6. Computable Database Queries and the Expressiveness of the Relational Algebra

One of the fundamental operations a database system needs to carry out is that of processing queries. The relational algebra was defined in Section 3.2 of Chapter 3 as a yardstick for the expressiveness of a query language for relational databases. The relational algebra on its own is not intended to be used as a general purpose database programming language for developing applications and as such does not provide the application programmer with iteration and recursion facilities. Thus in the context of database application programming the question that arises is: what are the possible queries that such a database language can and should be able to compute? In this chapter we define and investigate a general notion of a computable query in an attempt to answer the question we have just posed. There are essentially two parts to our investigation. The first part involves a categorisation of several subclasses of computable queries and the second part involves the presentation of a database programming language that is complete with respect to computable queries and an investigation of the expressive power of the relational algebra and how it can be made more expressive by adding to it iteration and/or recursion facilities. Since the early 1980's it was realised that the relational algebra is not expressive enough to carry out general database computations. The research into computable queries has been instrumental in motivating the necessity to develop more expressive query languages for relational databases and laying down the fundamental principles which provide the foundations for such development.

6.1 What is a Computable Database Query?

As we have shown in Subsection 3.2.1 of Chapter 3 there are many useful queries that the relational algebra cannot express such as computing the transitive closure of a relation and counting the number of tuples in a relation. In fact, from a computational complexity point of view the relational algebra is equivalent to the set of problems checkable in constant time on a concurrent parallel random access machine [Imm89]. This computational complexity class is very weak and is properly included in the deterministic logspace computational complexity class [Imm81, Var82a, Imm87]. Thus it is natural to investigate extensions of the relational algebra in order to enhance its computational expressiveness. In particular, our aim is to formalise the notion of a *computable query*.

There are two fundamental differences between computable queries and Turing-computable mappings (also known as *partially recursive functions*). The first difference is

that Turing-computable mappings are mappings from strings to strings (or alternatively, from natural numbers to natural numbers) whilst computable queries are mappings from finite sets of objects to finite sets of objects (cf. [Gan80]). The second difference is that the objects of computable queries are not strings or numbers but abstract objects (cf. [Fri71]), in our case records (cf. [CM91]), which are defined as total mappings from a finite set of attributes to a set of domain values.

We define the semantics of a computable query as a Turing-computable mapping together with an encoding from finite sets of records (or simply sets) to strings. Consequently, we can evaluate a computable query by encoding the input set into a string, then use this string as an input to a given Turing-computable mapping, and finally decode the resulting output string to obtain the output set, which yields the result of the computable query. This analysis provides clarification of the notion of a computable query by dealing with the problem of how a database language can be implemented on a standard Turing machine that does not cater directly for mappings from sets to sets. Thus, intuitively, a database query language is *computationally query complete* (or simply query complete) if it can express the set of all computable queries.

Once the formal definition of a computable query is established in Section 6.2 we investigate several important subclasses of computable queries in the following section, i.e. Section 6.3. Thereafter in Section 6.4 we further our understanding of computable queries by looking into the Turing-computable mappings that realise a given computable query. In Section 6.5 we define the notion of a database language being query complete, and present the database query language, QL, from [CH80]. In Section 6.6 we present a fundamental characterisation of the expressiveness of the relational algebra. Since the ability to express all the computable queries may be more than is needed from a database query language, in Section 6.7 we describe how the relational algebra can be made more expressive without making it query complete by adding to it a looping mechanism. The motivation for such an extension of the relational algebra is that it may be desirable to enhance the expressiveness of the relational algebra, say to compute queries such as the transitive closure of a relation (see Subsection 3.2.1 in Chapter 3), without making it query complete, since database users rarely require query complete database languages.

6.2 Formalising Computable Database Queries

For the purpose of this chapter we will consider a slightly different model of a relational database. Instead of viewing a database as a set of relations with each relation being a set of tuples, we will view a database as being a set of *records* [CM91].

We first recall from Definition 3.1 in Section 3.1 that $\mathcal{U}$ is the countably infinite universe of attributes and that $\mathcal{D}$ is the countably infinite underlying database domain.

Definition 6.1 (A record and a database) Let X be a finite subset of the set of attributes $\mathcal{U}$. An X-*record* (or simply a record whenever X is understood from context) is a total mapping, t, from X into $\mathcal{D}$ such that

$$\forall A \in X, t(A) \in DOM(A).$$

In the case when X = $\emptyset$, we take t to be the empty mapping corresponding to the *empty record*, which is denoted by <>. We denote an X-record, t, where X = $\{A_1, A_2, \ldots, A_m\}$ and $\forall i \in \{1, 2, \ldots, m\}$, $t(i) = v_i$, with $v_i \in \text{DOM}(A_i)$, by

$$<(A_1 : v_1), (A_2 : v_2), \ldots, (A_m : v_m)>.$$

In the following we will abbreviate $(A_i : v_i)$ by $A_i : v_i$.

Let RECS denote the countably infinite set of all finite sets of records, where a set of records may contain X-records and Y-records, where X $\neq$ Y. Then a *database of records* (or simply a database) is a member of RECS. ■

The reader can verify that every relational database can be translated into a database of records and vice versa; we can assume without loss of generality that whenever R_i and R_j are two distinct relation schemas then schema(R_i) $\neq$ schema(R_j). The notational advantage of databases of records is that the schema of the database is encoded in the database itself and, in addition, the database can be viewed as a single relation containing variable length records [LL95a].

Example 6.1 In Table 6.1 we show a database of records, say EMP. The translation of EMP into a relational database d comprising three relations r_1, r_2 and r_3 is shown in Tables 6.2, 6.3 and 6.4. The semantics of EMP are: an employee has a NAME, earns a SALary, works in one DEParTment and may have at most one SPouSe. In addition, a DEPT has one ManaGeR and a MGR has one SECretary. ■

Table 6.1 The database of records EMP

{<DEPT : Computing,	NAME : Iris,	SAL : 20>,	
<DEPT : Computing,	NAME : Reuven,	SAL : 25,	SPS : Hanna>,
<DEPT : Computing,	NAME : Brian,	SAL : 30,	SPS : Annette>,
<DEPT : Maths,	NAME : Naomi,	SAL : 22,	SPS : Sophia>,
<DEPT : Maths,	MGR : Naomi,	SEC : Sophia>,	
<DEPT : Computing,	MGR : Brian,	SEC : Rachel>,	
<DEPT : Philosophy,	MGR : Dan,	SEC : Naomi>}	

Table 6.2 The relation r_1

DEPT	NAME	SAL
Computing	Iris	20

Table 6.3 The relation r_2

DEPT	NAME	SAL	SPS
Computing	Reuven	25	Hanna
Computing	Brian	30	Annette
Maths	Naomi	22	Sophia

Table 6.4 The relation r_3

DEPT	MGR	SEC
Maths	Naomi	Sophia
Computing	Brian	Rachel
Philosophy	Dan	Naomi

We let CHAR be a finite and nonempty set of characters (the alphabet) and STR be the countably infinite set of strings over CHAR. Furthermore, we let TC denote the set of all Turing-computable mappings (also known as *partially recursive functions*) from STR to STR; see Subsection 1.9.4 of Chapter 1 for the necessary background material from the theory of computing. (Recall that we denote the composition of two mappings g and f by the juxtaposition fg of f and g.)

6.2.1 Encodings and Decodings

Herein we investigate in depth the semantics of encodings, an area which has been hitherto neglected in the database literature, and then formally define a computable query by making use of encodings. Informally, an encoding of a set of records consists of two components: an ordering function, which orders the records in the set as well as the values of each record in the set, and an isomorphism, which maps the values in the records of the set to strings. A decoding, which is the inverse of an encoding, also has two components: the inverse of the said isomorphism and a mapping which forgets the order imposed by the aforesaid ordering function used in the encoding. Since, in general, there may not be an algorithm to convert any two encodings into each other, we restrict the set of encodings to a set of *mutually convertible* encodings, which is a set of encodings that are algorithmically convertible to a given standard encoding [GJ79]. It follows that our (restricted set of) encodings are "reasonable" in the sense of [GJ79]. An important class of encodings, called *free encodings*, whose isomorphism maps record values to a corresponding natural string representation, is also defined. In a free encoding the isomorphism has the same semantics as the identity mapping on record values.

Definition 6.2 (Encoding) An encoding is a mapping from RECS which maps each database of records S $\in$ RECS to an (ordered) list of (ordered) lists of ordered pairs.

More specifically, an encoding ϱ is a mapping from RECS to a restricted subset of STR, which is the composition $\theta\phi$ of an *ordering function*, ϕ, together with a *naming function*, θ.

Given an input S the ordering function, ϕ, converts each record in S into a list of ordered pairs and then orders the resulting set of lists of pairs into a further list as follows:

1) if $t = <A_1 : v_1, A_2 : v_2, \ldots, A_m : v_m>$, with $t \in$ S, then $\phi(<A_1 : v_1, A_2 : v_2, \ldots, A_m : v_m>) = [(B_1, w_1), (B_2, w_2), \ldots, (B_m, w_m)]$, such that $A_i = B_j$ and $v_i = w_j$ if and only if $(A_i : v_i)$ is mapped onto the jth ordered pair in ϕ(t), where $i, j \in \{1, 2, \ldots, m\}$; and
2) if S $= \{t_1, t_2, \ldots, t_n\}$ then ϕ(S) $= [u_1, u_2, \ldots, u_n]$, with $\phi(t_i) = u_j$, is mapped as described in (1), if and only if t_i is mapped onto the jth record in ϕ(S), where $i, j \in \{1, 2, \ldots, n\}$.

The naming function, θ, is a one-to-one mapping that converts the attributes and values of records in S into strings in STR which do not contain any delimiters in the fixed set, {[,],(,),,}. The mapping θ is extended to ϕ(S) as follows:

$$\theta([(A_1, v_1), (A_2, v_2), \ldots, (A_m, v_m)]) = [(\theta(A_1), \theta(v_1)), (\theta(A_2), \theta(v_2)), \ldots, (\theta(A_m), \theta(v_m))].$$

We denote the set of encodings of databases in RECS by ENC and denote the range of ENC by LISTS; we call the elements of LISTS *databases of lists.* ■

Definition 6.3 (Decoding) A decoding is a mapping from LISTS to RECS which maps a database of lists in LISTS onto a database of records in RECS.

More specifically, a decoding, ϱ^{-1}, is a mapping from LISTS to RECS, which is the composition $\gamma\theta^{-1}$ of the inverse θ^{-1} of a naming function θ together with the *forgetful function*, γ.

The forgetful function γ maps a database of lists into a set of records simply by ignoring or forgetting the ordering imposed by the list structure representing the records in the database. Furthermore, the inverse mapping θ^{-1} converts all the strings in a database of lists to their original attributes and values, thus yielding an ordered database of records.

We denote the set of decodings of databases of lists in LISTS by DEC. Given an encoding $\varrho = \theta\phi \in$ ENC, $\varrho^{-1} = \gamma\theta^{-1} \in$ DEC is called its decoding. ■

We now define a special encoding, called a *standard encoding*, which allows us to formalise the notion of what we call a *reasonable encoding.*

Definition 6.4 (A standard encoding) Assume that χ is an ordering function corresponding to some fixed lexicographical ordering of the set $\mathcal{U} \cup \mathcal{D}$ of attributes and domain values. Furthermore, assume that ι is a naming function that maps the attributes and values in $\mathcal{U} \cup \mathcal{D}$ into some fixed values, which are considered to be their *natural representation*, that is, $\forall$A $\in$ $\mathcal{U}$, ι(A) = "A" and $\forall v \in \mathcal{D}$, $\iota(v)$ = "v". The encoding $\iota\chi$ is called a *standard encoding.* ■

We need to restrict the above definition of encodings so that we consider only encodings which can be algorithmically converted to a standard encoding.

Definition 6.5 (Mutually convertible encodings) An encoding $\varrho \in$ ENC is said to be *mutually convertible* to the encoding $\iota\chi$ if both the compositions $\varrho(\iota\chi)^{-1}$ (observe that $(\iota\chi)^{-1} = \gamma\iota^{-1}$) and $(\iota\chi)\varrho^{-1}$ are Turing-computable mappings. ■

From now on we will assume that all the encodings in ENC *are mutually convertible to* $\iota\chi$. We leave the proof of the following proposition to the reader.

Proposition 6.1 All encodings $\varrho_1, \varrho_2 \in$ ENC are mutually convertible. □

Thus all encodings are equivalent in the sense that they can all be algorithmically converted into each other. It is also useful to insist that any encoding of a database of records can be converted into another one in polynomial time in the size of the input database of records, but such a restriction is unnecessary in our context. In practice, it is also important that encodings be precise in the sense that naming functions, θ, do not *pad* the input database of records S with extraneous characters.

A free encoding is one which maps attributes and values to their natural representation. Free encodings are useful, since users will easily be able to interpret their output.

Definition 6.6 (Free encoding) An encoding $\varrho = \theta\phi$ is said to be *free* if $\theta = \iota$. ■

Example 6.2 In Table 6.5 we show a free encoding of the database of records, EMP, shown in Table 6.1 and in Table 6.6 we show an encoding of EMP which is not free. ■

Table 6.5 A free encoding of EMP

[[(DEPT, Computing),	(NAME, Iris),	(SAL, 20)],	
[(DEPT, Computing),	(NAME, Reuven),	(SAL, 25),	(SPS, Hanna)],
[(DEPT, Computing),	(NAME, Brian),	(SAL, 30),	(SPS, Annette)],
[(DEPT, Maths),	(NAME, Naomi),	(SAL, 22),	(SPS, Sophia)],
[(DEPT, Maths),	(MGR, Naomi),	(SEC, Sophia],	
[(DEPT, Computing),	(MGR, Brian),	(SEC, Rachel)],	
[(DEPT, Philosophy),	(MGR, Dan),	(SEC, Naomi)]]	

Table 6.6 An encoding of EMP which is not free

[[(1, COMPUTING),	(2, IRIS),	(3, 20)],	
[(1, COMPUTING),	(2, REUVEN),	(3, 25),	(4, HANNA)],
[(1, COMPUTING),	(2, BRIAN),	(3, 30),	(4, ANNETTE)],
[(1, MATHS),	(2, NAOMI),	(3, 22),	(4, SOPHIA)],
[(1, MATHS),	(5, NAOMI),	(6, SOPHIA)],	
[(1, COMPUTING),	(5, BRIAN),	(6, RACHEL)],	
[(1, PHILOSOPHY),	(5, DAN),	(6, NAOMI)]]	

6.2.2 Definition of Computable Database Queries

We begin this subsection with an example of some computable database queries.

Example 6.3 The following queries over the database of records EMP of Example 6.1, shown in Table 6.1, are intuitively computable.

1) Project EMP onto the set of attributes {DEPT, NAME}.
2) Select from EMP the records whose DEPT-value is Computing.
3) Return the nth record in EMP (with respect to an encoding of EMP) if $|\text{EMP}| \geq n$, otherwise return {<>}.
4) Select from EMP the records, t, where t has n attributes.
5) Return {<>} if $|\text{EMP}| \geq n$, otherwise return $\emptyset$.
6) Select from EMP all records t_i such that there exists another record t_j in EMP such that $t_i \neq t_j$ and both t_i and t_j contain a common attribute value pair, say $(A : v)$. ■

Informally, a computable database query τ is a mapping from RECS to RECS that can be computed via a Turing-computable mapping δ from LISTS to LISTS by encoding the input database of records S to τ via an encoding ϱ and decoding the output database of lists from δ via the decoding ϱ^{-1}. Figure 6.1 shows the commutative diagram describing the semantics of a computable database query, where CQ denotes the set of all computable database queries.

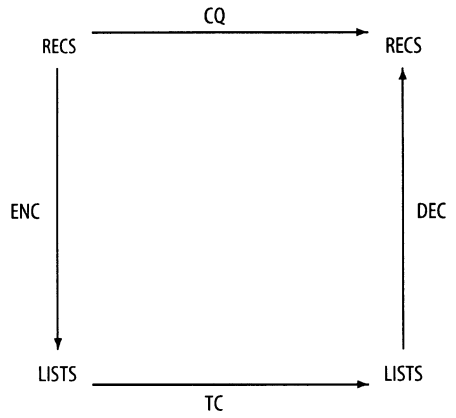

Fig 6.1 Commutative diagram describing the semantics of CQ

Definition 6.7 (Computable database query) A mapping τ from RECS to RECS is a *computable database query* (or simply a computable query or a query) if

$$\exists \delta \in \text{TC}, \exists \varrho \in \text{ENC} \text{ such that } \tau = \varrho^{-1}\delta\varrho.$$

As an abbreviation to the above equation we say that τ is *realised* via δ and ϱ; at times we simply say that τ is realised via δ meaning that $\exists \varrho \in \text{ENC}$ such that τ is realised via δ and ϱ. ■

Example 6.4 The reader can verify that all the queries given in Example 6.3 are in fact computable according to Definition 6.7. ■

The next proposition shows that if a computable query τ is realised via δ and ϱ then for any other encoding $\varrho_i \in \text{ENC}$ we can effectively find a Turing-computable mapping δ_i such that τ is realised via δ_i and ϱ_i.

Proposition 6.2 Let $\tau \in \text{CQ}$. Then $\forall \varrho_i \in \text{ENC}, \exists \delta_i \in \text{TC}$ such that τ is realised via δ_i and ϱ_i.

Proof. Assume that τ is realised via δ and ϱ. Let $\delta_i = \varrho_i\varrho^{-1}\delta\varrho\varrho_i^{-1}$. By Proposition 6.1 and the fact that TC is closed under composition of mappings it follows that $\delta_i \in \text{TC}$. Therefore, τ is realised via δ_i and ϱ_i as required. □

Our definition of a computable query differs from the standard definition given in [CH80] and [AV90], wherein a computable query is defined directly as a computable mapping from sets to sets without detailing the encoding and decoding process. Moreover, another nonstandard feature of a computable query is that, in addition to encoding attribute values of tuples, we also encode the attribute names of values. Research in the area of computable database queries can be found in [CH80, VS89, AV90, AV91a, AV91b, DM92, HS93, Saz93, Van93a, LL96a].

6.3 Subclasses of Computable Database Queries

Herein we define various subclasses of computable queries and investigate the relationships between these subclasses.

6.3.1 Order-Independent Computable Queries

Informally, a computable database query τ that is realised via $\varrho = \theta\phi$ is order-independent if its computation does not depend on the ordering function ϕ. That is, intuitively τ is order-independent if the diagram shown in Figure 6.1 commutes for all encodings $\varrho_i = \theta_i\phi_i \in$ ENC with $\theta = \theta_i$.

Prior to defining order-independent computable queries we define order-independence as a property of the Turing-computable mapping δ that realises τ.

Definition 6.8 (Order-independent Turing-computable mapping) A Turing-computable mapping $\delta \in$ TC is *order-independent* if

$$\forall \varrho_i = \theta\phi_i \in \text{ENC}, \forall \varrho = \theta\phi \in \text{ENC}, \delta\varrho_i = \delta\varrho.$$

Hereafter we let OITC denote the set of all order-independent Turing-computable mappings. ■

We observe that in the above definition $\varrho_i^{-1} = \gamma\theta^{-1} = \varrho^{-1}$ holds and thus it is true that $\varrho_i^{-1}\delta\varrho_i = \varrho^{-1}\delta\varrho$.

Definition 6.9 (Order-independent computable query) A computable query $\tau \in$ CQ is *order-independent* if $\exists\delta \in$ OITC such that τ is realised via δ.

Hereafter we let OI denote the set of all order-independent computable queries. ■

We observe that in Example 6.3 only query (3) is not order-independent and thus OI is a proper subset of CQ.

6.3.2 Isomorphism-Independent Computable Queries

Informally, a computable database query τ that is realised via $\varrho = \theta\phi$ is isomorphism-independent if its computation does not depend on the naming function θ. That is, intuitively τ is isomorphism-independent if the diagram shown in Figure 6.1 commutes for all encodings $\varrho_i = \theta_i\phi_i \in$ ENC with $\phi = \phi_i$.

Prior to defining isomorphism-independent computable queries we define isomorphism-independence as a property of the Turing-computable mapping δ that realises τ.

Definition 6.10 (Isomorphism-independent Turing-computable mapping) A Turing-computable mapping $\delta \in$ TC is *isomorphism-independent* if

$$\forall \varrho_i = \theta_i\phi \in \text{ENC}, \forall \varrho = \theta\phi \in \text{ENC}, \theta_i^{-1}\delta\theta_i = \theta^{-1}\delta\theta.$$

Hereafter we let IITC denote the set of all isomorphism-independent Turing-computable mappings. ■

We observe that in the above definition it is true that $\varrho_i^{-1}\delta\varrho_i = \varrho^{-1}\delta\varrho$.

Definition 6.11 (Isomorphism-independent computable query) A computable query $\tau \in$ CQ is *isomorphism-independent* if $\exists\delta \in$ IITC such that τ is realised via δ.

Hereafter we let II denote the set of all isomorphism-independent computable queries. ■

We observe that in Example 6.3 only queries (1) and (2) are not isomorphism-independent and thus II is a proper subset of CQ.

6.3.3 Encoding-Independent Computable Queries

Informally, a computable database query τ that is realised via ϱ is encoding-independent if its computation does not depend on any particular encoding ϱ. That is, intuitively τ is encoding-independent if the diagram shown in Figure 6.1 commutes for all encodings $\varrho \in$ ENC.

In essence encoding-independence with respect to computable queries means that, in practice, the same query executed on two distinct machines, whether they be different or not, will yield the same result irrespective of how the query is represented internally within each machine.

Definition 6.12 (Encoding-independent computable query) A computable query $\tau \in$ CQ is *encoding-independent* if

$$\exists\delta \in \text{TC such that } \forall\varrho \in \text{ENC}, \tau = \varrho^{-1}\delta\varrho.$$

Hereafter we let EI denote the set of all encoding-independent computable queries. ■

We observe that in the above definition it is true that $\forall\varrho_i, \varrho_j \in$ ENC, $\varrho_i^{-1}\delta\varrho_i = \varrho_j^{-1}\delta\varrho_j$. Therefore, EI is a subset of both OI and II. In fact, EI is a proper subset of both OI and II, since in Example 6.3 queries (1) and (2) are in OI but not in EI and query (3) is in II but not in EI. We now show that EI is the intersection of OI and II.

Theorem 6.3 EI = OI ∩ II.

Proof. EI ⊆ OI ∩ II, since EI ⊂ OI and EI ⊂ II as noted above.

It remains to show that OI ∩ II ⊆ EI. Let $\tau \in$ OI ∩ II be a computable query which is both order-independent and isomorphism-independent. Now, let $\delta \in$ IITC be a Turing-computable mapping and $\varrho \in$ ENC be an encoding such that τ is realised via δ and ϱ; δ and ϱ exist due to that fact that $\tau \in$ II. Let δ' be the mapping $\delta\varrho\varrho^{-1}$. Now, $\delta' \in$ TC, since by Proposition 6.1 ϱ is mutually convertible to itself. Furthermore, $\delta' \in$ OITC, since the mapping $\varrho\varrho^{-1}$ effectively reorders the encoded input so that it be suitable for δ. Thus τ is realised via δ'.

In order to conclude the proof we need to show that $\forall\varrho_i, \varrho_j \in$ ENC, $\varrho_i^{-1}\delta'\varrho_i = \varrho_j^{-1}\delta'\varrho_j$. That is, we need to show that τ is realised via δ' and any encoding ϱ in ENC. Let $\varrho = \theta\phi$, $\varrho_i = \theta_i\phi_i$

and $\varrho_j = \theta_j\phi_j$; $\varrho_i^{-1}\delta'\varrho_i = \gamma\theta_i^{-1}\delta\theta\phi\gamma\theta^{-1}\theta_i\phi_i = \gamma\theta_j^{-1}\delta\theta\phi\gamma\theta^{-1}\theta_j\phi_i$, since by the definition of IITC $\delta' \in$ IITC due to the fact that $\delta \in$ IITC. Thus, $\gamma\theta_j^{-1}\delta\theta\phi\gamma\theta^{-1}\theta_j\phi_i = \gamma\theta_j^{-1}\delta\theta\phi\gamma\theta^{-1}\theta_j\phi_j = \varrho_j^{-1}\delta'\varrho_j$, since $\delta' \in$ OITC. The result that $\tau \in$ EI follows as required. □

An interesting corollary to the above proof is that any isomorphism-independent computable query τ can be made to be encoding-independent by "hard-wiring" the ordering function into the Turing-computable mapping that realises τ.

A *generic* computable query is one which commutes with every one-to-one mapping from records to records, which is the composition of an encoding and a decoding. The next definition formalises this notion.

Definition 6.13 (Generic computable query) A computable query τ is generic if $\eta\tau = \tau\eta$, where $\eta = \varrho_1^{-1}\varrho_2$ and $\varrho_1, \varrho_2 \in$ ENC. ■

The notion of encoding-independence is closely related to the notion of genericity as the following theorem asserts.

Theorem 6.4 A computable query, τ, is encoding-independent if and only if τ is order-independent and generic.

Proof. If. Suppose that τ is realised via δ and $\varrho_1 \in$ ENC, where $\delta \in$ OITC. We need to show that $\tau = \varrho_1^{-1}\delta\varrho_1 = \varrho_2^{-1}\delta\varrho_2$, where $\varrho_2 \in$ ENC.

Let $\eta = \varrho_1^{-1}\varrho_2$. Then $\eta^{-1}\tau\eta = \tau$, since τ is generic. Therefore $\tau = \varrho_2^{-1}\varrho_1\varrho_1^{-1}\delta\varrho_1\varrho_1^{-1}\varrho_2$. Thus $\tau = \varrho_2^{-1}\delta\theta_2\phi$, since $\varrho_2^{-1}\varrho_1\varrho_1^{-1} = \varrho_2^{-1}$ and $\varrho_1\varrho_1^{-1}\varrho_2 = \theta_2\phi$ for some ordering function ϕ, where $\varrho_2 = \theta_2\phi_2$. The result that $\tau = \varrho_2^{-1}\delta\varrho_2$ follows, since $\delta \in$ OITC.

Only if. Suppose that τ is realised by δ and ϱ. Thus, by the definition of encoding-independence $\tau = \varrho^{-1}\delta\varrho = \eta^{-1}\varrho^{-1}\delta\varrho\eta$. The result follows, since $\eta\tau = \eta\eta^{-1}\varrho^{-1}\delta\varrho\eta = \varrho^{-1}\delta\varrho\eta = \tau\eta$. □

We next summarise the benefits of encoding-independent queries.

- The order of records in the database does not influence the result of the query.
- The result of the query is independent of the representation of the attributes and values in the database.

The concept of encoding-independence is related to the concept of data independence discussed in Section 1.6 of Chapter 1. If a database query is in EI, then the result of the query is unaffected by the physical representation of the database.

We close this section by indicating how the concept of encoding-independence can be weakened to allow a finite set C of attributes and values to be mentioned explicitly in a query.

Recall from Definition 6.4 that the naming function ι maps attributes and values to their natural representation. An encoding $\varrho = \theta\phi$ is said to be a C-*encoding* if $\forall c \in$ C, $\theta(c) = \iota(c)$, i.e. any constant in C can be identified by its natural representation. Intuitively, a computable query τ is C-*encoding-independent* if the diagram shown in Figure 6.1 commutes for all

C-encodings. When C = ∅ then the notion of C-encoding-independent reduces to the notion of encoding-independent. We denote the set of all C-encoding-independent computable queries by C-EI.

The reader can verify that, with respect to Example 6.3, query (1) is {DEPT, NAME}-encoding-independent and query (2) is {DEPT, Computing}-encoding-independent. In practice we would require C to include at least the attribute names and domain values in the database over which we are querying; that is, with respect to the latter, C should include at least all the constants in the active domain of the database. We leave it to the reader to verify that all the results in Sections 6.2 and 6.3 could be recast in terms of C-encodings rather than general encodings.

6.4 An Equivalence Relation on Computable Queries

The aim of this section is to add to our understanding of computable queries by investigating an equivalence relation on the set CQ of computable queries; two such queries are related if they are realised via the same Turing-computable mapping, say δ. In particular, we are interested in the cardinality of the equivalence class of a computable query, realised via δ and some encoding, with respect to the said equivalence relation; this equivalence class is denoted by Δ_δ. In the case when $|\Delta_\delta| = 0$, there is no computable query τ such that τ is realised via δ and some encoding. On the other hand, when $|\Delta_\delta| = 1$ and τ is realised via δ and all possible encodings in ENC, then τ corresponds to an encoding-independent computable query.

Definition 6.14 (Δ_δ) Let $\delta \in$ TC. Then $\Delta_\delta = \{\tau \mid \exists \varrho \in$ ENC such that τ is realised via δ and $\varrho\}$. That is, Δ_δ is the set of all computable queries that are realised via δ and some encoding. ■

The following theorem is proved using Rice's theorem (see Theorem 1.2 in Subsection 1.9.4 of Chapter 1).

Theorem 6.5 The two decision problems: is $|\Delta_\delta| = 0$? and is $|\Delta_\delta| = 1$? are undecidable.

Proof. On using Rice's theorem we need to show that the sets, CL_0 and CL_1, of Turing-computable mappings, δ, such that $|\Delta_\delta| = 0$ and $|\Delta_\delta| = 1$, respectively, are both nontrivial. That is, $CL_i \neq \emptyset$ and $CL_i \neq$ TC, for $i = 0, 1$. We prove the result by exhibiting $\delta_0, \delta_1 \in$ TC, such that $\delta_0 \in CL_0$ and $\delta_1 \in CL_1$.

Let δ_0 be a Turing-computable mapping that removes all the delimiters in the fixed set {[,],(,),„} from its input and then halts. It can easily be verified that $|\Delta_{\delta_0}| = 0$, since $\forall \varrho \in$ ENC, $\varrho^{-1}\delta_0\varrho$ is not in RECS.

Let δ_1 be a Turing-computable mapping that returns [[]] (i.e. it returns the string representing the singleton containing the empty record) if its input is an encoding of a set containing an even number of records, and returns [] (i.e. it returns the string representing the empty set of records) otherwise. (We could have also chosen the Turing-computable mapping that realises query (4), (5) or (6) from Example 6.3 to be δ_1.) It can easily be verified that $|\Delta_{\delta_1}| = 1$. □

The following theorem shows that either $|\Delta_\delta| = 0$ or $|\Delta_\delta| = 1$ or $|\Delta_\delta| = \omega$ recalling that ω is the set of all natural numbers. Our interpretation of this interesting result is that it is not possible to obtain a finer partition of the class of computable queries with respect to the Turing-computable mappings that realise them, without putting restrictions on the set of encodings.

Theorem 6.6 If $|\Delta_\delta| > 1$, then $|\Delta_\delta| = \omega$.

Proof. Suppose that $|\Delta_\delta| = n$, where $n > 1$, with $n \in \omega$. Then $\exists \varrho_1, \varrho_2, \ldots, \varrho_n \in \mathrm{ENC}$ such that τ_i is realised via δ and ϱ_i and $\forall i, j \in \{1, 2, \ldots, n\}$, $\tau_i \neq \tau_j$. In order to obtain a contradiction to $|\Delta_\delta| = n$, we use a diagonalisation argument to construct a computable query $\tau_{n+1} \in \Delta_\delta$ such that $\forall i \in \{1, 2, \ldots, n\}$, $\tau_{n+1} \neq \tau_i$.

By our assumption that $|\Delta_\delta| = n$, $\exists S_1, S_2, \ldots, S_{n-1} \in \mathrm{RECS}$ such that $\forall i \in \{1, 2, \ldots, n-1\}$, $\tau_i(S_i) \neq \tau_{i+1}(S_i)$. Without loss of generality let $S_n \in \mathrm{RECS}$ be a database of records that satisfies $\varrho_1(S_1) = \varrho_1(S_n)$ and $\varrho_2(S_1) = \varrho_2(S_n)$.

We construct an encoding $\varrho_{n+1} \in \mathrm{ENC}$ such that τ_{n+1} is realised via δ and ϱ_{n+1} as follows. Assume without loss of generality that $\forall i \in \{1, 2, \ldots, n-1\}$, $\varrho_{n+1}(S_i) = \varrho_i(S_i)$, and $\varrho_{n+1}(S_n) = \varrho_2(S_n)$. The result that $\forall i \in \{1, 2, \ldots, n\}$, $\tau_{n+1} \neq \tau_i$ then follows. □

6.5 Computational Query Completeness

We would like to utilise our notion of a computable database query to design expressive query languages. Relational completeness of a query language is only a minimal requirement for such a language, since there are many useful computable queries such as transitive closure and counting the number of tuples in a relation which are not expressible in the relational algebra. (See Definition 3.21 of relational completeness in Subsection 3.2.1 of Chapter 3.) We will further discuss the expressive power of the relational algebra in Section 6.6.

The set of all computable queries, CQ, is too expressive as a measure of the expressiveness of a database query language (or simply a query language), since databases are unordered collections of objects. Thus we should require all expressible queries to be at least order-independent. The class of order-independent computable queries, OI, may also be considered to be overly expressive, since it requires fixing a naming function. On the other hand, the class of encoding-independent computable queries, EI, is not expressive enough, since in practice the user would like to be able to refer to attributes and values which are present in the database. Therefore, we use the notion of C-encoding-independent queries as a measure of the computational completeness of a query language.

Definition 6.15 (Computational query completeness) A database query language is *computationally query complete* (or simply query complete) if it expresses exactly the union of all the classes, C-EI, of all C-encoding-independent computable queries, for some finite set C of attribute names and domain values.

When C is restricted to be a finite set of attribute names only (i.e. C does not include domain values) then we say that the query language is *computationally attribute query complete* (or simply attribute query complete). ■

In the above definition if we require that C = ∅, then a query language is query complete if it expresses exactly the class, EI, of all encoding-independent computable queries.

In the following let us disregard the differences between relational databases and databases of records recalling that every relational database can be translated into a database of records and vice versa. When a query language is attribute query complete the user can explicitly refer to attribute names in queries. Therefore in this case, without any loss of generality, the naming function used to encode a database can always encode a finite set, $\{A_1, A_2, \ldots, A_m\}$, of attribute names as the finite set of natural numbers, $\{1, 2, \ldots, m\}$, and the ordering function used to encode a database can always order the pair (i, v_i) before the pair (j, v_j) if and only if $i < j$. (See Table 6.6 for such an encoding.) This technicality is useful, since it allows us to view X-records as tuples over a relation schema R, with schema(R) = X. (Recall from Definition 3.3 in Section 3.1 of Chapter 3 that attribute names can be referred to in the fixed order induced by the mapping, att.)

We now describe the query language QL [CH80], where d over **R** is taken to be the input database to a QL program (recall from Definition 3.9 in Section 3.1 of Chapter 3 that ADOM(d) is the active domain of d).

Definition 6.16 (The syntax of QL) The syntax of QL is defined as follows:

- $\{y_1, y_2, \ldots\}$ is a countable set of *generic variables.*
- The set of *terms* in QL is defined inductively as follows:
  1) The equality relation **E**, a relation rel_i and a generic variable y_i, with i $\geq$ 1, are terms.
  2) $(e_1 \cap e_2)$, $(\neg e)$, $(e\uparrow)$, $(e\downarrow)$ and $(e\sim)$ are terms, where e, e_1 and e_2 are terms.
- The set of *programs* in QL is defined inductively as follows:
  1) $y_i \leftarrow e$ is a program, where y_i is a generic variable and e is a term.
  2) $(P_1; P_2)$ and **while** $y_i = \emptyset$ **do** P, are programs, where y_i is a generic variable and P, P_1 and P_2 are programs. ■

Prior to giving the formal semantics of QL programs we give informal descriptions of the operators of QL which appear in the definition of the set of terms:

1) $\cap$ is the intersection operator.
2) $\neg$ is the complementation operator.
3) $\uparrow$ is the extension operator, which extends a relation with an additional column.
4) $\downarrow$ is the projection operator, which projects out the first column of a relation.
5) $\sim$ is the permutation operator, which exchanges the values in the last two columns of a relation.

Definition 6.17 (The semantics of QL) The semantics of terms are defined as follows:

- $\mathbf{E} = \{(v, v) \mid v \in \text{ADOM}(d)\}$ is the equality relation.
- rel_i is the relation r_i over a relation schema R_i, if the input database d contains a relation r_i over R_i, otherwise rel_i is the empty set over the relation schema with the empty set of attributes.
- The value of a generic variable y_i is a relation; y_i is initialised to be the empty set over the relation schema with the empty set of attributes. The relation schema of the value of y_i may change during the computation of a QL program.
- Let the value of e be a relation r over R, with type(R) = m, the value of e_1 be a relation r_1 over R_1 and the value of e_2 be a relation r_2 over R_2.
  - If $R_1 = R_2$, then the value of $(e_1 \cap e_2)$ is $r_1 \cap r_2$, otherwise it is the empty set over the relation schema with the empty set of attributes.
  - The value of $(\neg e)$ is $s - r$, where s is the Cartesian product of ADOM(d) with itself m times.
  - The value of $(e \uparrow)$ is the relation $\{<v_1, v_2, \ldots, v_m, v> \mid <v_1, v_2, \ldots, v_m> \in r$ and $v \in \text{ADOM}(d)\}$.
  - The value of $(e \downarrow)$ is the relation $\{<v_2, v_3, \ldots, v_m> \mid <v_1, v_2, \ldots, v_m> \in r\}$ if $m \geq 1$, otherwise the value of $(e \downarrow)$ is defined to be the empty relation.
  - The value of $(e\sim)$ is the relation $\{<v_1, v_2, \ldots, v_m, v_{m-1}> \mid <v_1, v_2, \ldots, v_{m-1}, v_m> \in r\}$ if type(R) = $m > 1$, otherwise the value of $(e \sim)$ is r.

The semantics of QL programs are defined as follows:

- The value of $y_i \leftarrow e$ is the result of assigning the value of e to y_i.
- The value of $(P_1; P_2)$ is the result of sequentially composing P_1 and P_2; we omit parentheses whenever no ambiguity arises.
- The value of **while** $y_i = \emptyset$ **do** P is the result of iterating P while the value of y_i is equal to the empty set; if the while loop terminates, then the value of the program P is the value of y_i, otherwise it is undefined. ■

The proof of the next theorem can be found in [CH80].

Theorem 6.7 The query language QL is attribute query complete. □

Using the extension and projection operators we can simulate counting in QL as follows. The term, $((\mathbf{E} \downarrow) \downarrow)$, whose value is equal to $\{<>\}$ represents the natural number zero. Let $<i>$ be the representation of the natural number i. Then adding one to $<i>$ is given by $<i> \uparrow$. Similarly, subtracting one from $<i>$ is given by $<i> \downarrow$.

We next give some examples of QL programs.

Example 6.5 Let e_1 and e_2 be terms, where the value of e_1 is a relation r_1 over R_1 and the value of e_2 is a relation r_2 over R_2. If $R_1 = R_2$, then the union of e_1 and e_2, denoted by $e_1 \cup e_2$, is defined by

$$(e_1 \cup e_2) = \neg((\neg e_1) \cap (\neg e_2)).$$

Assume that P_1 and P_2 are programs and that y_1 and y_2 are generic variables that do not appear in P_1 or in P_2. The following program gives the expected semantics to the statement, **if** $y_i = \emptyset$ **then** P_1 **else** P_2:

$y_1 \leftarrow y_i$;
$y_2 \leftarrow (((\mathbf{E}\downarrow)\downarrow)\downarrow)$;
while $y_1 = \emptyset$ **do** $(P_1 ; y_1 \leftarrow \mathbf{E};\ y_2 \leftarrow \mathbf{E})$;
while $y_2 = \emptyset$ **do** $(P_2 ; y_2 \leftarrow \mathbf{E})$.

We observe that $(((\mathbf{E}\downarrow)\downarrow)\downarrow) = \emptyset$. Using the above semantics of an if statement we can simulate iterating a program P while $y_i \neq \emptyset$ with the following QL program, where y_1 is a generic variable that does not appear in P:

if $y_i = \emptyset$ **then** $y_1 \leftarrow \mathbf{E}$ **else** $y_1 \leftarrow (((\mathbf{E}\downarrow)\downarrow)\downarrow)$;
while $y_1 = \emptyset$ **do** $(P$; **if** $y_i = \emptyset$ **then** $y_1 \leftarrow \mathbf{E}$ **else** $y_1 \leftarrow (((\mathbf{E}\downarrow)\downarrow)\downarrow))$. ■

Another example of an abstract query language which was shown to be attribute query complete is the *generic machine* [AV91b]. The generic machine is based on extending the relational algebra with Turing-machine capability. An example of a query language which was shown to be query complete is detDL (deterministic transformation language) [AV90]. Finally, another example of a query language which was also shown to be query complete is an extension of Datalog presented in [AV91a], which allows the heads of rules to be negative literals.

An important difference between the semantics of QL and detDL is in the way these query languages simulate counting. As we have seen above QL simulates counting by using the extension operator ($\uparrow$) in order to simulate addition and the projection operator ($\downarrow$) in order to simulate subtraction. On the other hand, detDL simulates counting by generating (or inventing) new values, not in the active domain of the input database, by using a construct called **with new**. The invented values are generated nondeterministically. Furthermore, the invented values are not allowed to appear in the result of a query ensuring that the result of the query is deterministic.

A distinguished value, say v_0, is chosen to represent the natural number zero and another distinguished value, say $v_\star$, is chosen as a placeholder which allows detDL to determine how many natural numbers have already been generated. At any stage during the execution of a detDL program a finite sequence of values, $v_0, v_1, \ldots, v_n$, have already been generated representing the natural numbers $\{0, 1, \ldots, n\}$. In order to indicate the linear ordering $v_0 < v_1 < \ldots < v_n$, these values are stored in a binary relation, r_ω, over a relation schema R_ω, with schema$(R_\omega) = \{N_1, N_2\}$. The relation r_ω is shown in Table 6.7. Whenever addition is performed on the value, v_n, such that $<v_n, v_\star> \in r_\omega$, then a new value, v_{n+1}, is generated, and r_ω is replaced by $r_\omega - \{<v_n, v_\star>\} \cup \{<v_n, v_{n+1}>, <v_{n+1}, v_\star>\}$.

Table 6.7 The relation r_ω

N_1	N_2
v_0	v_1
v_1	v_2
...	...
v_{n-1}	v_n
v_n	$v_\star$

6.6 The Expressive Power of the Relational Algebra

Let us assume that the relational algebra includes only the basic set of relational operators, that is, union, difference, projection, selection, natural join and renaming. In selection we allow only simple selection formulae of the form A = B, where A and B are attributes. The reader can verify that all queries of the relational algebra are C-encoding-independent computable queries for some finite set C of attributes. As we have already mentioned in Subsection 3.2.1 of Chapter 3 the relational algebra cannot express the transitive closure operation, which is a C-encoding-independent computable query. Furthermore, the relational algebra cannot count the number of tuples in a relation or determine whether the number of tuples in a relation is even or odd; both these queries are encoding-independent computable queries. It therefore follows that the relational algebra is not an attribute complete query language. The fundamental reason for the limited expressiveness of the relational algebra is its lack of a looping mechanism (such as a while loop or a recursion facility) and its inability to simulate counting. Still, the relational algebra has become an important yardstick for measuring the expressiveness of a query language. Thus it is important to pinpoint the expressive power of the relational algebra, which is the objective of this section. In particular, we give a characterisation of the set of relations that can be computed as answers to a relational algebra query with respect to an input database.

Let d be a database over the database schema **R**, let E be a relational algebra expression (or equivalently, a relational algebra query) over the database schema **R** and let E(d) be the answer to E with respect to d (see Definition 3.20 in Subsection 3.2.1 of Chapter 3). We now define the *basic information* of a database to be the set of all relations that can be obtained by a relational algebra query over that database.

Definition 6.18 (The basic information of a database) The *basic information* of a database d over **R**, denoted by BI(d), is the set of all relations, r, for which there exists a relational algebra expression E such that E(d) = r. ■

Thus the basic information of d measures the expressive power of the relational algebra with respect to a database d. The operators of the relational algebra (as restricted at the beginning of this section) do not reference explicitly the values in the database, but rather only equality or inequality of values is expressed in relational algebra queries. This motivates us to investigate the connection between the basic information of a database and the mappings from values in the active domain of the database to themselves that leave the database unchanged. Such a mapping that leaves the database unchanged is called an *automorphism* of the database and the set of all automorphisms of a database is called the *cogroup* of the database.

Definition 6.19 (An automorphism of a database) Let $d = \{r_1, r_2, \ldots, r_n\}$ be a database over $\mathbf{R}$, $r = \{t_1, t_2, \ldots, t_k\}$ be a relation over $R \in \mathbf{R}$ and $t = <v_1, v_2, \ldots, v_m>$ be a tuple in r. An *automorphism* of d is a one-to-one mapping, φ, from V onto V, where $V \subseteq \mathcal{D}$ is a set of domain values, extended to tuples, t, relations, r, and databases, d, as follows:

- $\varphi(t) = \varphi(<v_1, v_2, \ldots, v_m>) = <\varphi(v_1), \varphi(v_2), \ldots, \varphi(v_m)>$.
- $\varphi(r) = \varphi(\{t_1, t_2, \ldots, t_k\}) = \{\varphi(t_1), \varphi(t_2), \ldots, \varphi(t_k)\}$.
- $\varphi(d) = \varphi(\{r_1, r_2, \ldots, r_n\}) = \{\varphi(r_1), \varphi(r_2), \ldots, \varphi(r_n)\}$. ■

Definition 6.20 (The cogroup of a database) The *cogroup* of a database, d, denoted by CG(d), is the set of all automorphisms of d. ■

The next definition shows how the cogroup of a database can be represented by a relation.

Definition 6.21 (The cogroup relation of a database) The cogroup of a database, d, can be expressed as a relation, r, over a relation schema R, called the *cogroup relation* of d as follows:

- $|\text{schema}(R)| = |\text{ADOM}(d)| = |\{v_1, v_2, \ldots, v_q\}|$, and
- $<\varphi(v_1), \varphi(v_2), \ldots \varphi(v_q)> \in r$ if and only if $\varphi \in \text{CG}(d)$. ■

From now on, for simplicity, we will not distinguish between the cogroup of a database, d, and its cogroup relation; we will denote both by CG(d). Observe that the cogroup relation of d is uniquely defined up to the linear order imposed on ADOM(d) and the attribute names of its database schema.

Example 6.6 The cogroup relation, CG(d), of a database $d = \{r\}$, where the relation r is shown in Table 6.8, is shown in Table 6.9. The latter relation is obtained by applying Algorithm 6.1 to r. ■

Table 6.8 The relation r

EMP	SEC	MGR
John	Jane	John
Jeff	John	Jenny
John	Jill	John
Jenny	John	Jeff

Table 6.9 The cogroup relation of $\{r\}$

A	B	C	D	E
John	Jeff	Jenny	Jane	Jill
John	Jenny	Jeff	Jane	Jill
John	Jeff	Jenny	Jill	Jane
John	Jenny	Jeff	Jill	Jane

We can now prove that the cogroup relation of d is included in the basic information of d.

Lemma 6.8 $\mathrm{CG}(d) \in \mathrm{BI}(d)$.

Proof. We prove the result when $d = \{r\}$ is a singleton; we leave it to the reader to generalise the result when d may contain more than one relation by constructing a database with a single relation resulting from taking the Cartesian product of all the relations in the database.

Assume that r is a relation over R with $|\mathrm{schema(R)}| = m$, $|r| = k$ and $|\mathrm{ADOM}(r)| = q$. The pseudo-code of an algorithm, designated CONSTRUCT_CG(r), which returns the unique cogroup relation of d (up to a permutation of its attribute names) over a relation schema, whose cardinality is q, is presented in the following algorithm. The reader can verify that CONSTRUCT_CG(r) returns a relational algebra expression whose answer with respect to $\{r\}$ is the cogroup relation $\mathrm{CG}(\{r\})$ of r.

Algorithm 6.1 (CONSTRUCT_CG(r))

1. **begin**
2. $r^k := \rho_{A_m \to B_m}(\ldots(\rho_{A_1 \to B_1}(r))\ldots) \times \cdots \times \rho_{A_m \to B_{km}}(\ldots(\rho_{A_1 \to B_{km-m+1}}(r))\ldots)$;
3. $t :=$ the tuple in r^k that is the concatenation of all the tuples in r;
4. $E^{cg} := r^k$;
5. **for each** $i \in \{1, 2, \ldots, mk - 1\}$ **do**
6. **for each** $j \in \{i + 1, i + 2, \ldots, mk\}$ **do**
7. **if** $t[B_j] = t[B_i]$ **then**
8. $E^{cg} := \sigma_{B_i = B_j}(E^{cg})$;
9. **else**
10. $E^{cg} := \sigma_{B_i \neq B_j}(E^{cg})$;
11. **end if**
12. **end for**
13. **end for**
14. X := a set of q attributes such that $\exists t \in E^{cg}$ with $\bigcup_{B_i \in X} t[B_i] = \mathrm{ADOM}(r)$;
15. $E^{cq} := \pi_X(E^{cg})$;
16. **return** E^{cg};
17. **end.** □

Theorem 6.9 $r \in \mathrm{BI}(d)$ if and only if $\mathrm{ADOM}(r) \subseteq \mathrm{ADOM}(d)$ and $\mathrm{CG}(d) \subseteq \mathrm{CG}(\{r\})$.

Proof. We only sketch the proof.

If. Suppose that $\mathrm{ADOM}(r) \subseteq \mathrm{ADOM}(d)$ and $\mathrm{CG}(d) \subseteq \mathrm{CG}(\{r\})$. We then need to show that $r = \mathrm{E}(d)$ for some relational algebra expression E. Let t be a tuple in r and let R be the relation schema of r. We know by Lemma 6.8 that $\mathrm{CG}(d) \in \mathrm{BI}(d)$. Furthermore, for all attributes A $\in$ schema(R), t[A] is a value in each tuple of $\mathrm{CG}(d)$. Thus t is a member of the answer to a relational algebra expression, say E_t, with respect to $\mathrm{CG}(d)$. The expression E_t is composed of a Cartesian product for each repeated value in t, appropriate renamings so that the schema of the output corresponds to R and a projection onto the set of attributes schema(R). All the tuples in $E_t(\mathrm{CG}(d))$ are of the form $\varphi(t)$, where $\varphi \in \mathrm{CG}(d) \subseteq \mathrm{CG}(\{r\})$ and thus all the tuples in

$E_t(\mathrm{CG}(d))$ are members of r. Thus on letting E be the relational expression $\cup_{t \in r} E_t$, it follows that $\mathrm{E}(d) = r$ as required.

Only if. If $r \in \mathrm{BI}(d)$, then by Definition 6.18 $r = \mathrm{E}(d)$ for some relational algebra expression E. The result that $\mathrm{ADOM}(r) \subseteq \mathrm{ADOM}(d)$ and $\mathrm{CG}(d) \subseteq \mathrm{CG}(\{r\})$ follows by induction on the number of relational algebra operators appearing in E. □

The above theorem was first proved in [Ban78, Par78]; a full proof can also be found in [AD93]. An alternative representation of the cogroup relation as a nested relation can be found in [AGV89].

6.7 Adding a Looping Mechanism to the Relational Algebra

The reason that the relational algebra is not expressive enough to express transitive closure queries is its lack of a looping mechanism. One possible extension suggested in [AU79] is to add a *fixpoint operator* to the relational algebra. The fixpoint operates on a relational algebra query, Q, with respect to a database d, by iterating Q with respect to d until no changes occur in the result of the query Q. More formally, we define the result of the fixpoint of Q with respect to d, denoted by $\mathrm{FIX}(\mathrm{Q}(d))$, by using the auxiliary query Q_i, where $i \geq 0$ is a natural number, as follows:

1) $Q_0(d) = \mathrm{Q}(d)$,
2) $Q_{i+1}(d) = Q_i(d) \cup \mathrm{Q}(Q_i(d))$; and
3) $\mathrm{FIX}(\mathrm{Q}(d)) = Q_k(d)$, where $k \geq 0$ is the least natural number such that $Q_k(d) = Q_{k+1}(d)$.

A query of the form $\mathrm{FIX}(\mathrm{Q}(d))$ is called a *fixpoint query*. We observe that fixpoint queries as we have defined them are *inflationary*, since the intermediate results Q_i are increasing for $i \geq 0$. We show that the cardinality of $\mathrm{FIX}(\mathrm{Q}(d))$ is polynomial in the size of the input database, d. Let s be the size of d, i.e. the number of symbols needed to encode d, and let m be the number of attributes in the schema of the relation resulting from answering the query. Then $|\mathrm{FIX}(\mathrm{Q}(d))| \leq s^m$, which is polynomial in the size of the input database.

Let ARC be a binary relation representing the arcs of a digraph. Then the following fixpoint query computes the transitive closure of the digraph:

$$\mathrm{FIX}(\mathrm{ARC} \cup (\pi_{\{\mathrm{A,B}\}}(\rho_{\mathrm{B} \to \mathrm{C}}(\mathrm{ARC}) \bowtie \rho_{\mathrm{A} \to \mathrm{C}}(\mathrm{ARC})))).$$

Instead of incorporating the fixpoint operator into the relational algebra, it has been suggested that we add an explicit looping mechanism to the algebra such as the while loop of the query language, QL. Such a while loop is *unbounded*, since it is not guaranteed to terminate, resulting in polynomial space computations in the size of the input database. In order to bound the number of iterations of a loop by a polynomial in the size of the input database, a *bounded* looping mechanism can be added to the relational algebra. Adding a bounded looping mechanism, such as the for loop introduced below, provides us with a query

language of intermediate expressive power between the relational algebra augmented with the fixpoint operator and QL.

Definition 6.22 (For loop) Let y_i be a generic variable and P be a QL program. Then the construct

$$\textbf{for } y_i \textbf{ do } P,$$

called a *for loop*, is also a QL program.

The semantics of the for loop are defined as follows, where r_i is the value of y_i:

- The program P is executed n times, where n is the cardinality of r_i upon entry to the for loop; when the for loop terminates the value of P is the value of y_i. ■

We leave it to the reader to verify that the for loop does not add any expressive power to QL, since it can be simulated in QL by a while loop that counts the number of times the for loop is executed and exits the while loop after a specified number of iterations. On the other hand, if we replace the while loop in QL by the for loop, then QL would not be an attribute query complete database query language, since intuitively QL's computations may be unbounded and nonterminating, as opposed to the computations of the for loop restriction of QL which are always bounded and terminating. More precisely it can be shown that the for loop restriction of QL computes exactly the set of primitive recursive queries (see Subsection 1.9.4), which are a proper subset of the set of computable queries [Cha81].

For the purpose of extending the relational algebra with a for loop mechanism we will restrict QL as follows.

Definition 6.23 (ForQL) A variable is *typed* if the similarity type of its relation schema is fixed and cannot change during the computation of a program. If a typed variable is assigned a relation over a relation schema with a different similarity type then the empty set is assigned to this variable.

ForQL is the query language which restricts QL by assuming that all variables are typed, that the terms of the language are relational algebra expressions (as defined at the beginning of Section 6.6) and that instead of the while loop we have the for loop **for** y_i **do** P, where y_i is a typed variable. ■

By a similar argument to the fixpoint, it can now be shown that the time complexity of ForQL programs is bounded by a polynomial in the size of the input database. We note that if we lift the restriction that variables be typed, then we cannot, in general, bound the time complexity of such QL programs to a polynomial in the size of the input database, since a nested for loop which uses the same generic variable can lead to a computation which is exponential in the size of the input database.

Assume that r is a relation over R, with schema(R) = {A,B} and that the input database to the next QL program is $d = \{r\}$, i.e. that the value of *rel* in the QL program is r. Also, assume that the schema of the equality relation E is R. The following ForQL program uses a for loop

in order to compute the transitive closure of r:

$$
\begin{array}{l}
y \leftarrow \pi_{\mathrm{A}}(\mathbf{E}) \times \pi_{\mathrm{B}}(\mathbf{E}); \\
y_1 \leftarrow rel; \\
\textbf{for } y \textbf{ do} \\
\quad (y_2 \leftarrow y_1 \cup (\pi_{\{\mathrm{A},\mathrm{B}\}}(\rho_{\mathrm{B}\rightarrow\mathrm{C}}(y_1) \bowtie \rho_{\mathrm{A}\rightarrow\mathrm{C}}(rel))); \\
\quad y_1 \leftarrow y_2; \\
\quad y \leftarrow y_1).
\end{array}
$$

The next ForQL program shows that by using a for loop it is possible to test whether the cardinality of a relation is even. In the ForQL program we assume that $\pi_{\mathrm{A}}(\mathbf{E})$ represents logical truth and $\emptyset$ represents logical falsity:

$$
\begin{array}{l}
y \leftarrow rel; \\
y_1 \leftarrow \pi_{\mathrm{A}}(\mathbf{E}); \\
\textbf{for } y \textbf{ do } y_1 \leftarrow (\neg y_1).
\end{array}
$$

When the above program terminates, y_1 is nonempty, i.e. it represents logical truth, if and only if $|r|$ is even. In [CH82] it was shown that testing whether the cardinality of a relation is even cannot be expressed by the relational algebra augmented with the fixpoint operator. It follows that adding a bounded looping mechanism to the relational algebra results in a query language that is strictly more expressive than the language resulting from adding the fixpoint operator to the relational algebra. Still, some natural queries such as checking whether two relations r_1 and r_2 have equal cardinality, i.e. checking whether $|r_1| = |r_2|$, cannot be expressed in ForQL, i.e. it cannot be expressed by the relational algebra augmented with a bounded looping mechanism [Cha88].

In [AV91a] it was shown that Datalog, whose programs may be recursive and contain rules having negative literals in their body (see Subsection 3.2.3 of Chapter 3), is equivalent to the relational algebra augmented with a fixpoint operator, in the sense that they both express exactly the same set of computable queries. This set of computable queries is a proper subset of the set of all polynomial-time computable queries, since as noted above determining whether the cardinality of a relation is even is not amongst such queries.

Definition 6.24 (Ordered relational database) A relational database d over $\mathbf{R}$ is an *ordered relational database* (or simply an ordered database) if $\mathbf{R}$ contains a designated binary relation schema, SUCC, such that the relation r in d over SUCC defines a linear ordering on the set of active domain values, ADOM(d). ■

We observe that SUCC is isomorphic to a finite fragment of the successor relation on the natural numbers. We further note that an ordered database induces a lexicographical ordering on the tuples of each relation in d. The next fundamental theorem, which characterises the computational expressiveness over ordered databases of the relational algebra augmented with a fixpoint operator, was shown in [Var82a, Imm86] (see also [AHV95b, Chapter 17]).

Theorem 6.10 Over ordered databases, the relational algebra augmented with a fixpoint operator expresses exactly the set of all polynomial-time computable queries.

Proof. We briefly sketch the proof leaving the reader to consult the above references for the full proof. We have already shown that the algebra augmented with a fixpoint can only express computable queries which can be evaluated in polynomial time in the size of the input database, so it suffices to show that it can express all such computable queries.

Let Q be a polynomial-time computable query. The idea is to simulate the Turing machine TM that computes Q with a fixpoint query. Firstly, we can encode TM's input tape by utilising the lexicographical ordering of the tuples of the relations in d. Secondly, we can encode an instantaneous description of TM, after the ith computation step, by using the lexicographical ordering to encode i and by having distinguished attributes whose values encode the current state of TM's finite state control and the current position of its head. The next move function can then be encoded via an algebraic expression, which given the current state and position of the finite state control performs the next computation step of TM. That is, the currently scanned symbol is overwritten, the head of the finite state control moves either left (add one) or right (subtract one) and a transition of state is effected. We assume that TM halts if and only if no more state transitions can be effected or the maximum number of computation steps has been performed. Thus the fixpoint operator is needed in order to repeat the next move function until TM halts. □

The next corollary follows from the previous theorem and the fact that Datalog, whose programs may be recursive and contain rules having negative literals in their body (see Subsection 3.2.3 of Chapter 3), is equivalent to the relational algebra with a fixpoint and less expressive than the relational algebra with bounded looping; we note that these three query languages can only compute queries whose time complexity is polynomial in the size of the input database.

Corollary 6.11 Over ordered databases, the relational algebra augmented with a fixpoint operator, the relational algebra augmented with a bounded for loop mechanism and Datalog are all equivalent and express exactly the set of all polynomial-time computable queries. □

As mentioned prior to Definition 6.22 augmenting the relational algebra with a while loop as in QL results in polynomial space computations in the size of the input database. For the purpose of extending the relational algebra with a while loop mechanism, we will restrict QL to allow only typed variables as follows.

Definition 6.25 (WhileQL) WhileQL is the query language which restricts QL by assuming that all variables are typed, and that the terms of the language are relational algebra expressions (as defined at the beginning of Section 6.6). ■

Thus WhileQL is the query language resulting from restricting QL in the same manner as the query language ForQL given in Definition 6.23, apart from maintaining the while loop, **while** $y_i = \emptyset$ **do** P, rather than swapping it with a for loop as in ForQL. It can be shown that WhileQL's computational power is included in the set of polynomial space queries [CH82]. (See PSPACE in Subsection 1.9.4 of Chapter 1.) This inclusion is proper, since, for example, WhileQL is not expressive enough to determine whether the cardinality of a relation is even or odd. Moreover, it can be shown that over ordered relational databases WhileQL expresses exactly the set of all polynomial space queries [Var82a].

WhileQL can be further extended with integer arithmetic by augmenting it with the following constructs:

- A countable set $\{c_1, c_2, \ldots\}$ of *counter* variables distinct from typed variables, whose values are natural numbers.
- Assignment statements of the form $c_i \leftarrow c_i + 1$ and $c_i \leftarrow c_i - 1$, which increment and decrement a counter variable c_i, assuming that counter variables are initialised to 0. We assume that if c_i has the value 0, then $c_i \leftarrow c_i - 1$ leaves c_i unchanged.
- Tests of the form **while** $c_i = 0$ **do** P, which terminate when the counter variable c_i has a value other than 0.

Let us call the extension of WhileQL with integer arithmetic as defined above WhileInt. The query language WhileInt is still not attribute query complete, since it still cannot determine whether the cardinality of a relation is even or odd. It can be viewed as providing an interface between a Turing-complete programming language and a first-order query language such as SQL, where SQL statements can be embedded in the statements of the programming language. (See also the recent *Java Database Connectivity* (JDBC) approach for executing SQL statements from within a Java program [HCF97].) It is evident that the expressive power of WhileInt properly includes that of WhileQL, since the ability to manipulate counters gives Turing-machine capability to the language (see Subsection 1.9.4 of Chapter 1) and thus over ordered relational databases WhileInt is attribute query complete.

The class of computable queries that can be expressed by the query language WhileInt is robust as can be seen by its equivalence to the class of computable queries expressed by two other query languages, which we now briefly describe [AV93].

The first query language is called a *relational machine*. Such a machine consists of a Turing machine and a relational store which holds a finite set of relations over a fixed set of relation schemas partitioned into input relations and output relations. The tape of the Turing machine is initially empty. The head of the relational machine can move left and right and can write on the tape in accordance with the state transitions of a standard Turing machine transition function with the following extensions. The machine can check whether the input relations in the relational store satisfy a relational algebra query, i.e. return a nonempty result, and the machine can also assign to an output relation in the relational store the result of computing a relational algebra query on the input relations in this store. In both cases the number of typed variables in any relational algebra query is bounded by some constant. The relational machine accepts its input relations if and only if it reaches its halting state. Its output can then be found in the output relations in the store.

The second query language is an effective fragment of the *infinitary logic* $L^{\omega}_{\infty\omega}$. The first subscript indicates that conjunctions and disjunctions can be taken over arbitrary, possibly infinite, sets of formulae and the second subscript indicates that only finite quantifier blocks are allowed. The superscript indicates that every formula can only have a finite number k of variables, for some natural number k, with $k \geq 1$.

The formulae of the infinitary logic $L^{k}_{\infty\omega}$ over a database schema $\mathbf{R}$ constitute the smallest set of formulae containing relational algebra expressions over $\mathbf{R}$ having at most k typed variables and closed under the usual logical connectives and quantifiers of first-order logic and, in addition, are closed under conjunction and disjunction of arbitrary sets of formulae. That is,

in the infinitary logic $L^k_{\infty\omega}$ the disjunction $\bigvee \Phi$ and the conjunction $\bigwedge \Phi$, where Φ is a set containing an infinite number of formulae, are both well defined. The semantics of the logical connectives including arbitrary conjunctions and disjunctions are the standard ones.

We can now define the formulae of the infinitary logic over a database schema $\mathbf{R}$ by

$$L^\omega_{\infty\omega} = \bigcup_{k=1}^{\infty} L^k_{\infty\omega}.$$

For more details on the model theory of infinitary logic see [BF85] and for recent research on infinitary logic in finite model theory see [KV92a, KV92b, AVV95, DLW95].

In order to define the effective fragment of the above infinitary logic, we say that a set $\mathcal{S}$ of databases over $\mathbf{R}$ is *recursively enumerable* if there exists a recursive enumeration $d_1, d_2, \ldots$ of all databases over $\mathbf{R}$ such that a database d is in $\mathcal{S}$ if and only if there is some database which is isomorphic to d and belongs to the enumeration. (See Subsection 1.9.4 of Chapter 1 for an overview of recursively enumerable languages or sets.)

The *effective fragment* of infinitary logic is now defined as the set of formulae in $L^\omega_{\infty\omega}$ whose set of finite models is recursively enumerable. The proof of the next theorem can be found in [AVV95].

Theorem 6.12 The following three query languages express the same class of computable queries over a database schema $\mathbf{R}$:

1) WhileInt.

2) Relational machines.

3) Effective fragment of infinitary logic.

Proof. We only sketch the proof.

The equivalence of (1) and (2) follows from the equivalence between Turing machines and counter machines (see Subsection 1.9.4 of Chapter 1).

To show that part (2) implies part (3), let $\mathcal{S}$ be the set of all databases over $\mathbf{R}$ that are accepted by some relational machine M and let $d \in \mathcal{S}$. Intuitively, a computation of M that accepts d can be described by a formula in $L^k_{\infty\omega}$ for some natural number k, due to the way in which a relational machine interfaces with the relational algebra. It follows that the set of databases $\mathcal{S}$ that are accepted by M can be described by a formula in the effective fragment of infinitary logic, which is a countably infinite disjunction of a recursive set of formulae in $L^k_{\infty\omega}$.

Finally, to show that part (3) implies part (2) suppose that φ is a formula over $\mathbf{R}$ in the effective fragment of infinitary logic. It can be shown that given a relational database d over $\mathbf{R}$, there exists an ordered relational database d' over a database schema $\mathbf{R}'$ and a formula ψ over $\mathbf{R}'$ in the effective fragment of infinitary logic such that d is a model of φ if and only if d' is a model of ψ. We can now use this fact to encode d' on the tape of a relational machine M. The relational machine M utilises its Turing-machine capability to accept d', if and only if d' is a model of ψ, by a recursive enumeration of those databases over $\mathbf{R}'$ which are models of ψ. □

6.8 Discussion

In this chapter we have introduced and developed the fundamental concept of a computable database query, which does not feature prominently in most of the current textbooks on database theory. Although the development is theoretical in nature, the subclass of computable queries implemented has an effect on the degree of portability of the database, in the sense that two queries may yield the same result on different machines if and only if they are C-encoding-independent. The notion of encoding-independence is thus strongly related to the notion of physical data independence, which is one of the fundamental reasons that relational databases are successful in practice. In addition, we have presented the concept of a database language which is *query complete*; such a language allows for both attribute names and domain values to be mentioned in queries. This concept refines the notion of *attribute query complete*, which allows only attribute names to be mentioned in queries.

The notion of computable queries has interested database researchers since the beginning of the 1980's and provides a link between relational database theory and the theory of computing. It also provides a firm basis for developing database programming languages possessing greater computational expressive power than the relational algebra.

Pioneering work on the computational complexity of various query languages can be found in [Fag74, AU79, Cha81, Imm81, CH82, Var82a]. More recent research in this area can be found in [Imm86, BG87, Imm87, Cha88, Imm89, AV90, AV91a, AV91b, AV92, AV93, Fag93, AV95, AVV97].

6.9 Exercises

Exercise 6.1 SQL3 is the emerging standard, which is to replace SQL2 [DD93, Mel96] (see Subsection 3.2.2 of Chapter 3 and Section 10.2 of Chapter 10). One of the features of SQL3 is the addition of procedural constructs to SQL2 including assignment, conditional and looping statements. Thus SQL3 is computationally query complete. Argue for the usefulness of these features of SQL3 and its potential impact on database programming.

Exercise 6.2 Aggregate functions provide an important and useful extension to the basic operators of the relational algebra (see Definition 3.24 in Subsection 3.2.1 of Chapter 3). Suggest how such an extension affects the expressive power of the relational algebra [Klu82, AB95].

Exercise 6.3 It is a standard assumption in relational database theory that domain elements are taken to be unordered. In practice domain elements are defined to be either strings or numbers and thus tuples in relations have a natural lexicographical ordering that can be utilised by the DBMS. Argue whether it is reasonable for the DBMS to use such an ordering when processing queries and how such an ordering can be taken into account in the definition of a computable database query.

Exercise 6.4 Discuss, with a motivating example, the connection between physical data independence and encoding-independent computable queries.

Exercise 6.5 Recall Definition 6.25 of the query language WhileQL. WhileQL is the result of augmenting the relational algebra with an unbounded while loop mechanism as in QL, with the restriction on generic variables that they be typed. Prove that WhileQL cannot determine whether the cardinality of a relation is even or odd [CH82].

Exercise 6.6 Recall that the query language WhileInt extends the query language WhileQL with integer arithmetic. Prove that when all the relation schemas of relations in the input database to a WhileInt program are monadic, i.e. contain a single attribute, then the resulting program is equivalent to a relational algebra query [AV95].

Exercise 6.7 Discuss the significance of Theorem 6.10 and Corollary 6.11 in Section 6.7 with respect to the implementation of database programming languages.

Exercise 6.8 Suggest a parallel model of computation for relational algebra queries, where given an input database d, such a model has available a polynomial number of processors in the size of d in order to speed up the computation [Imm89].

Exercise 6.9 It has been suggested that it is useful to add to query languages an operator that selects a tuple from a relation at random. Discuss this suggestion with a concrete example [ASV90].