Universality of Data Retrieval Languages

Alfred V. Aho
Bell Laboratories
Murray Hill, New Jersey

and

Jeffrey D. Ullman†
Princeton University
Princeton, New Jersey

Abstract. We consider the question of how powerful a relational query language should be and state two principles that we feel any query language should satisfy. We show that although relational algebra and relational calculus satisfy these principles, there are certain queries involving least fixed points that cannot be expressed by these languages, yet that also satisfy the principles. We then consider various extensions of relational algebra to enable it to answer such queries. Finally, we discuss our extensions to relational algebra in terms of a new programming language oriented model for queries

1. Introduction

One facility provided by a database system is a *query,* or *data manipulation,* language whose primary function is the extraction of information from the database. The data retrieved by a single query can range from a small simple subset of the database, as in "print the name and address of the employee with employee number 12345," to a large complex subset, as in "print the names of all employees under 40 whose last three raises have been above average." A query will be treated as a mapping on the contents of the database, which we here regard as a collection of relations [C1]. The value returned by a query will also be a relation, that is, a set of k-tuples for some $k \geqslant 1$.

One key question concerning query languages is what power they should have. For example, should a query language be able to specify any mapping whatsoever from lists of relations to relations? It is our point of view, and one held widely, that a query language should provide physical data independence; that is, the result of a query should not depend on the representation of the data. We also feel that the role of a query language should be primarily the selection of data from a database, rather than arithmetic computation on this data. The computational capability, if desired, should be separate from the retrieval capability. This separation of function has a number of benefits, such as simplifying the optimization of both the computational and the query operations.

We therefore postulate two principles that a query language should obey. In essence, these principles state (1) that the value produced by a query should be independent of the manner in which the data are actually stored in a database and (2) that a query language should treat data values as essentially uninterpreted objects, although certain properties, such as a linear ordering on certain domains can be built into the query language. These principles are defined more formally in Section 2.

The relational algebra and calculus of Codd [C1] satisfy these principles and are often used as models of a query language. One purpose for which Codd introduced these languages was to provide a yardstick for measuring the relative power of query languages.

There is, however, an important family of "least fixed point" operations that still satisfy our principles but yet cannot be expressed in relational algebra or calculus. Such fixed point operations arise naturally in a variety of common database applications. In an airline reservations system, for example, one may wish to determine the number of possible flights between two cities during a given time period. In a network analysis system, one may wish to determine whether there is an active circuit connecting two points. In a business management system, one may wish to determine the lowest-level manager common to a group of employees. None of these queries can be couched in relational algebra. In the appendix, we give a formal proof that the transitive closure of a binary

† Work partially supported by NSF grant MCS-76-15255

relation, an elementary example of a least fixed point operation, cannot be expressed in relational calculus.

In Sections 4 and 5 we consider extensions to relational algebra to enable queries such as these to be expressed. In Section 6 we discuss various methods by which relational algebra expressions containing least fixed point operators can be efficiently implemented.

Finally, in Section 7 we examine the universality of our extensions in terms of a new, stylized programming language that is intended to serve as a model of computation for query languages. We show that the programming language is at least as powerful as relational algebra with a least fixed point operator. We also show that Codd's original formulation of relational algebra is equivalent to this language with a restrictive interpretation of the semantics of the **for** statement in this language.

2. Two Principles for Data Retrieval Languages

The general consensus is that, at least for query languages based on the relational model, one does not wish to have general Turing machine (see [HU], e.g.) capabilities. In particular, one wants the query language to be sufficiently high level that it deals with relations as sets of tuples, meaning that the order in which the tuples are considered should not influence the result. In this sense, the Turing machine working on an input tape containing lists of tuples is too powerful a model. This is the basis of the first of our two principles regarding what properties a relational query language should have.

Principle 1. The result of a function on relations should depend only on the values of the relations as sets of tuples. The result should not depend on the order in which the tuples are stored.

Our formalization of the second principle is related to an idea used independently by **Paredaens** [P] and Bancilhon [B] to characterize certain aspects of relational queries. Their idea is that queries should preserve symmetries that exist among the values that appear in argument relations. More formally, assume D is the domain from which all values for the components of relations are taken. Let μ be a 1-1 mapping of D into itself. (D is presumably infinite.) Call μ a *renaming.* If f is a function that takes n relations as arguments, we say f *commutes* with μ if $\mu(f(r_1, \ldots, r_n)) = f(\mu(r_1), \ldots, \mu(r_n))$. Let us call f an *acceptable query* if f commutes with every renaming. In this sense, operations like the union or Cartesian product of relations are acceptable; so is the transitive closure of a binary relation.

There are, however, other queries we might well regard as acceptable that do not commute with arbitrary renamings. For example, "print the name of the employee with employee number 12345" does not commute with renamings that do not map 12345 to itself. Saying that the renaming should also apply to 12345 in the query skirts the issue that we are concerned with a class of abstract functions on relations, and such functions need not have a concrete representation where a specially treated constant like 12345 is explicitly named. Another problem comes up when we want to use a relationship like $<$, as in AGE $<$ 40. We would not expect such a function to commute with those renamings that did not preserve the $<$ order on D.

We thus extend the Bancilhon-**Paredaens** concept in the following way. We postulate the existence of a collection of predicates P on the domain D. These could be any sort of predicates; for example, a unary predicate like "$x = 12345$" or a binary predicate like "$x < y$." Intuitively, one might expect that a function f uses some finite subset of the available predicates in P to extract or select information from its argument relations. We would not, therefore, expect that f would commute with renamings that did not preserve the predicates.

We say μ *preserves* predicate $p(x_1, \ldots, x_n)$ if $\mu(p(x_1, \ldots, x_n))$ is true if and only if $p(\mu(x_1), \ldots, \mu(x_n))$ is true. For example, if p is "$x = 12345$", then μ preserves p if and only if it maps 12345 (and therefore no other value) to 12345. μ preserves $<$ if $\mu(x) < \mu(y)$ if and only if $x < y$. We can now state our second principle formally.

Principle 2. A function f is allowable with respect to a set of predicates P on domain D if and only if there is a finite subset $P_1 \subseteq P$ such that f commutes with every renaming that preserves the predicates in P_1.

In what follows, we shall always talk as though D were a set containing integers, reals, and character strings, with $<$ defined in the obvious manner when applied to values of the same type. We shall also take the set of predicates P to be boolean combinations of statements of the form $x < y$ and $x = c$, for some constant c in D. The ideas extend to other sets of predicates, although if we use too rich a set, such as "$x+y=z$" and "$u \times v = w$," we can with a finite set of predicates P_1 assure that only the identity mapping preserves P_1, thus making every f allowable.

We shall not take a position on the "correct" set of underlying predicates, but we shall take the two principles above as a definition of what properties a complete query language should have, relative to a given set of "built-in" predicates P.

Codd's relational algebra [C1] is often used as a model of a query language, that is, as a minimal set of operations that every query language should have. We shall describe relational algebra in the next section, whereupon it will be obvious that all queries posable in relational algebra satisfy the two principles above, if the allowable predicates are taken to be $x < y$ and $x = c$ for all constants c.

3. Relational Algebra

In this section we shall define a set of relational algebra expressions that is complete for *relational calculus,* which is the first order theory of relations and tuples with comparison operators $=$ and $<$ on components of tuples [C2]. The operands of an expression are either constant unary relations or variables representing relations. The operators of an expression are the following:

(1) *Cartesian Product.* If R and S are relations consisting of r-tuples and s-tuples, respectively, then $R \times S$ is the set of $(r+s)$-tuples of which the first r components are an r-tuple in R and the last s components are an

s-tuple in S.

(2) *Set Union.* If R and S are relations whose tuples are of the same length, then $R \cup S$ is the union of the tuples in R and S.

(3) *Set Difference.* If R and S are relations whose tuples are of the same length, then $R - S$ is the set of tuples that are in R but not in S.

(4) *Selection.* Let F be a predicate built from

(i) the logical operators: **and** ($\wedge$), **or** ($\vee$), **not** ($\neg$),

(ii) the arithmetic relational operators: $=$, $<$, $>$, $\leqslant$, $\geqslant$, $\neq$, and

(iii) operands of the form $\$i$ standing for the i^{th} component of a tuple.

Then $\sigma_F(R) = \{t \mid$ for some tuple t in R, F becomes true when $\$i$ is replaced by the i^{th} component of $t\}$.

(5) *Projection.* $\pi_{\$i_1,\$i_2,\ldots,\$i_k}(R) = \{t' \mid$ for some t in R, t' is the k-tuple whose j^{th} component is the i_j^{th} component of $t\}$. We assume $i_1, i_2, \ldots, i_k$ have no repetitions (although this constraint is not essential), and the i_j's are not assumed to be in any particular order.

All expressions in relational algebra produce relations as values, when given relations as arguments. Moreover, the value of an expression cannot involve a symbol in a tuple if that symbol was not part of some tuple of some argument relation. Therefore, if the operands in a relational expression are all finite relations, then the value of the expression is also a finite relation.

Example 1. Let $R = \{012, 121, 001\}$.† Then $\sigma_{\$1<\$2}(R) = \{012,121\}$, and $\pi_{\$3,\$1}(R) = \{20,11,10\}$.

As another example, we can define the *composition* of two binary (two-component) relations R and S by $R \circ S = \pi_{\$1,\$4}(\sigma_{\$2=\$3}(R \times S))$. This shorthand will be used subsequently. □

4. The Least Fixed Point Operator

Consider an equation of the form

$$R = f(R) \tag{1}$$

where $f(R)$ is a relational algebra expression with operand R, perhaps among other operands, such that the degree (i.e., the number of components in each tuple) of R and $f(R)$ are the same. A *least fixed point* of equation (1), denoted LFP($R = f(R)$), is a relation R^* such that

(i) $R^* = f(R^*)$ and

(ii) If R is any relation such that $R = f(R)$, then $R^* \subseteq R$.

In general, there may not be any R^* satisfying (i) or any satisfying (ii). However, Tarski [T] assures us that a unique least fixed point exists if f is *monotone,* which in the context of the partial order $\subseteq$ on relations means that

$$\text{if } R_1 \subseteq R_2, \text{ then } f(R_1) \subseteq f(R_2) \tag{2}$$

or equivalently

$$f(R_1 \cup R_2) \supseteq f(R_1) \cup f(R_2) \tag{3}$$

† Throughout, we shall indicate a tuple such as (0,1,2) by the sequence 012

There is a stronger condition called *additivity,* expressed by

$$f(R_1 \cup R_2) = f(R_1) \cup f(R_2) \tag{4}$$

Obviously (4) implies (3), so every additive f is monotone.

If f is monotone, then by induction on i we can show that

$$f^{i-1}(\varnothing) \subseteq f^i(\varnothing)$$

where f^i is f applied i times. If all argument relations are finite, then since no new component values are introduced by the relational algebra operators, we know that there is some finite relation of which each $f^i(\varnothing)$ is a subset. Therefore, there must be some n_0 such that

$$\varnothing \subsetneq f(\varnothing) \subsetneq f^2(\varnothing) \subsetneq \cdots \subsetneq f^{n_0}(\varnothing) = f^{n_0+1}(\varnothing) \tag{5}$$

We shall use the term

$$\lim_{n\to\infty} f^n(\varnothing)$$

for $f^{n_0}(\varnothing)$.

It is easy to check that $f^{n_0}(\varnothing)$ is a fixed point of R. An induction on i shows that $f^i(\varnothing)$ is contained in any fixed point of equation (1). Therefore $\lim_{n\to\infty} f^n(\varnothing)$ is the least fixed point of (1).

Example 2. The transitive closure of a binary relation R_0 is the least fixed point of

$$R = R \circ R_0 \cup R_0$$

If we let $f(R) = R \circ R_0 \cup R_0$, then

$$f^n(\varnothing) = \bigcup_{i=1}^{n} R_0 \circ R_0 \circ \cdots \circ R_0 \quad (i \text{ times})$$

as may be proved by an easy induction on n. Thus by (5), the least fixed point of $R = f(R)$ is what we normally call the transitive closure, that is,

$$\bigcup_{i=1}^{\infty} R_0 \circ R_0 \circ \cdots \circ R_0 \quad (i \text{ times})$$

Example 3. Suppose we have a database representing airline flights, containing the relation

FLIGHTS(SOURCE, DEST, D_TIME, A_TIME)

where SOURCE and DEST indicate the source and destination cities; D_TIME and A_TIME indicate the departure and arrival times. We might wish to compute a relation FLIGHTS* with the same four components that includes FLIGHTS and, in addition, represents all finite sequences of flights such that in each sequence

(i) the destination of each flight (except the last) is the source of the next, and

(ii) the arrival time of each flight (except the last) occurs before the departure time of the next.

We could express this relation as the least fixed point of the equation

$$\text{FLIGHTS}^* = \text{FLIGHTS} \cup \pi_{\$1,\$6,\$3,\$8}(\sigma_{\$2=\$5 \text{ and } \$4<\$7}(\text{FLIGHTS} \times \text{FLIGHTS}^*)) \tag{6}$$

Note that when we treat (6) as an equation of the form $R = f(R)$, then $f(R)$ is

$$\text{FLIGHTS} \cup \pi_{\$1,\$6,\$3,\$8}(\sigma_{\$2=\$5 \text{ and } \$4<\$7}(\text{FLIGHTS} \times R))$$

and FLIGHTS is an operand that is presumed to be a constant as far as the taking of a fixed point is concerned. □

The following theorem is easy to prove by induction on the number of operators in a relational expression.

Theorem 1. Any relational algebra expression that does not use the set difference operator is additive in all its variables. □

On the other hand, a nonmonotone expression can have a least fixed point and not every expression involving the difference operator fails to be monotone.

5. Embedding the Least Fixed Point Operator in a Query Language

The problem of how to incorporate a least fixed point (LFP) operator into a query language in a useful manner is not trivial. To begin, it may be hard just to determine whether the LFP of an equation exists, let alone to visualize what it is. Also, evaluating a LFP in a straightforward way may be computationally infeasible unless some advantage is taken of the sparseness of a relation or the special nature of the expression involved. In this section, we propose three syntactic mechanisms whereby the LFP operator can be made available to a query language modeled after relational algebra.

(1) Provide specialized operators such as * (reflexive-and-transitive closure) or $^+$ (transitive closure) that can be applied to certain relations. Such shorthands are clearly useful, but they do not provide us with the versatility of the LFP operator; witness Example 2.

Standard methods for computing the transitive closure [AHU] will probably be prohibitively expensive unless they take into consideration the likely sparseness of a relation. Recently, an algorithm for transitive closure that is optimal in an expected time sense has been given [S]. A facility to define transitive closures (but not general LFP's) is available in Query-by-Example [Z].

(2) Provide a syntactically sugared way of saying "let S be the LFP of $R = f(R)$," where f is an expression in relational algebra, perhaps with other fixed point operators. In essence, here we are proposing that the user be able to write down an equation that his desired relation satisfies, but no smaller relation satisfies. If he does so, and his intuition is correct, $R = f(R)$ will have a LFP. If f is monotone, the LFP can be computed using (5), where, as we have mentioned, only a finite number of terms need be taken before we obtain the limit; there is, however, no *a priori* upper bound on the number of terms to be taken.

(3) Provide a procedural method of constructing a relation R inductively. That is, we provide a basis rule that says $R_0 \subseteq R$ for some relation R_0, which may be an algebraic expression of existing relations. We then provide an inductive rule that says $g(R) \subseteq R$, where g is some function of R, expressed in relational algebra, that has as value a relation with the same degree as R. By implication, we assume an exclusion clause saying that nothing is in R unless it follows from the basis and induction rules. We might express such a procedure in terms of a simple **while**-loop:

$R \leftarrow R_0$
do
  $R' \leftarrow R$
  $R \leftarrow R \cup g(R)$
while $(R' \neq R)$

Now the value of R can be computed as the limit of the sequence $X_0, X_1, \ldots,$ where $X_0 = R_0$ and $X_i = g(X_{i-1}) \cup X_{i-1}$ for $i \geqslant 1$. If g is monotone, and $f(R) = R_0 \cup g(R) \cup R$, then f is also monotone, and $X_i \subseteq f^{i+1}(\varnothing) \subseteq X_{i+1}$ for all $i \geqslant 0$, as an easy induction on i shows. As the limit of $f^n(\varnothing)$ exists, the limit of X_n exists, and the limits are the same.

Conversely, if f is monotone, then the LFP of $R = f(R)$ can be expressed by giving the basis rule: $\varnothing \subseteq R$ and the induction rule $f(R) \subseteq R$. As $f^{i-1}(\varnothing) \subseteq f^i(\varnothing)$ follows from monotonicity and induction on i, we see that $X_i = f^i(\varnothing)$. Therefore, for monotone functions, methods (2) and (3) are equivalent in their ability to define relations. However, method (3) has the additional advantage that the limit $\lim_{n\to\infty} X_i$ is known to exist independent of whether g is monotone, since when g is a relational algebra expression, there are only a finite number of symbols that can appear in the tuples of any X_i, and $X_0 \subseteq X_1 \subseteq X_2 \subseteq \cdots$.

6. Optimization of Queries with the LFP Operator

The mechanical optimization of queries that is possible when the operators are union, selection, projection, and Cartesian product [CM, ASU, SY] does not seem to be available when the LFP operator is included. A "next-best" alternative is to develop commutation laws for the LFP operator, similar to those developed for other relational operators by [SC]. Most important is a method for evaluating selections ahead of LFP operators. For example, this change can convert a general transitive closure problem into a single source shortest path problem.

We shall present a method for commuting selections over a LFP operation, provided that in the expression $\text{LFP}(R = f(R))$ the function f is such that it has only one instance of R. There are certain restrictions on the form of the selection as well, since selection does not commute with Cartesian product, in general. We also assume either that f is monotone, or that the semantics of the LFP operator are such that the limit of the series (5) is what is wanted. Let us begin with an example that will introduce the general method.

Example 5. The transitive closure of a relation R_0 is $\text{LFP}(R = R \circ R_0 \cup R_0)$. If we want to know the points accessible from a given point a_0, we could write $\sigma_{\$1=a_0}(\text{LFP}(R = R \circ R_0 \cup R_0))$. However, computing the desired relation $S = \{a_0 b \mid a_0 b \text{ is in } R_0^+\}$ requires that we first compute R_0^+, an operation that is far more expensive than we need. Our goal is to, in a sense, move $\sigma_{\$1=a_0}$ inside the LFP operator, so S can be computed directly.

Our approach is to replace the expression

$$\sigma_{\$1=a_0}(\text{LFP}(R = R \circ R_0 \cup R_0)$$

by its infinite expansion

$$\sigma_{\$1=a_0}(((((\cdots) \circ R_0) \cup R_0) \circ R_0) \cup R_0) \qquad (7)$$

obtained by repeatedly replacing R by $R \circ R_0 \cup R_0$. Expression (7) has the form $\sigma_{\$1=a_0}(X \cup R_0)$. Since selection distributes across union, we can rewrite (7) as

$$\sigma_{\$1=a_0}(X) \cup \sigma_{\$1=a_0}(R_0) \qquad (8)$$

where expression X has the form $Y \circ R_0$, and

$$Y = (((((\cdots) \circ R_0) \cup R_0) \circ R_0) \cup R_0)$$

We would now like to distribute the selection operator across the composition, but we may not do this in general. If the selection involves only the first or second argument of the composition, however, then it is possible to move the selection through the composition operator. Since $\sigma_{\$1=a_0}$ involves only the first argument here, we can replace (8) by

$$\sigma_{\$1=a_0}(Y) \circ R_0 \cup \sigma_{\$1=a_0}(R_0) \qquad (9)$$

We now note that $\sigma_{\$1=a_0}(Y)$ looks exactly like (7), our original expression. If we use E to denote the value of (7), we obtain the following fixed point equation for E

$$E = E \circ R_0 \cup \sigma_{\$1=a_0}(R_0) \qquad (10)$$

The value of equation (10) can be computed relatively efficiently by expanding according to (5). Note that we never put into T any tuple ab with $a \neq a_0$. □

The above example works because several conditions hold.

(1) The function $f(R) = R \circ R_0 \cup R_0$ is monotone, so the infinite expansion "makes sense," and the constructed function (10) is also monotone.

(2) We are able to distribute the selection operator through every subexpression of (7).

(3) We obtain a subexpression which has the same form as our original expression.

Condition (3) turns out to hold without loss of generality, provided that (2) holds. This follows from the fact that the selection cannot change substantially as it is distributed; it can only apply to different components or disappear entirely. Thus only a finite number of selections are ever applied to the various subexpressions.

We can now state our general algorithm for taking an expression of the form $\sigma_F(\text{LFP}(R = f(R)))$, where $f(R)$ has only one occurrence of R, and is monotone, and converting it to a particular expression $h(\text{LFP}(S{=}g(S)))$. The work involved in computing the latter expression is of no greater order than that to compute the former, and in some circumstances will be an order of magnitude less.

Algorithm 1. Distribution of selection through the LFP operator.

Input. An expression $\sigma_F(\text{LFP}(R = f(R)))$ where $f(R)$ has only one occurrence of R and is monotone.

Output. An equivalent expression $h(\text{LFP}(S = g(S)))$.

Method. We construct a sequence of expressions $E_0, E_1, \ldots$, each with one occurrence of the operand R, by the following rules.

Basis. E_0 is $\sigma_F(R)$.

Induction. Suppose we have constructed E_i. If E_i has a subexpression of the form $\sigma_G(R)$, then in E_i replace $\sigma_G(R)$ by $\sigma_G(f(R))$. Since $f(R)$ has one occurrence of R, the resulting expression will also have one occurrence of R.

We now attempt to distribute the selection operator through $f(R)$ as far as possible using the following identities.

(1) If σ_G is applied to $Y{\times}Z$, the following rules apply.

(i) $\sigma_G(Y{\times}Z) = (\sigma_G(Y)){\times}Z$ if σ_G applies only to components that come from Y.

(ii) $\sigma_G(Y{\times}Z) = Y{\times}\sigma_{G'}(Z)$ if σ_G applies only to components from Z. G' is G with the component numbers adjusted appropriately.

(iii) If neither Y nor Z by itself includes all components mentioned in G, we cannot distribute the selection through the Cartesian product.

(2) $\sigma_G(Y \cup Z) = \sigma_G(Y) \cup \sigma_G(Z)$

(3) $\sigma_G(Y - Z) = \sigma_G(Y) - \sigma_G(Z)$

(4) $\sigma_G(\sigma_H(Y)) = \sigma_H(\sigma_G(Y))$†

(5) $\sigma_G(\pi_L(Y)) = \pi_L(\sigma_{G'}(Y))$ where G' is G with component numbers adjusted appropriately.

As a special case, if the selection is not distributed to the subexpression of $f(R)$ that contains R, then apply the trivial selection that is satisfied by every tuple to that subexpression. This is done for convenience, to ensure some selection will eventually reach the argument R.

Expression E_{i+1} results from applying these rules as far as possible. We continue the induction with E_{i+1} in place of E_i except in two cases.

(1) If we are unable to pass a selection through a Cartesian product of terms, one of which involves the argument R, then we fail to commute σ_F with the LFP operator.

(2) If E_{i+1} is of the form $h(g(\ \sigma_F(R)))$ and for some $j \leqslant i$, E_j is of the form $h(\sigma_F(R))$, then we have found g and h. From E_{i+1} we construct the expression $h(\text{LFP}(S = g(S)))$.

Theorem 2. If f is monotone, Algorithm 1 succeeds in producing an equivalent expression $h(\text{LFP}(S = g(S)))$. If no relation appearing as an operand in f, or as the result of evaluating a subexpression of f is empty, then the time to evaluate $h(\text{LFP}(S{=}g(S))$ is no greater than a constant times that needed to evaluate $\sigma_F(R{=}f(R))$ (and in general may be much less).

Some Extensions

The same technique as described above for selections can also be used to distribute some projections into a LFP operator. In fact, projections always pass through a Cartesian product, so there is no risk of failure from that

† We could combine the two selections, but we assume selections within f have been distributed as much as possible, and to combine selections here may make it impossible to pass our selection through a Cartesian product.

source. We may also generalize what we have done to LFP operators that simultaneously define several relations.

7. Toward an Improved Model for a Relational Query Language

In this section we define a language that we feel can serve as a model of computation for relational database retrieval operations. The language obeys the principles specified in Section 2 and has the ability to:

(1) Create new tuples from given tuples by selecting certain components from the given tuples and arranging them in some order.

(2) Create new relations.

(3) Copy relations.

(4) Iterate over all tuples of a relation in an unspecified order.

(5) Make tests based on some property of a given tuple, provided that property involves only arithmetic comparisons among components of the tuple and, perhaps, constants.

(6) Make tests based on the membership of a tuple in a relation.

Although we cannot prove that we have found the maximal class of queries that respect the two principles of Section 2, we nevertheless feel that our language serves as one natural benchmark against which other notations for operating on relations can be evaluated, in exactly the same way that relational calculus serves as a benchmark. Under one natural interpretation of the iteration statement our language is equivalent to relation calculus or algebra; under another more general interpretation it is at least as powerful as relational algebra with the fixed point operator, and we conjecture that it is strictly more powerful.

A Language for Data Retrieval

We now define our data retrieval language. The statements of the language are as follows.

(1) $t \leftarrow f(t_1, \ldots, t_n)$

Here t and t_i are tuple-valued variables and f is a function that produces from tuples $t_1, \ldots, t_n$ a particular tuple $t_{i_1}(j_1)\, t_{i_2}(j_2) \cdots t_{i_k}(j_k)$, where $t_i(j)$ is the jth component of tuple t_i.

(2) **insert**(t, R) and **delete**(t, R)

Here, t is a tuple variable and R a relation variable.

(3) $R \leftarrow S$

R is a relation variable and S is a relational variable or constant.

(4) **for** t **in** R **do** <statement>

Here t is a tuple variable whose scope is local to the **for** statement and R is a relation variable. Any tuple variable assigned within the **for** statement is assumed local to that **for** statement. We shall see that the precise semantics of the **for** statement determines the class of functions definable in the language.

(5) **if** $p(t)$ **then** <statement> [**else** <statement>]

Here t is a tuple variable and p is a predicate built from the arithmetic comparison operators and the logical connectives (and, or, not).

(6) **if** t **in** R **then** <statement> [**else** <statement>]

Again, t is a tuple variable and R a relation variable.

(7) **begin** <statement list> **end**

(8) **local** t

This statement defines a tuple variable t whose scope is local to the **begin-end** block in which this define satement is contained.

A program in this language is a statement, which can be a **begin-end** block. The input to a program is the set of relation variables in the program that are referenced but not previously defined. One relation variable is designated as being the output relation variable. Thus a program computes a function whose arguments are the input relation variables and whose value is that of the output relation variable.

We assume that for each relation variable there is a fixed degree, and all tuples in the relation have this number of components. We also assume that there is a fixed arity for each tuple variable. Finally, we assume each component of each relation is selected from a known finite domain for that component.

The Four Interpretations of the for Statement

The semantics of the **for** statement

for t **in** R **do** <statement>

can be defined in several different ways. There are two orthogonal issues. The <statement> should be executed for each tuple of R. But should the execution be in parallel, or should we iterate over each t in R serially? The second issue is, should the value of R be bound on entry to the **for** loop, or should it be allowed to change within the loop? In each case we get a different class of functions. Let us consider the four interpretations and the resulting classes of functions in turn.

7.1. R Bound Before the Loop; Parallel Execution

In this interpretation we associate with each tuple variable a set of tuples. The following rules are used to interpret the statements:

(1) $t \leftarrow f(t_1, \ldots, t_n)$. Assign to t the set of tuples formed by taking one tuple in each set for t_i, $1 \leqslant i \leqslant n$ and applying f to those tuples. For example, if the assignment is $t \leftarrow t_1 t_2$ (juxtaposition of tuples denotes concatenation), then the set for t is the Cartesian product of the sets for t_1 and t_2.

(2) **insert**(t, R) and **delete**(r, R) are executed by setting R to $R \cup T$ and $R - T$, respectively, where T is the set of tuples associated with t.

(3) $R \leftarrow S$ has the obvious meaning and does not affect the sets associated with tuple variables.

(4) **for** t **in** R **do** <statement>

The set associated with t is R at the beginning of <statement>. The tuple variable t is local to the **for** statement.

(5) **if** $p(t)$ **then** <statement1> [**else** <statement2>]

Let T be the set associated with the tuple variable t before the **if** statement. Let T' be the set $\{s \mid s \text{ is in } T \text{ and } p(s)\}$. Within <statement1>, set T' is associated with tuple variable t and within <statement2>, if it is

present, $T-T'$ is associated with t. After the **if** statement T is again associated with t, even if at that time the union of the two sets associated with t is not T. Note that we in effect define a new t local to the **if** statement.

(6) **if** t **in** R **then** <statement1> [**else** <statement2>]

The meaning of this statement is similar to (5) with $T' = T-R$.

(7) **begin**
<statement1>;
.
.
.
<statement$n$>
end

Any changes to the set associated with a tuple variable local to the block made by <statement$i$> carry over to the next statement, unless the rule for <statement$i$> causes the value for that variable to be changed (as in (1)).

Example 6. Consider the program in Fig. 5, which computes $T = R \circ S$. Suppose $R = \{ab,ac,bc\}$ and $S = \{aa,cb\}$ initially. After statement $t \leftarrow r\ s$ the sets associated with t, r and s are $\{abaa,abcb,acaa,accb,bcaa,bccb\}$, $\{ab,ac,bc\}$, and $\{aa,cb\}$, respectively. The statement **insert**(u, T) sets T to $\{ab,bb\}$, which is $R \circ S$. □

```
begin
    T ← ∅;
    for s in S do
        for r in R do
            begin
                local u;
                t ← r s;
                if t(2) = t(3) then
                    begin
                        u ← t(1) t(4);
                        insert(u, T)
                    end
            end
end
```

Fig. 5. Program to compute composition.

Theorem 3. The set of functions computable by programs with t bound to the current value of R in **for** T **in** R **do** and parallel interpretation of **for** loops is coextensive with the functions computed by relational algebra expressions.

7.2. R Not Bound Before the Loop; Parallel Execution

There is the possibility that R changes within a **for** loop, and if so, we might wish to consider the possibility that the set associated with t in the loop **for** t **in** R **do** changes as R changes. One way to effect such changes might be to modify the set associated with t after an assignment, insertion or deletion to R within the loop. It is not hard to show that such an interpretation does not add to the capability of the language to produce new functions; the language is still equivalent to relational algebra.

A second interpretation is to fix the set associated with t, but to repeat the **for** loop with t associated with the set of tuples added to R at the previous repetition, until at some point no new tuples are added to R. Let us call this interpretation the *iterative-parallel* interpretation of programs.

Theorem 4. Every function expressible by the basis-induction method (method 3) of specifying fixed points can be computed under the iterative-parallel interpretation.

7.3. Serial Execution

Let us now consider interpretations of **for** t **in** R **do** <statement> that select an arbitrary ordering for the members of R and execute the <statement> once for each value of t. We may either fix the set of t's before entering the loop or we may allow the set to adapt to changes in R within the loop. The method of adaptation could be either of those suggested in Section 7.2, or we could at each iteration of the **for** loop select a t currently in R that has not been selected before. Unlike the iterative-parallel interpretation, this interpretation permits the loop to respond to deletions within the loop.

We shall not fix on a serial interpretation with R not bound on entry to the loop because our negative remarks about the serial interpretation with R bound on entry to the loop apply to all these interpretations as well, and our conclusion is that serial interpretations must be rejected as candidates for universal database languages.

Our principal objection to serial interpretations is that they can violate the first principle: order independence. It is easy to construct programs whose output is determined by the order in which tuples in some relation are considered.

While we do not advocate serial interpretations of our language as candidates for the notion of a universal data manipulation language, we shall state the following theorem which places the interpretation above into perspective.

Theorem 5. Every relational algebra expression can be computed by a program under the fixed binding, serial interpretation.

References

[AHU] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.

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

[B] F. Bancilhon, "On the completeness of query languages for relational expressions," Rapport de Recherche No. 297, IRIA, Rocquencourt, France, May, 1978.

[C1] E. F. Codd, "A Relational Model for Large Shared Data Banks," *Comm. ACM* **13**:6 (June, 1970), 377-387.

[C2] E. F. Codd, "Relational Completeness of Data Base Sublanguages," in *Data Base Systems* (R. Rustin, ed.), Prentice-Hall, Englewood Cliffs, N. J., 1972, pp. 65-98.

[Ch] A. Church, *Introduction to Mathematical Logic,* Vol. 1, Princeton University Press, 1956.

[CM] A. K. Chandra and P. M. Merlin, "Optimal Implementation of Conjunctive Queries in Relational Data Bases," *Proc. Ninth Annual ACM Symposium on Theory of Computing,* May 1976, pp. 77-90.

[D] C. J. Date, *An Introduction to Database Systems,* Addison-Wesley, Reading, Mass., 1975.

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

[P] J. Paredaens, *Information Processing Letters* 7:2, 1978.

[S] C. P. Schnorr, "An Algorithm for Transitive Closure with Linear Expected Time," *SIAM J. Computing* 7:2 (May, 1978), 127-133.

[SC] J. M. Smith and P. Y.-T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," *Comm. ACM* **18**:10 (Oct., 1975), 568-579.

[SY] Y. Sagiv and M. Yannikakis, "Equivalence Among Relational Expressions with the Union and Difference Operations," *Proc. ACM International Conference on Very Large Data Bases,* Sept., 1978.

[T] A. Tarski, "A Lattice-Theoretical Fixpoint Theorem and its Applications," *Pacific J. Mathematics* 5:2 (June, 1955), 285-309.

[Z] M. M. Zloof, "Query-by-Example: a Database Language," *IBM Syst. J.,* **16**:4 (1977), pp. 324-343.

Appendix

In this appendix, we prove that the transitive closure of a relation cannot be couched as an expression of relational algebra.† It is interesting to note that both Bancilhon [B] and Paredaens[P] in essence characterize relational algebra as equivalent to the set of **mappings** obeying principle 2 with respect to an empty set of predicates. However, transitive closure obeys this principle. There is no contradiction. In [B,P] it is shown that for every relation r there is a relational algebra expression E such that $E(R)=R^+$, the transitive closure of R. What we show is that for no relational algebra expression E is $E(R)=R^+$ for all r.

Theorem 6. For an arbitrary binary relation R, there is no expression $E(R)$ in relational algebra equivalent to R^+, the transitive closure of R.

Suppose we have an expression $E(R)$ that is the transitive closure of R. Let $\Sigma_l = \{a_1, a_2, \ldots, a_l\}$ be a set of l arbitrary symbols. Let R_l be the finite relation $\{a_1a_2, a_2a_3, \ldots, a_{l-1}a_l\}$. R_l represents the graph

We shall show that, for any relational expression E, there is some value of l for which $E(R_l)$ is not R_l^+. In particular, we shall show by induction on the number of operators in E, that $E(R_l)$ can be expressed as

$$\{b_1b_2 \cdots b_k \mid \Psi(b_1, b_2, \ldots, b_k)\}$$

where Ψ is the logical "or" of *clauses*; each clause is the logical "and" of *atoms*. An atom is a formula $b_i = c$, $b_i \neq c$, $b_i = b_j + c$ or $b_i \neq b_j + c$, where c is a (not necessarily positive) constant, and $b_j + c$ is short for "that a_m such that $b_j = a_{m-c}$." The b's are assumed to range over $\{a_1, a_2, \ldots, a_l\}$, where l is understood. The assertion $b_i = b_j + c$ says that b_i is c nodes down the chain, formed by the graph of R_l, from b_j. Note that Ψ is not precisely a predicate of relational calculus, since the b's have a fixed domain $\{a_1, \ldots, a_l\}$. Also, it should be understood that the a_i's are abstract objects and may not be ordered by arithmetic $<$ (although superficially, R_l appears to do this). That is, the expression $\sigma_{\$1<\$2}(\pi_{\$1}(R_l) \times \pi_{\$2}(R_l))$ does not compute R_l^+; rather it is meaningless, because $\sigma_{\$1<\$2}$ makes no sense when applied to a set of pairs of abstract a_i's.

The following lemma states a useful fact about logical expressions we shall manipulate.

Lemma 1. Any logical expression consisting of the logical operators $\wedge$, $\vee$, $\neg$, applied to atoms is equivalent to an expression consisting of clauses separated by **or**'s. (Such an expression is said to be in *disjunctive normal form* or *DNF*).

Proof. Any expression in propositional calculus [Ch] can be written as the disjunction of clauses consisting of the logical "and" of *literals*, which are propositional variables or their negation. The lemma follows since the negation of an atom is an atom. □

We now prove the characterization of the values that $E(R_l)$ may take for any relational algebra expression E.

Lemma 2. If E is any relational algebra expression, then for sufficiently large l,

$$E(R_l) = \{b_1 \cdots b_k \mid \Psi(b_1, \ldots, b_k)\}$$

for some k and some DNF expression Ψ, where the b_i's range over the set $\{a_1, a_2, \ldots, a_l\}$.

Proof. The proof is an induction on the number of operators in E.

Basis. Zero operators. Either the operand is R_l or a constant relation of degree 1. The relation R_l can be expressed as

$$\{b_1b_2 \mid b_2 = b_1 + 1\}$$

while the constant set $\{c_1, c_2, \ldots, c_m\}$ can be expressed as

$$\{b_1 \mid b_1 = c_1 \vee b_1 = c_2 \vee \cdots \vee b_1 = c_m\}$$

Note the c_i's must be a_j's for expression E to make sense.

Induction. All but projections are easy.

Case 1. $E = E_1 \cup E_2$, $E_1 - E_2$, or $E_1 \times E_2$. Let E_1 have value $\{b_1 \cdots b_k \mid \Psi_1(b_1 \cdots b_k)\}$ and E_2 have value $\{b'_1 \cdots b'_m \mid \Psi_2(b'_1 \cdots b'_m\}$. If $E = E_1 \cup E_2$, or $E = E_1 - E_2$, then $m = k$ for E to make sense. The value of $E_1 \cup E_2$ is

$$\{b_1 \cdots b_k \mid \Psi_1(b_1, \ldots, b_k) \vee \Psi_2(b_1, \ldots, b_k)\}.$$

The value of $E_1 - E_2$ is

$$\{b_1 \cdots b_k \mid \Psi_1(b_1, \ldots, b_k) \wedge \neg\Psi_2(b_1, \ldots, b_k)\}.$$

In this case, $\Psi_1 \wedge \neg \Psi_2$ must be transformed to DNF, which we know can be done by Lemma 1. The value of $E_1 \times E_2$ is

$$\{b_1 \cdots b_k b'_1 \cdots b'_m \mid \Psi_1(b_1, \ldots, b_k) \wedge \Psi_2(b'_1, \ldots, b'_m)\}.$$

In this case, $\Psi_1 \wedge \Psi_2$ must be transformed into DNF.

Case 2. $E = \sigma_F(E_1)$. Let the value of E_1 be $\{b_1 \cdots b_k \mid \Psi_1(b_1, \ldots, b_k)\}$. If formula F involves an arithmetic comparison other than $=$ or $\neq$, then E makes no sense, since the a_i's are not comparable by $<$. Thus F involves only $=$, $\neq$ and logical operators. We may, therefore, express E as

$$\{b_1 \cdots b_k \mid \Psi_1(b_1, \ldots, b_k) \wedge F(b_1, \ldots, b_k)\}$$

and put the resulting formula into DNF.

Case 3. $E = \pi_s(E_1)$. To begin, we can express each projection as a cascade of

(1) projections that permute the order of components, and

(2) projections that eliminate the last component only.

Let E_1 have value $\{b_1 \cdots b_k \mid \Psi(b_1, \ldots, b_k)\}$. If π_s is a permutation, then the value of E is easily obtained by permuting $b_1, \ldots, b_k$ in Ψ appropriately.

Now consider the case where π_s projects out b_k. The value of E_1 is

$$\{b_1 \cdots b_{k-1} \mid (\exists b_k)\Psi(b_1, \ldots, b_k)\}.$$

As Ψ is in DNF, we can write

† This claim appears in [D], p. 145, without proof or citation.

substitute	into		
	$b_k = a_r$	$b_m = b_k + d$	$b_k = b_m + d$
a_j	*true* if $j = r$ *false* if $j \neq r$	$b_m = a_{j+d}$ (*false* if $d \leqslant -j$)	$b_m = a_{j-d}$ (*false* if $d \geqslant j$)
$b_i - c$	$b_i = a_{r+c}$ (*false* if $c \leqslant -r$)	$b_m = b_i + (d-c)$	$b_i = b_m + (c+d)$
$b_i + c$	$b_i = a_{r-c}$ (*false* if $c \leqslant r$)	$b_m = b_i + (c+d)$	$b_i = b_m + (d-c)$

Fig. 1. Result of substitutions.

$\Psi = \Psi_1 \vee \Psi_2 \vee \cdots \vee \Psi_m$, where each Ψ_i is the logical "and" of atoms. For sufficiently large l, the value of E is $\bigcup_{i=1}^{m} \{b_1 \cdots b_{k-1} \mid (\exists b_k)\, \Psi_i(b_1, \ldots, b_k)\}$. As we can handle unions by Case 1, let us consider only the case where Ψ itself is a single clause.

Subcase 1. There is no atom of the form $b_k = a_j$, $b_i = b_k + c$ or $b_k = b_i + c$ in Ψ. Then let Ψ' be the logical "and" of all the atoms of Ψ that do not involve b_k (any such must have the $\neq$ relation). Then the value of E is $\{b_1 \cdots b_{k-1} \mid \Psi'(b_1, \ldots, b_{k-1}\}$. In proof note that for any fixed tuple $b_1 \cdots b_{k-1}$ that satisfies Ψ', we can pick b_k to be a_l for some l sufficiently large that all atoms of the forms $b_k \neq a_j$, $b_i \neq b_k + c$ or $b_k \neq b_i + c$ are true. Thus if $b_1 \cdots b_{k-1}$ satisfies Ψ', it also satisfies $(\exists b_k)\Psi(b_1, \ldots, b_k)$.

Conversely, if $b_1 \cdots b_{k-1}$ satisfies

$$(\exists b_k)\Psi(b_1, \ldots, b_k),$$

then surely $b_1 \cdots b_{k-1}$ satisfies all atoms that do not mention b_k.

Subcase 2. There is some atom $b_k = a_j$, $b_i = b_k + c$ or $b_k = b_i + c$. Substitute for b_k in all the other atoms of Ψ the expression a_j, $b_i - c$ or $b_i + c$, respectively.

The result is always equivalent to some atom, or a logical constant, which we take to be the result of the substitution as indicated in Fig. 1. An atom with the value *true* disappears. A *false* atom causes the value of E to become the empty set, which we may represent by $\{b_1 \cdots b_{k-1} \mid b_1 \neq b_1\}$. If no atom is *false,* the value of E is $\{b_1 \cdots b_{k-1} \mid \Psi'(b_1, \ldots, b_{k-1})\}$, where Ψ' consists of the logical "and" of the following atoms:

(1) The atoms of Ψ modified by the substitution of Fig. 1,

(2) In the case $b_k = b_i + c$, $c \geqslant 0$, was the substituted atom, we add the atoms $b_i \neq a_j$, for $l - c < j \leqslant l$, and

(3) In the case $b_k = b_i - c$, $c \geqslant 0$ was the substituted atom, we add the atoms $b_i \neq a_j$ for $1 \leqslant j < c$. □

Now we return to the proof of the theorem. Suppose that $E(R) = R^+$ for some expression E and any relation R. Then, for sufficiently large l, R_l^+ can be expressed as $\{b_1 b_2 \mid \Psi(b_1, b_2)\}$, where Ψ is in DNF.

Case 1. Every clause of Ψ has an atom of the form $b_1 = a_i$, $b_2 = a_i$, or $b_1 = b_2 + c$ (or equivalently, $b_2 = b_1 - c$). Consider the pair $b_1 b_2 = a_m a_{m+d}$, and where m is larger than any i such that $b_1 = a_i$ or $b_2 = a_i$ is an atom in Ψ, where d is positive and larger in magnitude than any c such that $b_1 = b_2 + c$ is an atom of Ψ. Then $b_1 = a_m$, $b_2 = a_{m+d}$ satisfies no clause of Ψ. However, for sufficiently large l, $a_m\, a_{m+d}$ is in R_l^+ but not in $E(R_l)$, a contradiction.

Case 2. Some clause of Ψ has only atoms with the $\neq$ relation. Then consider the pair $a_{m+d} a_m$, where no atom $b_i \neq a_m$ or $b_i \neq a_{m+d}$ appears in Ψ, and d is positive and larger in magnitude than any c such that $b_1 \neq b_2 + c$ or $b_2 \neq b_1 + c$ appears in Ψ. Note that from the construction of Ψ' in the case of projections in Lemma 2 that all atoms $b_i \neq a_j$ added have j either close to zero or close to l, where "close" means within some constant that depends on E but not on l. Thus, for sufficiently large l, $a_{m+d} a_m$ is in $E(R_l)$ but not in R_l^+, another contradiction.

We conclude that for any E, there is always an l for which $E(R_l) \neq R_l^+$. □