JOURNAL OF COMPUTER AND SYSTEM SCIENCES 21, 156–178 (1980)

Computable Queries for Relational Data Bases

ASHOK K. CHANDRA AND DAVID HAREL

IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, N.Y. 10598

Received September 26, 1979; revised March 24, 1980

The concept of "reasonable" queries on relational data bases is investigated. We provide an abstract characterization of the class of queries which are computable, and define the completeness of a query language as the property of being precisely powerful enough to express the queries in this class. This definition is then compared with other proposals for measuring the power of query languages. Our main result is the completeness of a simple programming language which can be thought of as consisting of the relational algebra augmented with the power of iteration.

1. INTRODUCTION

The relational model of data bases, introduced by Codd [7, 8] is attracting increasing attention lately [2–6, 11, 12, 16, 22, 23]. One of the significant virtues of the relational approach is that, besides lending itself readily to investigations of a mathematical nature, its modelling of real data bases is quite honest.

One of the central themes of research in relational data bases is the investigation of query languages. A query language is a well-defined linguistic tool, the expressions of which correspond to requests one might want to make to a data base. With each request, or query, there is associated a response, or answer. For example, in a data base which involves information about the personnel of some company, a reasonable query might be one whose answer is the set of names and addresses of all employees earning over $15,000 a year.

A large portion of the work done on query languages involves the first-order relational calculus (or its closely related relational algebra [2, 3, 5, 8, 12, 22]). Besides the fact that this language resembles conventional predicate calculus and, as such, is known and is easy to comprehend, it seems that one of the reasons for this phenomenon is rooted in the choice made by Codd [8]. There, one version of this calculus is taken as the canonical query language, and any other language having at least its power of expression is said (in [8]) to be *complete*.

It has been shown by Aho and Ullman [4], however, that certain reasonable queries cannot be expressed in first-order relational calculus (in particular, the transitive closure of a binary relation cannot be so expressed). This poses the question of whether there is a natural definition of the set of all reasonable queries. One approach to this is to add various constructs such as transitive closure, fixpoint operators, and iteration [4, 9, 23] to

0022-0000/80/050156–23$02.00/0

query languages and to compare the expressive power of the resulting languages. This approach has the disadvantage that there is no guarantee that one has in fact captured *all* reasonable queries, whatever they might be. The other approach (which we follow) is to define an appropriately large set of queries, and then give query languages that can express all the queries in the set. The term completeness, if adopted, then becomes appropriate; if indeed a reasonable set of queries is chosen, a language expressing the queries in that set could conceivably be called a complete one. Bancilhon [5] favored this approach, questioning the choice of the word complete as referring to the expressive power of first-order relational calculus.

The purpose of this paper is to settle the question of maximal expressiveness of query languages. This is done in two steps. First, the set of computable queries is defined. These correspond to partial recursive functions which satisfy a consistency criterion reflecting the fact that the computation is on a data base rather than, say, on a natural number. Then a query language is demonstrated which expresses precisely all the computable queries.

In Section 2 relational data bases and queries are defined, and the class of queries, which we call *computable*, is identified. This class consists of those queries which (i) when regarded as functions on data bases, are partial recursive and which (ii) preserve isomorphisms between these data bases.

In Section 3 we introduce the general notion of a query language and define the properties of *boundedness*, *expressiveness* and *completeness*. The rest of that section is devoted to the introduction of and the comparison with two other definitions of completeness appearing in the literature, namely those of [8] and [5].

Section 4 contains the definition of our proposed query language QL, and a proof that it is "minimal" in the sense that removing any one of its constructs weakens its power. One can view the language QL as being essentially an iterative programming language in which the right-hand sides of assignment statements are taken to be expressions in a simple version of the relational algebra.

In Section 5 the main result, i.e., the completeness of QL, is proved. The nontrivial part of the proof is in showing that a program in QL can reconstruct a data base up to isomorphism by computing its set of automorphisms, and then, from this set, generate the desired output.

In Section 6 we provide a useful generalization of the simple basic model, and prove the appropriate generalized version of the main result.

2. Data Base Queries

Definition. Let U denote a fixed countable set, called the *universal domain*. Let $D \subset U$ be finite and nonempty, and let $R_1, \ldots, R_k$ for $k > 0$, be relations such that, for all i, $R_i \subset D^{a_i}$. $B = (D, R_1, \ldots, R_k)$ is called a *relational data base of type a* (*or data base* for short), where $a = (a_1, \ldots, a_k)$. R_i is said to be of *rank* a_i; D is called the *domain* of B and is also written $D(B)$.

Our formalism of data omits certain features that are often adopted in the literature. These include data base dependencies and the use of different domains for each column of a relation. However, features such as these can be viewed as merely restricting the allowed data bases, and our results carry over directly to these cases.

DEFINITION. Two data bases of type a, $B = (D, R_1, ..., R_k)$ and $B' = (D', R'_1, ..., R'_k)$ are said to be *isomorphic* by isomorphism h, (or *h-isomorphic*) written $B \leftrightarrow^h B'$, if $h: D \to D'$ is an isomorphism and $h(R_i) = R'_i$ for all i (i.e., h is one–one onto, and for all i, $(x_1, x_2, ..., x_{a_i}) \in R_i$ iff $(h(x_1), h(x_2), ..., h(x_{a_i})) \in R'_i$). An important special case is when $B = B'$. If $B \leftrightarrow^h B$ then h is an *automorphism* on B.

DEFINITION. A *data base query of type a* (or *query* for short) is a partial function giving, for each data base B of type a, an output (if any) which is a relation over $D(B)$. Formally, with $\multimap$ denoting partial,

$$Q: \{B \mid B \text{ is a data base of type } a\} \multimap \bigcup_j 2^{U^j},$$

where, if $Q(B)$ is defined, $Q(B) \subset D(B)^j$ for some j. A query Q is said to be *computable* if Q is partial recursive and satisfies the following *consistency criterion*: if $B \leftrightarrow^h B'$ then $Q(B') = h(Q(B))$, i.e., *Q preserves isomorphisms*.

EXAMPLES. Consider a data base $B = (D, R)$ consisting of a single ternary relation R whose columns correspond to *employee name*, *employee age*, *and manager's name*. The query

the name of the first employee in the data base

is not consistent because the value depends on how the data base is stored. The query

the name(s) of the youngest employee(s)

is also not consistent because the date base does not have a relation giving the total ordering on *employee age*. This second query, however, is a reasonable query to ask, and in Section 6 we extend our notion of consistency to handle queries such as this one. Finally, the query

the names of those managers for whom the number of employees they directly manage, encodes a true statement of first order arithmetic

is consistent but not computable because the set of true statements of arithmetic is not even recursively enumerable.

The set of computable queries satisfies the principles postulated in [4, 5], namely, that the result of a query should be independent of the representation of the data in a data base and should treat the elements of the data base as uninterpreted objects. Also, we will see in Section 3 that our consistency criterion is the appropriate generalization of the condition appearing in [5, 22]. There, the *outcome* of a query Q (which is the subject matter of [5, 22], not the query itself as a function) is to have the property that it cannot distinguish between tuples which are "equivalent" as far as the data base B is concerned. In other words, if $B \leftrightarrow^h B$, then $(d_1, \ldots, d_j) \in Q(B)$ iff $(h(d_1), \ldots, h(d_j)) \in Q(B)$.

Although our constraints on a computable query seem to be necessary for it to be "reasonable" it is not intuitively clear whether or not additional ones are also called for. However, we wish to enforce our belief in the sufficiency of these constraints, and hence to substantiate our argument that the set of computable queries plays a role in relational data bases analogous to the role played by the set of partial recursive functions in the framework of the natural numbers. Accordingly, in Section 4 we define an operationally computable data base query language and show that it expresses precisely the set of computable queries.

3. Query Languages and Their Completeness

Now that we have defined data bases, queries and computable queries, we can turn to the issue of designing languages for expressing queries.

We will think of a query language as consisting of a set L of expressions and a meaning function M, such that for any expression $E \in L$ and for any data base B, the *meaning of E in B*, denoted by $M_E(B)$, is either undefined, or is a relation over $D(B)$.

Throughout, for convenience, we assume that for any data base B each query language has at least one expression E for which $M_E(B)$ is undefined. (This can be achieved, e.g., by letting the meaning of an expression referring to more relations than the data base has, be undefined.)

As examples of query languages one might consider the relational algebra and (first-order) relational calculus of Codd [8]. Using the definitions appearing in [22] and [5], respectively, the following are expressions on these languages:

(1) $((R_1 \cup R_2) \times R_3)_{(2=5)}$,

(2) $(R_1(x_1, x_2) \vee R_2(x_1, x_2)) \wedge R_3(x_3, x_4, x_5) \wedge x_2 = x_5$.

Given a data base $B = (D, R_1, R_2, R_3)$ of type $(2, 2, 3)$, the meaning of both these expressions is the relation consisting of all 5-tuples over D whose first two components form a pair in either R_1 or R_2, whose last three components form a tuple in R_3, and whose second and fifth components are equal (i.e., the same element of D). We do not further define these languages here.

Definition. An expression E in a query language *expresses* the query Q of type a if for each data base of type a, either both $M_E(B)$ and $Q(B)$ are undefined or else $M_E(B) = Q(B)$. We write

$$(\forall B)(M_E(B) \equiv Q(B)),$$

where B is understood to range over data bases of type a.

Definition. A query language is *bounded* if its expressions express only computable queries. It is *expressive* if every computable query is expressed by some expression. A query language is *complete* if it is both bounded and expressive, i.e., it expresses exactly the set of computable queries. These definitions for boundedness and expressiveness can respectively be formalized as

$$\begin{aligned}&(\forall a)\,(\forall E)\,(\exists Q)\,(\forall B)\,(M_E(B) \equiv Q(B)),\\&(\forall a)\,(\forall Q)\,(\exists E)\,(\forall B)\,(M_E(B) \equiv Q(B)),\end{aligned} \qquad (*)$$

where a ranges over types, E ranges over expressions in the language and Q and B respectively range over computable queries and data bases, both of type a.

There have been at least two other definitions of completeness in the literature, both of which turn out to be different from ours, and their expressiveness components turn out to be strictly weaker than ours.

DEFINITION (Codd [8]). A query language is *C-bounded* if it is no more expressive than the relational calculus, it is *C-expressive* if it is at least as expressive as the relational calculus and it is *C-complete* if it is both C-bounded and C-expressive. (Here, language L is at least as expressive as L' if L can express any query that L' can.)

A note on notation. Codd used the term "complete" for our C-expressive and it is in this sense that the term is usually used in the literature. However, there has been some confusion even about the precise formulation of Codd's definition (e.g., the differences between the C-version and the BP-version, see below). We have adopted a slight change in terminology in order to maintain consistency between the terms "expressive," "bounded" and "complete," and because we feel that our definition of completeness is perhaps a more natural one.

Codd [8] showed that the relational algebra is C-expressive. Trivially it is also C-bounded; hence, C-complete. Aho and Ullman [4] have shown, however, that there are computable queries which are not expressible in the relational calculus or algebra. Consider the query *transitive closure* of type $a = (2)$, defined as

$$TC(B) = R^*$$

for $B = (D, R)$, where R is a binary relation over D and R^* is its reflexive and transitive closure.

THEOREM 3.1. (Aho and Ullman [4]). *There is no expression in Codd's relational algebra which expresses the query TC.*

Observing that TC is a computable query we can conclude

COROLLARY 3.2. *Neither the relational algebra nor the relational calculus is expressive (or complete).*

Thus, although an expressive query language is also C-expressive, the converse fails. Likewise, a C-bounded query language is also bounded but not vice versa. See Fig. 1.

In order to introduce the second definition of completeness, taken from Bancilhon [5], we define, for a data base B, the set of relations consistent with it in the sense of the consistency criterion of the previous section. Define

$$I_B = \{R \mid R \subset D(B)^m \text{ for some } m, \text{ and whenever } B \overset{h}{\leftrightarrow} B,\ R = h(R)\}.$$

DEFINITION (Bancilhon [5]; see also Paredaens [22]). A query language is *BP-bounded* if for every data base B and expression E, either $M_E(B)$ is undefined or $M_E(B) \in I_B$. It is *BP-expressive* if for every data base B and for every $R \in I_B$ there are expressions E, E' such that $M_E(B) = R$, and $M_{E'}(B)$ is undefined. A query language is *BP-complete* if it is both BP-bounded and BP-expressive.

Bancilhon [5] and Paredaens [22] proved, respectively, that the relational calculus and relational algebra are BP-complete. (It has been pointed out to us by L. Marcus and these results also follow directly from known facts in model theory. Bancilhon's result, in that framework, states that a relation is first-order definable in a finite structure iff it is invariant under the automorphisms of that structure.) Bancilhon used the word "complete" for our BP-expressiveness. Paredaens, on the other hand, did not use the term completeness, but rather regarded his results as a characterization of the power of the relational algebra to express relations. In order to better see the connection with our own definitions, we show

LEMMA 3.3. *A query language is BP-complete iff, using the notation of* (*),

$$(\forall a)\ (\forall E)\ (\forall B)\ (\exists Q)\ (M_E(B) \equiv Q(B)),$$

and

$$(\forall a)\ (\forall Q)\ (\forall B)\ (\exists E)\ (M_E(B) \equiv Q(B)), \qquad (**)$$

the first line asserting BP-boundedness and the second BP-expressiveness.

Proof. We prove that the second line of (**) asserts BP-expressiveness. The proof of the other claim is similar. Assume L is BP-expressive and let Q and B be a computable query and a data base, respectively, of type a. By the definition of a computable query $Q(B)$ is either undefined or is in I_B. In the former case take E to be an expression such that $M_E(B)$ is undefined, and in the latter take E to be the expression existing by the assumption.

Conversely, assume the second line of (**) and let $R \in I_B$ for some data base B of type a. Define the query Q of type a as

$$\begin{aligned} Q(B') &= h(R) && \text{if } h \text{ is a function such that } B \overset{h}{\leftrightarrow} B', \\ &= \text{undefined} && \text{if } B \text{ and } B' \text{ are non-isomorphic.} \end{aligned}$$

It can be shown that this definition is sound; in particular if $B \leftrightarrow^{h} B'$ and $B \leftrightarrow^{h'} B'$ then $h(R) = h'(R)$. Clearly, Q is a computable query with $R = Q(B)$ and by the second line of (**) there is an E such that $M_E(B) = Q(B) = R$. The existence of E' follows from (**) and the computable query that is undefined on B. ∎

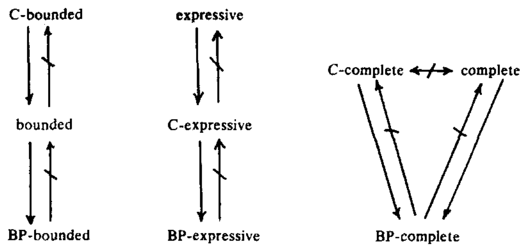

FIGURE 1

Comparing (*) with (**), *BP*-completeness can be seen to be a measure of the power of a language to express relations (as nicely captured by the title of [22]) and *not* of its power to express functions having relations as outputs, i.e., queries.

The notion of *BP*-boundedness is not restrictive enough for queries, in that a query language can contain expressions that are not partial recursive and still be *BP*-bounded. For example, consider the relational calculus augmented with the expression E_0, whose meaning is given by

$$\begin{aligned} M_{E_0}(B) &= \{\} && \text{if the } k\text{th Turing Machine halts on input } k, \text{ where } k = |D(B)|, \\ &= \{()\} && \text{otherwise.} \end{aligned}$$

M_{E_0} is not partial recursive but the language is *BP*-bounded. Also, the notion of *BP*-expressiveness is fairly weak for queries. One can define a query language, each expression of which has the property that its meaning is defined only for data bases of some fixed size. This language will clearly be neither expressive nor *C*-expressive, but can be made to be *BP*-expressive. For example, consider expressions of the form (E, k), where E is an expression of the relational calculus and $k \geqslant 0$, with meaning given by

$$\begin{aligned} M_{(E,k)}(B) &= M_E(B) && \text{if } \ k = |D(B)|, \\ &= \text{undefined} && \text{otherwise.} \end{aligned}$$

Thus, if a query language is bounded it is also *BP*-bounded but not vice versa, and if a query language is expressive it is also *BP*-expressive but not vice versa. See Fig. 1.

Our choice of a stronger notion of expressiveness, and hence of a different notion of completeness, cannot be justified solely on the basis of it apparently being a more natural one for queries (as opposed to relations), but must be accompanied by a feasibly "computable" language which is indeed complete in our sense. In the next section we supply such a language.

4. THE QUERY LANGUAGE QL

The language we define (*QL*) is essentially a programming language computing finite relations over some domain. Its access to a given data base, however, is only through a restricted set of operations: equality, complementation, intersection, a test for emptiness, and simplified versions of projection and cartesian product. The ability to simulate arbitrary Turing machines is an important consequence of this choice. Let us now define QL more formally.

Syntax

$y_1, y_2, \ldots$ are *variables* in QL. The set of *terms* of QL is defined inductively as follows:

(1) E is a term, and for $i \geqslant 1$, rel_i and y_i are terms.

(2) For any terms e and e',

$$(e \cap e'),\ (\neg e),\ (e\downarrow),\ (e\uparrow),\text{ and } (e\sim) \text{ are terms.}$$

The set of *programs* of QL is defined inductively as follows:

(1) $y_i \leftarrow e$ is a program for a term e and $i \geqslant 1$.

(2) For programs P and P'

$$(P; P') \qquad \text{and} \qquad \textit{while } y_i \textit{ do } P$$

are programs.

Semantics

Given a data base $B = (D, R_1, \ldots, R_k)$ of type $a = (a_1, \ldots, a_k)$, terms of QL take on values which are relations over $D = D(B)$. For technical reasons it will be convenient to associate a rank, ***rank***(e), with each term e, forcing us to distinguish between empty relations of various ranks. Thus, if rank$(e) = i$, $i \geqslant 0$, then e is either a nonempty subset of D^i or the empty set of rank i, ϕ^i. For example, there are precisely two relations of rank 0, ϕ^0 and $\{()\}$, the latter being D^0. Denote by EX_i the set of all relations over D of rank i, in the above sense, and let EX_{-1} denote $\{\phi^0\}$.

The term E, equality, is a fixed relation in EX_2 given by $E = \{(d, d) \mid d \in D\}$.

The value of rel$_i$ is given by

$$\begin{aligned} \text{rel}_i &= R_i, && i \leqslant k,\\ &= \phi^0, && i > k \end{aligned}$$

and is thus either of rank a_i or of rank 0.

$\cap$ is a binary operator on relations having the standard value (intersection) when both its arguments are of rank i (ϕ^i if the intersection is empty) and ϕ^0 otherwise. $\neg$, $\downarrow$, $\uparrow$, and $\sim$ are unary operators on relations acting as follows:

$\neg$: $EX_i \rightarrow EX_i$, complementation, is given by

$$\neg e = D^i - e.$$

$\downarrow$: $EX_i \rightarrow EX_{i-1}$, projecting out the first coordinate, is given by

$$e\downarrow = \{(d_2, \ldots, d_i) \mid (d_1, \ldots, d_i) \in e\}.$$

$\uparrow$: $EX_i \to EX_{i+1}$, projecting in on the right, is given by

$$e\uparrow = \{(d_1, \dots, d_i, d) \mid d \in D, (d_1, \dots, d_i) \in e\}.$$

$\sim$: $EX_i \to EX_i$, exchanging the two rightmost coordinates, is given by

$$\begin{aligned} e\sim &= \{(d_1, \dots, d_{i-2}, d_i, d_{i-1}) \mid (d_1, \dots, d_i) \in e\}, & i > 1, \\ &= e, & i \leqslant 1. \end{aligned}$$

Note the following: $\neg(D^i) = \phi^i$, $e\downarrow = \{()\}$ if rank$(e) = 1$ and $e \neq \phi^1$, $(\phi^{i+1})\downarrow = \phi^i$, $\phi^0\downarrow = \phi^0$, $\phi^i\uparrow = \phi^{i+1}$, and $\{()\}\uparrow = E\downarrow = D$.

Programs in QL act in the obvious way; all variables are initialized to ϕ^0, and the test y_i in the *while do* construct is true iff the value of y_i is empty, i.e., ϕ^j for some j.

Given a program P and a data base B, the *value of P in B*, $M_P(B)$, or $P(B)$ for convenience, is undefined if P does not terminate, and is otherwise defined to be the value of the variable y_1 upon termination (if y_1 has the value ϕ^i then the output is the empty set; the rank of empty sets is irrelevant as far as the output behavior of queries and programs is concerned). Given a query Q of type a, we know from Section 3 what it means for a program P to *express* Q. P expresses Q (we will also say *computes* Q) if for every data base of type a, $P(B) \equiv Q(B)$, i.e., either $P(B)$ and $Q(B)$ are both undefined or $P(B) = Q(B)$.

Our main result, to be proved in the next section is

THEOREM 4.1. *QL is complete.*

For the proof we will need the fact that several conventional operations on relations are expressible in QL. Observe first that in effect we have *counters*. $E\downarrow\downarrow$, which is $\{()\}$, plays the role of 0, and if e plays the role of the natural number i then $e\uparrow$ and $e\downarrow$ play the roles of $i + 1$ and $i - 1$, respectively. (Counters need never attempt to substract 1 from 0.) Testing whether e is "equal" to 0 is accomplished by testing $e\downarrow$ for emptiness. Note that this gives QL the power of general Turing machines (cf. [18]). Hence in the sequel, we will use n, m..., in programs to denote natural numbers, and will freely use Turing machine terminology.

An *if y_i then P else P'* construct can be simulated by the following program (assuming that y_2 and y_3 do not appear in P or P').

```
$y_2 \leftarrow y_i$;    $y_3 \leftarrow E\downarrow\downarrow\downarrow$;
while $y_2$ do($P$; $y_2 \leftarrow E$; $y_3 \leftarrow E$);
while $y_3$ do ($P'$; $y_3 \leftarrow E$).
```

The reader should also be convinced that we can simulate the test "non-empty" using the *if* and *while* constructs. Denote "y_i non-empty" by $\bar{y}_i$.

Now we show how to compute, in n, the value *rank*(e):

```
if $e$ then $y_1 \leftarrow \neg e$ else $y_1 \leftarrow e$;
$n \leftarrow 0$; while $\bar{y}_1$ do ($y_1 \leftarrow y_1\downarrow$; $n \leftarrow n + 1$);
$n \leftarrow n - 1$.
```

The following abbreviation is useful. Define $e(\uparrow\sim)^n$ to be the value of y_2 in the program

```
$y_2 \leftarrow e$;
*while* $n \neq 0$ *do* $(y_2 \leftarrow ((y_2\uparrow)\sim);\ n \leftarrow n - 1)$
```

in other words $e(\uparrow\sim)^n$ is e with n columns projected in to the left of the rightmost column. (We will use similar notations for single connectives, e.g. $e(\uparrow^n)$.)

Now, we can project in on the left of a relation, an operation we denote by $\uparrow e$:

```
$y_1 \leftarrow e\uparrow$;      $n \leftarrow rank(e)$;
*while* $n \neq 0$ *do*   $(y_2 \leftarrow E(\uparrow\sim)^{rank(e)}$;
                    $y_1 \leftarrow (y_1\uparrow \cap y_2)\downarrow$;
                    $n \leftarrow n - 1)$.
```

It may be checked that this program generates the desired result. We can now compute the *cartesian product* of two relations, i.e.,

$$e_1 \times e_2$$
$$= \{(d_1, \ldots, d_{rank(e_1)}, b_1, \ldots, b_{rank(e_2)}) \mid (d_1, \ldots, d_{rank(e_1)}) \in e_1 \text{ and } (b_1, \ldots, b_{rank(e_2)}) \in e_2\}$$

This is done by

$$y_1 \leftarrow (e_1(\uparrow^{rank(e_2)}) \cap (\uparrow^{rank(e_1)})\, e_2).$$

It should be clear that one can also compute the *generalized projection of e*,

$$e_{[i_1,\ldots,i_p]} = \{(d_{i_1}, \ldots, d_{i_p}) \mid (d_1, \ldots, d_m) \in e\},$$

where for all $1 \leqslant j \leqslant p$, $i_j \leqslant rank(e) = m$. In order to do this, observe that,

$$e_{[i_1,\ldots,i_p]} = ((e \times D^p) \cap \bigcap_{1 \leqslant j \leqslant p} (D^{i_j - 1} \times E(\uparrow\sim)^{rank(e) - i_j + j - 1} \times D^{p-j}))(\downarrow^{rank(e)}).$$

As an example of a naturally arising query, consider again the transitive closure. The following program, in which $\neg(\neg e_1 \cap \neg e_2)$ is abbreviated by $e_1 \cup e_2$, computes TC:

```
$y_1 \leftarrow E$;    $y_2 \leftarrow rel_1$;    $y_3 \leftarrow E \cup y_2$;    $y_4 \leftarrow \neg E \cap y_2$;
*while* $\bar{y}_4$ *do*  $(y_1 \leftarrow y_3$;
  $y_3 \leftarrow y_3 \cup (((y_3 \times y_2) \cap \uparrow E\uparrow)_{[1,4]})$;
  $y_4 \leftarrow \neg y_1 \cap y_3)$.
```

It is clear that no power is lost if, in QL, the $\uparrow$, $\downarrow$, and $\sim$ operators are replaced by cartesian product and generalized projections. Thus, the reader might want to regard our language QL as being, in a sense, the "closure" of the relational algebra [8] under sequencing and iteration, with the ability to test for emptiness. R. Parikh has pointed out to us that a language as powerful as ours can be obtained in a similar fashion from the relational calculus by "closing" it under a certain kind of infinitary quantification. This remark, upon which we do not further dwell here is reminiscent of the use of infinitary logic to reason about dynamic situations such as those involved in proving the correctness of programs. See [15].

We now argue that in a sense our language is minimal, i.e., that no constructs of QL can be eliminated without weakening its power. As we have remarked, QL without the *while* statements has no more power than the relational algebra, and hence, by Corollary 3.2 is weaker than QL. Also it is clear that without assignments the language can only compute the empty set. We leave to the reader the task of showing that the composition operator on programs, ";", cannot be eliminated, as is the case with the equality relation E. We now show that none of the operators on terms can be eliminated.

For $\delta \in \{\cap, \neg, \downarrow, \uparrow, \sim\}$, let QL_δ be the language obtained by omitting, from the definition of the terms of QL, the clause corresponding to δ. Let "$<$" stand for "strictly weaker in expressive power."

LEMMA 4.1. *For every* $\delta \in \{\cap, \neg, \downarrow, \uparrow, \sim\}$, $QL_\delta < QL$.

Proof. For each such δ we supply a data base B_δ and a relation R_δ, defining the query Q_δ as

$$\begin{aligned} Q_\delta(B) &= h(R_\delta) && \text{if } B_\delta \overset{h}{\leftrightarrow} B, \\ &= \text{undefined} && \text{if } B_\delta \text{ and } B \text{ are non-isomorphic.} \end{aligned}$$

From our choice of the B_δ and R_δ it will become clear that the Q_δ are computable queries. The claim is that for each δ, there is no program in QL_δ which computes Q_δ. First, though, note that it is sufficient to prove that no term in QL_δ has value R_δ when interpreted in B_δ. The reason for this is that if a program P in QL_δ computes Q_δ, then $P(B_\delta)$ is defined and by unravelling the *while* loops sufficiently, the "execution sequence" of P in B_δ can be found and collapsed into one assignment of the form $y_1 \leftarrow e$, for some (large) term e, with the value of e being the value of y_1 upon termination, i.e. R_δ.

We supply a proof that no term in QL_δ has value R_δ in B_δ only for $\delta = \sim$. The other cases are either trivial or similar. The B_δ's are defined as

(i) $B_\cap = (\{a, b\}, \{(b)\})$; $R_\cap = \{(b, b)\}$.

In other words, $B_\cap$ is a data base with domain $\{a, b\}$ and the single-tuple unary relation $\{(b)\}$. The intuition here is that without intersecting with E, there is no way of "doubling" the element b.

(ii) $B_\neg = (\{a, b\})$; $R_\neg = \{(a, b), (b, a)\}$.

$B_\neg$ has no relations (or to fit into the strict definition of a data base $(D, R_1, \dots, R_k)$ with $k > 0$ we would let $R_1 = E$ be its single relation). The intuition is that complementation is the only way to get rid of the equality pairs (b, b) and (a, a).

(iii) $B_\downarrow = (\{a\})$; $R_\downarrow = \{(a)\}$.

$B_\downarrow$ is the trivial data base with a 1-element domain and no relations. The intuition is that no unary relation is expressible without $\downarrow$.

(iv) $B_\uparrow = (\{a\})$; $R_\uparrow = \{(a, a, a)\}$.

Similarly, without $\uparrow$ no relation of rank 3 is expressible.

(v) $B_\sim = (\{a, b\})$; $R_\sim = \{(a, b, a), (b, a, b), (a, a, a), (b, b, b)\}$.

$R_\sim$ is in fact (in QL) $E\uparrow\sim$. We have to prove that no term in $QL_\sim$ has value $R_\sim$. Define

$$A = \{\phi^i, D^i\}_{i=0}^{\infty} \cup \{E \times D^i, \neg(E \times D^i)\}_{i=0}^{\infty}.$$

First, note that the only relations of rank 3 in A are ϕ^3, D^3, $E \times D$, and $\neg(E \times D)$, none of which is equal to $R_\sim$. Thus, $R_\sim \in A$. However, we now show that the value (in $B_\sim$) of any term e in $QL_\sim$, is in A. This is done by induction on the structure of e: E and rel_i, for any i, are in A. A is clearly closed under complementation. Also, $D^{i+1}\downarrow = D^i$, $D^i\uparrow = D^{i+1}$, $(E \times D^i)\downarrow = D^{i+1}$, $(E \times D^i)\uparrow = E \times D^{i+1}$, and similarly for their complements. Thus A is also closed under $\uparrow$ and $\downarrow$. Finally, if $e = e' \cap e''$, then e' and e'' are either of different ranks, in which case $e = \phi^0 \in A$, or else the intersection is trivial in the sense that e is either one of e' or e'', or else is ϕ^i. Thus, A is closed under intersection, and the claim is proved. ∎

To summarize, we can extend our notation of QL_δ to include assignments, composition, while statements, etc., obtaining

THEOREM 4.2. *For every* $\delta \in \{E, rel_i, y_i, ;, \leftarrow, while, \cap, \neg, \downarrow, \uparrow, \sim\}$, $QL_\delta < QL$.

We refer the reader at this point to Aho and Ullman [4], where it is suggested that the deficiency of the relational algebra expressed in Theorem 3.1 be remedied by augmenting that language with a least-fixpoint operator on monotonic functionals over relations. The transitive closure TC can then be expressed. We show in [25] that such a language is not complete. Another point raised in [4] is that of allowing some predicates and constants over the domain to be fixed by the isomorphisms of a data base when considering a candidate query for consistency. This issue is dealt with in an extension to our basic notion of a data base, in Section 6.

We now turn to the proof of Theorem 4.1.

5. PROOF OF THEOREM 4.1

It is straightforward to show that QL is bounded: given a program P and query Q of type a such that P computes Q, it is obvious that Q is partial recursive. Furthermore, to see that Q preserves isomorphisms, consider the simultaneous behaviors of P on two h-isomorphic data bases B and B' of type a. One can easily show that all expressions of QL preserve isomorphism (an isomorphism on empty sets preserves their rank). For example, if e_1 and e_2 are, respectively, h-isomorphic to e_1' and e_2', then $e_1 \cap e_2$ is h-isomorphic to $e_1' \cap e_2'$. Also, if e and e' are h-isomorphic then e is empty iff e' is. Hence, tests evaluate to the same truth values in both computations and it is clear, therefore, that $P(B)$ is defined iff $P(B')$ is, and that if both are defined then $P(B)$ and $P(B')$ are h-isomorphic.[1]

Turning to the other direction, i.e., expressiveness, let Q be a computable query of type $a = (a_1, \ldots, a_k)$. We will describe the construction of a program P_Q such that P_Q computes Q. The computation of P_Q, given an input data base B of type a, will consist of the following four main steps:

(1) Compute the set of automorphisms of B.

(2) Compute an internal, "model" data base B_N isomorphic to B.

(3) Compute $Q(B_N)$ using the Turing machine capability.

(4) Compute $Q(B)$ from $Q(B_N)$ using the set of automorphisms computed in step 1.

In order to be able to spell out this process more precisely and show how to program it in QL, we will need some additional notation. Let $B = (D, R_1, \ldots, R_k)$ be a data base of type a. Let $n = |D|$, and denote by *perm*(D) the n-ary relation over D consisting of all permutations of the n elements of D. Assume, without loss of generality, that $\{1, 2, 3, \ldots\} \subset U$.

Now, let $d = (d_1, \ldots, d_n)$ be some tuple of *perm*(D), i.e., d is some ordering of D. For $R \subset D^r$ denote by R/d the *index set*

$$\{(i_1, \ldots, i_r) \mid (d_{i_1}, \ldots, d_{i_r}) \in R\}.$$

We note that two different elements, d and d', of *perm*(D) may give rise to the same index set. Accordingly, define $d \sim_R d'$ iff $R/d = R/d'$. It is clear that $\sim_R$ is an equivalence relation. The equivalence class of d with respect to $\sim_R$ will be called (following [22]) the *cogroup of R via d*

$$\begin{aligned} CG_d(R) &= \{d' \mid d' \in perm(D) \wedge d \sim_R d'\}, \\ &= \{(d_{\alpha(1)}, \ldots, d_{\alpha(n)}) \mid \alpha \text{ is a permutation of } \{1, \ldots, n\} \text{ and} \\ &\qquad R = \{(d_{\alpha(i_1)}, \ldots, d_{\alpha(i_r)}) \mid (d_{i_1}, \ldots, d_{i_r}) \in R\}\}. \end{aligned}$$

Observe that $CG_d(R)/d$ gives the indices corresponding to the permutations of D which preserve R. Also, note that $d \in CG_d(R)$.

EXAMPLE. Let $d = (d_1, d_2, d_3, d_4)$ and $R = \{(d_1, d_2), (d_2, d_1), (d_3, d_3), (d_4, d_4)\}$. Then $R/d = \{(1, 2), (2, 1), (3, 3), (4, 4)\}$ and $CG_d(R)/d = \{(1, 2, 3, 4), (2, 1, 3, 4), (1, 2, 4, 3), (2, 1, 4, 3)\}$.

Now, for our data base $B = (D, R_1, \ldots, R_k)$, let $CG_d{}^B$ abbreviate $\bigcap_{1 \leqslant i \leqslant k} CG_d(R_i)$. Certainly $CG_d{}^B \subset perm(D)$, and $CG_d{}^B/d$ can be thought of as representing the set of automorphisms of D relative to the ordering d, which preserve the relations of B. Here too, note that $d \in CG_d{}^B$.

[1] This argument is analogous to Theorem 1 of [5] and Lemma 2 of [22] in which, respectively, the constructs of Codd's [8] relational calculus and algebra were shown to preserve automorphisms.

We now give a more precise description of the four steps of the computation of P_Q on input B (describing how to program these steps in QL will be done below).

(1) Compute CG_d^B for some $d \in perm(D)$. (CG_d^B is an n-ary relation over D.)

(2) Compute and "store on tape" the data base

$$B_N = (\{1, 2, \ldots, n\}, R_1/d, \ldots, R_k/d)$$

(Each R_i/d is an a_i-ary relation over $\{1, 2, \ldots, n\}$.)

(3) Compute, using the Turing machine capability, the value $Q(B_N)$ of the given function Q applied to the argument B_N. ($Q(B_N)$ is, say, an m-ary relation over $\{1, 2, \ldots, n\}$.)

(4) Compute (in y_1)

$$S = \bigcup_{(j_1, \ldots, j_m) \in Q(B_N)} (CG_d^B)_{[j_1, \ldots, j_m]} .$$

(S is an m-ary relation over D.)

Step 3 makes the execution of P_Q depend on the given computable query Q. The fact that Q is partial recursive is what enables the "Turing machine part" of QL to carry out this step, and the fact that Q preserves isomorphisms will be essential in establishing that $S = Q(B)$. Note that if $Q(B_N)$ is undefined the Turing machine will not halt and $P_Q(B)$ will be undefined too.

LEMMA 5.1. *$Q(B) \subset S$, where S is as described above.*

Proof. We will show that in fact $Q(B)$ corresponds to a very "small" part of S, namely that part obtained by replacing the relation CG_d^B in the definition of S by the singleton $\{(d_1, \ldots, d_n)\}$, a subrelation of CG_d^B. Indeed, we now show that

$$Q(B) = \{(d_{j_1}, \ldots, d_{j_m}) \mid (j_1, \ldots, j_m) \in Q(B_N)\}.$$

First, observe that $B_N \leftrightarrow^h B$, where for $1 \leqslant i \leqslant n$, $h(i) = d_i$. This follows immediately from the definition of R/d. Hence since Q is a computable query, we must have $Q(B_N) \leftrightarrow^h Q(B)$, or $h(Q(B_N)) = Q(B)$, which is precisely what was required. ∎

LEMMA 5.2. *$S \subset Q(B)$, where S is as described above.*

Proof. Let $s = (s_1, \ldots, s_m) \in S$. Then there is $(j_1, \ldots, j_m) \in Q(B_N)$ and $(d_{\alpha(1)}, \ldots, d_{\alpha(n)}) \in CG_d^B$, such that for $1 \leqslant i \leqslant m$, $s_i = d_{\alpha(j_i)}$. We show that $(d_{\alpha(j_1)}, \ldots, d_{\alpha(j_m)}) \in Q(B)$. Note that, by definition of CG_d^B, $B \leftrightarrow^{\alpha'} B$, where $\alpha'(d_i) = d_{\alpha(i)}$. It follows that $\alpha'(Q(B)) = Q(B)$ or that $(d_{j_1}, \ldots, d_{j_m}) \in Q(B)$ iff $(d_{\alpha(j_1)}, \ldots, d_{\alpha(j_m)}) \in Q(B)$. But $(j_1, \ldots, j_m)$ being in $Q(B_N)$ by assumption, implies $(d_{j_1}, \ldots, d_{j_m}) \in Q(B)$ by the characterization of $Q(B)$ in the proof of Lemma 5.1. ∎

Hence we have established that the above four steps, if executed, correctly compute $Q(B)$. We now set out to show how (1)–(4) can be programmed in QL.

We first show how to compute *perm*(D) in some variable, say y_2, and simultaneously compute $n = |D|$ in a "numerical" variable n. For any expression e and $1 \leqslant i < j \leqslant$ *rank*(e), denote by $e_{(i \neq j)}$ the expression

$$e \cap \neg(D^{i-1} \times E(\uparrow\sim)^{(j-i-1)} \times D^{rank(e)-j}).$$

Thus, $e_{(i \neq j)}$ contains all tuples in e for which the elements in the ith and jth positions are unequal. (The same expression, but without the "$\neg$", is denoted $e_{(i=j)}$). Denote by $e_{(\neq)}$ the relation obtained by executing

```
$y \leftarrow e$;
for all $1 \leqslant i < j \leqslant rank(e)$ do
$y \leftarrow y_{(i \neq j)}$,
```

where y is a suitable new variable. Certainly this is programmable in QL. Now *perm*(D) and n are calculated by

```
$n \leftarrow 0; y_2 \leftarrow E\downarrow$;
while $\bar{y}_2$ do $(y_3 \leftarrow y_2\uparrow; y_2 \leftarrow y_{3(\neq)}; n \leftarrow n + 1); y_2 \leftarrow y_3\downarrow$.
```

We now show how to calculate $CG_d{}^B$ in QL, for some $d \in$ *perm*(D). Let $N = \{1, 2,..., n\}$. Consider the function $\psi(V, r, R, X)$, where $V \subset$ *perm*(D), $R \subset D^r$ and $X \subset N^r$, defined as follows:

$$\begin{aligned} \psi(V, r, R, X) &= \phi^r && \text{if } \forall d \in V,\ X \neq R/d, \\ &= CG_d(R) \cap V && \text{if } X = R/d,\ d \in V. \end{aligned}$$

Assume for the moment that we can compute ψ. The way in which $CG_d{}^B$ is computed, for some $d \in$ *perm*(D), is by utilizing the Turing machine power of QL to cycle through all possible choices of a set $\{X_1 ,..., X_k\}$ where, for each i, $X_i \subset N^{a_i}$. For each such choice the following program is executed:

```
$y_3 \leftarrow perm(D)$;
for all $1 \leqslant i \leqslant k$ do $y_3 \leftarrow \psi(y_3, a_i, R_i, X_i)$
```

and upon its completion y_3 is tested for emptiness. It is easy to see that y_3 is nonempty (i.e., $\forall j$, $y_3 \neq \phi^j$) iff for some $d \in$ *perm*(D), $X_i = R_i/d$ for every $1 \leqslant i \leqslant k$. In fact, y_3 will then have the value $(\cdots((perm(D) \cap CG_d(R_1)) \cap CG_d(R_2)) \cap \cdots \cap CG_d(R_k)) = CG_d{}^B$. Moreover, cycling through all possibilities of $\{X_1 ,..., X_k\}$ must result in our falling upon a nonempty y_3. Note that the "successful" set $\{X_1 ,..., X_k\}$ is that required in step (2) of the computation of P_Q, so that it can be essentially stored on tape and used for step (3).

Turning now to ψ, given V and R as relations, we show that the following program computes $\psi(V, r, R, X)$ in y_3. (As earlier, the reader should convince himself that (***) can be programmed in QL.)

```
$y_3 \leftarrow V$;
for all $(i_1, ..., i_r) \in N^r$ do
    (if $(i_1, ..., i_r) \in X$ then $y_4 \leftarrow R \times y_3$                (***)
      else $y_4 \leftarrow \neg R \times y_3$;
    for all $1 \leqslant j \leqslant r$ do $y_4 \leftarrow y_{4(j=r+i_j)}$;
    $y_3 \leftarrow y_4(\uparrow^r)$).
```

For each element of X (respectively of $\neg X$), (***) eliminates from V all permutations with which no tuple of R (respectively of $\neg R$) is consistent. Denote by T the final value of y_3 in (***). Noting that $T \subset V$, we now prove the following two lemmas which serve to establish the validity of (***):

LEMMA 5.3. *If $X \neq R/d$ for every $d \in V$, then $T = \phi^r$.*

Proof. Let $d = (d_1, ..., d_n) \in T$. We show that $X = R/d$. Let $(i_1, ..., i_r) \in X$. We have to show $(i_1, ..., i_r) \in R/d$, or equivalently $(d_{i_1}, ..., d_{i_r}) \in R$. In order to be in T, d had to "survive" each execution of the body of the main loop of (***). In particular, d had to be left in y_4, concatenated with come element $(d_{\alpha(1)}, ..., d_{\alpha(r)})$ of R, and such that for all $1 \leqslant j \leqslant r$, $d_{\alpha(j)} = d_{i_j}$. But this implies $(d_{i_1}, ..., d_{i_r}) \in R$.

Conversely, let $(i_1, ..., i_r) \in R/d$, or $(d_{i_1}, ..., d_{i_r}) \in R$. Using a similar argument, if $(i_1, ..., i_r) \notin X$, then d would have survived the inner loop of (***) with the given $(i_1, ..., i_r)$, from which it would follow that $(d_{i_1}, ..., d_{i_r}) \notin R$. ▌

LEMMA 5.4. *If $X = R/d$ and $d \in V$ then $T = CG_d(R) \cap V$.*

Proof. Assuming that $X = R/d$ for some $d = (d_1, ..., d_n) \in V$, we first let $d' = (d_{\alpha(1)}, ..., d_{\alpha(n)}) \in T$ and show that $d \sim_R d'$. By our assumption we need only show that $X = R/d'$. Indeed, if $(i_1, ..., i_r) \in X$ then the appropriate inner loop of (***) with $(i_1, ..., i_r)$ would have eliminated d' from y_4 if it were not the case that $(d_{\alpha(i_1)}, ..., d_{\alpha(i_r)}) \in R$. But this implies that $(i_1, ..., i_r) \in R/d'$. Conversely, if $(i_1, ..., i_r) \in R/d'$ then $(d_{\alpha(i_1)}, ..., d_{\alpha(i_r)}) \in R$, and similarly, if $(i_1, ..., i_r) \notin X$ then we would have eliminated d' in the inner loop of (***) with $(i_1, ..., i_r)$.

For the other direction, let $R/d' = X$. We have to show that $d' \in T$. The reader should be able to use arguments similar to the previous ones in order to show that if d' was eliminated in a "positive" inner loop, i.e., where $(i_1, ..., i_r) \in X$, then $(i_1, ..., i_r) \notin R/d'$, and if d' was eliminated in a "negative" one, i.e., where $(i_1, ..., i_r) \notin X$, then $(i_1, ..., i_r) \in R/d'$, in both cases a contradiction to $X = R/d$. ▌

To complete the proof of Theorem 4.1, note that S of step (4) in the computation of P_Q is easily programmed in QL using the computed $CG_d{}^B$ and the program described earlier for the generalized projection operator.

6. The Extended Query Language EQL

When relational data bases are used in practice, several operations outside the formal relational framework are useful. Consider the query "sum the salaries of all employees." Answering this query requires the ability to recognize numbers in the data base, to add, and to produce a number as output. Or consider the query "what is the length of the longest name of a department." Answering this query requires a length operator on strings. The problem with these additional operations is that their results can be in a potentially infinite domain. We abstract the essence of these additional operations to produce the set of extended queries as follows.

In addition to the universal domain U, there is another countable, enumerable domain $F = \{f_0, f_1, f_2, ...\}$, where $F \cap U = \phi$. F is intended to include interpreted features such as numbers, strings (if needed), etc. An *extended data base* $B = (D, R_1, ..., R_k, S_1, ..., S_m)$ has a finite domain $D \subset U$, finite relations R_i on $D \cup F$, and operations S_i: $D^{b_i} \rightarrow F$ which serve to connect the "uninterpreted" domain D to the interpreted domain F. Thus the requirement $F \cap U = \phi$ is not restrictive since if overlap is needed, F could contain a "copy" which is obtained by applying an S_i of rank 1 performing the "identity" operation. The *rank* a_i of relation R_i is (not a natural number but) a finite 0, 1 sequence, with $R_i \subset Z_{a_i}$, where Z_{a_i} is defined recursively as follows

$$Z_\lambda = \{()\}, \qquad Z_{0c} = D \times Z_c, \qquad Z_{1c} = F \times Z_c.$$

The *type* of B is $(a_1, ..., a_k, b_1, ..., b_m)$. It should be noted that the operations S_i are not really necessary in this formalism since they can simply be treated as relations R_i (see also comment on functional dependencies in Section 7).

Two extended data bases $B = (D, R_1, ..., S_1, ...)$ and $B' = (D', R_1', ..., S_1', ...)$ of the same type are said to be *isomorphic* by isomorphism h (or *h-isomorphic*, $B \leftrightarrow^h B'$) if h: $D \rightarrow D'$ is an isomorphism and for all i, $h(R_i) = R_i'$ (where h is extended to be the identity function on F) and $h(S_i) = S_i'$ (where S_i is treated as a relation for purposes of applying h).

Let Y_c be defined as was Z_c but replacing D by (the universal domain) U. An *extended data base query of type a* (*extended query* for short) is a partial function

$$Q: \{B \mid B \text{ is a data base of type } a\} \multimap \bigcup_c 2^{Y_c}$$

where, if $Q(B)$ is defined, $Q(B) \subset Z_c$ for some c and $Q(B)$ is finite. An extended query is said to be *computable* if it is partial recursive and satisfies the *consistency criterion*: if $B \leftrightarrow^h B'$ then $Q(B') = h(Q(B))$.

Examples. The query

sum the salaries of all employees

can be modeled as follows. $B = (D, R)$ where D is the set of employee names, $F = \{0, 1, 2, ...\}$, and $R \subset D \times F$ is of rank 01 and associates salaries with names. The desired query has output $\{\sum_{(x,i)\in R} i\}$ and is a computable extended query. The same query could also have been modeled by a data base $B = (D, R, S)$, where D is the set of employee names and salaries (tagged to make them disjoint from F), $F = \{0, 1, 2, \ldots\}$, $R \subset D \times D$ is of rank 00 and associates salaries with names, and $S\colon D \to F$ maps salaries in D to the corresponding values in F. The desired query has output $\{\sum_{(x,i)\in R} S(i)\}$ and is a computable extended query. In this data base, the query

output the names of people who make the highest salary

has output $\{x \mid \exists i.\, (x, i) \in R \wedge S(i) = \mathrm{Max}\{S(j) \mid (x, j) \in R\}\}$ and is also a computable extended query. An example in which S is not used merely for providing "copies" of elements of D, is the query

length of the longest name of a department

in which $S\colon D \to F$ might associate with each department name, viewed as a string of characters, its length. All other elements in the domain would be mapped to a special element in F which may be called the "undefined" element.

We define the extended query language (EQL) which contains the constructs of QL:

$$E,\ rel_i,\ y_i,\ \cap,\ \neg,\ \downarrow,\ \uparrow,\ \sim,\ \leftarrow,\ ;,\ \textit{while}.$$

In addition, terms in EQL can also be of the forms:

$S_i(y_j)$

f_{y_i}

$(e_1 \times e_2)$, where e_1, and e_2 are terms

$(e_1 \cup e_2)$

and $(e_1 - e_2)$.

The semantics of EQL is the appropriate extension to that of QL. Values of variables have ranks in $\{0, 1\}^*$; variables are initialized to ϕ^λ; $\neg e$ has value $Z_c - e$, where e has rank c; the value of $e_1 \cap e_2$ is the set intersection of the values of e_1 and e_2 if they have the same rank, otherwise it is ϕ^λ; $\downarrow$ as before projects out the first coordinate, and $\phi^\lambda\!\downarrow\, = \phi^\lambda$; $\uparrow$ maps Z_c to Z_{c0}, and projects in D to the right; and $\sim$ interchanges the two rightmost coordinates (the operators $\neg$ and $\uparrow$ are redundant in EQL). The new term f_{y_i} has value $\{(f_k)\}$ if y_i has rank 0^k, and has value ϕ^λ otherwise. $S_i(y_j)$ has value $\{S_i(x_1, \ldots, x_m) \mid (x_1, \ldots, x_m) \in y_j\}$ if y_j has rank 0^m and $b_i = m$, and has value ϕ^λ otherwise. The terms $e_1 \times e_2$, $e_1 \cup e_2$ and $e_1 - e_2$ are respectively, cartesean product (function $Y_c \times Y_d \to Y_{cd}$), set union and set difference (for $\cup$ and $-$, if e_1 and e_2 do not have the same rank, the value is ϕ^λ). Programs $y_i \leftarrow e$, $(P; P')$, and *while* y_i *do* P have the obvious semantics.

THEOREM 6.1. *EQL is complete in the extended sense, i.e., the set of queries computed by programs in EQL is precisely the set of extended computable queries.*

Outline of Proof. As in the proof of Theorem 4.1, it may be seen that *EQL* computes only extended computable queries: if program P computes query Q, then Q is partial recursive, and by considering simultaneous executions of P on two h-isomorphic data bases it is seen that the outputs, if any, are also h-isomorphic (recall that isomorphisms induce the identity function on the interpreted domain F).

The proof for the other direction is also quite similar to the proof of Theorem 4.1. Let $\{1, 2, 3,...\} \subset U$. For $d = (d_1,..., d_n)$ in *perm*(D) and R of rank $c = c_1c_2...c_r$ let

$$R/d = \{(i_1,..., i_r) \mid \exists(g_1,..., g_r) \in R.\ \forall k.\ \text{if } c_k = 0 \text{ then } g_k = d_{i_k} \text{ else } i_k = g_k\}.$$

Note that R/d has the same rank as R. Given $B = (D, R_1,..., R_k, S_1,..., S_m)$, S_i/d is defined as for R_i/d (treating S_i as a relation) and is a function of the same rank as S_i; $d \sim_R d'$ iff $R/d = R/d'$, and $CG_d(R)$ is the equivalence class of d with respect to $\sim_R$ (and likewise for an S_i). Also, let

$$CG_d^B = \bigcap_{1 \leqslant i \leqslant k} CG_d(R_i) \cap \bigcap_{1 \leqslant i \leqslant m} CG_d(S_i).$$

Given a computable query Q, the program P_Q in *EQL* computes Q as follows:

(1) Compute CG_d^B for some $d \in perm(D)$.

(2) Compute the data base ($n = |D|$)

$$B_N = (\{1, 2,..., n\}, R_1/d,..., S_1/d,...).$$

(3) Compute, using the Turing machine capability of *EQL*, the value $Q(B_N)$.

Let the rank of $Q(B_N)$ be $c = c_1,..., c_r$.

(4) Compute (in y_1)

$$S = \bigcup_{(j_1,...,j_m) \in Q(B_N)} (CG_d^B \times \{(g_1, g_2,..., g_r)\})_{[q_1,...,q_r]},$$

where if $c_i = 0$ then $q_i = j_i$ and $g_i = f_0$, otherwise $q_i = n + j_i$ and $g_i = j_i$. Each term in S is the set of all $h(j_1,..., j_m)$ where h is an automorphism on B.

Along the lines of Lemmas 5.1, 5.2 it can be shown that $S = Q(B)$.

A program in *EQL* can enumerate the elements in the interpreted domain F, and determine the set F' of such elements occurring in the data base (the operators $\times$, $\cup$, and $-$ are used for this). Steps (1), (2) above, can be implemented as in Section 5 by treating the extended data base as if it were a (nonextended) data base with domain $D \cup F'$ with elements in F' all being distinguished (e.g., by additional relations $\{(f)\}$ for each $f \in F'$). Using the Turing machine capability, step (3) can be executed, and the set F'' of interpreted elements occurring in $Q(B_N)$ determined. The implementation of step (4) then follows. ∎

Other Extensions. It is sometimes desirable to have *typed queries*. For every data base, the value of such a query is either undefined, or is a relation of a fixed given rank. If

$a = (a_1, \ldots, a_k)$, and $a' \geqslant 0$, a *typed computable query* Q of type $a \to a'$ is a partial recursive function mapping (nonextended) data bases of type a into relations (on the domain of the data base) of rank a', and which satisfies the consistency criterion (Section 2). Typed computable queries form a subset of the computable queries. The query language QL may be modified so as to compute precisely the typed computable queries as follows. The output variable y_1 is given a special rank a' and is initialized to the empty set of rank a'. The semantics is changed so that any assignment $y_1 \leftarrow e$ where e is an expression of rank a' has the standard semantics. However, if e has any other rank, it assigns the empty relation of rank a' to y_1. Note that only the variable y_1 has a rank associated with it. It can then be shown that this query language computes precisely the typed computable queries. Similar restrictions can be imposed on the extended computable queries and the language EQL to handle the case when part of the data is interpreted.

One advantage of typed queries is that a sequence of them can be combined to produce a data base as output (as an alternative to a sequence of typed queries, one might just modify the definition of a query, and the query language, so that a single query produces a data base as output). Since data bases are both inputs and outputs of such sequences, the sequences could be composed to develop an algebra of data base queries (see [25]).

A second extension is that any query could refer to a fixed number of elements in the uninterpreted domain U (or in some subset of U). Consider, for example, a query like, "give me the names of all faculty members in the Computer Sciences and Mathematics departments". Such a query treats the names "Mathematics" and "Computer Sciences" as interpreted, but all other names in the data base, such as "Physics" or "Smith" as uninterpreted. For any finite set of "constants" $C \subset U$, $C = \{c_1, c_2, \ldots, c_m\}$, we can define a *query with* C to be a partial recursive function which, for data bases $B = (D, R_1, \ldots, R_k)$, produces as output a relation on $D \cup C$, and preserves isomorphisms on $B_C = (D \cup C, R_1, \ldots, R_k, \{c_1\}, \ldots, \{c_m\})$. A *query with constants* is then a query with C for some C. It is not hard to show that the set of queries with constants is exactly the set of queries computed by the query language QL augmented such that $\{c\}$ is also an expression for every $c \in U$. This exercise and the preceding one with extended queries shows that while such extensions (and others like column headings and dependencies) are useful for modelling real data bases and queries, they serve from a theoretical point of view largely to obfuscate basic issues in data base queries (see also [24, 25]). Typed queries, however, are probably of greater theoretical interest.

Another extension to queries is to add nondeterminism. For example, queries such as "give me the name of anyone in the Toy department" can be handled by having an EQL program output the names of all employees in the Toy department, and then having a "back end" choice operator that chooses an element (in general, a tuple) from this set of elements (in general, from a relation). More general versions of nondeterminism, e.g., "give me any one in the Toy department, or the entire employee-manager relation" require both a choice operator on relations as above (see also [11]) as well as nondeterminism on the flow of control (see [10] for a discussion of some primitives). In this case, however, a reasonable definition of all nondeterministic computable queries is not known, and remains a topic for future research.

A related extension is that of probabilistic choice. An example is where employees' salaries are randomly probed to estimate the average salary (see [20] for a general probabilistic construct). Another extension is where a query language is augmented with user-supplied procedures. In this case, the query can be thought of as a query schema, and concepts from program schemata (see, e.g., [13, 17]) could be used. These extensions also remain largely unexplored.

7. Directions for Future Work

Several interesting possibilities present themselves for further research. First, results similar to ours can be proved for systems other than relational data bases. For example, analogous results could be obtained for the hierarchical and network models of data bases. Perhaps the most general framework is that of computable functions over arbitrary algebraic structures [21]. Even for relational data bases, several constructs such as nondeterminism, probabilistic choice, and program schemata based queries remain largely unexplored as mentioned in the preceding section. It should be noted, however, that our results do apply to relational data bases on which certain constraints (such as functional or multivalued dependencies [1, 6, 7, 16]) are specified. Such data base definitions can be thought of as defining a subset of our data bases, and as such, no additional queries can be computed. The computable queries on such data bases are precisely the queries determined by our query languages restricted to the applicable data bases.

A second research area is that of usable query languages which are complete. Our language *QL* is intended to be a minimal language, and as such could be useful for proofs in the theory of query languages but not for writing queries. This is analogous to the role played by Turing machines in computability and complexity theory. Query languages that are more powerful than first-order predicate calculus can be obtained by various approaches, one of which involves embedding a query language such as Query-by-Example or *SQL* into a programming language such as Cobol, *PL*/1, etc. (see, e.g., [19]). A second approach is to add constructs such as summing a column in a relation, or obtaining the length in bits, of a field [14], operations that may (depending on the formalism) violate the consistency criterion. A third approach, suggested by [4] and indicated by our results, is that of augmenting relational algebra or predicate calculus with constructs such as transitive closure, fixpoint operators, while-do, or other programming features. This seems to be a promising approach. A possible disadvantage, however, is that it might be more difficult to produce efficient code as compared with the other two approaches. Such questions of efficiency provide several interesting topics for research.

Perhaps of more interest to the theoretically inclined reader, the fundamental questions of complexity theory should now be asked in the realm of relational data bases, with an eye towards singling out those queries which are not only computable, but also tractable or efficiently computable. One could start by providing, for queries, sensible definitions of measures of time and space and perhaps other relevant measures of resources (such as the number of times a data base access is required in the course of evaluating a query—this, in a practical situation, being possibly of great importance since an action may require physical manipulation such as disc seek and access). Then it would be of interest to check the appropriateness of such definitions by observing whether certain complexity classes of queries with respect to these measures are invariant within a large class of computational models (our *QL* being perhaps one of those models). Are there naturally arising queries which are, in one or more of the senses defined, provably intractable ? Are there interesting classes of queries which arise naturally in practice and which are tractable ? A particular open question: give a complexity-theoretic characterization of the first-order definable queries. This question is not presented by virtue of it being particularly hard, but by virtue of it being dependent for its solution on providing the right kinds of definitions.

For some purposes one might be interested in different kinds of subclasses of the class of computable queries, say "monotonic" or "continuous" queries, i.e., ones which not only preserve isomorphisms but which preserve small changes in data bases (that is, if B' is obtained from B by, say, removing the record of one employee, then $Q(B')$ should be "close" to $Q(B)$). Are there simple query languages which are complete for classes such as these ?

Some initial answers to these questions have been provided in [24, 25]. We feel that many other interesting questions await to be asked and answered.

REFERENCES

1. W. W. ARMSTRONG, Dependency structures of data base relationships, *in* "Proceedings, IFIP 74," North-Holland, Amsterdam, 1974, pp. 580–583.
2. A. V. AHO, C. BEERI, AND J. D. ULLMAN, The theory of joins in relational data bases, *in* "Proceedings, 18th IEEE Symp. on Foundations of Computer Science, Providence, R.I., Oct. 1977."
3. A. V. AHO, Y. SAGIV, AND J. D. ULLMAN, Equivalences among relational expressions, *SIAM J. Comput.* 8, 2 (1978), 218–246.
4. A. V. AHO AND J. D. ULLMAN, Universality of data retrieval languages, *in* "Proceedings, 6th ACM Symp. on Principles of Programming Languages, San-Antinio, Texas, Jan. 1979," pp. 110–117.
5. F. BANCILHON, On the completeness of query languages for relational data bases, *in* "Proceedings, 7th Symp. on Mathematical Foundations of Computer Science, Zakopane, Poland, Sept. 1978," Lecture Notes in Computer Science, Springer-Verlag, Berlin/New York/Heidelberg.
6. C. BEERI, R. FAGIN, AND J. H. HOWARD, A complete axiomatization for functional and multivalued dependencies in database relations, *in* "Proceedings, 1977 SIGMOD Conf." (D.C.P. Smith, Ed.), Toronto, pp. 47–61.
7. E. F. CODD, A Relational Model for Large Shared Data Bases. *Comm. ACM* **13**, 6 (June 1970).
8. E. F. CODD, Relational completeness of data base sublanguages, *in* "Data Base Systems" (Rustin, Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1972.
9. D. D. CHAMBERLIN *et al.* SEQUEL 2: a unified approach to data definition, manipulation, and control, *IBM J. Res. Dev.* **20**, 6 (Nov. 1976), 560–575.
10. A. K. CHANDRA, Computable nondeterministic functions, *in* "Proceedings, 19th Ann. Symp. on Foundations of Comp. Sci., Ann Arbor, Michigan, Oct. 1978," pp. 127–131.
11. M. A. CASANOVA AND P. A. BERNSTEIN. The logic of a relational data manipulation language, *in* "Proceedings, 6th ACM Symp. on Principles of Programming Languages, San Antonio, Texas, Jan. 1979," pp. 101–109.
12. A. K. CHANDRA AND P. M. MERLIN, Optimal implementation of conjunctive queries in relational data bases, *in* "Proceedings, 9th ACM Symp. on Theory of Computing, Boulder, Colorado, May 1977."

13. A. K. Chandra and Z. Manna, On the power of features in programming, *J. Comput. Lang.* 1, 3 (1975), 219–232.
14. H. W. Davis and L. E. Winslow, "Recursion in Retrieval Languages," technical report, Computer Science Dept., Wright State University, Dayton, Ohio, 1979. Also, The retrieval power of predicate calculus based query languages, *in* "Proceedings, ACM Computer Science Conference, Dayton, Ohio, Feb. 1979."
15. E. Engeler, Algorithmic properties of structures, *Math. Systems Theory* 1 (1967), 183–195.
16. R. Fagin, Multivalued dependencies and a new normal form for relational databases, *ACM Trans. Database Systems* 2, 3 (Sept. 1977), 262–278.
17. S. A. Greibach, "Theory of Program Structures: Schemes, Semantics, Verification," Lecture Notes in Computer Science No. 36, Springer-Verlag, Berlin/New York/Heidelberg, 1975.
18. J. E. Hopcroft and J. D. Ullman, "Formal Languages and their Relation to Automata," Addison–Wesley, Reading, Mass., 1969.
19. IBM Manual SH20-2077-0. "Query By Example, Program Description/Operations Manual," (1978).
20. D. Kozen, Semantics of probabilistic programs, *in* "Proceedings, 20th Ann. Symp. on Foundations of Comp. Sci., San Juan, Puerto Rico, Oct. 1979," pp. 101–114.
21. J. Moldestad, V. Stoltenberg-Hansen, and J. V. Tucker, Finite algorithmic procedures and inductive definability, Preprint Series, Matematisk Institutt, University of Oslo. May 1978.
22. J. Paredaens, On the expressive power of the relational algebra, *Inform. Processing Lett.*, 7, 2 (Feb. 1978).
23. M. Zloof, "Query-by-Example: Operations on the Transitive Closure," RC5526, IBM Research, Yorktown Heights, Oct. 1976.
24. A. K. Chandra, Programming primitives for database languages, *in* "Proceedings, 8th Symp. on Principles of Programming Languages, Williamsburg, Virginia, Jan. 1981."
25. A. K. Chandra and D. Harel, Structure and complexity of relational queries, *in* "Proceedings, 21st Ann. Symp. on Foundations of Comp. Sci., Syracuse, New York, Oct. 1980," pp. 333–347.