Universal Queries for Relational Query Languages

Lawrence V. Saxton
Department of Computer Science
University of Regina
Regina, SK S4S 0A2, Canada
e-mail: saxton@cs.uregina.ca

Dirk Van Gucht and Munish Gandhi
Computer Science Department
Indiana University
Bloomington Indiana 47405
email: vgucht,gandhim@cs.indiana.edu

Proposed running head: Universal queries

Lawrence V. Saxton
Department of Computer Science
University of Regina
Regina, SK S4S 0A2, Canada
e-mail: saxton@cs.uregina.ca

1 Introduction

The theoretical study of query languages began with Codd's introduction of the relational algebra and calculus for flat relational databases (Codd72). It was however quickly realized and formally established that many natural queries could not be formulated in these languages (Aho79). These limitations of first-order query languages led to the addition of limited forms of recursion into query languages. Examples of such query languages are the fixpoint queries (Aho79, Chandra82, Immerman86, Vardi82) logic programming query languages (Abiteboul-Vianu88, Abiteboul-Vianu90, Bancilhon86, Chandra85, Ullman88), and query languages with simple programming constructs (Abiteboul-Vianu90, Aho79, Chandra80, Chandra82). Parallel to these efforts was the introduction of language-independent characteristics for query languages. Notable among these are the by now accepted requirements for query languages to be 1) generic (or symmetry preserving), 2) domain preserving (or safe) and 3) typed. Furthermore, efficiency considerations led to the formal study of time and space complexity of such languages (Abiteboul-Vianu88,Abiteboul-Vianu90,Chandra82,Immerman86,Vardi82).

The requirements of more complex database applications sparked the research into data models and query languages with higher-order objects (Abiteboul et.al.89, Abiteboul-Kanellakis90, Gyssens88, Hull88, Hull89, Jaeschke82, Roth88, Thomas85). It was shown that most of the flat relational query languages could be naturally extended to these more complex data objects. Furthermore, many of the properties established for the flat query languages remained valid in this context. There were however some fundamental differences. Two notable results are 1) whereas flat relational query languages can be organized with respect to their expressive power into a strict lattice, in the world of complex objects this lattice partially collapses (Abiteboul-Beeri88, Gyssens88) and 2) without special restrictions, the time and space complexity of these languages can increase dramatically (Hull88, Hull89, Kuper88).

A less explicitly stated, but nevertheless fruitful and insightful research direction consisted of the study of query language features resembling those of general purpose programming languages. The best-known results of this kind came from the observation that many query languages often already have either implicitly or explicitly programming constructs such as variables, assignment statements, if-then statements, compound-statements, and sometimes while-statements (Abiteboul-Vianu88, Abiteboul-Vianu90, Aho79, Chandra80, Chandra82, Gyssens88, VandenBussche93). The most extensive and recent example occurs in (Vanden-

Bussche93) where the reflection construct is added to relational algebra. Probably, lesser-known results came from the observation that many query languages shared with general programming languages the property that their queries could be transformed into equivalent queries in a certain normal form. For example, it was shown by Immerman (Immerman86) (see also (Leivant89)) that a fixpoint query can be rewritten into an equivalent fixpoint query in which the least fixpoint constructor appears only once[1]. These results are of course analogous to Kleene-Normal Form of (partial) recursive functions (Kleene52).

In view of these results, it appears natural to continue the investigation into the similarities of query languages and general purpose programming languages. In this paper we take up this issue by looking at one of the most powerful aspect of programming languages, namely their ability to interpret other languages. Specifically we ask *whether query languages can be interpreted by queries of other query languages*? To show this possibility, we introduce the concept of *universal queries*; we will say that a query is a universal query for a query language L_1 if it can emulate the effect of an arbitrary query of L_1. These results will of course be more interesting if we can show that such queries belong to query languages of limited expressiveness, since it is well-known that Turing-complete languages have a universal property directly. The main results of this paper will be an affirmative answer to the above question. We will first show how a (relational) database instance and a (relational) query in L_1 can be encoded as an unique database instance. This encoding is reminiscent of the approach of Ross, (Ross92), who proposed a model and an algebra where relations can contain relation names. Using these encodings we will show the existence of a universal query u_1 for the flat relational algebra, and a universal query u_2 for the while-loop queries with typed flat relation variables. The query u_1 will be shown to be a for-loop query with typed nested relation variables of set-height two (which is therefore also equivalent to a powerset algebra and a nested calculus query of set-height two (Abiteboul-Beeri88, Gyssens88, Hull88). The query u_2 will be shown to be a while-loop query with typed nested relation variables of set-height two. By a result in (Gyssens88) this query can also be written as an (extended) powerset algebra query of set-height two. We will further discuss the existence of universal queries for flat relational algebra queries which use up to k attributes, and for while-loop queries with typed nested relation variables of set-height at most $h \geq 0$.

We believe that the existence of universal queries offers yet another technique in the

[1]Similar result are known for datalog (Chandra80) the while-loop queries with typed flat relation variables (Abiteboul-Vianu88) and the while-loop queries with typed nested relation variables (Gyssens91)

measurement of the relative expressiveness of query languages.

The paper is organized as follows. In Section 2 we give the definition of nested relations and databases. In Section 3 we describe the various query languages we will deal with in this paper. In Section 4, we will show the existence of the universal queries u_1 and u_2. In Section 5 we discuss the existence of universal queries for other query languages.

2 Basic Definitions

Since nested relations are more general than flat relations, we will state our definitions in terms of nested relations. Our definitions are based on (Beeri90, Gyssens88).

2.1 Nested Relations

Assume an infinitely enumerable set U of *elementary attributes*. Attributes are either elementary or composite, where a composite attribute is a set of elementary or composite attributes.

Definition 1 *The set of all attributes $\mathcal{U}$ is the smallest set containing U such that every finite subset X of $\mathcal{U}$, in which no elementary attribute appears more than once, is in $\mathcal{U}$.*

Elements of U are called *elementary*; those of $\mathcal{U} - U$ are called *composite* (or relation-valued). We denote elementary attributes by $A, B, C, \ldots$, composite attributes by $X, Y, Z, \ldots$, and general attributes, also called *types*, by $T_1, T2, \ldots$. A *relation scheme* Ω is a composite attribute, i.e. an element of $\mathcal{U} - U$. The structural complexity of types is measured by the depth of set constructs in their definition. The *set-height* of a type T, denoted $sh(T)$, is defined as 0 for $T \in U$, and $1 + max\{sh(X) : X \in T\}$ for $T \in \mathcal{U}$.

Next we define simultaneously the notions of value, tuple and instance. Assume an infinitely enumerable set V of *elementary values*.

Definition 2 *The set $\mathcal{V}$ of all values, the set $\mathcal{I}_X$ of all instances over $X \in \mathcal{U} - U$, the set $\mathcal{T}_X$ of all tuples over $X \in \mathcal{U} - U$, and the set $\mathcal{I}$ of all instances are the smallest sets satisfying:*

- $\mathcal{V} = V \cup \mathcal{I}$;
- $\mathcal{I} = \bigcup_{X \in \mathcal{U} - U} \mathcal{I}_X$;
- $\mathcal{I}_X$ *consists of all finite subsets of* $\mathcal{T}_X$;
- $\mathcal{T}_X$ *consists of all mappings* t *from* X *into* $\mathcal{V}$*, called tuples, such that* $t(A) \in V$ *for all* $A \in X \cap U$ *and* $t(Y) \in \mathcal{I}_Y$ *for all* $Y \in X - U$.

Definition 3 *A (nested) relation is a pair* (Ω, ω) *where* $\Omega \in \mathcal{U} - U$ *and* $\omega \in \mathcal{I}_\Omega$. Ω *is called the scheme of the relation and* ω *is called the instance of the relation. The set of all relations with scheme* Ω *will be denoted by* $\mathcal{R}_\Omega$*, and the set of all relations will be denoted by* $\mathcal{R}$.

Assume an infinitely enumerable set N of *relation names*. A (nested) *database scheme* is a sequence $\Delta = [R_1 : \Omega_1, \ldots, R_n : \Omega_n]$, where $R_i \in N$, and Ω_i is a relation scheme. A (nested) *database instance* over Δ is a sequence $\delta = [R_1 : \omega_1, \ldots, R_n : \omega_n]$, where $\omega_i \in \mathcal{I}_{\Omega_i}$. The set of databases instances over Δ is denoted $\mathcal{I}_\Delta$. A (nested) *database* is a pair (Δ, δ), where $\delta \in \mathcal{D}_\Delta$. The set of all databases over Δ will be denoted by $\mathcal{D}_\Delta$. The set of all databases will be denoted by $\mathcal{D}$.

The traditional (non-nested) relational model consists of the restriction of relation schemes to sets of elementary attributes. In the sequel, we will call the traditional relational model the *flat* relational model, and refer to the above defined concepts in its context with the adjective flat.

2.2 Queries

The *active domain* of a tuple t (relation instance ω, database instance δ), denoted $adom(t)$ ($adom(\omega)$, $adom(\delta)$), is the set of elementary values appearing in t (ω, δ).

Definition 4 *A query from a database scheme* Δ_{in} *to a database scheme* Δ_{out}*, denoted* $q : \Delta_{in} \rightarrow \Delta_{out}$[2]*, is a (partial) mapping from* $\mathcal{D}_{\Delta_{in}}$ *to* $\mathcal{D}_{\Delta_{out}}$*, such that for some finite set* $C \subset V$ *the following holds:*

[2]In the case where Δ_{out} consists of a single component $[R_{out} : \Omega_{out}]$, we will also talk about a query from a database scheme Δ_{in} to a relation scheme Ω_{out}.

- *q is domain preserving w.r.t C: $\forall \delta \in \mathcal{I}_{\Delta_{in}}$, $adom(q(\delta)) \subseteq adom(\delta) \cup C$;*
- *q is C-generic: for each permutation σ over V (extended in the natural way to $\mathcal{I}$ and $\mathcal{V}$), such that σ is the identity on C and $q \circ \sigma = \sigma \circ q$.*

Two query languages L_1, L_2 are *equivalent*, if each query expressible in L_1 is expressible in L_2 and vice versa.

3 Relational query languages

We will consider several relational query languages. In particular, for relational databases we will consider the flat relational algebra, the flat relational calculus (Codd72). and the while-loop and for-loop queries with *typed* flat relation variables (Chandra80). For nested relational databases we will consider the powerset algebra, the nested calculus, the while-loop and for-loop queries with *typed* nested relation variables.

3.1 Algebraic query languages

Algebraic query languages for (nested) relations are obtained by extending the flat relation algebra operators to deal with (nested) relations, and adding the restructuring operators unnest and powerset.

Definition 5
- *The classical relational operators of union ($\cup$), difference ($-$), cartesian product ($\times$) and projection (π). Cartesian product is applicable only to relations whose schemes are built from disjoint sets of elementary attributes. Required renaming of attributes is performed by the rename operator. If (Ω, ω) is a relation and T an attribute in Ω then the renaming of T by T' in (Ω, ω) is denoted $\rho_{T \to T'}(\Omega, \omega)$.*
- *Selection of tuples from a relation (Ω, ω) is defined relative to a predicate Ψ on tuples as $\sigma_\Psi(\Omega, \omega) = (\Omega, \{t : t \in \omega \wedge \Psi(t) = \texttt{true}\})$. We consider the following predicates: Elementary attribute equality, $A = B$, for A, $B \in \Omega \cap U$; Elementary attribute-constant equality, $A = a$, for $A \in \Omega \cap U$ and $a \in V$; Composite attribute equality, $X = Y$, for compatible attributes X, $Y \in \Omega - U$; Composite attribute-constant equality, $X = x$, for $X \in \Omega - U$ and $x \in \mathcal{I}_X$.*

- *Let $X \in \Omega - U$. The unnesting $\mu_X(\Omega, \omega)$ equals (Ω', ω') where $\Omega' = (\Omega - \{X\}) \cup X$ and $\omega' = \{t \in \mathcal{T}_{\Omega'} | \exists t' \in \omega : t$ restricted to $\Omega - \{X\}$ equals t' restricted to $\Omega - \{X\}$ and t restricted to X is an element of $t'(X)\}$.*
- *Let 2^ω denote the set of all subsets of ω. The powerset $\Pi(\Omega, \omega)$ equals $(\{\Omega\}, 2^\omega)$.*

To avoid extensive use of brackets, we assume the following precedence on these operators:

- unary operators;
- cartesian product;
- set operators.

The *powerset algebra* $\mathcal{P}$ is defined by expressions built from typed relation variables and constant relations using the operators above. The *flat relational algebra* is the subset of $\mathcal{P}$ built with flat relation variables and flat relation constants not using powerset.

3.2 A calculus for nested relations

The calculus uses typed variables ranging over tuples. The *terms* are constants, variables, and expressions of the form $x.Z$, where x is a tuple variable and $Z \in \mathcal{U}$. The *atomic formulas* are (well-typed) expressions of the form $t_1 = t_2$, $t_1 \in t_2$, or $R(t_1)$ where t_1, t_2 are terms and R is a typed relation name. Formulas are built using connectives and quantifiers in the usual manner. A *calculus query* from a database schema $\Delta = [R_1 : \Omega_1, \ldots, R_n : \Omega_n]$ to a relation scheme Ω (see footnote 1) is an expression $\{y|\Phi\}$ where Φ is built from the relation names $R_1, \ldots, R_n$ in Δ, has only y as a free tuple variable, and y has the type Ω.

Clearly, a calculus query defines a generic mapping with domain $\mathcal{D}_\Delta$. However, this mapping need not be domain preserving. We therefore also consider the notion of *domain preserving* calculus queries, and we will call the *nested calculus* the set of domain preserving calculus queries. A fundamental result in the theory of nested relation query languages is:

Theorem 1 *(Abiteboul-Beeri88) The powerset algebra and the nested calculus are equivalent.*

```
                  <query>  →  <constant>*<statement>* <final-statement>
               <constant>  →  <typed-relation-name>"←"(<constant-relation> |
                                  <typed-relation-name>)";"
              <statement>  →  <statement-number> (<assignment-statement> |
                              <if-statement> | <loop-statement> | <compound-statement>)
   <assignment-statement>  →  <typed-relation-variable>"←"<subquery>";"
           <if-statement>  →  "if"<boolean-expression>"then<statement>
         <loop-statement>  →  <while-statement> | <for-loop>
        <while-statement>  →  "while"<boolean-expression><statement>
               <for-loop>  →  "for" "|" <typed-relation-variable>"|"<statement>
     <compound-statement>  →  "{"<statement>*"}"
              <sub-query>  →  <powerset-algebra-query> | <nested-calculus-query>
     <boolean-expression>  →  <typed-relation-variable><comparison-operator>
                              (<typed-relation-constant> | <typed-relation-variable>)
    <comparison-operator>  →  "="|"≠"|"⊆"
        <final-statement>  →  "Result" "←" <sub-query>"."
```

Figure 1: The syntax for the while-loop query language

3.3 Query languages with programming constructs

To add expressiveness to the flat relational algebra and calculus, several extensions to these query languages have been introduced (for an excellent survey, see (Chandra88)). In particular, Chandra and Harel introduced a (typed) query language which incorporate the standard features of an imperative programming language (Chandra80), i.e. *typed* relation variables and constants, (correctly typed) assignment statements, if-then statements, loop statements, and compound statements[3]. Similar extensions for nested relational query languages were introduced in (Gyssens88). The precise syntax for these queries is shown in Figure 3.3.

The set of queries satisfying the syntax of Figure 3.3 will be called the *while-loop query language*. The subset of queries in which no while-statement occurs will be called the *for-loop query language*. Since while-loop queries in general define *partial* functions, we also consider the subset of *total while-loop queries*.

[3]Chandra and Harel also introduced an *untyped* version of this language. This untyped language is considerably more expressive than the corresponding typed one. In fact, in (Chandra80) it is shown that the untyped language can express all computable flat relational queries!

As an example, assume that $\Delta = [R : \{A, B\}]$ is a (flat) relational database scheme. Then the following (total) while-loop query computes the transitive closure of R.

```
$Empty \leftarrow \emptyset$;
$New \leftarrow R$;
$TC \leftarrow R$;
**while** $New \neq Empty$
{
    $TC \leftarrow TC \cup \rho_{A' \to A} \pi_{A',B} (\sigma_{B'=A} (\rho_{A \to A'} \rho_{B \to B'} TC \times R))$;
    $New \leftarrow TC - New$;
}
$Result \leftarrow TC$.
```

This particular while-loop query can also be written as a for-loop query.

```
$TC \leftarrow R$;
**for** $|R|$
    $TC \leftarrow TC \cup \rho_{A' \to A} \pi_{A',B} (\sigma_{B'=A} (\rho_{A \to A'} \rho_{B \to B'} TC \times R))$;
$Result \leftarrow TC$.
```

The following result relates the various nested relational query languages.

Theorem 2 *(Gyssens88) The powerset algebra (and therefore, by Theorem 1, also the nested calculus), the total while-loop query language and the for-loop query language are equivalent*[4].

To take into account that while-loop queries are in general only partial functions, the powerset algebra is extended with an operator which when applied to the empty set yields an undefined result. With this extension, the following result holds.

Theorem 3 *(Gyssens88) The extended powerset algebra and the while-loop query language are equivalent.*

[4]It should be noted that a similar result does *not* hold for flat relational query languages (Chandra88).

4 Universal queries

One of the main properties of general purpose programming languages is their ability to interpret other programming languages. This observation of course dates back to Turing's proof about the existence of Universal Turing Machines (Turing37). Another closely related result to this is the existence of universal (partial) recursive functions. More precisely, if e is an encoding for the (partial) recursive functions then there exists a $n+1$-ary function u_{n+1} such that for each n-ary function f, and each set of n natural numbers $m_1, \ldots, m_n$, one has that $u_{n+1}(e(f), m_1, \ldots, m_n) = f(m_1, \ldots, m_n)$, where $e(f)$ denotes the natural number corresponding to the encoding of f (see e.g. (Hennie77,Kleene52)).

Clearly, we can't expect such strong properties for the query languages we introduced in the previous section. In fact, to show such strong properties, we would need query languages that can express all computable queries (Abiteboul88,Chandra80). We will however show that even in the context of these much weaker languages, there is the notion of interpreting a query language by a query in another query language. To show this, we will prove the existence of universal queries for various query languages. What makes our result interesting is that that these universal queries themselves belong to query languages much weaker than the class of all computable queries. We begin with the definition of encodings and universal queries.

Definition 6 *Let L_1, L_2 be query languages. A query $u_{(L_1,L_2)}$ in L_2 is a universal query for L_1, or, $u_{(L_1,L_2)}$ interprets L_1, if there exist database schemes Δ^{db}, Δ^{q}, a database encoding mapping de and a query encoding mapping qe with:*

- *de a computable one-to-one mapping from the set of databases for queries in L_1 to the set of databases over Δ^{db}, i.e. $de : \mathcal{D}_{L_1} \rightarrow \mathcal{D}_{\Delta^{db}}$;*
- *qe a computable mapping from the set of queries in L_1 to the set of databases over Δ^{q}, i.e. $qe : L_1 \rightarrow \mathcal{D}_{\Delta^{q}}$;*
- *the input scheme of $u_{(L_1,L_2)}$ is the combination of Δ^{q} and Δ^{db}, denoted by $[query : \Delta^{q}, inputdb : \Delta^{db}]$, and its output scheme is Δ^{db},*

such that for each query $q : \Delta_{in} \rightarrow \Delta_{out}$ in L_1 and each database $d \in \mathcal{D}_{\Delta_{in}}$ the following equation holds:

$$de^{-1}(u_{(L_1,L_2)}([query : qe(q), inputdb : de(d)])) = q(d)$$

with de^{-1} a partial computable mapping from $\mathcal{D}_{\Delta^{db}}$ to $\mathcal{D}_{L_1}$ such that de^{-1} is the inverse of de on the range of de and undefined everywhere else.

Thus, the query $u_{(L_1,L_2)}$ when presented with input the encoding of the query q and the encoding of the database d yields as result (when decoded by de^{-1}) the result of applying the query q directly to d; i.e. $u_{(L_1,L_2)}$ is an *interpreter* for the language L_1.

It ought to be stressed that neither de nor qe are themselves queries of either L_1 or L_2. Also note that since de is a one-to-one mapping, its inverse, de^{-1}, is well-defined and is also a one-to-one mapping.

4.1 A universal query for the flat relational algebra

In this section we will prove that there exists a universal for-loop query with *nested* relation variables (and therefore, by Theorem 2, also a powerset algebra and a nested calculus query) for the flat relational algebra[5]. More precisely, we will specify a flat database encoding fde, a flat relational algebra query encoding $faqe$, and a for-loop query $u_{(FRA,\mathbf{for}LQ)}$ such that for each flat relational algebra query faq and each (input) flat database fd, we have that

$$fde^{-1}(u_{(FRA,\mathbf{for}LQ)}([query : faqe(faq), inputdb : fde(fd)])) = faq(fd)$$

In other words we will prove the following theorem.

Theorem 4 *The flat relational algebra can be interpreted by a for-loop query with nested relation variables (and therefore by Theorem 2 also by a total while-loop query, a powerset algebra query and a the nested calculus query). Furthermore, this query is of set-height two.*

We will begin with the flat database encoding fde. The best way to illustrate this encoding is to consider an example. Let $\Delta_1 = [R_1 : \{A, B\}, R_2 : \{B, C, D\}, R_2 : \{A, E\}]$ be a flat relational database scheme and let $\delta_1 = [R_1 : r_1, R_2 : r_2, R : r_3]$ be the flat relational database shown in Figure 4.1. Notice that r_3 is empty. The encoding of this database, $fde(\Delta_1, \delta_1)$, is shown in Figure 4.1. As can be seen, the encoding of a flat database is of a single *nested* relation with scheme $\{RelationName, \{Attribute, Value\}\}$. For each tuple in the flat database there is a corresponding tuple in this nested relation. We will therefore from now on refer to the composite attribute $\{Attribute, Value\}$ as the $Tuple$ attribute.

r_1

A	B
0	0
1	1
0	1

r_2

B	C	D
0	5	0
1	2	1

r_3

A	E

Figure 2: The flat relational database δ_1 (r_3 is empty)

Relation-name	{ Attribute	Value }
R_1	A	0
	B	0
R_1	A	1
	B	1
R_1	A	0
	B	1
R_2	B	0
	C	5
	D	0
R_2	B	1
	C	2
	D	1

Figure 3: The encoding $fde(\Delta_1, \delta_1)$

$$\begin{array}{lll} 1 & C_1 \leftarrow & \begin{array}{ccc} B' & C & D \\ \hline 0 & 1 & 0 \\ 1 & 2 & 0 \end{array} ; \\ & & \\ 2 & S_1 \leftarrow \rho_{B \to B'}(R_2); & \\ 3 & S_2 \leftarrow R_1 \times S_1; & \\ 4 & S_3 \leftarrow \sigma_{B=B'}(S_2); & \\ 5 & S_4 \leftarrow \pi_{B',C}(S_3); & \\ 6 & S_5 \leftarrow \rho_{B \to B'}(R_2); & \\ 7 & S_6 \leftarrow S_4 \cup S_5; & \\ & & \\ & \mathit{Result} \leftarrow S_6 - C_1. & \end{array}$$

Figure 4: A loop-less query equivalent to query q_1

Next, we consider the flat relational algebra query encoding $faqe$. To simplify matters, given a flat relational algebra query, we will first transform this query into an equivalent *loop-less* for-loop query. Consider the following relational algebra query q_1.

$$(\Pi_{B',C}(\sigma_{B=B'}(R_1 \times \rho_{B \to B'}(R_2))) \cup \rho_{B \to B'}(R_2)) \quad - \quad \begin{array}{ccc} \text{B'} & \text{C} & \text{D} \\ \hline 0 & 1 & 0 \\ 1 & 2 & 0 \end{array}$$

The query shown in Figure 4.1 is an equivalent (loop-less) for-loop query. Notice how there is a one-to-one correspondence between the operators in the relational algebra query and the assignment statements in the (loop-less) for-loop query.

The encoding of this (loop-less) for-loop query, i.e. $faqe(q_1)$, is shown in Figure 4.1. It is built from two nested relations: the first nested relation, denoted $aqe(q_1)$, corresponds to the actual query. The second nested relation, denoted $qce(q_1)$ is the encoding of the constant relations occurring in q_1[6].

Ross (Ross92) developed a model and an algebra for storing relation names directly into a relation. His α operator, suitably generalized and extended to allow attribute names, could

[5]In Section 4.2 we will strengthen this result.

[6]The encoding technique for the constants is identical to the encoding technique for flat databases.

PSN	SN	R	$TypeR$	Opr	$UOpd$	$LOpd$	$Ropd$	$Pars$	LP	RP
1	2	S_1	$\{B', C, D\}$	ρ	R2	na	na	$\emptyset$	B	B’
2	3	S_2	$\{A, B, B', C, D\}$	$\times$	na	R1	S1	$\emptyset$	na	na
3	4	S_3	$\{A, B, B', C, D\}$	σ	S2	na	na	$\emptyset$	B	B’
4	5	S_4	$\{B', C, D\}$	π	S3	na	na	{B’,C}	na	na
5	6	S_5	$\{B', C, D\}$	ρ	R2	na	na	$\emptyset$	B	B’
6	7	S_6	$\{B', C, D\}$	$\cup$	na	S4	S1	$\emptyset$	na	na
7	8	$Result$	$\{B', C, D\}$	$-$	na	S_6	C_1	$\emptyset$	na	na

The encoding aqe of the query q_1

Relation-name	{	Attribute	Value	}
C_1		B'	0	
		C	1	
		D	0	
C_1		B'	1	
		C	2	
		D	0	

The encoding qce of the constant appearing in q_1

Figure 5: The encoding $faqe$ of the query q_1

be used to provide an encoding for the database instance, similar to our encoding. He shows that adding his operator to relational algebra would not drastically increase the expressive power of the language. However, the encoding of the query which follows and the decoding function are not available in his language.

The encoding of the query, i.e. $aqe(q_1)$, consists of a nested relation with scheme {Previous-Statement-Number (PSN), Statement-Number (SN), Relation-Name (R), Type-of-R ($TypeR$), Operator (*Opr*), Unary-Operand (*UOpd*), Left-Operand (*LOpd*), Right-Operand (*ROpd*), {Projection-Attribute} ({*Patt*}, abbreviated *Patts*), Left-Parameter (LP), Right-Parameter (RP)}. Each tuple in $aqe(q_1)$ corresponds to an assignment statement, say *astat*, q_1. PSN denotes the statement number of the statement preceding *astat*. SN denotes the statement number of *astat*. R is the relation variable name to which the result of *astat* is assigned. $TypeR$ denotes the type of the relation variable name R. *Opr* denotes the single operator involved in *astat*. If *Opr* is a unary operator, $UOpd$ gives the relation name of its operand (otherwise, $UOpd$ is set to "na"; we assume w.l.o.g. that "na" is a distinguished value in V). If *Opr* is a binary operator, *LOpd* (*ROpd*) gives its left (right) operand (otherwise, *LOpd* (*ROpd*) is set to "na"). If *Opr* is the projection operator, *Patts* is the *set* of attributes over which the projection is applied (otherwise *Patts* is set to $\emptyset$). If *Opr* is the renaming (selection[7]) operator, $LPar$ ($RPar$) gives the attribute name of its left (right) parameter (otherwise $LPar$ (*Rpar*) is set to "na").

The relation $eqa(q_1)$ has two important properties:

1. It is possible to retrieve from it the tuple corresponding to the top statement of the query q_1. This is because this tuple in $aqe(q_1)$ is the only one with its particular PSN value. In fact the following query retrieves the top statement in $aqe(q_1)$.

$$\{t|\ qe(q_1)(t)\ \wedge\ \neg\ \exists\ t'(qe(q_1)(t')\ \wedge\ t'.SN = t.PSN)\}$$

2. There exists an implicit order among its tuples corresponding to the order that exists among the statements in the query q_1. This order is implied by the matches that exists between PSN and SN values of the tuples in $aqe(q_1)$.

[7]We will make the simplifying assumption that the boolean expression involved in the selection is of the form $A = B$ where A and B are attribute names. It can easily be shown that this is not a restriction of the expressive power of the flat relational algebra.

We are now ready to specify the universal for-loop query $u_{(FRA,\mathbf{for}LQ)}$ such that for each flat relational algebra query faq and each (input) flat relational database fd, we have that

$$fde^{-1}(u_{(FRA,\mathbf{for}LQ)}([query : faqe(faq), inputdb : fde(fd)])) = faq(fd)$$

The query $u_{(FRA,\mathbf{for}LQ)}$ takes as input a nested database with scheme $[query : FlatQuery,$ $inputdb : Inputdb]$. $Inputdb$ is simply the scheme of $fde(fd)$. $FlatQuery$ is a scheme with two composite attributes: the first, denoted by AQE, corresponds to the scheme of $aqe(faq)$, the second, denoted by QCE, corresponds to the scheme of $qce(faq)$. The encoding $faqe(faq)$ is then the nested relation over scheme $FlatQuery = \{AQE, QCE\}$ and contains the single tuple $(AQE : aqe(faq), QCE : qce(faq))$.

The query $u_{(FRA,\mathbf{for}LQ)}$ itself is essentially a single loop which at each iteration selects the current top statement in $aqe(faq)$, applies this statement to appropriate intermediate results and adds its result to the intermediate results. This loop ends when all statements in $aqe(faq)$ have been selected and applied. In Figure 4.1, we show the appropriate initializations and the main loop of $u_{(FRA,\mathbf{for}LQ)}$.

The code for `TOP-STATEMENT(`$Query$`)` is

$$\{t|\ Query(t)\ \wedge\ \neg\ \exists\ t'\ (Query(t')\ \wedge\ t'.SN = t.PSN)\}$$

Note that this query, when applied to a valid query encoding results in a singleton set containing a statement (tuple).

We now specify the statements to implement `APPLY(`$Statement$`,`IRs`)`. Essentially, these statements form a case statement controlled by the value of the operator (specified by the Opr attribute) in $Statement$.

```
        Initialize the set of intermediate results ($IRs$) with $Inputdb$
1       $IRs \leftarrow Inputdb$;

        Initialize the variable $Query$ to the encoding
        of the input query stored in $FlatQuery$
2       $Query \leftarrow \mu_{QE}\pi_{QE}(FlatQuery)$;

        Add the constants occurring in the input query constants to $IRs$
2       $IRs \leftarrow IRs \cup \mu_{QCE}\pi_{QCE}(FlatQuery)$;

        Enter the main loop of $u_{(FRA,\mathbf{for}LQ)}$ and execute each statement (tuple)
        in $Query$
4       for $|Query|$
5       {
          Retrieve the top statement of $Query$
6         $Statement \leftarrow$ TOP-STATEMENT($Query$);
          Remove the top statement of $Query$
7         $Query \leftarrow Query - Statement$;

          Apply $Statement$ to $IRs$; the result will be assigned to $Temp$
8         APPLY($Statement$,$IRs$);
          Add $Temp$ to $IRs$
9         $IRs \leftarrow IRs \cup Temp$;
        }

        Select the result of applying $faq$ to $fd$
9       $Result \leftarrow \sigma_{RelationName=Result}(IRs)$.
```

Figure 6: The main loop of $u_{(FRA,\mathbf{for}LQ)}$

```
8.1     $Temp \leftarrow \emptyset;$

        Select the operator in $Statement$
8.2     $Operator \leftarrow \pi_{Opr}(Statement);$

        Consider the various cases for $Operator$
8.3     **if** $Operator = \{\rho\}$ **then** RENAME($Statement, IRs$);
8.4     **if** $Operator = \{\pi\}$ **then** PROJECT($Statement, IRs$);
8.5     **if** $Operator = \{\sigma\}$ **then** SELECT($Statement, IRs$);
8.6     **if** $Operator = \{\cup\}$ **then** UNION($Statement, IRs$);
8.7     **if** $Operator = \{-\}$ **then** DIFFERENCE($Statement, IRs$);
8.8     **if** $Operator = \{\times\}$ **then** PRODUCT($Statement, IRs$);
```

The statement corresponding to `RENAME`$(Statement, IRs)$ is:

```
8.2.1   $Temp \leftarrow \{\ t \mid\ \exists\ s\ (Statement(s)\ \wedge\ t.RelationName = s.R \wedge$
                $\exists\ i\ (IRs(i)\ \wedge\ i.RelationName = s.Uopd\ \wedge$
                    $\forall\ j\ ((\exists\ k\ (k \in i.Tuple\ \wedge\ k.Attribute \neq s.LP\ \wedge\ j = k))\ \vee$
                        $(\exists\ k\ (k \in i.Tuple\ \wedge\ k.Attribute = s.LP\ \wedge$
                            $j.Attribute = s.RP\ \wedge\ j.Value = k.Value))$
                        $\Longleftrightarrow$
                        $j \in t.Tuple)))\}$
```

The statement corresponding to `PROJECT`$(Statement, IRs)$ is:

```
8.3.1   $Temp \leftarrow \{\ t \mid\ \exists\ s\ (Statement(s)\ \wedge\ t.RelationName = s.R \wedge$
                $\exists\ i\ (IRs(i)\ \wedge\ i.RelationName = s.Uopd\ \wedge$
                    $\forall\ j\ ((\exists\ k\ (k \in i.Tuple\ \wedge\ k = j\ \wedge\ k.Attribute \in s.Patts$
                        $\Longleftrightarrow$
                        $j \in t.Tuple)))\}$
```

The statement corresponding to `SELECT`$(Statement, IRs)$ is:

```
8.4.1   $Temp \leftarrow \{\ t \mid\ \exists\ s\ (Statement(s)\ \wedge\ t.RelationName = s.R \wedge$
                $\exists\ i\ (IRs(i)\ \wedge\ i.RelationName = s.Uopd\ \wedge$
                    $\exists\ j_1\ \exists\ j_2$
                        $(j_1 \in i.Tuple\ \wedge\ j_2 \in i.Tuple\ \wedge$
                          $j_1.Attribute = s.LP\ \wedge\ j_2.Attribute = s.RP\ \wedge$
                          $j1.Value = j2.Value)\ \wedge$
                    $t.Tuple = i.Tuple))\}$
```

The statement corresponding to UNION$(Statement, IRs)$ is:

8.5.1 $$\begin{aligned} Temp \leftarrow \{\, t \mid\ & \exists\, s\ (Statement(s)\ \wedge\ t.RelationName = s.R \wedge (\\ & \exists\, i_1\ (IRs(i_1)\ \wedge\ i_1.R = s.LOpd\ \wedge\ t.Tuple = i_1.Tuple) \vee \\ & \exists\, i_2\ (IRs(i_2)\ \wedge\ i_2.R = s.ROpd\ \wedge\ t.Tuple = i_2.Tuple)))\} \end{aligned}$$

The statement corresponding to DIFFERENCE$(Statement, IRs)$ is:

8.5.1 $$\begin{aligned} Temp \leftarrow \{\, t \mid\ & \exists\, s\ (Statement(s)\ \wedge\ t.RelationName = s.R \wedge (\\ & \exists\, i_1\ (IRs(i_1)\ \wedge\ i_1.R = s.LOpd\ \wedge\ t.Tuple = i_1.Tuple) \wedge \\ & \neg(\exists\, i_2\ (IRs(i_2)\ \wedge\ i_2.R = s.ROpd\ \wedge\ t.Tuple = i_2.Tuple))))\} \end{aligned}$$

The statement corresponding to PRODUCT$(Statement, IRs)$ is:

8.6.1 $$\begin{aligned} Temp \leftarrow \{\, t \mid\ & \exists\, s\ (Statement(s)\ \wedge\ t.RelationName = s.R \wedge \\ & (\exists\, i_1\ \exists\, i_2\ (IRs(i_1)\ \wedge\ IRs(i_2)\ \wedge \\ & \quad i_1.RelationName = s.LOpd\ \wedge\ i_2.RelationName = s.ROpd\ \wedge \\ & \quad\quad \forall\, j\ (j \in t.Tuple \Longleftrightarrow \\ & \quad\quad\quad j \in i_1.Tuple\ \vee\ j \in i_2.Tuple))))\} \end{aligned}$$

This completes the description of the universal query $u_{(FRA,\mathbf{for}LQ)}$ and therefore also the proof of Theorem 4.

4.2 A universal query for the flat while-loop queries

It turns out that Theorem 6 can be substantially strengthened. In this section we will shown that there exists a universal nested while-loop query (and therefore also an (extended) powerset algebra query) for the flat while-loop queries.

To show this result, we will make the syntactic restriction that the flat while-loop queries to be interpreted are in a certain *normal form*. This normal form requires that there is at most one loop statement in the flat while-loop query under consideration.[8] This syntactic restriction does *not* decrease the expressive power since the following results holds.

[8]This normal form is closely related to the Kleene normal form of (partial) recursive functions which states that in its formulation a single application of the minimization operator suffices (Kleene52).

Theorem 5 *(Abiteboul-Vianu88) The set of flat while-loop queries and the set of flat while-loop queries containing at most one loop-statement are equivalent.*[9]

More precisely, we will specify a flat database encoding fde, a flat while-loop query encoding $fwqe$, and a while-loop query $u_{(FWLQ,WLQ)}$ such that for each flat while-loop query fwq and each (input) flat relational database fd, we have that

$$fde^{-1}(u_{(FWLQ,WLQ)}([query : fwqe(fwq), inputdb : fde(fd)])) = fwq(fd)$$

In other words we will prove the following theorem.

Theorem 6 *The set of flat while-loop queries can be interpreted by a (nested) while-loop query (and therefore by Theorem 3 also by an extended powerset algebra query). Furthermore, this query is of set-height two.*

To prove this theorem we can keep the encoding for flat databases fde introduced in Section 4.1. However, we will need adjustments to the encoding $faqe$. Assume that fwq is a flat while-loop query with at most one while-loop statement. The new encoding for fwq, denoted $fqwe(fwq)$, is built from *three* nested relations. The first nested relation, denoted $qwe(fwq)$, will correspond to the body of the complete query. The second nested relation, denoted $we(fwq)$, will correspond to the single while-loop statement[10] in fwq. (The scheme for these two nested relation is identical to the scheme for the corresponding construction in Section 4.1.) The third nested relation, denoted $cwe(fwq)$, will encode the constants in fwq.

To illustrate the encoding fwq, reconsider the query to compute the transitive closure of a binary (flat) relation introduced in Section 3.3. The encoding of this query is shown in Figure 4.2, provided we make two simple changes to the original query[11]

1. The statements 2, 3 and 8 in the program, i.e.

[9]It turns out that there are variants of this theorem for datalog (Chandra82), the least fixed point queries (Immerman86,Leivant89) and the nested while-loop queries (Gyssens91).

[10]We will make the additional assumption that the boolean expression controlling this while-loop is of the form $R_1 \theta R_2$ (with θ either $=$, $\neq$, or $\subseteq$)where R_1 and R_2 are composite attribute names, but not constants. It can be shown that this is not a limiting assumption and it will make the encoding problem slightly more straightforward.

[11]These changes are merely technical and are done to make the encoding simpler.

```
2      $New \leftarrow R$;
3      $TC \leftarrow R$;

8      $Result \leftarrow TC$;
```

are replaced by the following three statements:

```
2      $New \leftarrow R \cup R$;
3      $TC \leftarrow R \cup R$;

8      $Result \leftarrow TC \cup TC$;
```

So now each statement involves a single (although redundant) operation.

2. The statement 6, i.e.

```
6      $TC \leftarrow TC \cup \rho_{A' \to A} \pi_{A',B}(\sigma_{B'=A}(\rho_{A \to A'} \rho_{B \to B'} TC \times R))$;
```

is replaced by the following assignment statements:

```
6.1    $S_1 \leftarrow \rho_{A \to A'}(TC)$;
6.2    $S_2 \leftarrow \rho_{B \to B'}(S_1)$;
6.3    $S_3 \leftarrow S_2 \times R$;
6.4    $S_4 \leftarrow \sigma_{B'=A}(S_3)$;
6.5    $S_5 \leftarrow \pi_{A',B}(S_4)$;
6.6    $TC \leftarrow \rho_{A' \to A}$;
```

We are now ready to specify the while-loop query $u_{(FWLQ,WFQ)}$ such that for each flat while-loop query fwq and each (input) flat relational database fd, we have that

$$fde^{-1}(u_{()}([query : fwqe(fwq), inputdb : fde(fd)])) = fwq(fd)$$

The query $u_{(FWLQ,WLQ)}$ takes as input a database with scheme $[query : FlatWhileQuery$, $inputdb : Inputdb]$. $Inputdb$ is simply the scheme of $fde(fd)$. $FlatWhileQuery$ is a scheme with three composite attributes: the first corresponds to the scheme of $wqe(fwq)$ and is denoted by WQE, the second corresponds to the scheme of $we(fwq)$ and is denoted by WE, and the third corresponds to the scheme of $cwe(fwq)$ and is denoted by CWE. The encoding $fwqe(fwq)$ is then the nested relation over $FlatWhileQuery$ and contains the single tuple $(WQE : wqe(fwq), WE : we(fwq), CWE : ce(fwq))$.

PSN	SN	R	$TypeR$	Opr	$UOpd$	$LOpd$	$Ropd$	$Pars$	LP	RP
1	2	New	$\{A, B\}$	$\cup$	na	R	R	$\emptyset$	na	na
2	3	TC	$\{A, B\}$	$\cup$	na	R	R	$\emptyset$	na	na
3	4	na	$\emptyset$	**while**	$\neq$	$Temp$	$Empty$	$\emptyset$	na	na
4	8	$Result$	$\{A, B\}$	$\cup$	na	TC	TC	$\emptyset$	na	na

The encoding of the main body of the flat while-loop query

PSN	SN	R	$TypeR$	Opr	$UOpd$	$LOpd$	$Ropd$	$Pars$	LP	RP
5	6.1	S_1	$\{A', B\}$	ρ	TC	na	na	$\emptyset$	A	A'
6.1	6.2	S_2	$\{A', B'\}$	ρ	S_1	na	na	$\emptyset$	B	B'
6.2	6.3	S_3	$\{A', B', A, B\}$	$\times$	na	S_2	R	$\emptyset$	na	na
6.3	6.4	S_4	$\{A', B', A, B\}$	σ	S_3	na	na	$\emptyset$	B'	A
6.4	6.5	S_5	$\{A', B\}$	π	S_4	na	na	$\{A', B\}$	na	na
6.5	6.6	TC	$\{A, B\}$	ρ	S_5	na	na	$\emptyset$	A'	A
6.6	7	New	$\{A, B\}$	$-$	TC	New	na	$\emptyset$	na	na

The encoding of the main body of the flat while-loop query

Figure 7: The encoding of a flat while loop query

The query $u_{(FWLQ,WLQ)}$ contains a main while-loop which at each iteration selects the current top statement in $wqe(fwq)$, applies it to the appropriate intermediate results and adds its result to the intermediate results. This loop ends when all statements in $wqe(fwq)$ have been selected. The query $u_{(FWLQ,WLQ)}$ differs from that for $u_{(FRA,\mathbf{for}LQ)}$ in essentially two ways

1. The main loop of the query $u_{(FWLQ,WLQ)}$ becomes a while loop rather than a for loop, i.e. the code for the main loop changes from the for loop of $u_{(FRA,\mathbf{for}LQ)}$:

    ```
           Enter the main loop of $u_{(FRA,\mathbf{for}LQ)}$ and execute each
           statement (tuple) in $Query$
    4      for $|Query|$
    5      { ...
           }
    ```

    into the while loop:

    ```
           Enter the main loop of $u_{(FWLQ,WLQ)}$ and execute each
           statement (tuple) in $Query$
    4      while $Query \neq \emptyset$
    5      { ...
           }
    ```

    i.e. the query $u_{(FWLQ,WLQ)}$ needs the ability to interpret a while-loop statement which might, depending on the given input query, potentially loop forever.

We are now ready to describe the query $u_{(FWLQ,WLQ)}$ in detail. Reconsider the query $u_{(FRA,\mathbf{for}LQ)}$ in Section 4.1. Replace the statement

```
       Initialize the variable $Query$ to the encoding
       of the input query stored in $FlatQuery$
3      $Query \leftarrow \mu_{QE}\pi_{QE}(FlatQuery)$;
```

by the following statements

```
       Initialize the variable $Query$ to the encoding
       of the input query stored in $FlatWhileQuery$
3.1    $Query \leftarrow \mu_{WQE}\pi_{WQE}(FlatWhileQuery)$;
       Initialize the variable $WhileStat$ to the encoding
       of the input query stored in $FlatWhileQuery$
3.1    $WhileStat \leftarrow \mu_{WE}\pi_{WE}(FlatWhileQuery)$;
```

Secondly, add to the `APPLY(`$Statement, IR$`)` the case

```
8.9        if $Operator$ = { while } then WHILE-STATEMENT($Statement, IR$);
```

The query corresponding to the `WHILE-STATEMENT` ($Statement, IR$) is

```
                Keep a temporary copy of $WhileStat$
8.8.1           $TempWhileStat \leftarrow WhileStat$;

                Decode the boolean expression controlling
                the while-loop statement
8.8.2           $CompOpr \leftarrow \pi_{UOpd}(Statement)$;
8.8.3           $LeftOperandName \leftarrow \pi_{LOpd}(Statement)$;
8.8.4           $RightOperandName \leftarrow \pi_{ROpd}(Statement)$;

                Introduce a ``boolean'' variable $BooleanConditionSatisfied$
                to control whether an iteration of the while statement
                should be executed
8.8.5           $BooleanConditionSatisfied \leftarrow \{\texttt{false}\}$;

                Depending on the value of $CompOpr$, $LeftOperandName$
                $RightOperandName$, and $IRs$, we need to check if
                the body of while-loop should be should be executed
8.8.6           if $CompOpr = \{=\}$ then
                    CHECK-CONDITION-$=(LeftOperandName, RightOperandName, IRs)$;
8.8.7           if $CompOpr = \{\neq\}$ then
                    CHECK-CONDITION-$\neq(LeftOperandName, RightOperandName, IRs)$;
8.8.8           if $CompOpr = \{\subseteq\}$ then
                    CHECK-CONDITION-$\subseteq(LeftOperandName, RightOperandName, IRs)$;

                Apply the body of the while-loop if a CHECK-CONDITION
                set $BooleanConditionSatisfied$ equal to $\{\texttt{true}\}$
8.8.9           if $BooleanConditionSatisfied = \{\texttt{true}\}$ then
8.8.10          {
8.8.10.1            Apply the for loop statement of query $u_{(FRA,\mathbf{for}LQ)}$ but
                    with variable $WhileStat$ instead of $Query$

                    Restore the $WhileStat$ variable
8.8.10.2            $WhileStat \leftarrow TempWhileStat$;

                    Execute the while-loop statement again
8.8.10.3            $Query \leftarrow WhileQuery \cup Statement$;
                }
```

The query corresponding to CHECK-CONDITION-$=(LeftOperandName, RightOperandName, IRs)$ is

```
8.8.6.1        $LOperand \leftarrow \pi_{Tuple}(LeftOperandName \bowtie IRs)$;
8.8.6.2        $LOperand \leftarrow \pi_{Tuple}(RightOperandName \bowtie IRs)$;

8.8.6.3        **if** $LOperand = ROperand$ **then**
                   $BooleanConditionSatisfied \leftarrow \{\texttt{true}\}$
```

The cases for CHECK-CONDITION-$\neq(LeftOperandName, RightOperandName, IRs)$ and CHECK-CONDITION-$\subseteq(LeftOperandName, RightOperandName, IRs)$ are similar.

This completes the proof of Theorem 6.

5 Additional results

Although we will not prove this, one can also show that for a fixed $k \geq 2$, there is a universal query for *flat* typed while-loop queries which use up to k attributes, although the arity of the encoding relations is greater than k. What is interesting is that this universal query can be shown to be a flat while-loop query. Furthermore, it can be shown that there exists a universal query for the *nested* while-loop queries with typed nested relation variables of set-height three. Not surprisingly, this query has set-height three. We also conjecture that for a fixed $h \geq 2$ there exists a nested while loop query of set-height $h + 1$ for the set of nested while-loop queries of set-height h. In our given encoding, the encoding relation is a ternary nested relation with set height 2. It is still an open question whether there is a way to build a fixed arity encoding for the universal query which is also of set height 1 (that is, flat), although this is conjectured to be false. The above conjectures remind one of the work of (Meyer67), where they exhibited a hierarchy of program expressiveness depending on the depth of nesting.

References

Abiteboul, S. and Beeri, C. (1988), On the power of languages for the manipulation of complex objects. Technical report, INRIA, 1988.

Abiteboul, S., Beeri, C., Gyssens, M., and Van Gucht, D. (1989), An introduction to the completeness of languages for complex objects and nested relations. In S. Abiteboul and H.J. Schek, editors, *Nested Relations and Complex Objects, LNCS 361*, pages 117–138. Springer-Verlag, 1989.

Abiteboul, S. and Kanellakis, P. (1990), Database theory column: Query languages for complex object databases. *SIGACT News*, 76:9–18, 1990.

Abiteboul, S. and Vianu, V. (1988), Datalog extensions for database queries and updates. Technical Report INRIA-900, INRIA, 1988.

Abiteboul, S. and Vianu, V. (1990), Procedural and declarative database update languages. *JCSS*, 41:181–229, 1990.

Aho, A.V. and Ullman, J.D. (1979), Universality of data retrieval languages. In *Proc. of 6th ACM Symposium on Principles of Programming Languages*, pages 110–117, 1979.

Bancilhon, F. and Ramakrishnan, R. (1986), An amateur's introduction to recursive query-processing strategies. In *Proceedings of ACM-SIGMOD '86 International Conference on Management of Data, Washington, D.C.*, pages 16–52, 1986.

Beeri, C. and Kornatzky, Y. (1990), The many faces of query monotonicity. In *Proc. of Advances in Database Technology-EDBT '90*, pages 120–135, 1990.

Chandra, A. (1988), Theory of database queries. In *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Austin*, pages 1–9, 1988.

Chandra, A. and Harel, D. (1980), Computable queries for relational data bases. *JCSS*, 21:156–178, 1980.

Chandra, A. and Harel, D. (1982), Structure and complexity of relational queries. *JCSS*, 25:99–128, 1982.

Chandra, A. and Harel, D. (1985), Horn clause queries and generalizations. *J. of Logic*

Programming, 1:1–15, 1985.

Codd, E.F. (1972) Relational completeness of database sublanguages. In R. Rustin, editor, *Database Systems.* Prentice-Hall, Englewood Cliffs, 1972.

Gyssens, M. and Van Gucht, D. (1991), A Comparison between Algebraic Query Languages for Flat and Nested Databases, *Theoretical Computer Science*, 87 (1991), pp. 263–286.

Gyssens, M. and Van Gucht, D. (1988), The powerset algebra as a result of adding programming constructs to the nested relation algebra. In *Proceedings of ACM-SIGMOD 1988 Annual Conference, Chicago*, pages 225–232, 1988.

Hennie, F. (1977), *Introduction to computability.* Addison-Wesley, Reading, Mass., 1977.

Hull, R. and Su, J. (1988), On the expressive power of database queries with intermediate types. In *Proc. of 7th ACM Symposium on Principles of Database Systems*, pages 39–51, 1988.

Hull, R. and Su, J. (1989), Untyped sets, invention, and computable queries. In *Proc. of 8th ACM Symposium on Principles of Database Systems*, pages 347–360, 1989.

Immerman, N. (1986), Relational queries computable in polynomial time. *Inform. and Comp*, 68:86–104, 1986.

Jaeschke, G.H. and Schek, H.J. (1982), Remarks on the algebra of non first normal form relations. In *Proceedings of the First ACM Symposium on Principles of Database Systems*, pages 124–138, 1982.

Kleene, S.C. (1952). *Introduction to metamathematics.* Amsterdam (North-Holland Pub. Co.), 1952.

Kuper, G.M. and Vardi, M.Y. (1984), A new approach to database logic. In *Proceedings of 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 86–96, 1984.

Kuper, G.M. and Vardi, M.Y. (1988), On the complexity of queries in the logical data model. In *Proc. of the 2nd International Conference on Database Theory*, pages 267–280, 1988.

Leivant, D. (1989), Inductive definitions over finite structures. Technical Report CMU-CS-89-153, CMU, 1989.

Meyer, A.R. and Ritchie, D.M. (1967), The complexity of loop programs. In *Proc. of 22nd National Conference, ACM*, pages 465–469, 1967.

Roth, M.A., Korth, H.F. and Silberschatz, A. (1988), Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–419, 1988.

Ross, K. (1992), Relations with relation names as arguments: algebra and calculus. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 346–353, 1992.

Thomas, S.J. and Fischer, P.C. (1985), Nested relational structures. In P. C. Kanellakis, editor, *Advances in Computing Research, Volume 3: The Theory of Databases*, pages 269–307. JAI Press, 1985.

Turing, A.M. (1937), On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 42:230–265, 1937.

Ullman, J.D. (1988), *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Rockville, MD, 1988.

Vardi, M.Y. (1982) Complexity and relational query languages. In *Proc. of 14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.

Van den Bussche, J., Van Gucht, D. and Vossen, G. (1993), Reflective programming in the relational algebra. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1993.