup_i \underset{B\ A}{R}(s_i)$$

For this reason the mapping is generally called a disjunctive mapping. For simplicity the term mapping in this paper always refers to a disjunctive mapping.

In certain instances (e.g. with built-in functions SUM, COUNT, etc.) the elimination of duplicates from the set of returned values is undesirable. Consequently, a special type of mapping, denoted by a *prime symbol* on the relation name, which does not remove duplicates is defined. For example,

Q2. Find the average salary of employees in the Shoe Department.

$$AVG\left(\underset{SAL}{EMP'}(\underset{DEPT}{\text{'SHOE'}})\right)$$

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

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

$$\underset{ITEM}{SALES} \circ \underset{DEPT}{LOC}(\underset{FLOOR}{2})$$

The floor '2' is first mapped to the departments located there, and then to the items which they sell. The range of the inner mapping must be compatible with the domain of the outer mapping, but they need not be identical, as illustrated by Q4.

Q4. Find the salary of Anderson's manager.

$$\underset{SAL}{EMP} \circ \underset{MGR}{EMP}(\underset{NAME}{\text{'ANDERSON'}})$$

Q3 is repeated in Section 3 to demonstrate the different perception that is required in order to express the query in a language based on relational calculus.

The next important building block of relational expressions is called a *free variable*. A relational expression containing a free variable takes the following form:

free-variable-list : test

On the left side of the colon are listed the free variables to be used in the query and the relations to which they belong. Each free variable represents a row of a relation. Free variables may be given arbitrary names provided they do not conflict with the names of relations. On the right side of the colon is a logical test which may be true or false for each set of values of the free variables. The value of the expression is the set of free-variable values for which the test is true. A subscripted free variable represents a particular field value from the row represented by the free variable. For example:

Q5. Find the names of employees who earn more than their managers.

$$x_{NAME} \in EMP : x_{SAL} > \underset{NAME}{EMP}(x_{MGR})$$

The following types of operators are permissible in tests:

numeric comparisons:	$= \neq < \leq > \geq$
set comparisons:	$= \neq \supset \supseteq \subset \subseteq$
arithmetic operators:	$+ - \times /$
set operators:	$\cap \cup -$
logical connectives:	$\wedge \vee \sim$
parentheses for grouping:	( )
built in functions:	SUM, COUNT, AVG, MAX, MIN, etc.

The following example constructs a binary relation:

Q6. List the name and salary of all managers who manage more than ten employees.

$$x_{NAME,\ SAL} \in EMP :$$
$$COUNT\left(\underset{NAME}{EMP}(x_{NAME})\right) > 10$$

The absence of a subscript on a free variable denotes "all columns." Thus in the above example x would mean the same as $x_{\text{NAME, SAL, MGR, DEPT}}$.

The free variable is introduced into queries where it becomes necessary to correlate information pertaining to a specific row in a table with another row or set of rows from some table. Consequently, this variable is introduced only for queries that are more complex than simple selection. As can be seen in Section 3, all queries regardless of complexity require free variables in calculus-based languages.

Another important concept is that of projection. If a relation name appears subscripted on the left by one or more column names, it represents the set of unique tuples of values occurring in those columns of the relation. For example, $_{ITEM}SUPPLY$ is the set of all item values in the SUPPLY relation. This feature is useful in constructing expressions like the following:

Q7. Find those companies, each of which supplies every item.

$$x_{COMP} \in SUPPLY :$$
$$\underset{ITEM}{SUPPLY}(x_{COMP}) = \underset{ITEM}{SUPPLY}$$

Note that equality here is set equality.

We will now discuss some extensions to the concept of mapping. A mapping may specify more than one domain column in which case each domain column must be compatible with its respective argument. If an argument is a set then the value of the domain column must match some element of the set. This facility is useful in dealing with n-ary associations. For example:

Q8. Find the volume of guns sold by the Toy Department.

$$\underset{VOL}{SALES}(\underset{DEPT,\ ITEM}{\text{'TOY', 'GUN'}})$$

Similarly, a mapping may specify more than one range column, in which case it returns tuples of values from the columns specified. If no range columns are specified for a mapping, the range is assumed to be all columns.

When one of the numeric comparison operations $\neq$, $<$, $\leq$, $>$, $\geq$, is used as prefix to the argument of a mapping, the argument effectively becomes the set of all values which compare by the given operator with the given argument. This type of mapping often avoids the use of a free variable, as illustrated in Q9.

Q9. List the names and managers of employees in the Shoe Department with salary greater than 10,000.

$$\underset{\text{NAME, MGR}}{\text{EMP}}\ (\underset{\text{DEPT, SAL}}{\text{'SHOE'}},\ >\ 10{,}000)$$

The numeric comparison operators $>$, $\geq$, $<$, $\leq$, $\neq$ may also be extended so that a number may be compared to a set. This is done by placing the word SOME or ALL on the side(s) of the comparison operator which is a set. For example, X > ALL Y is true if the number X is greater than all elements of the set Y, and Y ALL $\neq$ Z is true if all elements of Y are unequal to the number Z (i.e. each element of Y is unequal to Z). This facility is useful in queries like the following:

Q10. Find the names of those employees who make more than any employee in the Shoe Department.

$$x\ \underset{\text{NAME}}{\epsilon\ \text{EMP}} : x\ \underset{\text{SAL}}{>\ \text{ALL}}\ \underset{\text{SAL}}{\text{EMP}}\ (\underset{\text{DEPT}}{\text{'SHOE'}})$$

In understanding Q10, it is important to remember that the free variable x represents a row of the EMP relation. If the test (which uses the SAL value of the row) is true, the NAME value of the row is returned. All rows of the relation are tested in this way, and duplicate values are eliminated from the returned set.

It should be noted that the functions of ALL and SOME when used in conjunction with the comparison operators $<$, $\leq$, $>$, $\geq$ could be accomplished equally well by the built-in functions MAX and MIN. This fact is shown by the following table, which specifies how any modifier may be replaced by a built-in function:

	$>$, $\geq$	$<$, $\leq$
SOME θ[1]	MAX θ	MIN θ
ALL θ	MIN θ	MAX θ
θ SOME	θ MIN	θ MAX
θ ALL	θ MAX	θ MIN

[1] θ represents a comparison operator.

Examples:

$$X > \text{ALL } Y \equiv X > \text{MAX}(Y)$$
$$Y \text{ SOME} < \text{ALL } Z \equiv \text{MIN}(Y) < \text{MIN}(Z)$$

Another language feature which is occasionally useful is a special type of mapping called a *conjunctive mapping*. As in a disjunctive mapping, a relation name, domain, range, and argument are specified, but the domain name is underlined to denote the conjunctive mapping. The conjunctive mapping differs from a disjunctive mapping only when the argument is a set. In this case, the conjunctive mapping returns the set of range values whose corresponding domain values match *all* elements of the argument set. Formally, we write the following definitions for a disjunctive mapping and a conjunctive mapping on a set S of values s_i:

$$\underset{\text{B A}}{\text{R}}\ (\text{S}) \equiv \bigcup_i\ \underset{\text{B A}}{\text{R}}\ (s_i)$$

$$\underset{\text{B }\underline{\text{A}}}{\text{R}}\ (\text{S}) \equiv \bigcap_i\ \underset{\text{B A}}{\text{R}}\ (s_i)$$

As an example of the use of a conjunctive mapping, we might express Q7 as follows, eliminating the free variable:

$$\underset{\text{COMP}}{\text{SUPPLY}}\ (\underset{\text{ITEM ITEM}}{\text{SUPPLY}})$$

In the case of a mapping with more than one domain, each of the domains may participate conjunctively or disjunctively in the mapping; those domains which participate conjunctively are underlined. This is illustrated by Q11.

Q11. Find companies, each of which supplies every item of type A to some department on the second floor.

$$\underset{\text{COMP}}{\text{SUPPLY}}\ (\underset{\text{DEPT}}{\underset{\text{DEPT, }\underline{\text{ITEM}}}{\text{LOC}}}\ (2),\ \underset{\text{ITEM}}{\underset{\text{ITEM TYPE}}{\text{CLASS}}}\ (\text{'A'}))$$

The formal definition of a mapping in which some domains participate conjunctively is as follows (before applying the definition, permute the domains so that the conjunctive domains are on the right):

If $S_1 = \{s_{i_1}\}$, $S_2 = \{s_{i_2}\}$, etc., then

$$\underset{\text{B }\underline{\text{A}_1 \ldots \text{A}_k\text{A}_{k+1} \ldots \text{A}_n}}{\text{R}}\ (\text{S}_1\text{S}_2 \ldots \text{S}_n) \equiv$$

$$\bigcup_{i_1} \cdots \bigcup_{i_k} \bigcap_{i_{k+1}} \cdots \bigcap_{i_n} \underset{\text{B A}_1 \cdots \text{A}_n}{\text{R}}\ (s_{i_1} s_{i_2} \cdots s_{i_n}).$$

In the foregoing, we have discussed various kinds of relational expressions. When the user inputs a relational expression, it is evaluated and displayed. From a human factor viewpoint it is frequently desirable to decompose a complex query into a sequence of simple queries. Consequently, a SQUARE user may also assign a relational expression to a named variable. This variable then denotes a temporary relation in the user's workspace, which may be used in the same ways as the permanent relations in the database. This permits multiple-step queries such as the following:

Q12. Among all departments with total salary greater than 100,000, find those departments which sell dresses.

$$\text{BIGDEPTS} \leftarrow x\ \epsilon\ \text{SALES}:$$
$$\text{SUM}(\underset{\text{SAL}}{\underset{\text{DEPT}}{\text{EMP}}}\ (x\ \underset{\text{DEPT}}{)})\ >\ 100{,}000;$$
$$\underset{\text{DEPT}}{\underset{\text{ITEM}}{\text{BIGDEPTS}}}\ (\text{'DRESS'})$$

The first two lines create a temporary relation BIGDEPTS, consisting of those rows of the SALES relation which pertain to departments with total salary greater than 100,000. The third line is a relational expression which completes the query by making use of BIGDEPTS.

In constructing a temporary relation, it is not always obvious what the column-names of the new relation will be. In such cases, the user must give the

new column-names as subscripts on the name of the new relation, as illustrated in the next example.

It is occasionally necessary to construct a new relation containing a column not taken from any existing relation, but computed by some arithmetic operations. This is permitted by means of a "computed variable" which is given a name on the left side of the colon and defined on the right side, as follows:

Q13. Create a new relation ANNUAL having columns ITEM and TOTVOL. The new relation must contain all items and, for each, the total volume of that item sold by all departments.

$$ANNUAL_{ITEM,TOTVOL} \leftarrow x_{ITEM} \in SALES,\ Q :$$

$$Q = SUM({}_{VOL}SALES'_{ITEM}(x_{ITEM}))$$

This query has one free variable (x) and one computed variable (Q).

SQUARE also provides facilities for inserting, deleting, and updating rows of a relation. These facilities operate in exactly the same way on permanent relations in the database and on temporary relations in the user's work space.

The insert operation has the following general form:

$$\downarrow RELATION_{field\text{-}list}(argument)$$

New rows are inserted into the named relation. The values for the fields named in the field-list are determined by the argument, and the remaining fields, if any, are made null. The argument may be a sequence of constants or a relational expression. For example:

Q14. Insert into EMP a new employee named Adams in the shoe department, leaving his salary and manager null.

$$\downarrow EMP_{NAME,DEPT}(\text{'ADAMS'}, \text{'SHOE'})$$

If the argument is omitted, the downward arrow denotes creation of a new (empty) permanent relation in the database, having columns as named in the field-list. For example:

Q15. Create a new persistent relation named INVENTORY, having columns ITEM and QUANTITY.

$$\downarrow INVENTORY_{ITEM,QUANTITY}$$

The syntax of the delete operator is similar to that for insertion:

$$\uparrow RELATION_{field\text{-}list}(argument)$$

The relation is searched for rows whose named fields match the argument (or match any row of the argument if it evaluates to a relation). All matching rows are deleted. For example:

Q16. Delete from EMP all the employees who are in the same department as Anderson.

$$\uparrow EMP_{DEPT}({}_{DEPT}EMP_{NAME}(\text{'ANDERSON'}))$$

An entire relation may be deleted from the database by omitting the field-list and argument:

Q17. Delete the INVENTORY relation from the database.

$$\uparrow INVENTORY$$

The operation for updating rows is somewhat more complex. In general, it is necessary to give a criterion for identifying rows to be updated, and to specify the update to be done. The general syntax is as follows:

$$\rightarrow RELATION_{match\ fields;\ update\ fields}(argument)$$

If m match fields and n update fields are specified, the argument must evaluate to have $m + n$ columns. If the match fields of a row in the relation to be updated match the first m fields of an argument row, then the update fields of that relation row are replaced by the last n fields of that argument row. For example:

Q18. Move the location of the toy department to the second floor.

$$\rightarrow LOC_{DEPT;FLOOR}(\text{'TOY'}, 2)$$

When one of the update fields is followed by one of the symbols $+$, $-$, $\times$, $/$, the argument values are used to update that field relative to its current value by the indicated operation:

Q19. Give a 10 percent raise to all employees in the shoe department who make less than 10,000.

$$\rightarrow EMP_{DEPT,SAL;SAL\times}(\text{'SHOE'}, < 10{,}000, 1.1)$$

The feature of updating a relation relative to its current value permits a row of a relation to be updated multiple times in a single statement, as in the following:

Q20. Assume existence of a relation NEWSALES having columns DEPT, ITEM, and VOL. For each DEPT, ITEM pair in NEWSALES, add VOL to the VOL entry in the corresponding row of SALES.

$$\rightarrow SALES_{DEPT,ITEM;VOL+}(NEWSALES)$$

If a given DEPT, ITEM pair occurs more than once in NEWSALES, the corresponding VOL entries will be successively added to the proper row of SALES.

All tests for matching rows are done before the update begins, to avoid the possibility that an updated row might match another argument row and be updated again. This is illustrated by Q21, which also shows how multiple rows of constants may be represented in an argument:

Q21. Renumber floors in the LOC relation so that floor 2 becomes floor 1 and floor 1 becomes floor 0.

$$\rightarrow LOC_{FLOOR;FLOOR}([<2, 1>, <1, 0>])$$

Q21 might update the LOC relation as shown below.

(Before)

LOC:

DEPT	FLOOR
SHOE	1
TOY	2
FURNI-TURE	2
CREDIT	3

(After)

LOC:

DEPT	FLOOR
SHOE	0
TOY	1
FURNI-TURE	1
CREDIT	3

3. Comparison with Relational Calculus

In this section we illustrate the difference in perception between queries expressed in the relational calculus and those expressed in SQUARE. As we have already mentiond, ALPHA [9], COLARD [3], RIL [11], and QUEL [19] are examples of languages based on the relational calculus. They permit the description of sets of data but require the description to be in terms of tests on individual rows of the relations in question. Thus, in the relational calculus Q1 is expressed as follows:

$\{\nu\,[\text{NAME}] \in \text{EMP} : \nu\,[\text{DEPT}] = \text{'TOY'}\}$

where ν is a variable which ranges over rows of EMP and ν [NAME] is the projection of ν on the domain NAME. Even for this simple query the user must invent a variable to be used as a cursor for selection of rows.

Q3 shows how a "composed mapping" is expressed in the relational calculus.

Q3. $\{\nu_0[\text{ITEM}] \in \text{SALES} : \exists(\nu_1 \in \text{LOC})[(\nu_1[\text{FLOOR}] = 2) \wedge (\nu_1[\text{DEPT}] = \nu_0[\text{DEPT}])]\}$

Here the distinction between the user's perception of the languages becomes clearer. In Section 2 we saw that the SQUARE user could view this query as a simple combination of table lookups. The calculus user must be concerned with: (1) setting up two variables, ν_0 and ν_1, which represent rows of the two tables; (2) the notions of existential quantifier and bound variable; (3) the explicit linking term, "ν_0[DEPT] = ν_1[DEPT]," which describes the interrelationship between the variables; and (4) the actual criterion, "ν_1[FLOOR] = 2," which completes the definition of the answer set.

As queries become more complex the differences between the languages become greater. More variables and linking terms are required in the calculus and the management of quantifiers becomes more complex. SQUARE can express complex queries without the use of bound variables, quantifiers, or linking terms.

4. Alternative Syntaxes

An informal syntax for the SQUARE notation used in this paper is presented in the Appendix. This notation was chosen for expository reasons. However, it is not suited for input technology which requires one-dimensional input. Moreover, an unsophisticated user would feel more comfortable with an English-like notation. Therefore, a block-structured English-keyword syntax, based like SQUARE on the concept of mapping, has been developed. This syntax, called SEQUEL, is described elsewhere [4]. A direct translation of most SQUARE queries to SEQUEL is possible, as illustrated below by several examples. A prototype implementation of the SEQUEL language is now operational at the IBM Research Laboratory in San Jose, California [20].

Q1. (SQUARE)

$_{\text{NAME}}\text{EMP}_{\text{DEPT}}\,(\text{'TOY'})$

Q1. (SEQUEL)

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

Q3. (SQUARE)

$_{\text{ITEM}}\text{SALES}_{\text{DEPT}} \circ {}_{\text{DEPT}}\text{LOC}_{\text{FLOOR}}\,(2)$

Q3. (SEQUEL)

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

Q5. (SQUARE)

$x_{\text{NAME}} \in \text{EMP} :$

$x_{\text{SAL}} > {}_{\text{SAL}}\text{EMP}_{\text{NAME}}\,(x_{\text{MGR}})$

Q5. (SEQUEL)

```
SELECT  NAME
FROM    X IN EMP
WHERE   SAL >
        SELECT  SAL
        FROM    EMP
        WHERE   NAME = X.MGR
```

Clearly it is also possible to "linearize" the basic notation of SQUARE without introducing English keywords. Although we will not treat a linearized mathematical notation in this paper, we present Q3 in such a notation as an example:

SALES[ITEM|DEPT] (LOC[DEPT|FLOOR] (2))

To form a more complete facility for development of database application programs, the SQUARE language might be imbedded in a host programming language. The most compatible host would be a high level, set-oriented language such as APL.

5. Completeness

Traditionally, files are structured to optimize a particular application program. The treelike structures which frequently result have the property that certain queries can be answered only with great difficulty, if at all. In this type of environment as the queries change the data must be restructured, and then previously written queries will no longer work on the new database. A modern database management system should be capable of responding to any new unanticipated query in a reasonably uniform time without requiring a restructuring of the data and without impacting previously written queries. Such a system requires a query language which is able to express any query whose answer is semantically contained in the database. A language possessing this property is said to be "complete."

Codd [8] has proposed a measure for completeness of database sublanguages and has shown that both relational algebra and relational calculus are "relationally complete" according to this measure. In this section we provide a set of algorithms for translating a relational algebra expression into a semantically equivalent SQUARE expression. Consequently, SQUARE is shown to be relationally complete.

A number of operators in relational algebra have direct counterparts in SQUARE. Other algebra operators have direct SQUARE counterparts to their relational calculus definitions as given in Codd [8]. Below we list each operator of relational algebra together with its defining expression in calculus and the equivalent expression in SQUARE. The equivalence is clear in every case except division, which is substantiated by a short proof.

In the expressions below, R and S are relations over the domains $A_1 \ldots A_l$ and $B_1 \ldots B_m$ respectively. A and B represent subsets of $\{A_i\}$ and $\{B_i\}$ respectively, and $\bar{A}$ is the subset complementary to A.

UNION
Algebra: $R \cup S$
Calculus: $\{r : r \in R \vee r \in S\}$
SQUARE: $R \cup S$

INTERSECTION
Algebra: $R \cap S$
Calculus: $\{r : r \in R \wedge r \in S\}$
SQUARE: $R \cap S$

DIFFERENCE
Algebra: $R - S$
Calculus: $\{r : r \in R \wedge r \notin S\}$
SQUARE: $R - S$

PROJECTION
Algebra: $R[A]$
Calculus: $\{r[A] : r \in R\}$
SQUARE: R_A

CARTESIAN PRODUCT
Algebra: $R \otimes S$
Calculus: $\{(r \frown s) : r \in R \wedge s \in S\}$
SQUARE: $r \in R, s \in S$

θ-JOIN where $\theta \in \{=, \neq, >, \geq, <, \leq\}$
Algebra: $R[A\ \theta\ B]S$
Calculus: $\{(r \frown s) : r \in R \wedge s \in S \wedge (r[A] \theta s[B])\}$
SQUARE: $r \in R, s \in S : {}^{r}_{A}\theta\, {}^{s}_{B}$

RESTRICTION
Algebra: $R[A\ \theta\ B]$
Calculus: $\{r : r \in R \wedge (r[A]\ \theta\ r[B])\}$
SQUARE: $r \in R : {}^{r}_{A}\theta\, {}^{r}_{B}$

DIVISION
Algebra: $R[A \div B]S$
Calculus: $\{r[\bar{A}] : r \in R \wedge \{y : (r[\bar{A}], y) \in R\} \supseteq S[B]\}$
SQUARE: $R\ (\ S)$
$\bar{A}\ \underline{A}\ B$

Proof: If we let $S[B] = \{s_1, s_2, \ldots, s_n\}$ then the calculus definition above becomes

$$\{r[\bar{A}] : r \in R \wedge (r[\bar{A}], s_1) \in R \wedge (r[\bar{A}], s_2) \in R \wedge \ldots \wedge (r[\bar{A}], s_n) \in R\}$$

$$= \bigcap_{i=1}^{n} \{r[\bar{A}] : r \in R \wedge (r[A] = s_i)\}$$

$$= \bigcap_{i=1}^{n} {}_{\bar{A}}R\ (s_i) \text{ by definiinition of mapping in SQUARE.}_{A}$$

$$= {}_{\bar{A}}R\ (\ S) \text{ by definition of conjunctive mapping.}_{\underline{A}\ B}$$

(Note. The division operator has been shown to be definable in terms of the other operators of relational algebra, and hence is not strictly necessary for relational completeness [8]).

6. Conclusions

This paper has described SQUARE, a data sublanguage for retrieving, inserting, deleting, and updating data in relational form. The query facilities of SQUARE provide a natural means of expression for the ways in which people use tables to answer questions. SQUARE and its companion language SEQUEL are intended for interactive, ad hoc problem solving by users who are not required to have any knowledge of traditional computer programming. SQUARE exploits the advantages of Codd's relational data model for simplicity, symmetry, and data independence.

The query capabilities of SQUARE have been proven to be relationally complete. Like the relational calculus, SQUARE has the advantage of being a nonprocedural language. However, SQUARE does not require the user to have the mathematical sophistication demanded by languages based on the relational calculus. The user describes the data to be accessed by expressions based on "mappings" rather than by variables and quantifiers. Consequently, SQUARE queries are simpler and more concise than their equivalents in the relational calculus.

Acknowledgments. The authors wish to thank L.Y. Liu, B.M. Leavenworth, E.F. Codd, and P. L. Fehder for their useful discussions.

Received October 1973; revised October 1974

References

1. Boyce, R.F., Chamberlin, D.D., King, W.F., and Hammer, M.M. Specifying queries as relational expressions. Proc. ACM SIGPLAN/SIGIR Interface Meeting, Gaithersburg, Md., Nov. 1973.
2. Boyce, R.F., and Chamberlin, D.D. Using a structured English query language as a data definition facility. Res. Report RJ 1318, IBM Res. Lab., San Jose, Calif., Dec. 1973.
3. Bracchi, G., Fedeli, A., and Paolini, P. A language for a relational data base management system. Proc. Sixth Annual Princeton Conf. Inf. Science and Systems, March 1972, pp. 84–92.
4. Chamberlin, D.D., and Boyce, R.F. SEQUEL: A structured English query language. Proc. 1974 ACM SIGFIDET Workshop, Ann Arbor, Michigan.
5. Childs, D.L. Description of a set-theoretic data structure. Proc. 1968 AFIPS Fall Joint Comp. Conf., pp. 557–564.
6. Codasyl Development Committee. An information algebra. *Comm. ACM 5*, 4 (Apr. 1962), 190–204.
7. Codd, E.F. A relational model of data for large shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
8. Codd, E.F. Relational completeness of data base sublanguages. *Courant Computer Science Symposia, Vol. 6: Data Base Systems.* Prentice-Hall, New York, 1971.

9. Codd, E.F. A data base sublanguage founded on the relational calculus. Proc. 1971 ACM SIGFIDET Workshop, San Diego, Calif., pp 35–68.
10. Codd, E.F. Normalized data base structure: a brief tutorial. Proc. 1971 ACM SIGFIDET Workshop, San Diego, Calif., pp. 1–18.
11. Fehder, P.L. The representation-independent language. Res. Rep. RJ 1121, IBM Research Laboratory, San Jose, Calif., Nov. 1972.
12. Goldstein, R.C., and Strnad, A.L. The MACAIMS data management system. Proc. 1970 ACM SIGFIDET Workshop, Houston, Texas, pp. 201–230.
13. Kuhns, J.L. Logical aspects of question-answering by computer. In *Software Engineering: COINS 2*, J. Tou (Ed.), Academic Press, New York, 1971, pp. 89–104.
14. Kuhns, J.L. Quantification in query systems. Proc. Symp. Inf. Storage and Retrieval, ACM, New York, 1971, pp. 81–94.
15. Levien, R.E., and Maron, M.E. A computer system for inference execution and data retrieval. *Comm. ACM 10*, 11 (Nov. 1967), 715–721.
16. Notley, M.G. The Peterlee IS/1 System. Tech. Rep. UKSC-0018, IBM Scientific Centre, Peterlee, U.K., March 1972.
17. Whitehead, A.N., and Russell, B. *Principia Mathematica.* Cambridge University Press, 1950.
18. Whitney, V.K. A relational data management system. In *Information Systems: COINS IV* J. Tou (Ed.), Plenum Press, New York, 1974, pp. 55–66.
19. Held, G.D., Stonebraker, M.R., and Wong, E. INGRES—a relational data base system. Proc. 1975 AFIPS Nat. Computer Conf., pp. 409–416.
20. Astrahan, M.M., and Chamberlin, D.D. Implementation of a structured English query language. *Comm. ACM 18*, 10 (Oct. 1975), 580–588.

Appendix

The syntax below is intended primarily to give the reader an intuitive understanding of the structure of SQUARE. Some liberties have been taken with the usual BNF notation (e.g. the use of subscripts). The syntax is not intended for use by an automatic parsing algorithm. Because the syntax is simplified for ease of understanding, it does not exclude all semantically meaningless programs; however, all semantically meaningful programs may be expressed in this syntax. The symbols ⟨name⟩, ⟨number⟩, and ⟨string⟩ are considered to be terminal symbols and are not further defined.

⟨transaction⟩ ::= ⟨statement⟩ ;
| ⟨transaction⟩ ⟨statement⟩ ;
⟨statement⟩ ::= ⟨query⟩ | ⟨assignment⟩ | ⟨insertion⟩
| ⟨deletion⟩ | ⟨update⟩
⟨assignment⟩ ::= ⟨rel-name⟩ ← ⟨query⟩
| ⟨rel-name⟩ ⟨col-list⟩ ← ⟨query⟩
⟨insertion⟩ ::= ↓ ⟨rel-name⟩ ⟨col-list⟩ (⟨arg-list⟩)
| ↓ ⟨rel-name⟩ ⟨col-list⟩
⟨deletion⟩ ::= ↑ ⟨rel-name⟩ ⟨col-list⟩ (⟨arg-list⟩)
| ↑ ⟨rel-name⟩
⟨update⟩ ::= → ⟨rel-name⟩ ⟨m-list⟩ ; ⟨u-list⟩ (⟨arg-list⟩)
⟨m-list⟩ ::= ⟨col-list⟩
⟨u-list⟩ ::= ⟨update-col⟩
| ⟨u-list⟩, ⟨update-col⟩
⟨update-col⟩ ::= ⟨col-name⟩
| ⟨col-name⟩ ⟨update-op⟩
⟨update-op⟩ ::= + | − | × | /
⟨col-list⟩ ::= ⟨col-name⟩
| ⟨col-list⟩, ⟨col-name⟩
⟨query⟩ ::= ⟨expression⟩
| ⟨var-list⟩ : ⟨test⟩
| ⟨var-list⟩
⟨var-list⟩ ::= ⟨variable⟩
| ⟨var-list⟩, ⟨variable⟩
⟨variable⟩ ::= ⟨free-variable⟩
| ⟨computed-variable⟩
⟨free-variable⟩ ::= ⟨var-name⟩ ϵ ⟨rel-name⟩
| ⟨var-name⟩ ⟨col-list⟩ ϵ ⟨rel-name⟩
⟨computed-variable⟩ ::= ⟨var-name⟩
⟨test⟩ ::= ⟨boolean⟩
| ⟨test⟩ ∧ ⟨boolean⟩
| ⟨test⟩ ∨ ⟨boolean⟩
| ∼ ⟨boolean⟩
⟨boolean⟩ ::= ⟨expression⟩ ⟨boolean-op⟩ ⟨expression⟩
| (⟨test⟩)

⟨boolean-op⟩ ::= ⟨set-comp⟩
| ⟨arith-comp⟩
| ⟨modifier⟩ ⟨arith-comp⟩
| ⟨arith-comp⟩ ⟨modifier⟩
| ⟨modifier⟩ ⟨arith-comp⟩ ⟨modifier⟩
⟨modifier⟩ ::= ALL | SOME
⟨set-comp⟩ ::= ⊃ | ⊇ | ⊂ | ⊆ | = | ≠
⟨arith-comp⟩ ::= > | ≥ | < | ≤ | = | ≠
⟨expression⟩ ::= ⟨term⟩
| ⟨expression⟩ ⟨operator⟩ ⟨term⟩
⟨operator⟩ ::= + | − | × | / | ∩ | ∪
⟨term⟩ ::= ⟨built-in-fn⟩ (⟨expression⟩)
|⟨var-name⟩ ⟨col-list⟩
| ⟨computed-variable⟩
| ⟨mapping⟩
| ⟨projection⟩
| ⟨constant-term⟩
| ∅
| (⟨expression⟩)
⟨mapping⟩ ::= ⟨map-header⟩ (⟨arg-list⟩)
| ⟨map-header⟩ ∘ ⟨mapping⟩
⟨map-header⟩ ::= ⟨rel-spec⟩ ⟨target-list⟩ ⟨source-list⟩
| ⟨rel-spec⟩ ⟨source-list⟩
⟨projection⟩ ::= ⟨rel-spec⟩ ⟨target-list⟩
⟨rel-spec⟩ ::= ⟨rel-name⟩
| ⟨rel-name⟩′
⟨target-list⟩ ::= ⟨col-list⟩
⟨source-list⟩ ::= ⟨source-col⟩
| ⟨source-list⟩, ⟨source-col⟩
⟨source-col⟩ ::= ⟨col-name⟩
| ⟨col-name⟩
⟨arg-list⟩ ::= ⟨argument⟩
| ⟨arg-list⟩, ⟨argument⟩
⟨argument⟩ ::= ⟨expression⟩
| ⟨arith-comp⟩ ⟨expression⟩
⟨constant-term⟩ ::= ⟨tuple-list⟩
| ⟨tuple⟩ | ⟨constant⟩
⟨tuple-list⟩ ::= [⟨tuple-list-content⟩]
⟨tupe-list-content⟩ ::= ⟨tuple⟩
| ⟨tuple-list-content⟩, ⟨tuple⟩
⟨tuple⟩ ::= < ⟨tuple-content⟩ >
⟨tuple-content⟩ ::= ⟨constant⟩
| ⟨tuple-content⟩, ⟨constant⟩
⟨constant⟩ ::= ⟨number⟩
| ‘ ⟨string⟩ ’
⟨built-in-fn⟩ ::= AVG | MAX | MIN | SUM | COUNT
⟨rel-name⟩ ::= ⟨name⟩
⟨col-name⟩ ::= ⟨name⟩
⟨var-name⟩ ::= ⟨name⟩