COMPUTABLE QUERIES FOR RELATIONAL DATA BASES

(Preliminary Report)

Ashok K. Chandra and David Harel

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

Abstract

The concept of a "reasonable" query in a relational data base 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. 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 [C1,C2] is attracting increasing attention lately [ABU, ASU, AU, B, BFH, CM, P]. 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. And so, theoretical results in the area are often of immediate practical value.

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 such 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 is the one with which the names and addresses of all employees earning over $15,000 a year is associated as an answer.

A large portion of the work done on query languages involves the first order relational calculus (or its closely related relational algebra, cf. [ABU, B, CM, C2, P]). 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 [C2]. There, one version of this calculus is taken as the canonical query language, to the point of terming any other language having at least its power of expression - *complete*.

In a recent paper, Bancilhon [B] questions this particular choice and voices the opinion that a universal class of queries should be defined, and its choice justified, independently of any language considerations. 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. The motivation of the present paper is that of [B].

In Section 2 relational data bases and queries are defined, and a class of queries, which we call *computable*, is identified. These are those queries which, when regarded as functions acting on data bases, are partially recursive and which furthermore are consistent with the data bases they operate on in the sense that they 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 and comparison of the other two definitions of completeness appearing in the literature, namely those of [C2] and [B].

Section 4 contains our proposed query language QL. In Section 5 the main result, i.e. the completeness of QL, is proved, and in Section 6 we provide a generalization of the simple basic model, and the appropriate generalized version of the main result. The nontrivial part of the proof of the theorem 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.

One can view the language QL as being essentially the iterative programming language obtained by taking the right-hand side of assignment statements to be expressions in a simple version of the relational algebra. Alternatively, it will become apparent that our language can be simulated by the relational algebra augmented with a least-fixpoint operator, as suggested in [AU].

2. Data Base Queries

Let U be a fixed countable set, called the *universal domain*. Let $D \subset U$ be finite 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).

Two data bases of type a, $B = (D,R_1,\ldots,R_k)$ and $B' = (D',R_1',\ldots,R_i')$ are said to be *isomorphic* by isomorphism h, (or *h-isomorphic*) written $B \xleftrightarrow{h} B'$, if $h:D \to D'$ is an isomorphism and $h(R_i) = R_i'$ for all i. An important special case is when $B = B'$. If $B \xleftrightarrow{h} B$ then h is an *automorphism* on B.

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 $\dashrightarrow$ denoting partial,

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

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 $B \xleftrightarrow{h} B'$ then $Q(B') = h(Q(B))$ i.e. Q *preserves isomorphism.*

The set of computable queries satisfies the principles postulated in [AU], 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 [B, P]. There, the *outcome* of a query Q (which is the subject matter of [B, P], 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 $B \xleftrightarrow{h} B$, 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 sets of natural numbers. Accordingly, in Section 4 we define an operationally computable data base query language and show that it computes 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 question 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 $E \in L$ 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 [C2]. Using the definitions appearing in [P] and [B] respectively, the following are expressions in 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 are a pair in either R_1 or R_2, whose last three components are 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 computable queries only. 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.

Our notations for boundedness and expressiveness, respectively, are

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

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

There have been two other definitions of completeness in the literature, both of which turn out to be, in a sense, strictly weaker than ours.

Definition (Codd [C2]): 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 strong in expressive power as the relational calculus, and it is *C-complete* if it is both C-bounded and C-expressive.

Codd used the term "complete" for our C-expressive. We have taken the liberty, here and below, of renaming notions to be more consistent with our terminology.

Codd [C] showed that the relational algebra is C-expressive (it is in fact C-complete also). Aho and Ullman [AU] 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 [AU]): 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 are expressive (or complete).

If a query language is C-bounded it is also bounded but not necessarily vice versa. Likewise, an expressive query language is also C-expressive but not vice versa. See fig. 1.

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

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

Definition (Bancilhon [B]; see also Paredaens [P]): A query language is ***BP-bounded*** if for every data base B and expression E, either $M_E(B)$ is undefined or there is an $R \in I_B$ such that $M_E(B) = R$. It is ***BP-expressive*** if for every data base B and for every $R \in I_B$ there is an expression E such that $M_B(E) = R$. A query language is ***BP-complete*** if it is both BP-bounded and BP-expressive.

Bancilhon [B] and Paredaens [P] proved, respectively, that the relational calculus and relational algebra are BP-complete. (It has been pointed out to us by L. Marcus that these results follow quite easily 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 at all, but rather regarded his result as a characterization of the power of the relational algebra to express relations. In order to better see the connection with our definition, we show:

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

$$(**) \quad (\forall a)(\forall E)(\forall B)(\exists Q)(M_E(B) \equiv Q(B)), \quad \text{and}$$
$$(\forall a)(\forall Q)(\forall B)(\exists E)(M_E(B) \equiv Q(B))$$

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

Proof: We prove that the second line 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 (**) and let $R \in I_B$ for some data base B of type a. Define the query Q of type a as

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

It can be shown that this definition is sound; in particular if $B \xleftrightarrow{h} B'$ and $B \xleftrightarrow{h'} B'$, then $h(R) = h'(R)$. Clearly, Q is a computable query with $R=Q(B)$ and by (**) there is an E such that $M_E(B) = Q(B) = R$. □

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 [P]) 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, the relational calculus augmented with the expression E_0, whose meaning is given by

$$M_{E_0}(B) = \begin{cases} \phi^0 & \text{if the } k\text{-th Turing Machine halts on input } k, \text{ where } k = |D(B)| \\ \{()\} & \text{otherwise} \end{cases}$$

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 $k \geq 0$, with meaning given by

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

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 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. Before presenting such a language, we refer the reader to [AU] in which it is suggested that the deficiency of the relational algebra as expressed in Theorem 3.1 be remedied by augmenting that language with a least-fixpoint operator on monotonic functionals over relations. Our results in the present paper can be viewed as justifying this approach; it will turn out that an appropriate way of carrying out the suggestion of [AU] indeed results in a complete language.

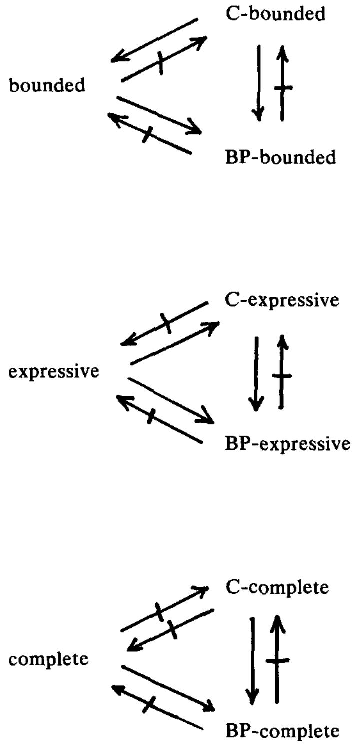

Fig. 1.

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 primitive 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 $i \geq 1$, rel_i and y_i are terms.

(2) For any terms e and e′,

$e \cap e'$, $\neg e$, $e\downarrow$, $e\uparrow$ and $e\sim$ 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 $i \geq 1$.

(2) For programs P and P′

(P; P′),

if t *then* P *else* *P′*, and

while t *do* P are programs, where t is y_i for some $i \geq 1$.

Semantics:

Assume given a data base $B = (D, R_1, .., R_k)$ of type $a = (a_1, ..., a_k)$. Expressions of QL take on values as 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, $i \geq 0$, 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 $\{()\}$. Denote by EX_i the set of all relations over D of rank i, in the above sense.

The expression *E*, equality, is a fixed relation of rank 2 given by $E = \{(d,d) \mid d \in D\}$.

The value of rel_i is given by

$$rel_i = \begin{cases} R_i & i \leq k \\ \phi^0 & i > k \end{cases}$$

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

$$\neg e = \begin{cases} D^i - e & e \neq D^i,\ rank(e) = i \\ \phi^i & e = D^i \end{cases}$$

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

$$e\downarrow = \begin{cases} \{(d_2, ..., d_i) \mid (d_1, ..., d_i) \in e\} & rank(e) = i,\ i > 1,\ e \neq \phi \\ \{()\} & rank(e) = 1,\ e \neq \phi^1 \\ \phi^i & e = \phi^{i+1} \\ \phi^0 & rank(e) = 0 \end{cases}$$

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

$$e\uparrow = \begin{cases} \{(d_1, ..., d_i, d) \mid (d \in D, (d_1, .., d_i) \in e\} & rank(e) = i,\ i \geq 1,\ e \neq \phi^i \\ \{(d) \mid d \in D\} & e = \{()\} \\ \phi^{i+1} & e = \phi^i \end{cases}$$

(Note that in fact $\{()\}\uparrow = E\downarrow$.)

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

$$e\sim = \begin{cases} \{(d_1, ..., d_{n-2}, d_n, d_{n-1}) \mid (d_1, ..., d_n) \in e\} & rank(e) > 1 \\ e & rank(e) \leq 1 \end{cases}$$

Programs in QL act in the obvious way; all variables are initialized to ϕ^0, and the test y_i in the *if then else* and *while do* constructs 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; empty sets are not typed as far as the output behavior of queries and programs is concerned.) Given a query Q of type a, we now know 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.

Before embarking on the proof we show how several conventional operations on relations are expressible in QL. Observe first that in effect we have *counters*. $I = E\downarrow\downarrow$, 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, respectively, the roles of i+1 and i-1. (Counters need never attempt to subtract 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. [HU]). Hence in the sequel, besides using n,m... in programming to denote natural numbers, we will freely use Turing machine terminology. At this point the reader should also be convinced that we can simulate the test "non-empty" in 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 $(y_1 \leftarrow y_1\downarrow$ ; $n \leftarrow n + 1)$;

$n \leftarrow n-1$;
```

Define $e(\uparrow\sim)^n$ to be the value of y_2 in the program

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

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 $e\dot{\uparrow}$:

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

Projecting out the last (rightmost) coordinate, denoted by $e\dot{\downarrow}$, is similar and is left to the reader.

We can now compute the cartesian product of two relations:

$$e_1 \times e_2 = \{(d_1,\dots,d_{rank(e_1)},b_1,\dots,b_{rank(e_2)}) \mid$$

$$(d_1,\dots,d_{rank(e_1)}) \in e_1 \text{ and } (b_1,\dots,b_{rank(e_2)}) \in e_2\}$$

This is done by:

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

We denote $\neg(\neg e_1 \cap \neg e_2)$ by $e_1 \cup e_2$, and call it the *union* of e_1 and e_2.

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

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

where for all $1 \le j \le p$, $i_j \le rank(e) = m$. In order to do this, observe that, denoting $E\downarrow(\uparrow^i)$ by D^i,

$$e_{[i_1,\dots,i_p]} =$$

$$((e \times D^p) \cap \bigcap_{1 \le j \le p} (D^{i_j-1} \times E(\uparrow\sim)^{rank(e)-i_j+j-1} \times D^{p-j}))(\underset{\bullet}{\downarrow}^{rank(e)})$$

We leave the programming to the reader.

As an example of a naturally arising query, consider again the transitive closure. The following program 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   $(y_1 \leftarrow y_3$;

      $y_3 \leftarrow y_3 \cup (((y_3 \times y_2) \cap E(\uparrow\sim)^2)_{[1,4]})$;

      $y_4 \leftarrow \neg y_1 \cap y_3)$
```

It is clear that the $\uparrow$, $\downarrow$ and $\sim$ operators can be 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 under sequencing and iteration, with the ability to test for emptiness. Also, the reader familiar with Aho and Ullman's [AU] independent work on the power of query languages, will be able to see that their suggestion of augmenting the relational algebra with least fixpoints gives rise to a complete language. This follows from Theorem 4 of [AU] and the (easily checked) fact that the language described in section 7.2 of [AU] is as powerful as QL. In Section 6 of the present paper an extension of the model of a data-base is given, with the aid of which one can specify predicates which are to be "preserved by the renamings" in the sense of [AU], and still have a complete query language.

We can 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 isomorphism, 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. (E.g. 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 defined P(B) and P(B′) are h-isomorphic.†

Turning to the other direction, i.e. expressiveness, let Q be a computable query of type $a = (a_1,...,a_k)$. 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.

In order to be able to spell out this process more precisely and show how to program it in QL, we will be needing some additional notation. Let $B = (D,R_1,..,R_k)$ 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.

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

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

We note that two different elements, d and d', of *perm*(D) may have 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 [P]) the *cogroup of* R *via* d:

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

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 $B = (D,R_1,...,R_k)$ let CG_d^B abbreviate $\bigcap_{1 \le i \le k} CG_d(R_i)$. 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 $d \in CG_d^B$.

Let us now give a more precise desription 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 tuple of sets
$$B_N = (\{1,2,...,n\}, R_1/d,...,R_k/d)$$
(Each R_i/d is an a_i – ary relation over $\{1,2,...,n\}$.)
(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 an m-ary relation over $\{1,2,...,n\}$.)
(4) Compute (in y_1)
$$S = \bigcup_{(j_1,...,j_m) \in Q(B_N)} (CG_d^B)_{[j_1,...,j_m]}.$$
(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 isomorphism 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 $\{(d_1,...,d_n)\}$, a subrelation of CG_d^B. Indeed, we now show that

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

First, observe that $B_N \xleftrightarrow{h} B$, where for $1 \le i \le n$, $h(i) = d_i$. This

† This argument is analogous to theorem 1 of [B] and lemma 2 of [P] in which, respectively the constructs of Codd's [C2] relational calculus and algebra were shown to preserve isomorphism.

follows immediately from the definition of R/d. Hence, Q being a computable query, we must have $Q(B_N) \xleftarrow{h} Q(B)$, 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 $s = (s_1,...,s_m) \in S$. Then there is $(j_1,...,j_m) \in Q(B_N)$ and $(d_{\alpha(1)},...,d_{\alpha(n)}) \in CG_d^B$, such that for $1 \leq i \leq m$, $s_i = d_{\alpha(j_i)}$. We show that $(d_{\alpha(j_1)},...,d_{\alpha(j_m)}) \in Q(B)$. Note that, by definition of CG_d^B, $B \xleftarrow{\alpha'} B$, where $\alpha'(d_i) = d_{\alpha(i)}$. It follows that $\alpha'(Q(B)) = Q(B)$ or that $(d_{j_1},...,d_{j_m}) \in Q(B)$ iff $(d_{\alpha(j_1)},...,d_{\alpha(j_m)}) \in Q(B)$. But $(j_1,...,j_m)$ being in $Q(B_N)$ by assumption, implies $(d_{j_1},...,d_{j_m}) \in Q(B)$ by the characterization of Q(B) in the proof of Lemma 5.1. □

Hence we have established that the above four steps, if executable, 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 $1 \leq i < j \leq rank(e)$, denote by $e_{(i \neq j)}$ the expression

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

(The same expression, but without the "$\neg$", is denoted $e_{(i=j)}$). Denote by $e_{(\neq)}$ the relation obtained by executing

```
(*)   y ← e;
      for all 1≤i<j≤rank(e) do
          y ← y_(i≠j)
```

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

```
n ← 0; y_2 ← E ↓ ;

while ȳ_2 do (y_3 ← y_2↑ ; y_2 ← y_3(≠) ; n ← n + 1) ; y_2 ← y_3↓.
```

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:

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

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 (assuming for the moment that we can compute ψ):

```
y_3 ← perm(D)
for all 1≤i≤k do y_3 ← ψ(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 $1 \leq i \leq k$. In fact, y_3 will then have the value

$$(...((perm(D) \cap CG_d(R_1)) \cap CG_d(R_2)) \cap ..... \cap CG_d(R_k)) = CG_d^B.$$

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, the following program computes $\psi(V,r,R,X)$ in y_3 (As earlier, the reader should convince himself that (**) can be rewritten precisely in QL.)

```
(**) y_3 ← V ;
     for all  (i_1,...,i_r) ∈ N^r do
         (if  (i_1,...,i_r) ∈ X  then  y_4 ← R × y_3
              else y_4 ← ¬R × y_3;
          for all  1≤j≤r  do  y_4 ← y_4(j=r+i_j);
          y_3 ← y_4(↓^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 $T = \phi$.

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 some element $(d_{\alpha(1)},...,d_{\alpha(r)})$ of R, and such that for all $1 \leq j \leq r$, $d_{\alpha(j)} = d_{i_j}$. 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 $CG_d(R) \cap V = T$.

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 Theorem4.1, 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 programs described earlier for the union and generalized projection operators.

6. Extensions

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, and operations $S_i:D^{b_i} \to F$ 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 obtainedby applying on 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: $Z_\lambda = \{()\}$, $Z_{0c} = D \times Z_c$, $Z_{1c} = F \times Z_c$. The type of B is $(a_1,...,a_k,b_1,...,b_m)$.

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, $B \xleftarrow{h} \to B'$) if $h:D \to D'$ 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).

An *extended data base query of type a* (*extended query* for short) is a partial function

$$Q:\{B \mid B \text{ a data base of type } a\} \dashrightarrow \bigcup_c 2^{Z_c}$$

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 following consistency criterion: if $B \xleftarrow{h} \to B'$ then $Q(B') = h(Q(B))$.

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), $F = \{0,1,2,...\}$, $R \subset D \times D$ is of rank 00 and associates salaries with names, and $S:D \to F$ 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

$\{x \mid \exists i, (x,i) \in R \land S(i) = \text{Max}\{S(j) \mid (x,j) \in R\}\}$

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 $S:D \to F$ might associate with each department name, viewed as a string of characters, its length.

We define an extended query language (EQL) consisting of the same constructs as those in QL with the extension that a term can also have the forms $S_i(y_j)$ and f_{y_i}. 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, and $\neg e$ has value $Z_c - e$ where e has rank c. 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.

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.

The theorem is proved essentially the same way as was Theorem 4.1, but here, computing *rank*(e) (say as a string in $\{f_0,f_1\}^*$) is slightly more complicated. Also, the program will compute, on its Turing machine tape, the set of all elements in F which are either in the R_i's, or are reachable from D via the S_i's. We leave the details of this proof, as well as additional generalizations of Theorem 4.1, to an extended version of the paper.

Acknowledgement:

We are grateful for the superb typing assistance of Marcia Bollard.

References:

[ABU] Aho, A. V., C. Beeri and J. D. Ullman. The Theory of Joins in Relational Data Bases. Proceedings 18th IEEE Symp. on Foundations of Computer Science. Providence, R.I., Oct. 1977.

[ASU] Aho, A. V., Y. Sagiv and J. D. Ullman. Equivalences Among Relational Expressions. *SIAM J. Computing*, 1978.

[AU] Aho, A. V. and J. D. Ullman. Universality of Data Retrieval Languages. Proceedings 6th ACM Symp. on Principles of Programming Languages. San-Antonio, TX, Jan. 1979.

[B] Bancilhon, F. On the Completeness of Query Languages for Relational Data Bases. Proceedings 7th Symp. on Mathematical Foundations of Computer Science. Zakopane, Poland. (Springer-Verlag Lecture Notes in Computer Science.) Sept. 1978.

[BFH] Beeri, C., R. Fagin and J. H. Howard. A Complete Axiomatization for Functional and Multivalued Dependencies. IBM San Jose Research Report RJ 1977.

[CM] Chandra, A. K. and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. Proceedings 9th ACM Symp. on Theory of Computing. Boulder, CO, May 1977.

[C1] Codd, E. F. A Relational Model for Large Shared Data Bases. *Communications of the ACM*, Vol. 13, No. 6. June 1970.

[C2] Codd, E. F. Relational Completeness of Data Base Sublanguages. In *Data Base Systems* (Rustin, Ed.), Prentice Hall, 1972.

[HU] Hopcroft, J. E. and J. D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, Reading, MA, 1969.

[P] Paredaens, J. On the Expressive Power of the Relational Algebra. *Information Processing Letters*, Vol. 7, No. 2, Feb. 1978.