PART

B Basics: Relational Query Languages

The area of query languages, and more generally providing access to stored data, is one of the most important topics in databases and one of the most deeply studied ones in database theory. This part introduces three paradigms that have been developed for querying relational databases. Each yields a family of query languages, and there are close connections among the different families.

The first paradigm provides simple algebraic operations for manipulating relations to construct answers to queries. This yields a language called relational *algebra*. It uses three operators tailored specially for the relational model, in addition to the natural set operators. The second paradigm is logic based. It is a variant of the predicate calculus of first-order logic, called relational *calculus*, and has expressive power equivalent to the algebra. The third paradigm stems from logic programming. Its most prominent representative in databases is *datalog*, which can be viewed as logic programming without function symbols.

The three paradigms yield languages that enjoy fundamental properties that have become standard for virtually all database access languages. First, they are *set-at-a-time*, in the sense that they focus on identifying and uniformly manipulating sets of tuples rather than identifying tuples individually and using loops to manipulate groups of tuples. Second, they are *associative* in that tuples are identified through a specification of their properties rather than by chasing pointers. And third, they are abstract, high-level languages that are separated from the physical storage of the data and from the specific algorithms that are used to implement them.

The relational algebra is conceptually a "procedural" language because queries are specified by a sequence of operations that constructs the answer. The relational calculus and datalog are conceptually "declarative" because the tuples in the answer are specified by properties they satisfy, with no reference to an algorithm for producing them. However, in modern database implementations the optimized translation of queries from any of the three languages may have little resemblance to the original; in this sense all three are essentially declarative.

In this part and Part D we introduce a variety of query languages based on the three paradigms. This is done in a natural progression, starting with simple, commonly asked

queries and building up to very powerful ones. At each stage, we provide equivalent languages in the three paradigms. In this part we focus on the simplest queries and incorporate negation into them. In Part D we incorporate recursion into the languages.

The simplest queries are based on extracting certain values as soon as a simple pattern of tuples is found in the database. These are discussed in Chapter 4. In terms of the calculus, the corresponding language is based on conjunction and existential quantification; hence the term *conjunctive queries.* Although these queries are simple, they constitute the vast majority of relational database queries arising in practice. In addition, conjunctive queries have many pleasing theoretical properties. For instance, there is an elegant characterization of two conjunctive queries being equivalent (discussed in Chapter 6). Chapter 4 also considers the inclusion of a union (or disjunction) capability into these languages, yielding the *conjunctive queries with union*.

In Chapter 5, the conjunctive queries (with union) are extended with *negation*. Adding negation (or set difference, in the algebraic paradigm) yields the full relational calculus and relational algebra. The calculus and algebra turn out to be equivalent; this is one of the earliest significant results in database theory. An equivalent language is also obtained from the datalog version of the conjunctive queries. In general, the positive results for conjunctive queries fail with the relational calculus. In Chapter 5 we also touch on the subject of infinite databases with finite representation and, in particular, the emerging area of "constraint databases."

Chapter 6 considers the conjunctive queries and the relational calculus from the perspective of static analysis (in the sense of programming languages). An elegant theorem shows that properties such as containment and equivalence are decidable for the conjunctive queries. Interestingly, these properties are undecidable for relational calculus. Query optimization is also considered in Chapter 6. The focus is on conjunctive queries (most practical and early theoretical work on optimization has been focused on these as well). Three topics in optimization are considered. First, some of the fundamental approaches taken by practical optimizers are considered. Second, the theory for testing equivalence of conjunctive queries is extended to develop a technique for minimizing the number of joins needed to compute a conjunctive query. Third, a family of interesting results concerning a natural subclass of conjunctive queries is presented based on a theory of "acyclic hypergraphs." (More recent work on optimizing recursive datalog queries is presented in Chapter 13.)

This concludes the presentation of the basic theory of the simple query languages. The theory of query languages is again taken up in Part D, which considers languages with recursion and resumes the parallel development along the three paradigms. Expressiveness and complexity of query languages are discussed in Part E.

Some connections between the abstract languages described in Chapters 4 and 5 and practical languages are considered in Chapter 7. We focus largely on SQL, which has become the industry standard for relational database access, and we briefly describe two visual languages, QBE and Access.

4 Conjunctive Queries

Alice: *Shall we start asking queries?*
Sergio: *Very simple ones for the time being.*
Riccardo: *But the system will answer them fast.*
Vittorio: *And there is some nice theory.*

In this chapter we embark on the study of queries for relational databases, a rich topic that spans a good part of this book. This chapter focuses on a limited but extremely natural and commonly arising class of queries called *conjunctive queries*. Five equivalent versions of this query family are presented here: one from each of the calculus and datalog paradigms, two from the algebra paradigm, and a final one that has a more visual form. In the context of conjunctive queries, the three nonalgebraic versions can be viewed as minor syntactic variants of each other; but these similarities diminish as the languages are generalized to incorporate negation and/or recursion. This chapter also discusses query composition and its interaction with user views, and it extends conjunctive queries in a straightforward manner to incorporate union (or disjunction).

The conjunctive queries enjoy several desirable properties, including, for example, decidability of equivalence and containment. These results will be presented in Chapter 6, in which a basic tool, the Homomorphism Theorem, is developed. Most of these results extend to conjunctive queries with union.

In the formal framework that we have developed in this book, we distinguish between a *query*, which is a syntactic object, and a *query mapping*, which is the function defined by a query interpreted under a specified semantics. However, we often blur these two concepts when the meaning is clear from the context. In the relational model, query mappings generally have as domain the family of all instances of a specified relation or database schema, called the *input schema*; and they have as range the family of instances of an *output schema*, which might be a database schema or a relation schema. In the latter case, the relation name may be specified as part of the syntax of the query or by the context, or it may be irrelevant to the discussion and thus not specified at all. We generally say that a query (mapping) is *from* (or *over*) its input schema *to* its output schema. Finally, two queries q_1 and q_2 over $\mathbf{R}$ are *equivalent*, denoted $q_1 \equiv q_2$, if they have the same output schema and $q_1(\mathbf{I}) = q_2(\mathbf{I})$ for each instance $\mathbf{I}$ over $\mathbf{R}$.

This chapter begins with an informal discussion that introduces a family of simple queries and illustrates one approach to expressing them formally. Three versions of conjunctive queries are then introduced, and all of them have a basis in logic. Then a brief

(4.1) Who is the director of "Cries and Whispers"?
(4.2) Which theaters feature "Cries and Whispers"?
(4.3) What are the address and phone number of the Le Champo?
(4.4) List the names and addresses of theaters featuring a Bergman film.
(4.5) Is a film directed by Bergman playing in Paris?
(4.6) List the pairs of persons such that the first directed the second in a movie, and vice versa.
(4.7) List the names of directors who have acted in a movie they directed.
(4.8) List pairs of actors that acted in the same movie.
(4.9) On any input produce ⟨"Apocalypse Now", "Coppola"⟩ as the answer.
(4.10) Where can I see "Annie Hall" or "Manhattan"?
(4.11) What are the films with Allen as actor or director?
(4.12) What films with Allen as actor or director are currently featured at the Concorde?
(4.13) List all movies that were directed by Hitchcock or that are currently playing at the Rex.
(4.14) List all actors and director of the movie "Apocalypse Now."

Figure 4.1: Examples of conjunctive queries, some of which require union

digression is made to consider query composition and database views. The algebraic perspectives on conjunctive queries are then given, along with the theorem showing the equivalence of all five approaches to conjunctive queries. Finally, various forms of union and disjunction are added to the conjunctive queries.

4.1 Getting Started

To present the intuition of conjunctive queries, consider again the **CINEMA** database of Chapter 3. The following correspond to conjunctive queries:

(4.1) Who is the director of "Cries and Whispers"?
(4.2) Which theaters feature "Cries and Whispers"?
(4.3) What are the address and phone number of the Le Champo?

These and other queries used in this section are gathered in Fig. 4.1. Each of the queries just given calls for extracting information from a single relation. In contrast, queries (4.4) through (4.7) involve more than one relation.

In queries (4.1–4.4 and 4.6–4.9), the database is asked to find values or tuples of values for which a certain pattern of data holds in the database, and in query (4.5) the database is asked whether a certain pattern of data holds. We shall see that the patterns can be described simply in terms of the existence of tuples that are connected to each other by equality of some of their coordinates. On the other hand, queries (4.10) through (4.14) cannot be expressed in this manner unless some form of disjunction or union is incorporated.

EXAMPLE 4.1.1 Consider query (4.4). Intuitively, we express this query by stating that

if there are tuples r_1, r_2, r_3 respectively in relations
Movies, *Pariscope*, *Location* **such that**
the *Director* in r_1 is "Bergman"
and the *Titles* in tuple r_1 and r_2 are the same
and the *Theaters* in tuple r_2 and r_3 are the same
then we want the *Theater* and *Address* coordinates from tuple r_3.

In this formulation we essentially use variables that range over tuples. Although this is the basis of the so-called (*relational*) *tuple calculus* (see Exercise 5.23 in the next chapter), the focus of most theoretical investigations has been on the *domain calculus*, which uses variables that range over constants rather than tuples. This also reflects the convention followed in the predicate calculus of first-order logic. Thus we reformulate the preceding query as

if there are tuples $\langle x_{ti}$, "Bergman", $x_{ac}\rangle$, $\langle x_{th}, x_{ti}, x_s\rangle$, and $\langle x_{th}, x_{ad}, x_p\rangle$,
respectively, in relations *Movies, Pariscope,* and *Location*
then include the tuple $\langle Theater : x_{th},\ Address : x_{ad}\rangle$ in the answer,

where $x_{ti}, x_{ac}, \ldots$ are variables. Note that the equalities specified in the first formulation are achieved implicitly in the second formulation through multiple occurrences of variables.

The translation of this into the syntax of *rule-based conjunctive queries* is now obtained by

$$ans(x_{th}, x_{ad}) \leftarrow Movies(x_{ti}, \text{"Bergman"}, x_{ac}),\ Pariscope(x_{th}, x_{ti}, x_s),\ Location(x_{th}, x_{ad}, x_p)$$

where *ans* (for "answer") is a relation over {*Theater*, *Address*}. The atom to the left of the $\leftarrow$ is called the rule *head*, and the set of atoms to the right is called the *body*.

The preceding rule may be abbreviated as

$$ans(x_{th}, x_{ad}) \leftarrow Movies(x_{ti}, \text{"Bergman"}, \_),\ Pariscope(x_{th}, x_{ti}, \_),\ Location(x_{th}, x_{ad}, \_)$$

where _ is used to replace all variables that occur exactly once in the rule. Such variables are sometimes called *anonymous*.

In general, a rule-based conjunctive query is a single rule that has the form illustrated in the preceding example. The semantics associated with rule-based conjunctive queries ensures that their interpretation corresponds to the more informal expressions given in the preceding example. Rule-based conjunctive queries can be viewed as the basic building block for datalog, a query language based on logic programming that provides an elegant syntax for expressing recursion.

A second paradigm for the conjunctive queries has a more visual form and uses tables with variables and constants. Although we present a more succinct formalism for this

Movies	*Title*	*Director*	*Actor*
	_The Seventh Seal	Bergman	

Pariscope	*Theater*	*Title*	*Schedule*
	_Rex	_The Seventh Seal	

Location	*Theater*	*Address*	*Phone number*
	P._Rex	P._1 bd. Poissonnière	

Figure 4.2: A query in QBE

paradigm later in this chapter, we illustrate it in Fig. 4.2 with a query presented in the syntax of the language Query-By-Example (QBE) (see also Chapter 7). The identifiers starting with a _ designate variables, and P. indicates what to output. Following the convention established for QBE, variable names are chosen to reflect typical values that they might take. Note that the coordinate entries left blank correspond, in terms of the rule given previously, to distinct variables that occur exactly once in the body and do not occur in the head (i.e., to anonymous variables).

The third version of conjunctive queries studied in this chapter is a restriction of the predicate calculus; as will be seen, the term *conjunctive query* stems from this version. The fourth and fifth versions are algebraic in nature, one for the unnamed perspective and the other for the named perspective.

4.2 Logic-Based Perspectives

In this section we introduce and study three versions of the conjunctive queries, all stemming from mathematical logic. After showing the equivalence of the three resulting query languages, we extend them by incorporating a capability to express equality explicity, thereby yielding a slightly more powerful family of languages.

Rule-Based Conjunctive Queries

The rule-based version of conjunctive queries is now presented formally. As will be seen later, the rule-based paradigm is well suited for specifying queries from database schemas to database schemas. However, to facilitate the comparison between the different variants of the conjunctive queries, we focus first on rule-based queries whose targets are relation schemas. We adopt the convention of using the name *ans* to refer to the name of the target relation if the name itself is unimportant (as is often the case with relational queries).

Definition 4.2.1 Let $\mathbf{R}$ be a database schema. A *rule-based conjunctive query* over $\mathbf{R}$ is an expression of the form

$$ans(u) \leftarrow R_1(u_1), \ldots, R_n(u_n)$$

where $n \geq 0$, $R_1, \ldots, R_n$ are relation names in $\mathbf{R}$; *ans* is a relation name not in $\mathbf{R}$; and $u, u_1, \ldots, u_n$ are free tuples (i.e., may use either variables or constants). Recall that if $v = \langle x_1, \ldots, x_m \rangle$, then '$R(v)$' is a shorthand for '$R(x_1, \ldots, x_m)$'. In addition, the tuples $u, u_1, \ldots, u_n$ must have the appropriate arities (i.e., u must have arity of *ans*, and u_i must have the arity of R_i for each $i \in [1, n]$). Finally, each variable occurring in u must also occur at least once in $u_1, \ldots, u_n$. The set of variables occurring in q is denoted $var(q)$.

Rule-based conjunctive queries are often more simply called *rules*. In the preceding rule, the subexpression $R_1(u_1), \ldots, R_n(u_n)$ is the *body* of the rule, and '$ans(u)$' is the *head*. The rule here is required by the definition to be *range restricted* (i.e., each variable occurring in the head must also occur in the body). Although this restriction is followed in most of the languages based on the use of rules, it will be relaxed in Chapter 18.

Intuitively, a rule may be thought of as a tool for deducing new facts. If one can find values for the variables of the rule such that the body holds, then one may deduce the head fact. This concept of "values for the variables in the rules" is captured by the notion of "valuation." Formally, given a finite subset V of **var**, a *valuation* ν over V is a total function ν from V to the set **dom** of constants. This is extended to be identity on **dom** and then extended to map free tuples to tuples in the natural fashion.

We now define the semantics for rule-based conjunctive queries. Let q be the query given earlier, and let $\mathbf{I}$ be an instance of $\mathbf{R}$. The *image* of $\mathbf{I}$ under q is

$$q(\mathbf{I}) = \{\nu(u) \mid \nu \text{ is a valuation over } var(q) \text{ and } \nu(u_i) \in \mathbf{I}(R_i), \text{ for each } i \in [1, n]\}.$$

The *active domain* of a database instance $\mathbf{I}$, denoted $adom(\mathbf{I})$, is the set of all constants occurring in $\mathbf{I}$, and the active domain $adom(I)$ of relation instance I is defined analogously. In addition, the set of constants occurring in a query q is denoted $adom(q)$. We use $adom(q, \mathbf{I})$ as an abbreviation for $adom(q) \cup adom(\mathbf{I})$.

Let q be a rule and $\mathbf{I}$ an input instance for q. Because q is range restricted, it is easily verified that $adom(q(\mathbf{I})) \subseteq adom(q, \mathbf{I})$ (see Exercise 4.2). In other words, $q(\mathbf{I})$ contains only constants occurring in q or in $\mathbf{I}$. In particular, $q(\mathbf{I})$ is finite, and so it is an instance.

A straightforward algorithm for evaluating a rule q is to consider systematically all valuations with domain the set of variables occurring in q, and range the set of all constants occurring in the input or q. More efficient algorithms may be achieved, both by performing symbolic manipulations of the query and by using auxiliary data structures such as indexes. Such improvements are considered in Chapter 6.

Returning to the intuition, under the usual perspective a fundamental difference between the head and body of a rule $R_0 \leftarrow R_1, \ldots, R_n$ is that body relations are viewed as being stored, whereas the head relation is not. Thus, referring to the rule given earlier, the values of relations $R_1, \ldots, R_n$ are known because they are provided by the input instance

I. In other words, we are given the extension of $R_1, \ldots, R_n$; for this reason they are called *extensional* relations. In contrast, relation R_0 is not stored and its value is computed on request by the query; the rule gives only the "intension" or definition of R_0. For this reason we refer to R_0 as an *intensional* relation. In some cases, the database instance associated with $R_1, \ldots, R_n$ is called the *extensional database* (edb), and the rule itself is referred to as the *intensional database* (idb). Also, the defined relation is sometimes referred to as an *idb relation*.

We now present the first theoretical property of conjunctive queries. A query q over $\mathbf{R}$ is *monotonic* if for each $\mathbf{I}$, $\mathbf{J}$ over $\mathbf{R}$, $\mathbf{I} \subseteq \mathbf{J}$ implies that $q(\mathbf{I}) \subseteq q(\mathbf{J})$. A query q is *satisfiable* if there is some input $\mathbf{I}$ such that $q(\mathbf{I})$ is nonempty.

PROPOSITION 4.2.2 Conjunctive queries are monotonic and satisfiable.

Proof Let q be the rule-based conjunctive query

$$ans(u) \leftarrow R_1(u_1), \ldots, R_n(u_n).$$

For monotonicity, let $\mathbf{I} \subseteq \mathbf{J}$, and suppose that $t \in q(\mathbf{I})$. Then for some valuation ν over $var(q)$, $\nu(u_i) \in \mathbf{I}(R_i)$ for each $i \in [1, n]$, and $t = \nu(u)$. Because $\mathbf{I} \subseteq \mathbf{J}$, $\nu(u_i) \in \mathbf{J}(R_i)$ for each i, and so $t \in q(\mathbf{J})$.

For satisfiability, let $\mathbf{d}$ be the set of constants occurring in q, and let $a \in \mathbf{dom}$ be new. Define $\mathbf{I}$ over the relation schemas R of the rule body so that

$$\mathbf{I}(R) = (\mathbf{d} \cup \{a\})^{arity(R)}$$

[i.e., the set of all tuples formed from $(\mathbf{d} \cup \{a\})$ having arity $arity(R)$]. Finally, let ν map each variable in q to a. Then $\nu(u_i) \in \mathbf{I}(R_i)$ for $i \in [1, n]$, and so $\nu(u) \in q(\mathbf{I})$. Thus q is satisfiable. ■

The monotonicity of the conjunctive queries points to limitations in their expressive power. Indeed, one can easily exhibit queries that are nonmonotonic and therefore not conjunctive queries. For instance, the query "Which theaters in New York show only Woody Allen films?" is nonmonotonic.

We close this subsection by indicating how rule-based conjunctive queries can be used to express yes-no queries. For example, consider the query

(4.5) Is there a film directed by Bergman playing in Paris?

To provide an answer, we assume that relation name *ans* has arity 0. Then applying the rule

$$ans() \leftarrow Movies(x, \text{"Bergman"}, y), Pariscope(z, x, w)$$

returns the relation $\{\langle\rangle\}$ if the answer is yes, and returns $\{\}$ if the answer is no.

Tableau Queries

If we blur the difference between a variable and a constant, the body of a conjunctive query can be seen as an instance. This leads to a formulation of conjunctive queries called "tableau", which is closest to the visual form provided by QBE.

DEFINITION 4.2.3 The notion of *tableau* over a schema $\mathbf{R}$ (R) is defined exactly as was the notion of instance over $\mathbf{R}$ (R), except that both variables and constants may occur. A *tableau query* is simply a pair $(\mathbf{T}, u)$ [or (T, u)] where $\mathbf{T}$ is a tableau and each variable in u also occurs in $\mathbf{T}$. The free tuple u is called the *summary* of the tableau query.

The summary tuple u in a tableau query $(\mathbf{T}, u)$ represents the tuples included in the answer to the query. Thus the answer consists of all tuples u for which the pattern described by $\mathbf{T}$ is found in the database.

EXAMPLE 4.2.4 Let $\mathbf{T}$ be the tableau

Movies	*Title*	*Director*	*Actor*
	x_{ti}	"Bergman"	x_{ac}

Pariscope	*Theater*	*Title*	*Schedule*
	x_{th}	x_{ti}	x_s

Location	*Theater*	*Address*	*Phone Number*
	x_{th}	x_{ad}	x_p

The tableau query $(\mathbf{T}, \langle \mathit{Theater} : x_{th}, \ \mathit{Address} : x_{ad} \rangle)$ expresses query (4.4). If the unnamed perspective on tuples is used, then the names of the attributes are not included in u.

The notion of valuation is extended in the natural fashion to map tableaux[1] to instances. An *embedding* of tableau $\mathbf{T}$ into instance $\mathbf{I}$ is a valuation ν for the variables occurring in $\mathbf{T}$ such that $\nu(\mathbf{T}) \subseteq \mathbf{I}$. The semantics for tableau queries is essentially the same as for rule-based conjunctive queries: The output of $(\mathbf{T}, u)$ on input $\mathbf{I}$ consists of all tuples $\nu(u)$ where ν is an embedding of $\mathbf{T}$ into $\mathbf{I}$.

Aside from the fact that tableau queries do not indicate a relation name for the answer, they are syntactically close to the rule-based conjunctive queries. Furthermore, the alternative perspective provided by tableaux lends itself to the development of several natural results. Perhaps the most compelling of these arises in the context of the chase (see

[1] One *tableau*, two *tableaux*.

Chapter 8), which provides an elegant characterization of two conjunctive queries yielding identical results when the inputs satisfy certain dependencies.

A family of restricted tableaux called *typed* have been used to develop a number of theoretical results. A tableau query $q = (T, u)$ under the named perspective, where T is over relation schema R and $sort(u) \subseteq sort(R)$, is typed if no variable of T or t is associated with two distinct attributes in q. Intuitively, the term 'typed' is used because it is impossible for entries from different attributes to be compared. The connection between typed tableaux and conjunctive queries in the algebraic paradigm is examined in Exercises 4.19 and 4.20. Additional results concerning complexity issues around typed tableau queries are considered in Exercises 6.16 and 6.21 in Chapter 6. Typed tableaux also arise in connection with data dependencies, as studied in Part C.

Conjunctive Calculus

The third formalism for expressing conjunctive queries stems from predicate calculus. (A review of predicate calculus is provided in Chapter 2, but the presentation of the calculus in this and the following chapter is self-contained.)

We begin by presenting conjunctive calculus queries that can be viewed as syntactic variants of rule-based conjunctive queries. They involve simple use of conjunction and existential quantification. As will be seen, the full conjunctive calculus, defined later, allows unrestricted use of conjunction and existential quantification. This provides more flexibility in the syntax but, as will be seen, does not increase expressive power.

Consider the conjunctive query

$$ans(e_1, \ldots, e_m) \leftarrow R_1(u_1), \ldots, R_n(u_n).$$

A conjunctive calculus query that has the same semantics is

$$\{e_1, \ldots, e_m \mid \exists x_1, \ldots, x_k(R_1(u_1) \wedge \cdots \wedge R_n(u_n))\},$$

where $x_1, \ldots, x_k$ are all the variables occurring in the body and not the head. The symbol $\wedge$ denotes conjunction (i.e., "and"), and $\exists$ denotes existential quantification (intuitively, $\exists x \ldots$ denotes "there exists an x such that ..."). The term 'conjunctive query' stems from the presence of conjunctions in the syntax.

Example 4.2.5 In the calculus paradigm, query (4.4) can be expressed as follows:

$$\{x_{th},\ x_{ad} \mid \exists x_{ti}\, \exists x_{ac}\, \exists x_s\, \exists x_p\, (Movies(x_{ti}, \text{“Bergman”}, x_{ac}) \wedge Pariscope(x_{th}, x_{ti}, x_s) \wedge Location(x_{th}, x_{ad}, x_p))\}.$$

Note that some but not all of the existentially quantified variables play the role of anonymous variables, in the sense mentioned in Example 4.1.1.

The syntax used here can be viewed as a hybrid of the usual set-theoretic notation,

used to indicate the form of the query output, and predicate calculus, used to indicate what should be included in the output. As discussed in Chapter 2, the semantics associated with calculus formulas is a restricted version of the conventional semantics found in first-order logic.

We now turn to the formal definition of the syntax and semantics of the (full) conjunctive calculus.

DEFINITION 4.2.6 Let **R** be a database schema. A (*well-formed*) *formula* over **R** for the conjunctive calculus is an expression having one of the following forms:

(a) an atom over **R**;

(b) $(\varphi \wedge \psi)$, where φ and ψ are formulas over **R**; or

(c) $\exists x \varphi$, where x is a variable and φ is a formula over **R**.

In formulas we permit the abbreviation of $\exists x_1 \ldots \exists x_n$ by $\exists x_1, \ldots, x_n$.

The usual notion of "free" and "bound" occurrences of variables is now defined. An occurrence of variable x in formula φ is *free* if

(i) φ is an atom; or

(ii) $\varphi = (\psi \wedge \xi)$ and the occurrence of x is free in ψ or ξ; or

(iii) $\varphi = \exists y \psi$, x and y are distinct variables, and the occurrence of x is free in ψ.

An occurrence of x in φ is *bound* if it is not free. The set of *free variables* in φ, denoted $free(\varphi)$, is the set of all variables that have at least one free occurrence in φ.

DEFINITION 4.2.7 A *conjunctive calculus query* over database schema **R** is an expression of the form

$$\{e_1, \ldots, e_m \mid \varphi\},$$

where φ is a conjunctive calculus formula, $\langle e_1, \ldots, e_m \rangle$ is a free tuple, and the set of variables occurring in $\langle e_1, \ldots, e_m \rangle$ is exactly $free(\varphi)$. If the named perspective is being used, then attributes can be associated with output tuples by specifying a relation name R of arity m. The notation

$$\{\langle e_1, \ldots, e_m \rangle : A_1 \ldots A_m \mid \varphi\}$$

can be used to indicate the sort of the output explicitly.

To define the semantics of conjunctive calculus queries, it is convenient to introduce some notation. Recall that for finite set $V \subset \mathbf{var}$, a *valuation* over V is a total function ν from V to **dom**. This valuation will sometimes be viewed as a syntactic expression of the form

$$\{x_1/a_1, \ldots, x_n/a_n\},$$

where $x_1, \ldots, x_n$ is a listing of V and $a_i = \nu(x_i)$ for each $i \in [1, n]$. This may also be interpreted as a set. For example, if x is not in the domain of ν and $c \in \mathbf{dom}$, then $\nu \cup \{x/c\}$ denotes the valuation with domain $V \cup \{x\}$ that is identical to ν on V and maps x to c.

Now let $\mathbf{R}$ be a database schema, φ a conjunctive calculus formula over $\mathbf{R}$, and ν a valuation over $free(\varphi)$. Then $\mathbf{I}$ *satisfies* φ under ν, denoted $\mathbf{I} \models \varphi[\nu]$, if

(a) $\varphi = R(u)$ is an atom and $\nu(u) \in \mathbf{I}(R)$; or

(b) $\varphi = (\psi \wedge \xi)$ and[2] $\mathbf{I} \models \psi[\nu|_{free(\psi)}]$ and $\mathbf{I} \models \xi[\nu|_{free(\xi)}]$; or

(c) $\varphi = \exists x \psi$ and for some $c \in \mathbf{dom}$, $\mathbf{I} \models \psi[\nu \cup \{x/c\}]$.

Finally, let $q = \{e_1, \ldots, e_m \mid \varphi\}$ be a conjunctive calculus query over $\mathbf{R}$. For an instance $\mathbf{I}$ over $\mathbf{R}$, the *image* of $\mathbf{I}$ under q is

$$q(\mathbf{I}) = \{\nu(\langle e_1, \ldots, e_n \rangle) \mid \mathbf{I} \models \varphi[\nu] \text{ and } \nu \text{ is a valuation over } free(\varphi)\}.$$

The *active domain* of a formula φ, denoted $adom(\varphi)$, is the set of constants occurring in φ; and as with queries q, we use $adom(\varphi, \mathbf{I})$ to abbreviate $adom(\varphi) \cup adom(\mathbf{I})$. An easy induction on conjunctive calculus formulas shows that if $\mathbf{I} \models \varphi[\nu]$, then the range of ν is contained in $adom(\mathbf{I})$ (see Exercise 4.3). This implies, in turn, that to evaluate a conjunctive calculus query, one need only consider valuations with range contained in $adom(\varphi, \mathbf{I})$ and, hence, only a finite number of them. This pleasant state of affairs will no longer hold when disjunction or negation is incorporated into the calculus (see Section 4.5 and Chapter 5).

Conjunctive calculus formulas φ and ψ over $\mathbf{R}$ are *equivalent* if they have the same free variables and, for each $\mathbf{I}$ over $\mathbf{R}$ and valuation ν over $free(\varphi) = free(\psi)$, $\mathbf{I} \models \varphi[\nu]$ iff $\mathbf{I} \models \psi[\nu]$. It is easily verified that if φ and ψ are equivalent, and if Ψ' is the result of replacing an occurrence of φ by ψ in conjunctive calculus formula Ψ, then Ψ and Ψ' are equivalent (see Exercise 4.4).

It is easily verified that for all conjunctive calculus formulas φ, ψ, and ξ, $(\varphi \wedge \psi)$ is equivalent to $(\psi \wedge \varphi)$, and $(\varphi \wedge (\psi \wedge \xi))$ is equivalent to $((\varphi \wedge \psi) \wedge \xi)$. For this reason, we may view conjunction as a polyadic connective rather than just binary.

We next show that conjunctive calculus queries, which allow unrestricted nesting of $\exists$ and $\wedge$, are no more powerful than the simple conjunctive queries first exhibited, which correspond straightforwardly to rules. Thus the simpler conjunctive queries provide a normal form for the full conjunctive calculus. Formally, a conjunctive calculus query $q = \{u \mid \varphi\}$ is in *normal form* if φ has the form

$$\exists x_1, \ldots, x_m (R_1(u_1) \wedge \cdots \wedge R_n(u_n)).$$

Consider now the two *rewrite* (or *transformation*) *rules* for conjunctive calculus queries:

Variable substitution: replace subformula

$$\exists x \ \psi \ \text{ by } \ \exists y \ \psi^x_y,$$

[2] $\nu|_V$ for variable set V denotes the restriction of ν to V.

if y does not occur in ψ, where ψ_y^x denotes the formula obtained by replacing all free occurrences of x by y in ψ.

Merge-exists: replace subformula

$$(\exists y_1, \ldots, y_n \psi \wedge \exists z_1, \ldots, z_m \xi) \text{ by } \exists y_1, \ldots, y_n, z_1, \ldots, z_m (\psi \wedge \xi)$$

if $\{y_1, \ldots, y_n\}$ and $\{z_1, \ldots, z_m\}$ are disjoint, none of $\{y_1, \ldots, y_n\}$ occur (free or bound) in ξ, and none of $\{z_1, \ldots, z_m\}$ occur (free or bound) in ψ.

It is easily verified (see Exercise 4.4) that (1) application of these transformation rules to a conjunctive calculus formula yields an equivalent formula, and (2) these rules can be used to transform any conjunctive calculus formula into an equivalent formula in normal form. It follows that:

Lemma 4.2.8 Each conjunctive calculus query is equivalent to a conjunctive calculus query in normal form.

We now introduce formal notation for comparing the expressive power of query languages. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be two query languages (with associated semantics). Then $\mathcal{Q}_1$ is *dominated* by $\mathcal{Q}_2$ (or, $\mathcal{Q}_1$ is *weaker than* $\mathcal{Q}_2$), denoted $\mathcal{Q}_1 \sqsubseteq \mathcal{Q}_2$, if for each query q_1 in $\mathcal{Q}_1$ there is a query q_2 in $\mathcal{Q}_2$ such that $q_1 \equiv q_2$. $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are *equivalent*, denoted $\mathcal{Q}_1 \equiv \mathcal{Q}_2$, if $\mathcal{Q}_1 \sqsubseteq \mathcal{Q}_2$ and $\mathcal{Q}_2 \sqsubseteq \mathcal{Q}_1$.

Because of the close correspondence between rule-based conjunctive queries, tableau queries, and conjunctive calculus queries in normal form, the following is easily verified (see Exercise 4.15).

Proposition 4.2.9 The rule-based conjunctive queries, the tableau queries, and the conjunctive calculus are equivalent.

Although straightforward, the preceding result is important because it is the first of many that show equivalence between the expressive power of different query languages. Some of these results will be surprising because of the high contrast between the languages.

Incorporating Equality

We close this section by considering a simple variation of the conjunctive queries presented earlier, obtained by adding the capability of explicitly expressing equality between variables and/or constants. For example, query (4.4) can be expressed as

$$\begin{aligned} ans(x_{th}, x_{ad}) \leftarrow\ & Movies(x_{ti}, x_d, x_{ac}), x_d = \text{“Bergman”}, \\ & Pariscope(x_{th}, x_{ti}, x_s),\ Location(x_{th}, x_{ad}, x_p) \end{aligned}$$

and query (4.6) can be expressed as

$$ans(y_1, y_2) \leftarrow Movies(x_1, y_1, z_1),\ Movies(x_2, y_2, z_2),\ y_1 = z_2,\ y_2 = z_1.$$

It would appear that explicit equalities like the foregoing can be expressed by conjunctive queries without equalities by using multiple occurrences of the same variable or constant. Although this is basically true, two problems arise. First, unrestricted rules with equality may yield infinite answers. For example, in the rule

$$ans(x, y) \leftarrow R(x), y = z$$

y and z are not tied to relation R, and there are infinitely many valuations satisfying the body of the rule. To ensure finite answers, it is necessary to introduce an appropriate notion of range restriction. Informally, an unrestricted rule with equality is *range restricted* if the equalities require that each variable in the body be equal to some constant or some variable occurring in an atom $R(u_i)$; Exercise 4.5 explores the notion of range restriction in more detail. A *rule-based conjunctive query with equality* is a range-restricted rule with equality.

A second problem that arises is that the equalities in a rule with equality may cause the query to be unsatisfiable. (In contrast, recall that rules without equality are always satisfiable; see Proposition 4.2.2.) Consider the following query, in which R is a unary relation and a, b are distinct constants.

$$ans(x) \leftarrow R(x), x = a, x = b.$$

The equalities present in this query require that $a = b$, which is impossible. Thus there is no valuation satisfying the body of the rule, and the query yields the empty relation on all inputs. We use $q^{\emptyset:\mathbf{R},R}$ (or $q^{\emptyset}$ if $\mathbf{R}$ and R are understood) to denote the query that maps all inputs over $\mathbf{R}$ to the empty relation over R. Finally, note that one can easily check if the equalities in a conjunctive query with equality are unsatisfiable (and hence if the query is equivalent to $q^{\emptyset}$). This is done by computing the transitive closure of the equalities in the query and checking that no two distinct constants are required to be equal. Each satisfiable rule with equality is equivalent to a rule without equality (see Exercise 4.5c).

One can incorporate equality into tableau queries in a similar manner by adding separately a set of required equalities. Once again, no expressive power is gained if the query is satisfiable. Incorporating equality into the conjunctive calculus is considered in Exercise 4.6.

4.3 Query Composition and Views

We now present a digression that introduces the important notion of query composition and describe its relationship to database views. A main result here is that the rule-based conjunctive queries with equality are closed under composition.

Consider a database $\mathbf{R} = \{R_1, \ldots, R_n\}$. Suppose that we have a query q (in any of the preceding formalisms). Conceptually, this can be used to define a relation with new relation name S_1, which can be used in subsequent queries as any ordinary relation from $\mathbf{R}$. In particular, we can use S_1 in the definition of a new relation S_2, and so on. In this context, we could call each of $S_1, S_2, \ldots$ *intensional* (in contrast with the extensional relations of $\mathbf{R}$).

This perspective on query composition is expressed most conveniently within the rule-

based paradigm. Specifically, a *conjunctive query program* (with or without equality) is a sequence P of rules having the form

$$\begin{aligned} S_1(u_1) &\leftarrow body_1 \\ S_2(u_2) &\leftarrow body_2 \\ &\vdots \\ S_m(u_m) &\leftarrow body_m, \end{aligned}$$

where each S_i is distinct and not in $\mathbf{R}$; and for each $i \in [1, m]$, the only relation names that may occur in $body_i$ are $R_1, \ldots, R_n$ and $S_1, \ldots, S_{i-1}$. An instance $\mathbf{I}$ over $\mathbf{R}$ and the program P can be viewed as defining values for all of $S_1, \ldots, S_m$ in the following way: For each $i \in [1, m]$, $[P(\mathbf{I})](S_i) = q_i([P(\mathbf{I})])$, where q_i is the i^{th} rule and defines relation S_i in terms of $\mathbf{I}$ and the previous S_j's. If P is viewed as defining a single output relation, then this output is $[P(\mathbf{I})](S_m)$. Analogous to rule-based conjunctive queries, the relations in $\mathbf{R}$ are called *edb* relations, and the relations occurring in rule heads are called *idb* relations.

Example 4.3.1 Let $\mathbf{R} = \{Q, R\}$ and consider the conjunctive query program

$$\begin{aligned} S_1(x, z) &\leftarrow Q(x, y), R(y, z, w) \\ S_2(x, y, z) &\leftarrow S_1(x, w), R(w, y, v), S_1(v, z) \\ S_3(x, z) &\leftarrow S_2(x, u, v), Q(v, z). \end{aligned}$$

Figure 4.3 shows an example instance $\mathbf{I}$ for $\mathbf{R}$ and the values that are associated to S_1, S_2, S_3 by $P(\mathbf{I})$.

It is easily verified that the effect of the first two rules of P on S_2 is equivalent to the effect of the rule

$$\begin{aligned} S_2(x, y, z) \leftarrow Q(x_1, y_1), R(y_1, z_1, w_1), x = x_1, w = z_1, \\ R(w, y, v), Q(x_2, y_2), R(y_2, z_2, w_2), v = x_2, z = z_2. \end{aligned}$$

Alternatively, expressed without equality, it is equivalent to

$$S_2(x, y, z) \leftarrow Q(x, y_1), R(y_1, w, w_1), R(w, y, v), Q(v, y_2), R(y_2, z, w_2).$$

Note how variables are renamed to prevent undesired "cross-talk" between the different rule bodies that are combined to form this rule. The effect of P on S_3 can also be expressed using a single rule without equality (see Exercise 4.7).

It is straightforward to verify that if a permutation P' of P (i.e., a listing of the elements of P in a possibly different order) satisfies the restriction that relation names in a rule body must be in a previous rule head, then P' will define the same mapping as P. This kind of consideration will arise in a richer context when stratified negation is considered in Chapter 15.

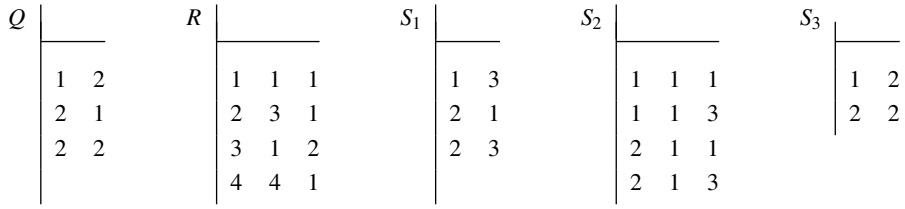


Figure 4.3: Application of a conjunctive query program

Example 4.3.2 Consider the following program P:

$$T(a, x) \leftarrow R(x)$$
$$S(x) \leftarrow T(b, x).$$

Clearly, P always defines the empty relation S, so it is not equivalent to any rule-based conjunctive query without equality. Intuitively, the use of the constants a and b in P masks the use of equalities, which in this case are contradictory and yield an unsatisfiable query.

Based on the previous examples, the following is easily verified (see Exercise 4.7).

Theorem 4.3.3 (Closure under Composition) If conjunctive query program P defines final relation S, then there is a conjunctive query q, possibly with equality, such that on all input instances $\mathbf{I}$, $q(\mathbf{I}) = [P(\mathbf{I})](S)$. Furthermore, if P is satisfiable, then q can be expressed without equality.

The notion of programs is based on the rule-based formalism of the conjunctive queries. In the other versions introduced previously and later in this chapter, the notation does not conveniently include a mechanism for specifying names for the output of intermediate queries. For the other formalisms we use a slightly more elaborate notation that permits the specification of these names. In particular, all of the formalisms are compatible with a functional, purely expression-based paradigm:

$$\begin{array}{lll} \textbf{let} & S_1 = q_1 & \textbf{in} \\ \textbf{let} & S_2 = q_2 & \textbf{in} \\ & \vdots & \\ \textbf{let} & S_{m-1} = q_{m-1} & \textbf{in} \\ & q_m & \end{array}$$

and with an imperative paradigm in which the intermediate query values are assigned to relation variables:

$$
\begin{aligned}
S_1 &:= q_1;\\
S_2 &:= q_2;\\
&\vdots\\
S_{m-1} &:= q_{m-1};\\
S_m &:= q_m.
\end{aligned}
$$

It is clear from Proposition 4.2.9 and Theorem 4.3.3 that the conjunctive calculus and tableau queries with equality are both closed under composition.

Composition and User Views

Recall that the top level of the three-level architecture for databases (see Chapter 1) consists of user *views* (i.e., versions of the data that are restructured and possibly restricted images of the database as represented at the middle level). In many cases these views are specified as queries (or query programs). These may be *materialized* (i.e., a physical copy of the view is stored and maintained) or *virtual* (i.e., relevant information about the view is computed as needed). In the latter case, queries against the view generate composed queries against the underlying database, as illustrated by the following example.

Example 4.3.4 Consider the view over schema {*Marilyn*, *Champo-info*} defined by the following two rules:

$$
\begin{aligned}
\mathit{Marilyn}(x_t) &\leftarrow \mathit{Movies}(x_t, x_d, \text{“Monroe”})\\
\mathit{Champo\text{-}info}(x_t, x_s, x_p) &\leftarrow \mathit{Pariscope}(\text{“Le Champo”}, x_t, x_s),\\
&\qquad\qquad \mathit{Location}(\text{“Le Champo”}, x_a, x_p).
\end{aligned}
$$

The conjunctive query “What titles in *Marilyn* are featured at the Le Champo at 21:00?” can be expressed against the view as

$$
\mathit{ans}(x_t) \leftarrow \mathit{Marilyn}(x_t),\ \mathit{Champo\text{-}info}(x_t, \text{“21:00”}, x_p).
$$

Assuming that the view is virtual, evaluation of this query is accomplished by considering the composition of the query with the view definition. This composition can be rewritten as

$$
\begin{aligned}
\mathit{ans}(x_t) \leftarrow\ & \mathit{Movies}(x_t, x_d, \text{“Monroe”}),\\
& \mathit{Pariscope}(\text{“Le Champo”}, x_t, \text{“21:00”})\\
& \mathit{Location}(\text{“Le Champo”}, x_a, x_p).
\end{aligned}
$$

An alternative expression specifying both view and query now follows. (Expressions from the algebraic versions of the conjunctive queries could also be used here.)

$$\begin{aligned} \mathit{Marilyn} &:= \{x_t \mid \exists x_d(\mathit{Movies}(x_t, x_d, \text{“Monroe”}))\}; \\ \mathit{Champo\text{-}info} &:= \{x_t, x_s, x_p \mid \exists x_a(\mathit{Location}(\text{“Le Champo”}, x_t, x_s) \\ &\qquad \wedge \mathit{Location}(\text{“Le Champo”}, x_a, x_p)\}; \\ \mathit{ans} &:= \{x_t \mid \mathit{Marilyn}(x_t) \wedge \exists x_p(\mathit{Champo\text{-}info}(x_t, \text{“21:00”}, x_p))\}. \end{aligned}$$

This example illustrates the case in which a query is evaluated over a single view; evaluation of the query involves a two-layer composition of queries. If a series of nested views is defined, then query evaluation can involve query compositions having two or more layers.

4.4 Algebraic Perspectives

The use of algebra operators provides a distinctly different perspective on the conjunctive queries. There are two distinct algebras associated with the conjunctive queries, and they stem, respectively, from the named, ordered-tuple perspective and the unnamed, function-based perspective. After presenting the two algebras, their equivalence with the conjunctive queries is discussed.

The Unnamed Perspective: The SPC Algebra

The algebraic paradigm for relational queries is based on a family of unary and binary operators on relation instances. Although their application must satisfy some typing constraints, they are polymorphic in the sense that each of these operators can be applied to instances of an infinite number of arities or sorts. For example, as suggested in Chapter 3, the union operator can take as input any two relation instances having the same sort.

Three primitive algebra operators form the *unnamed conjunctive algebra*: selection, projection, and cross-product (or Cartesian product). This algebra is more often referred to as the *SPC algebra*, based on the first letters of the three operators that form it. (This convention will be used to specify other algebras as well.) An example is given before the formal definition of these operators.

Example 4.4.1 We show how query (4.4) can be built up using the three primitive operators. First we use selection to extract the tuples of *Movies* that have Bergman as director.

$$I_1 := \sigma_{2=\text{“Bergman”}}(\mathit{Movies})$$

Next a family of “wide” (six columns wide, in fact) tuples is created by taking the cross-product of I_1 and *Pariscope*.

$$I_2 := I_1 \times \mathit{Pariscope}$$

Another selection is performed to focus on the members of I_2 that have first and fifth columns equal.

$$I_3 := \sigma_{1=5}(I_2)$$

In effect, the cross-product followed by this selection finds a matching of tuples from I_1 and *Pariscope* that agree on the *Title* coordinates.

At this point we are interested only in the theaters where these films are playing, so we use projection to discard the unneeded columns, yielding a unary relation.

$$I_4 := \pi_4(I_3)$$

Finally, this is paired with *Location* and projected on the *Theater* and *Address* columns to yield the answer.

$$I_5 := \pi_{2,3}(\sigma_{1=2}(I_4 \times \mathit{Location}))$$

The development just given uses SPC expressions in the context of a simple imperative language with assignment. In the pure SPC algebra, this query is expressed as

$$\pi_{2,3}(\sigma_{1=2}(\pi_4(\sigma_{1=5}(\sigma_{2=\text{“Bergman”}}(\mathit{Movies}) \times \mathit{Pariscope})) \times \mathit{Location})).$$

Another query that yields the same result is

$$\pi_{4,8}(\sigma_{4=7}(\sigma_{1=5}(\sigma_{2=\text{“Bergman”}}(\mathit{Movies} \times \mathit{Pariscope} \times \mathit{Location})))).$$

This corresponds closely to the conjunctive calculus query of Example 4.2.5.

Although the algebraic operators have a procedural feel to them, algebraic queries are used by most relational database systems as high-level specifications of desired output. Their actual implementation is usually quite different from the original form of the query, as will be discussed in Section 6.1.

We now formally define the three operators forming the SPC algebra.

Selection: This can be viewed as a “horizontal” operator. The two primitive forms are $\sigma_{j=a}$ and $\sigma_{j=k}$, where j, k are positive integers and $a \in \mathbf{dom}$. [In practice, we usually surround constants with quotes (“ ”).] The operator $\sigma_{j=a}$ takes as input any relation instance I with arity $\geq j$ and returns as output an instance of the same arity. In particular,

$$\sigma_{j=a}(I) = \{t \in I \mid t(j) = a\}.$$

The operator $\sigma_{j=k}$ for positive integers j, k is defined analogously for inputs with arity $\geq \max\{j, k\}$. This is sometimes called *atomic* selection; generalizations of selection will be defined later.

Projection: This "vertical" operator can be used to delete and/or permute columns of a relation. The general form of this operator is $\pi_{j_1,\ldots,j_n}$, where $j_1, \ldots, j_n$ is a possibly empty sequence of positive integers (the empty sequence is written [ ]), possibly with repeats. This operator takes as input any relation instance with arity $\geq \max\{j_1, \ldots, j_n\}$ (where the max of $\emptyset$ is 0) and returns an instance with arity n. In particular,

$$\pi_{j_1,\ldots,j_n}(I) = \{\langle t(j_1), \ldots, t(j_n)\rangle \mid t \in I\}.$$

Cross-product (or Cartesian product): This operator provides the capability for combining relations. It takes as inputs a pair of relations having arbitrary arities n and m and returns a relation with arity $n + m$. In particular, if $arity(I) = n$ and $arity(J) = m$, then

$$I \times J = \{\langle t(1), \ldots, t(n), s(1), \ldots, s(m)\rangle \mid t \in I \text{ and } s \in J\}.$$

Cross-product is associative and noncommutative and has the nonempty 0-ary relation $\{\langle\rangle\}$ as left and right identity. Because it is associative, we sometimes view cross-product as a polyadic operator and write, for example, $I_1 \times \cdots \times I_n$.

We extend the cross-product operator to tuples in the natural fashion—that is $u \times v$ is a tuple with arity = $arity(u) + arity(v)$.

The SPC algebra is the family of well-formed expressions containing relation names and one-element unary constants and closed under the application of the selection, projection, and cross-product operators just defined. Each expression is considered to be defined over a given database schema and has an associated output arity. We now give the formal, inductive definition.

Let **R** be a database schema. The *base SPC (algebra) queries* and output arities are

Input relation: Expression R; with arity equal to $arity(R)$.

Unary singleton constant: Expression $\{\langle a\rangle\}$, where $a \in$ **dom**; with *arity* equal to 1.

The family of *SPC (algebra) queries* contains all base SPC queries and, for SPC queries q_1, q_2 with arities α_1, α_2, respectively,

Selection: $\sigma_{j=a}(q_1)$ and $\sigma_{j=k}(q_1)$ whenever $j, k \leq \alpha_1$ and $a \in$ **dom**; these have arity α_1.

Projection: $\pi_{j_1,\ldots,j_n}(q_1)$, where $j_1, \ldots, j_n \leq \alpha_1$; this has arity n.

Cross product: $q_1 \times q_2$; this has arity $\alpha_1 + \alpha_2$.

In practice, we sometimes use brackets to surround algebraic queries, such as $[R \times \sigma_{1=a}(S)](\mathbf{I})$. In addition, parentheses may be dropped if no ambiguity results.

The *semantics* of these queries is defined in the natural manner (see Exercise 4.8).

The SPC algebra includes unsatisfiable queries, such as $\sigma_{1=a}(\sigma_{1=b}(R))$, where $arity(R) \geq 1$ and $a \neq b$. This is equivalent to $q^{\emptyset}$.

As explored in Exercise 4.22, permitting as base SPC queries constant queries that are not unary (i.e., expressions of the form $\{\langle a_1\rangle, \ldots, \langle a_n\rangle\}$) yields expressive power greater than the rule-based conjunctive queries with equality. This is also true of selection formulas in which disjunction is permitted. As will be seen in Section 4.5, these capabilities

are subsumed by including an explicit union operator into the SPC algebra. Permitting negation in selection formulas also extends the expressive power of the SPC algebra (see Exercise 4.27b).

Before leaving SPC algebra, we mention three operators that can be simulated by the primitive ones. The first is intersection ($\cap$), which is easily simulated (see Exercise 4.28). The other two operators involve generalizations of the selection and cross-product operators. The resulting algebra is called the *generalized SPC algebra*. We shall introduce a normal form for generalized SPC algebra expressions.

The first operator is a generalization of selection to permit the specification of multiple conditions. A *positive conjunctive selection formula* is a conjunction $F = \gamma_1 \wedge \cdots \wedge \gamma_n$ ($n \geq 1$), where each conjunct γ_i has the form $j = a$ or $j = k$ for positive integers j, k and $a \in \mathbf{dom}$; and a *positive conjunctive selection operator* is an expression of the form σ_F, where F is a positive conjunctive selection formula. The intended typing and semantics for these operators is clear, as is the fact that they can be simulated by a composition of selections as defined earlier.

The second operator, called *equi-join*, is a binary operator that combines cross-product and selection. A (well-formed) equi-join operator is an expression of the form $\bowtie_F$ where $F = \gamma_1 \wedge \cdots \wedge \gamma_n$ ($n \geq 1$) is a conjunction such that each conjunct γ_i has the form $j = k$. An equi-join operator $\bowtie_F$ can be applied to any pair I, J of relation instances, where the $arity(I) \geq$ the maximum integer occurring on the left-hand side of any equality in F, and $arity(J) \geq$ the maximum integer occurring on the right-hand side of any equality in F. Given an equi-join expression $I \bowtie_F J$, let F' be the result of replacing each condition $j = k$ in F by $j = arity(I) + k$. Then the semantics of $I \bowtie_F J$ is given by $\sigma_{F'}(I \times J)$. As with cross-product, equi-join is also defined for pairs of tuples, with an undefined output if the tuples do not satisfy the conditions specified.

We now develop a normal form for SPC algebra. We stress that this normal form is useful for theoretical purposes and, in general, represents a costly way to compute the answer of a given query (see Chapter 6).

An SPC algebra expression is in *normal form* if it has the form

$$\pi_{j_1,\ldots,j_n}(\{\langle a_1\rangle\} \times \cdots \times \{\langle a_m\rangle\} \times \sigma_F(R_1 \times \cdots \times R_k)),$$

where $n \geq 0$; $m \geq 0$; $a_1, \ldots, a_m \in \mathbf{dom}$; $\{1, \ldots, m\} \subseteq \{j_1, \ldots, j_n\}$; $R_1, \ldots, R_k$ are relation names (repeats permitted); and F is a positive conjunctive selection formula.

Proposition 4.4.2 For each (generalized) SPC query q there is a generalized SPC query q' in normal form such that $q \equiv q'$.

The proof of this proposition (see Exercise 4.12) is based on repeated application of the following eight *equivalence-preserving SPC algebra rewrite rules* (or *transformations*).

Merge-select: replace $\sigma_F(\sigma_{F'}(q))$ by $\sigma_{F \wedge F'}(q)$.

Merge-project: replace $\pi_{\vec{j}}(\pi_{\vec{k}}(q))$ by $\pi_{\vec{l}}(q)$, where $l_i = k_{j_i}$ for each term l_i in $\vec{l}$.

Push-select-through-project: replace $\sigma_F(\pi_{\vec{j}}(q))$ by $\pi_{\vec{j}}(\sigma_{F'}(q))$, where F' is obtained from F by replacing all coordinate values i by j_i.

Push-select-through-singleton: replace $\sigma_{1=j}(\langle a \rangle \times q)$ by $\langle a \rangle \times \sigma_{(j-1)=a}(q)$.

Associate-cross: replace $((q_1 \times \cdots \times q_n) \times q)$ by $(q_1 \times \cdots q_n \times q)$, and replace $(q \times (q_1 \times \cdots q_n))$ by $(q \times q_1 \times \cdots q_n)$.

Commute-cross: replace $(q \times q')$ by $\pi_{\vec{j}\,\vec{j'}}(q' \times q)$, where $\vec{j} = arity(q') + 1, \ldots, arity(q') + arity(q)$, and $\vec{j'} = 1, \ldots, arity(q')$.

Push-cross-through-select: replace $(\sigma_F(q) \times q')$ by $\sigma_F(q \times q')$, and replace $(q \times \sigma_F(q'))$ by $\sigma_{F'}(q \times q')$, where F' is obtained from F by replacing all coordinate values i by $i + arity(q)$.

Push-cross-through-project: replace $(\pi_{\vec{j}}(q) \times q')$ by $\pi_{\vec{j}}(q \times q')$, and replace $(q \times \pi_{\vec{j}}(q'))$ by $\pi_{\vec{j'}}(q \times q')$, where $\vec{j'}$ is obtained from $\vec{j}$ by replacing all coordinate values i by $i + arity(q)$.

For a set $\mathcal{S}$ of rewrite rules and algebra expressions q, q', write $q \to_{\mathcal{S}} q'$, or simply $q \to q'$ if $\mathcal{S}$ is understood from the context, if q' is the result of replacing a subexpression of q according to one of the rules in $\mathcal{S}$. Let $\overset{*}{\to}_{\mathcal{S}}$ denote the reflexive, transitive closure of $\to_{\mathcal{S}}$.

A family $\mathcal{S}$ of rewrite rules is *sound* if $q \to_{\mathcal{S}} q'$ implies $q \equiv q'$. If $\mathcal{S}$ is sound, then clearly $q \overset{*}{\to}_{\mathcal{S}} q'$ implies $q \equiv q'$.

It is easily verified that the foregoing set of rewrite rules is sound and that for each SPC query q there is a normal form SPC query q' such that q' is in normal form, and $q \overset{*}{\to} q'$ (see Exercise 4.12).

In Section 6.1, we describe an approach to optimizing the evaluation of conjunctive queries using rewrite rules. For example, in that context, the merge-select and merge-project transformations are helpful, as are the *inverses* of the push-cross-through-select and push-cross-through-project.

Finally, note that an SPC query may require, as the result of transitivity, the equality of two distinct constants. Thus there are unsatisfiable SPC queries equivalent to $q^{\emptyset}$. This is analogous to the logic-based conjunctive queries with equality. It is clear, using the normal form, that one can check whether an SPC query is $q^{\emptyset}$ by examining the selection formula F. The set of SPC queries that are not equivalent to $q^{\emptyset}$ forms the *satisfiable SPC algebra*.

The Named Perspective: The SPJR Algebra

In Example 4.4.1, the relation I_3 was constructed using selection and cross-product by the expression $\sigma_{1=5}(I_1 \times Pariscope)$. As is often the case, the columns used in this selection are labeled by the same attribute. In the context of the named perspective on tuples, this suggests a natural variant of the cross-product operator (and of the equi-join operator) that is called *natural join* and is denoted by $\bowtie$. Informally, the natural join requires the tuples that are concatenated to agree on the common attributes.

EXAMPLE 4.4.3 The natural join of *Movies* and *Pariscope* is

$$\begin{aligned}
&\mathit{Movies} \bowtie \mathit{Pariscope} \\
&\quad = \{u \text{ with sort } \mathit{Title\ Director\ Actor\ Theater\ Schedule} \mid \\
&\qquad\qquad \text{for some } v \in \text{Movies and } w \in \text{Pariscope}, \\
&\qquad\qquad u[\mathit{Title\ Director\ Actor}] = v \text{ and } u[\mathit{Theater\ Title\ Schedule}] = w\} \\
&\quad = \pi_{1,2,3,4,6}(\mathit{Movies} \bowtie_{1=2} \mathit{Pariscope})
\end{aligned}$$

(assuming that the sort of the last expression corresponds to that of the previous expression). More generally, using the natural analog of projection and selection for the named perspective, query (4.4) can be expressed as

$$\pi_{\mathit{Theater},\mathit{Address}}((\sigma_{\mathit{Director}=\text{“Bergman”}}(\mathit{Movies}) \bowtie \mathit{Pariscope}) \bowtie \mathit{Location}).$$

As suggested by the preceding example, natural join can be used in the named context to replace certain equi-joins arising in the unnamed context. However, a problem arises if two relations sharing an attribute A are to be joined but without forcing equality on the A coordinates, or if a join is to be formed based on the equality of attributes not sharing the same name. For example, consider the query

(4.8) List pairs of actors that acted in the same movie.

To answer this, one would like to join the *Movies* relation with itself but matching only on the *Title* column. This will be achieved by first creating a copy *Movies′* of *Movies* in which the attribute *Director* has been renamed to *Director′* and *Actor* to *Actor′*; joining this with *Movies*; and finally projecting onto the *Actor* and *Actor′* columns. Renaming is also needed for query (4.6) (see Exercise 4.11).

The *named conjunctive algebra* has four primitive operators: *selection*, essentially as before; *projection*, now with repeats not permitted; (*natural*) *join*; and *renaming*. It is thus referred to as the *SPJR algebra*. As with the SPC algebra, we define the individual operators and then indicate how they are combined to form a typed, polymorphic algebra. In each case, we indicate the sorts of input and output. If a relation name is needed for the output, then it is assumed to be chosen to have the correct sort.

Selection: The selection operators have the form $\sigma_{A=a}$ and $\sigma_{A=B}$, where $A, B \in \mathbf{att}$ and $a \in \mathbf{dom}$. These operators apply to any instance I with $A \in \mathit{sort}(I)$ [respectively, $A, B \in \mathit{sort}(I)$] and are defined in analogy to the unnamed selection, yielding an output with the same sort as the input.

Projection: The projection operator has the form $\pi_{A_1,\ldots,A_n}$, $n \geq 0$ (repeats not permitted) and operates on all inputs having sort containing $\{A_1, \ldots, A_n\}$, producing output with sort $\{A_1, \ldots, A_n\}$.

(*Natural*) *join:* This operator, denoted $\bowtie$, takes arbitrary inputs I and J having sorts V and

W, respectively, and produces an output with sort equal to $V \cup W$. In particular,

$$I \bowtie J = \{t \text{ over } V \cup W \mid \text{for some } v \in I \text{ and } w \in J, \\ t[V] = v \text{ and } t[W] = w\}.$$

When $sort(I) = sort(J)$, then $I \bowtie J = I \cap J$, and when $sort(I) \cap sort(J) = \emptyset$, then $I \bowtie J$ is the cross-product of I and J. The join operator is associative, commutative, and has the nonempty 0-ary relation $\{\langle\rangle\}$ as left and right identity. Because it is associative, we sometimes view join as a polyadic operator and write, for example, $I_1 \bowtie \cdots \bowtie I_n$.

As with cross-product and equi-join, natural join is extended to operate on pairs of tuples, with an undefined result if the tuples do not match on the appropriate attributes.

Renaming: An *attribute renaming* for a finite set U of attributes is a one-one mapping from U to **att**. An attribute renaming f for U can be described by specifying the set of pairs $(A, f(A))$, where $f(A) \neq A$; this is usually written as $A_1A_2 \ldots A_n \to B_1B_2 \ldots B_n$ to indicate that $f(A_i) = B_i$ for each $i \in [1, n]$ $(n \geq 0)$. A *renaming operator* for inputs over U is an expression δ_f, where f is an attribute renaming for U; this maps to outputs over $f[U]$. In particular, for I over U,

$$\delta_f(I) = \{v \text{ over } f[U] \mid \text{for some } u \in I,\ v(f(A)) = u(A) \text{ for each } A \in U\}.$$

Example 4.4.4 Let I, J be the two relations, respectively over R, S, given in Fig. 4.4. Then $I \bowtie J$, $\sigma_{A=1}(I)$, $\delta_{BC \to B'A}(J)$, and $\pi_A(I)$ are also shown there. Let K be the one-tuple relation $\langle A : 1, C : 9\rangle$. Then $\pi_{A,B}(I \bowtie K)$ coincides with $\sigma_{A=1}(I)$ and $J \bowtie K = \{\langle A : 1, B : 8, C : 9\rangle\}$.

The *base* SPJR algebra queries are:

Input relation: Expression R; with *sort* equal to $sort(R)$.

Unary singleton constant: Expression $\{\langle A : a\rangle\}$, where $a \in$ **dom**; with *sort* A.

The remainder of the syntax and semantics of the SPJR algebra is now defined in analogy to those of the SPC algebra (see Exercise 4.8).

Example 4.4.5 Consider again Fig. 4.4. Let $\mathbf{I}$ be the instance over $\{R, S\}$ such that $\mathbf{I}(R) = I$ and $\mathbf{I}(S) = J$. Then $[R]$ is a query and the answer to that query, denoted $R(\mathbf{I})$, is just I. Figure 4.4 also gives the values of $S(\mathbf{I})$, $[R \bowtie S](\mathbf{I})$, $[\sigma_{A=1}(R)](\mathbf{I})$, $[\delta_{BC \to B'A}(S)](\mathbf{I})$, and $[\pi_A(R)](\mathbf{I})$. Let $K_A = \{\langle A : 1\rangle\}$ and $K_C = \{\langle C : 9\rangle\}$. Then $[K_A]$ and $[K_C]$ are constant queries, and $[K_A \bowtie K_C]$ is a query that evaluates (on all inputs) to the relation K of Example 4.4.4.

As with the SPC algebra, we introduce a natural generalization of the selection operator for the SPJR algebra. In particular, the notions of *positive conjunctive selection formula* and *positive conjunctive selection operator* are defined for the context in complete

R

A	B
1	2
4	2
6	6
7	7
1	7
1	6

S

B	C
2	3
2	5
6	4
8	9

$[R \bowtie S]$

A	B	C
1	2	3
1	2	5
4	2	3
4	2	5
6	6	4
1	6	4

$[\sigma_{A=1}(R)]$

A	B
1	2
1	7
1	6

$[\delta_{BC \to B'A}(S)]$

B'	A
2	3
2	5
6	4
8	9

$[\pi_A(R)]$

A
1
4
6
7

Figure 4.4: Examples of SPJR operators

analogy to the unnamed case. Including this operator yields the *generalized SPJR algebra*.

A normal form result analogous to that for the SPC algebra is now developed. In particular, an SPJR algebra expression is in *normal form* if it has the form

$$\pi_{B_1,\ldots,B_n}(\{\langle A_1 : a_1\rangle\} \bowtie \cdots \bowtie \{\langle A_m : a_m\rangle\} \bowtie \sigma_F(\delta_{f_1}(R_1) \bowtie \cdots \bowtie \delta_{f_k}(R_k))),$$

where $n \geq 0$; $m \geq 0$; $a_1, \ldots, a_m \in \mathbf{dom}$; each of $A_1, \ldots, A_m$ occurs in $B_1, \ldots, B_n$; the A_i's are distinct; $R_1, \ldots, R_k$ are relation names (repeats permitted); δ_{f_j} is a renaming operator for $sort(R_j)$ for each $j \in [1, k]$ and no A_i's occur in any $\delta_{f_j}(R_j)$; the sorts of $\delta_{f_1}(R_1), \ldots, \delta_{f_k}(R_k)$ are pairwise disjoint; and F is a positive conjunctive selection formula. The following is easily verified (see Exercise 4.12).

Proposition 4.4.6 For each (generalized) SPJR query q, there is a generalized SPJR query q' in normal form such that $q \equiv q'$.

The set of SPJR queries not equivalent to $q^{\emptyset}$ forms the *satisfiable SPJR algebra*.

Equivalence Theorem

We now turn to the main result of the chapter, showing the equivalence of the various formalisms introduced so far for expressing conjunctive queries. As shown earlier, the three logic-based versions of the conjunctive queries are equivalent. We now show that the SPC and SPJR algebras are also equivalent to each other and then obtain the equivalence of the algebraic languages and the three logic-based languages.

LEMMA 4.4.7 The SPC and SPJR algebras are equivalent.

Crux We prove the inclusion SPC algebra $\sqsubseteq$ SPJR algebra; the converse is similar (see Exercise 4.14). Let q be the following normal form SPC query:

$$\pi_{j_1,\ldots,j_n}(\{\langle a_1\rangle\} \times \cdots \times \{\langle a_m\rangle\} \times \sigma_F(R_1 \times \cdots \times R_k)).$$

We now describe an SPJR query q' that is equivalent to q; q' has the following form:

$$\pi_{A_{j_1},\ldots,A_{j_n}}(\{\langle A_1 : a_1\rangle\} \bowtie \cdots \bowtie \{\langle A_m : a_m\rangle\} \bowtie \sigma_G(\delta_{f_1}(R_1) \bowtie \cdots \bowtie \delta_{f_k}(R_k))).$$

We use the renaming functions so that the attributes of $\delta_{f_t}(R_t)$ are $A_s, \ldots, A_{s'}$, where $s, \ldots, s'$ are the coordinate positions of R_t in the expression $R_1 \times \cdots \times R_k$ and modify F into G accordingly. In a little more detail, for each $r \in [1, k]$ let $\beta(t) = m + \Sigma_{s=0}^{t} arity(R_s)$, and let $A_{m+1}, \ldots, A_{\beta(k)}$ be new attributes. For each $t \in [1, k]$, choose δ_{f_t} so that it maps the i^{th} attribute of R_t to the attribute $A_{\beta(t-1)+i}$. To define G, first define the function γ from coordinate positions to attribute names so that $\gamma(j) = A_{m+j}$, extend γ to be the identity on constants, and extend it further in the natural manner to map unnamed selection formulas to named selection formulas. Finally, set $G = \gamma(F)$. It is now straightforward to verify that $q' \equiv q$. ■

It follows immediately from the preceding lemma that the satisfiable SPC algebra and the satisfiable SPJR algebra are equivalent.

The equivalence between the two algebraic languages and the three logic-based languages holds with a minor caveat involving the empty query $q^{\emptyset}$. As noted earlier, the SPC and SPJR algebras can express $q^{\emptyset}$, whereas the logic-based languages cannot, unless extended with equality. Hence the equivalence result is stated for the satisfiable SPC and SPJR algebras.

Theorem 4.3.3 (i.e., the closure of the rule-based conjunctive queries under composition) is used in the proof of this result. The closures of the SPC and SPJR algebras under composition are, of course, immediate.

THEOREM 4.4.8 (Equivalence Theorem) The rule-based conjunctive queries, tableau queries, conjunctive calculus queries, satisfiable SPC algebra, and satisfiable SPJR algebra are equivalent.

Proof The proof can be accomplished using the following steps:

(i) satisfiable SPC algebra $\sqsubseteq$ rule-based conjunctive queries; and

(ii) rule-based conjunctive queries $\sqsubseteq$ satisfiable SPC algebra.

We briefly consider how steps (i) and (ii) might be demonstrated; the details are left to the reader (Exercise 4.15). For (i), it is sufficient to show that each of the SPC algebra operations can be simulated by a rule. Indeed, then the inclusion follows from the fact that rule-based conjunctive queries are closed under composition by Theorem 4.3.3 and that

satisfiable rules with equality can be expressed as rules without equality. The simulation of algebra operations by rules is as follows:

1. $P \times Q$, where P and Q are not constant relations, corresponds to $ans(\vec{x}, \vec{y}) \leftarrow P(\vec{x}), Q(\vec{y})$, where $\vec{x}$ and $\vec{y}$ contain no repeating variables; in the case when P (Q) are constant relations, $\vec{x}$ ($\vec{y}$) are the corresponding constant tuples.
2. $\sigma_F(R)$ corresponds to $ans(\vec{x}) \leftarrow R(\sigma_F(\vec{y}))$, where $\vec{y}$ consists of distinct variables, $\sigma_F(\vec{y})$ denotes the vector of variables and constants obtained by merging variables of $\vec{y}$ with other variables or with constants according to the (satisfiable) selection formula F, and $\vec{x}$ consists of the distinct variables in $\sigma_F(\vec{y})$.
3. $\pi_{j_1 \ldots j_n}(R)$ corresponds to $ans(x_{j_1} \ldots x_{j_n}) \leftarrow R(x_1 \ldots x_m)$, where $x_1, \ldots, x_m$ are distinct variables.

Next consider step (ii). Let $ans(\vec{x}) \leftarrow R_1(\vec{x}_1), \ldots, R_n(\vec{x}_n)$ be a rule. There is an equivalent SPC algebra query in normal form that involves the cross-product of $R_1, \ldots, R_n$, a selection reflecting the constants and repeating variables occurring in $\vec{x}_1, \ldots, \vec{x}_n$, a further cross-product with constant relations corresponding to the constants in $\vec{x}$, and finally a projection extracting the coordinates corresponding to $\vec{x}$. ■

An alternative approach to showing step (i) of the preceding theorem is explored in Exercise 4.18.

4.5 Adding Union

As indicated by their name, conjunctive queries are focused on selecting data based on a conjunction of conditions. Indeed, each atom added to a rule potentially adds a further restriction to the tuples produced by the rule. In this section we consider a natural mechanism for adding a disjunctive capability to the conjunctive queries. Specifically, we add a *union* operator to the SPC and SPJR algebras, and we add natural analogs of it to the rule-based and tableau-based paradigms. Incorporating union into the conjunctive calculus raises some technical difficulties that are resolved in Chapter 5. This section also considers the evaluation of queries with union and introduces a more restricted mechanism for incorporating a disjunctive capability.

We begin with some examples.

EXAMPLE 4.5.1 Consider the following query:

(4.10) Where can I see "Annie Hall" or "Manhattan"?

Although this cannot be expressed as a conjunctive query (see Exercise 4.22), it is easily expressed if union is added to the SPJR algebra:

$$\pi_{Theater}(\sigma_{Title=\text{"Annie Hall"}}(Pariscope) \cup \sigma_{Title=\text{"Manhattan"}}(Pariscope)).$$

An alternative formulation of this uses an extended selection operator that permits disjunctions in the selection condition:

$$\pi_{Theater}(\sigma_{Title=\text{“Annie Hall”}\vee Title=\text{“Manhattan”}}(Pariscope)).$$

As a final algebraic alternative, this can be expressed in the original SPJR algebra but permitting nonsingleton constant relations as base expressions:

$$\pi_{Theater}(Pariscope \bowtie \{\langle Title\text{: “Annie Hall”}\rangle, \langle Title\text{: “Manhattan”}\rangle\}).$$

The rule-based formalism can accommodate this query by permitting more than one rule with the same relation name in the head and taking the union of their outputs as the answer:

$$ans(x_t) \leftarrow Pariscope(x_t, \text{“Annie Hall”}, x_s)$$
$$ans(x_t) \leftarrow Pariscope(x_t, \text{“Manhattan”}, x_s).$$

Consider now the following query:

(4.11) What are the films with Allen as actor or director?

This query can be expressed using any of the preceding formalisms, except for the SPJR algebra extended with nonsingleton constant relations as base expressions (see Exercise 4.22).

Let I_1, I_2 be two relations with the same arity. As standard in mathematics, $I_1 \cup I_2$ is the relation having this arity and containing the union of the two sets of tuples. The definition of the *SPCU algebra* is obtained by extending the definition of the SPC algebra to include the *union* operator. The *SPJRU algebra* is obtained in the same fashion, except that union can only be applied to expressions having the same sort.

The SPCU and SPJRU algebras can be generalized by extending the selection operator (and join, in the case of SPC) as before. We can then define *normal forms* for both algebras, which are expressions consisting of one or more normal form SPC (SPJR) expressions combined using a polyadic union operator (see Exercise 4.23). As suggested by the previous example, disjunction can also be incorporated into selection formulas with no increase in expressive power (see Exercise 4.22).

Turning now to rule-based conjunctive queries, the simplest way to incorporate the capability of union is to consider sets of rules all having the same relation name in the head. These queries are evaluated by taking the union of the output of the individual rules.

This can be generalized without increasing the expressive power by incorporating something analogous to query composition. A *nonrecursive datalog program* (*nr-datalog program*) over schema **R** is a set of rules

$$S_1 \leftarrow body_1$$
$$S_2 \leftarrow body_2$$
$$\vdots$$
$$S_m \leftarrow body_m,$$

where no relation name in $\mathbf{R}$ occurs in a rule head; the same relation name may appear in more than one rule head; and there is some ordering $r_1, \ldots, r_m$ of the rules so that the relation name in the head of r_i does not occur in the body of a rule r_j whenever $j \leq i$.

The term 'nonrecursive' is used because recursion is not permitted. A simple example of a recursive rule is

$$ancestor(x, z) \leftarrow parent(x, y),\ ancestor(y, z).$$

A fixpoint operator is used to give the semantics for programs involving such rules. Recursion is the principal topic of Part D.

As in the case of rule-based conjunctive query programs, the query is evaluated on input $\mathbf{I}$ by evaluating each rule in (one of) the order(s) satisfying the foregoing property and forming unions whenever two rules have the same relation name in their heads. Equality atoms can be added to these queries, as they were for the rule-based conjunctive queries.

In general, a nonrecursive datalog program P over $\mathbf{R}$ is viewed as having a database schema as target. Program P can also be viewed as mapping from $\mathbf{R}$ to a single relation (see Exercise 4.24).

Turning to tableau queries, a *union of tableaux query* over schema $\mathbf{R}$ (or R) is an expression of the form $(\{\mathbf{T}_1, \ldots, \mathbf{T}_n\}, u)$, where $n \geq 1$ and $(\mathbf{T}_i, u)$ is a tableau query over $\mathbf{R}$ for each $i \in [1, n]$. The semantics of these queries is obtained by evaluating the queries $(\mathbf{T}_i, u)$ independently and then taking the union of their results. Equality is incorporated into these queries by permitting each of the queries $(\mathbf{T}_i, u)$ to have equality.

We can now state (see Exercise 4.25) the following:

THEOREM 4.5.2 The following have equivalent expressive power:

1. the nonrecursive datalog programs (with single relation target),
2. the SPCU queries,
3. the SPJRU queries.

The union of tableau queries is weaker than the aforementioned languages with union. This is essentially because the definition of union of tableau queries does not allow separate summary rows for each tableau in the union. With just one summary row, the nonrecursive datalog query

$$ans(a) \leftarrow$$
$$ans(b) \leftarrow$$

cannot be expressed as a union of tableaux query.

As with conjunctive queries, it is easy to show that the conjunctive queries with union and equality are closed under composition.

Union and the Conjunctive Calculus

At first glance, it would appear that the power of union can be added to the conjunctive calculus simply by permitting *disjunction* (denoted $\vee$) along with conjunction as a binary connective for formulas. This approach, however, can have serious consequences.

EXAMPLE 4.5.3 Consider the following "query":

$$q = \{x, y, z \mid R(x, y) \vee R(y, z)\}.$$

Speaking intuitively, the "answer" of q on nonempty instance I will be (using a slight abuse of notation)

$$q(I) = (I \times \mathbf{dom}) \cup (\mathbf{dom} \times I).$$

This is an infinite set of tuples and thus not an instance according to the formal definition.

Informally, the query q of the previous example is not "safe." This notion is one of the central topics that needs to be resolved when using the first-order predicate calculus as a relational query language, and it is studied in Chapter 5. We return there to the issue of adding union to the conjunctive calculus (see also Exercise 4.26).

Bibliographic Notes

Codd's pioneering article [Cod70] on the relational model introduces the first relational query language, a named algebra. The predicate calculus was adapted to the relational model in [Cod72b], where it was shown to be essentially equivalent to the algebra. The conjunctive queries, in the calculus paradigm, were first introduced in [CM77]. Their equivalence with the SPC algebra is also shown there.

Typed tableau queries appeared as a two-dimensional representation of a subset of the conjunctive queries in [ASU79b] along with a proof that all typed restricted SPJ algebra expressions over one relation can be expressed using them. A precursor to the typed tableau queries is found in [ABU79], which uses a technique related to tableaux to analyze the join operator. [ASU79a, ASSU81, CV81] continued the investigation of typed tableau queries; [SY80] extends tableau queries to include union and a limited form of difference; and [Klu88] extends them to include inequalities and order-based comparators. Tableau queries have also played an important role in dependency theory; this will be discussed in Part C.

Many of the results in this chapter (including, for example, the equivalence of the SPC and SPJR algebras and closure of conjunctive queries under composition) are essentially part of the folklore.

Exercises

Exercise 4.1 Express queries (4.1–4.3) and (4.5–4.9) as (a) rule-based conjunctive queries, (b) conjunctive calculus queries, (c) tableau queries, (d) SPC expressions, and (e) SPJR expressions.

Exercise 4.2 Let $\mathbf{R}$ be a database schema and q a rule.

(a) Prove that $q(\mathbf{I})$ is finite for each instance $\mathbf{I}$ over $\mathbf{R}$.

(b) Show an upper bound, given instance $\mathbf{I}$ of $\mathbf{R}$ and output arity for conjunctive query q, for the number of tuples that can occur in $q(\mathbf{I})$. Show that this bound can be achieved.

Exercise 4.3 Let $\mathbf{R}$ be a database schema and $\mathbf{I}$ an instance of $\mathbf{R}$.

(a) Suppose that φ is a conjunctive calculus formula over $\mathbf{R}$ and ν is a valuation for $free(\varphi)$. Prove that $\mathbf{I} \models \varphi[\nu]$ implies that the image of ν is contained in $adom(\mathbf{I})$.

(b) Prove that if q is a conjunctive calculus query over $\mathbf{R}$, then only a finite number of valuations need to be considered when evaluating $q(\mathbf{I})$. (Note: The presence of existential quantifiers may have an impact on the set of valuations that need to be considered.)

Exercise 4.4

(a) Let φ and ψ be equivalent conjunctive calculus formulas, and suppose that Ψ' is the result of replacing an occurrence of φ by ψ in conjunctive calculus formula Ψ. Prove that Ψ and Ψ' are equivalent.

(b) Prove that the application of the rewrite rules *rename* and *merge-exists* to a conjunctive calculus formula yields an equivalent formula.

(c) Prove that these rules can be used to transform any conjunctive calculus formula into an equivalent formula in normal form.

Exercise 4.5

(a) Formally define the syntax and semantics of rule-based conjunctive queries with equality and conjunctive calculus queries with equality.

(b) As noted in the text, logic-based conjunctive queries with equality can generally yield infinite answers if not properly restricted. Give a definition for *range-restricted* rule-based and conjunctive calculus queries with equality that ensures that queries satisfying this condition always yield a finite answer.

(c) Prove for each rule-based conjunctive query with equality q that either $q \equiv q^{\emptyset}$ or $q \equiv q'$ for some rule-based conjunctive query q' without equality. Give a polynomial time algorithm that decides whether $q \equiv q^{\emptyset}$, and if not, constructs an equivalent rule-based conjunctive query q'.

(d) Prove that each rule-based conjunctive query with equality but no constants is equivalent to a rule-based conjunctive query without equality.

Exercise 4.6 Extend the syntax of the conjunctive calculus to include equality. Give a syntactic condition that ensures that the answer to a query q on $\mathbf{I}$ involves only constants from $adom(q, \mathbf{I})$ and such that the answer can be obtained by considering only valuations whose range is contained in $adom(q, \mathbf{I})$.

Exercise 4.7 Give a proof of Theorem 4.3.3.

Exercise 4.8

(a) Give a formal definition for the semantics of the SPC algebra.

(b) Give a formal definition for the syntax and semantics of the SPJR algebra.

Exercise 4.9 Consider the algebra consisting of all SPJR queries in which constants do not occur.

(a) Define a normal form for this algebra.

(b) Is this algebra closed under composition?

(c) Is this algebra equivalent to the rule-based conjunctive queries without constants or equality?

Exercise 4.10 Under the named perspective, a selection operator is *constant based* if it has the form $\sigma_{A=a}$, where $A \in \mathbf{att}$ and $a \in \mathbf{dom}$. Prove or disprove: Each SPJR algebra expression is equivalent to an SPJR algebra expression all of whose selection operators are constant based.

Exercise 4.11 Prove that queries (4.6 and 4.8) cannot be expressed using the SPJ algebra (i.e., that renaming is needed).

Exercise 4.12

(a) Prove that the set of SPC transformations presented after the statement of Proposition 4.4.2 is sound (i.e., preserves equivalence).

(b) Prove Proposition 4.4.2.

(c) Prove that each SPJR query is equivalent to one in normal form. In particular, exhibit a set of equivalence-preserving SPJR algebra transformations used to demonstrate this result.

Exercise 4.13

(a) Prove that the nonempty 0-ary relation is the left and right identity for cross product and for natural join.

(b) Prove that for a fixed relation schema S, there is an identity for union for relations over S. What if S is not fixed?

(c) Let S be a relational schema. For the binary operations $\alpha \in \{\bowtie, \cup\}$, does there exist a relation I such that $I \alpha J = I$ for each relation J over S?

Exercise 4.14 Complete the proof of Lemma 4.4.7 by showing the inclusion SPJR algebra $\sqsubseteq$ SPC algebra.

Exercise 4.15

(a) Prove Proposition 4.2.9.

(b) Complete the proof of Theorem 4.4.8.

Exercise 4.16 Consider the problem of defining restricted versions of the SPC and SPJR algebras that are equivalent to the rule-based conjunctive queries without equality. Find natural restricted versions, or explain why they do not exist.

Exercise 4.17 Let q be a tableau query and q' the SPC query corresponding to it via the translation sketched in Theorem 4.4.8. If q has r rows and q' has j joins of database (nonconstant) relations, show that $j = r - 1$.

♠ **Exercise 4.18**

(a) Develop an inductive algorithm that translates a satisfiable SPC query q into a tableau query by associating a tableau query to each subquery of q.

(b) Do the same for SPJR queries.

(c) Show that if q is a satisfiable SPC (SPRJ) query with n joins (not counting joins involving constant relations), then the tableau of the corresponding tableau query has $n + 1$ rows.

♠ **Exercise 4.19** [ASU79b] This exercise examines the connection between typed tableaux and a subset of the SPJ algebra. A *typed restricted* SPJ algebra expression over R is an SPJR algebra expression that uses only $[R]$ as base expressions and only constant-based selection (i.e., having the form $\sigma_{A=a}$ for constant a), projection, and (natural) join as operators.

(a) Describe a natural algorithm that maps typed restricted SPJ queries q over R into equivalent typed tableau queries $q' = (T, u)$ over R, where $|T| =$ (the number of join operations in q) $+ 1$.

(b) Show that $q = (\{\langle x, y_1\rangle, \langle x_1, y_1\rangle, \langle x_1, y\rangle\}, \langle x, y\rangle)$ is not the image of any typed restricted SPJ query under the algorithm of part (a).

★ (c) [ASSU81] Prove that the tableau query q of part (b) is not equivalent to any typed restricted SPJ algebra expression.

Exercise 4.20 [ASU79b] A typed tableau query $q = (T, u)$ with T over relation R is *repeat restricted* if

1. If $A \in sort(u)$, then no variable in $\pi_A(T) - \{u(A)\}$ occurs more than once in T.
2. If $A \notin sort(u)$, then at most one variable in $\pi_A(T)$ occurs more than once in T.

Prove that if $q = (T, u)$ is a typed repeat-restricted tableau query over R, then there is a typed restricted SPJ query q' such that the image of q' under the algorithm of Exercise 4.19 part (a) is q.

Exercise 4.21 Extend Proposition 4.2.2 to include disjunction (i.e., union).

Exercise 4.22 The following query is used in this exercise:

(4.15) Produce a binary relation that includes all tuples $\langle t$, "excellent"$\rangle$ where t is a movie directed by Allen, and all tuples $\langle t$, "superb"$\rangle$ where t is a movie directed by Hitchcock.

(a) Show that none of queries (4.10–4.15) can be expressed using the SPC or SPJR algebras.

A *positive selection formula* for the SPC and SPJR algebras is a selection formula as before, except that disjunction can be used in addition to conjunction. Define the *S+PC algebra* to be the SPC algebra extended to permit arbitrary positive selection operators; and define the *S+PJR algebra* analogously.

(b) Determine which of queries (4.10–4.15) can be expressed using the S+PJR algebra.

Define the *SPC-1* algebra* to be the SPC algebra, except that nonsingleton unary constant relations can be used as base queries; and define the *SPC-n* algebra* to be the SPC algebra,

except that nonsingleton constant relations of arbitrary arity can be used as base queries. Define the *SPJR-1** and *SPJR-n** algebras analogously.

(c) Determine which of queries (4.10–4.15) can be expressed using the SPJR-1* and SPJR-n^* algebras.

(d) Determine the relative expressive powers of the S+PC, SPC-1*, SPC-n^*, and SPCU algebras.

Exercise 4.23 Give precise definitions for normal forms for the SPCU and SPJRU algebras, and prove that all expressions from these algebras have an equivalent in normal form.

Exercise 4.24 An nr-datalog program P is in *normal form* if all relation names in rule heads are identical. Prove that each nonrecursive datalog query with single relation target has an equivalent in normal form.

Exercise 4.25 Prove Theorem 4.5.2.

★ **Exercise 4.26** Recall the discussion in Section 4.5 about disjunction in the conjunctive calculus.

(a) Consider the query $q = \{x|\varphi(x)\}$, where

$$\varphi(x) \equiv R(x) \wedge \exists y, z(S(y, x) \vee S(x, z)).$$

Let $\mathbf{I}$ be an instance over $\{R, S\}$. Using the natural extension of the notion of *satisfies* to disjunction, show for each subformula of φ with form $\exists\omega\psi$, and each valuation ν over $free(\exists\omega\psi)$ with range contained in $adom(\mathbf{I})$ that: there exists $c \in \mathbf{dom}$ such that $\mathbf{I} \models \psi[\nu \cup \{w/c\}]$ iff there exists $c \in adom(\mathbf{I})$ such that $\mathbf{I} \models \psi[\nu \cup \{w/c\}]$. Conclude that this query can be evaluated by considering only valuations whose range is contained in $adom(\mathbf{I})$.

(b) The *positive existential* (*relational*) *calculus* is the relational calculus query language in which query formulas are constructed using $\wedge$, $\vee$, $\exists$. Define a condition on positive existential calculus queries that guarantees that the answer involves only constants from $adom(q, \mathbf{I})$ and such that the answer can be obtained by considering only valuations whose range is contained in $adom(q, \mathbf{I})$. Extend the restriction for the case when equality is allowed in the calculus.

(c) Prove that the family of restricted positive existential calculus queries defined in the previous part has expressive power equivalent to the rule-based conjunctive queries with union and that this result still holds if equality is added to both families of queries.

Exercise 4.27

(a) Consider as an additional algebraic operation, the *difference*. The semantics of $q - q'$ is given by $[q - q'](\mathbf{I}) = q(\mathbf{I}) - q'(\mathbf{I})$. Show that the difference cannot be simulated in the SPCU or SPJRU algebras. (*Hint:* Use the monotonicity property of these algebras.)

(b) Negation can be added to (generalized) selection formulas in the natural way—that is, if γ is a selection formula, then so is $(\neg\gamma)$. Give a precise definition for the syntax and semantics of selection with negation. Prove that the SPCU algebra cannot simulate selections of the form $\sigma_{\neg 1=2}(R)$ or $\sigma_{\neg 1=a}(R)$.

Exercise 4.28 Show that intersection can be expressed in the SPC algebra.

★ **Exercise 4.29**

(a) Prove that there is no redundant operation in the set $\chi = \{\sigma, \pi, \times, \cup\}$ of unnamed algebra operators (i.e., for each operator α in the set, exhibit a schema and an algebraic query q over that schema such that q cannot be expressed with $\chi - \{\alpha\}$).

(b) Prove the analogous result for the set of named operators $\{\sigma, \pi, \bowtie, \delta, \cup\}$.

Exercise 4.30 An *inequality atom* is an expression of the form $x \neq y$ or $x \neq a$, where x, y are variables and a is a constant. Assuming that the underlying domain has a total order, a *comparison atom* is an expression of the form $x\theta y$, $x\theta a$, or $a\theta x$, where θ ranges over $<$, $\leq$, $>$, and $\geq$.

(a) Show that the family of rule-based conjunctive queries with equality and inequality strictly dominates the family of rule-based conjunctive queries with equality.

(b) Assuming that the underlying domain has a total order, describe the relationships between the expressive powers of the family of rule-based conjunctive queries with equality; the family of rule-based conjunctive queries with equality and inequality; the family of rule-based conjunctive queries with equality and comparison atoms; and the family of rule-based conjunctive queries with equality, inequality, and comparison atoms.

(c) Develop analogous extensions and results for tableau queries, the conjunctive calculus, and SPC and SPJR algebras.

★ **Exercise 4.31** For some films, we may not want to store any actor name. Add to the domain a constant $\perp$ meaning unknown information. Propose an extension of the SPJR queries to handle unknown information (see Chapter 19).

5 Adding Negation: Algebra and Calculus

Alice: *Conjunctive queries are great. But what if I want to see a movie that doesn't feature Woody Allen?*
Vittorio: *We have to introduce negation.*
Sergio: *It is basically easy.*
Riccardo: *But the calculus is a little feisty.*

As indicated in the previous chapter, the conjunctive queries, even if extended by union, cannot express queries such as the following:

(5.1) What are the Hitchcock movies in which Hitchcock did not play?

(5.2) What movies are featured at the Gaumont Opera but not at the Gaumont les Halles?

(5.3) List those movies for which all actors of the movie have acted under Hitchcock's direction.

This chapter explores how negation can be added to all forms of the conjunctive queries (except for the tableau queries) to provide the power needed to express such queries. This yields languages in the various paradigms that have the same expressive power. They include relational algebra, relational calculus, and nonrecursive datalog with negation. The class of queries they express is often referred to as the *first-order queries* because relational calculus is essentially first-order predicate calculus without function symbols. These languages are of fundamental importance in database systems. They provide adequate power for many applications and at the same time can be implemented with reasonable efficiency. They constitute the basis for the standard commercial relational languages, such as SQL.

In the case of the algebras, negation is added using the set difference operator, yielding the language(s) generally referred to as *relational algebra* (Section 5.1). In the case of the rule-based paradigm, we consider negative literals in the bodies of rules, which are interpreted as the absence of the corresponding facts; this yields *nonrecursive datalog*$^{\neg}$ (Section 5.2).

Adding negation in the calculus paradigm raises some serious problems that require effort and care to resolve satisfactorily. In the development in this chapter, we proceed in two stages. First (Section 5.3) we introduce the calculus, illustrate the problematic issues of "safety" and domain independence, and develop some simple solutions for them. We also show the equivalence between the algebra and the calculus at this point. The material in this section provides a working knowledge of the calculus that is adequate for understanding the study of its extensions in Parts D and E. The second stage in our study of the calculus

(Section 5.4) focuses on the important problem of finding syntactic restrictions on the calculus that ensure domain independence.

The chapter concludes with brief digressions concerning how aggregate functions can be incorporated into the algebra and calculus (Section 5.5), and concerning the emerging area of constraint databases, which provide a natural mechanism for representing and manipulating infinite databases in a finite manner (Section 5.6).

From the theoretical perspective, the most important aspects of this chapter include the demonstration of the equivalence of the algebra and calculus (including a relatively direct transformation of calculus queries into equivalent algebra ones) and the application of the classical proof technique of structural induction used on both calculus formulas and algebra expressions.

5.1 The Relational Algebras

Incorporating the *difference* operator, denoted '$-$', into the algebras is straightforward. As with union and intersection, this can only be applied to expressions that have the same sort, in the named case, or arity, in the unnamed case.

EXAMPLE 5.1.1 In the named algebra, query (5.1) is expressed by

$$\pi_{Title}\sigma_{Director=\text{“Hitchcock”}}(Movies) - \pi_{Title}\sigma_{Actor=\text{“Hitchcock”}}(Movies).$$

The *unnamed relational algebra* is obtained by adding the difference operator to the SPCU algebra. It is conventional also to permit the *intersection* operator, denoted '$\cap$' in this algebra, because it is simulated easily using cross-product, select, and project or using difference (see Exercise 5.4). Because union is present, nonsingleton constant relations may be used in this algebra. Finally, the selection operator can be extended to permit negation (see Exercise 5.4).

The *named relational algebra* is obtained in an analogous fashion, and similar generalizations can be developed.

As shown in Exercise 5.5, the family of unnamed algebra operators $\{\sigma, \pi, \times, \cup, -\}$ is nonredundant, and the same is true for the named algebra operators $\{\sigma, \pi, \bowtie, \delta, \cup, -\}$. It is easily verified that the algebras are not monotonic, nor are all algebra queries satisfiable (see Exercise 5.6). In addition, the following is easily verified (see Exercise 5.7):

PROPOSITION 5.1.2 The unnamed and named relational algebras have equivalent expressive power.

The notion of *composition* of relational algebra queries can be defined in analogy to the composition of conjunctive queries described in the previous chapter. It is easily verified that the relational algebras, and hence the other equivalent languages presented in this chapter, are closed under composition.

5.2 Nonrecursive Datalog with Negation

To obtain a rule-based language with expressive power equivalent to the relational algebra, we extend nonrecursive datalog programs by permitting negative literals in rule bodies. This yields the *nonrecursive datalog with negation* also denoted *nonrecursive datalog*$^\neg$ and *nr-datalog*$^\neg$.

A *nonrecursive datalog*$^\neg$ (*nr-datalog*$^\neg$) rule is a rule of the form

$$q: \quad S(u) \leftarrow L_1, \ldots, L_n,$$

where S is a relation name, u is a free tuple of appropriate arity, and each L_i is a *literal* [i.e., an expression of the form $R(v)$ or $\neg R(v)$, where R is a relation name and v is a free tuple of appropriate arity and where S does not occur in the body]. This rule is *range restricted* if each variable x occurring in the rule occurs in at least one literal of the form $R(v)$ in the rule body. Unless otherwise specified, all datalog$^\neg$ rules considered are assumed to be range restricted.

To give the *semantics* of the foregoing rule q, let $\mathbf{R}$ be a relation schema that includes all of the relation names occurring in the body of the rule q, and let $\mathbf{I}$ be an instance of $\mathbf{R}$. Then the *image* of $\mathbf{I}$ under q is

$$\begin{aligned} q(\mathbf{I}) = \{\nu(u) \mid \nu \text{ is a valuation and for each } i \in [1, n], \\ \nu(u_i) \in \mathbf{I}(R_i), \text{ if } L_i = R_i(u_i), \text{ and} \\ \nu(u_i) \notin \mathbf{I}(R_i), \text{ if } L_i = \neg R_i(u_i)\}. \end{aligned}$$

In general, this image can be expressed as a difference $q_1 - q_2$, where q_1 is an SPC query and q_2 is an SPCU query (see Exercise 5.9).

Equality may be incorporated by permitting literals of the form $s = t$ and $s \neq t$ for terms s and t. The notion of *range restriction* in this context is defined as it was for rule-based conjunctive queries with equality. The semantics are defined in the natural manner.

To obtain the full expressive power of the relational algebras, we must consider sets of nr-datalog$^\neg$ rules; these are analogous to the nr-datalog programs introduced in the previous chapter. A *nonrecursive datalog*$^\neg$ *program* (with or without equality) over schema $\mathbf{R}$ is a sequence

$$\begin{aligned} S_1 &\leftarrow body_1 \\ S_2 &\leftarrow body_2 \\ &\vdots \\ S_m &\leftarrow body_m \end{aligned}$$

of nr-datalog$^\neg$ rules, where no relation name in $\mathbf{R}$ occurs in a rule head; the same relation name may appear in more than one rule head; and there is some ordering $r_1, \ldots, r_m$ of the rules so that the relation name in the head of a rule r_i does not occur in the body of a rule r_j whenever $j \leq i$. The *semantics* of these programs are entirely analogous to

the semantics of nr-datalog programs. An *nr-datalog¬ query* is a query defined by some nr-datalog¬ program with a specified target relation.

Example 5.2.1 Assume that each movie in *Movies* has one director. Query (5.1) is answered by

$$ans(x) \leftarrow Movies(x, \text{“Hitchcock”}, z), \\ \neg Movies(x, \text{“Hitchcock”}, \text{“Hitchcock”}).$$

Query (5.3) is answered by

$$Hitch\text{-}actor(z) \leftarrow Movies(x, \text{“Hitchcock”}, z)$$
$$not\text{-}ans(x) \leftarrow Movies(x, y, z), \neg Hitch\text{-}actor(z)$$
$$ans(x) \leftarrow Movies(x, y, z), \neg not\text{-}ans(x).$$

Care must be taken when forming nr-datalog¬ programs. Consider, for example, the following program, which forms a kind of merging of the first two rules of the previous program. (Intuitively, the first rule is a combination of the first two rules of the preceding program, using variable renaming in the spirit of Example 4.3.1.)

$$bad\text{-}not\text{-}ans(x) \leftarrow Movies(x, y, z), \neg Movies(x', \text{“Hitchcock”}, z), \\ Movies(x', \text{“Hitchcock”}, z'),$$
$$ans(x) \leftarrow Movies(x, y, z), \neg bad\text{-}not\text{-}ans(x)$$

Rather than expressing query (5.3), it expresses the following:

(5.3′) (Assuming that all movies have only one director) list those movies for which all actors of the movie acted in *all* of Hitchcock's movies.

It is easily verified that each nr-datalog¬ program with equality can be simulated by an nr-datalog¬ program not using equality (see Exercise 5.10). Furthermore (see Exercise 5.11), the following holds:

Proposition 5.2.2 The relational algebras and the family of nr-datalog¬ programs that have single relation output have equivalent expressive power.

5.3 The Relational Calculus

Adding negation in the calculus paradigm yields an extremely flexible query language, which is essentially the predicate calculus of first-order logic (without function symbols). However, this flexibility brings with it a nontrivial cost: If used without restriction, the calculus can easily express queries whose “answers” are infinite. Much of the theoretical development in this and the following section is focused on different approaches to make

the calculus "safe" (i.e., to prevent this and related problems). Although considerable effort is required, it is a relatively small price to pay for the flexibility obtained.

This section first extends the syntax of the conjunctive calculus to the full calculus. Then some intuitive examples are presented that illustrate how some calculus queries can violate the principle of "domain independence." A variety of approaches have been developed to resolve this problem based on the use of both semantic and syntactic restrictions.

This section focuses on semantic restrictions. The first step in understanding these is a somewhat technical definition based on "relativized interpretation" for the semantics of (arbitrary) calculus queries; the semantics are defined relative to different "underlying domains" (i.e., subsets of **dom**). This permits us to give a formal definition of domain independence and leads to a family of different semantics for a given query.

The section closes by presenting the equivalence of the calculus under two of the semantics with the algebra. This effectively closes the issue of expressive power of the calculus, at least from a semantic point of view. One of the semantics for the calculus presented here is the "active domain" semantics; this is particularly convenient in the development of theoretical results concerning the expressive power of a variety of languages presented in Parts D and E.

As noted in Chapter 4, the calculus presented in this chapter is sometimes called the *domain calculus* because the variables range over elements of the underlying domain of values. Exercise 5.23 presents the *tuple calculus*, whose variables range over tuples, and its equivalence with the domain calculus and the algebra. The tuple calculus and its variants are often used in practice. For example, the practical languages SQL and Quel can be viewed as using tuple variables.

Well-Formed Formulas, Revisited

We obtain the relational calculus from the conjunctive calculus with equality by adding negation ($\neg$), disjunction ($\vee$), and universal quantification ($\forall$). (Explicit equality is needed to obtain the full expressive power of the algebras; see Exercise 5.12.) As will be seen, both disjunction and universal quantification can be viewed as consequences of adding negation, because $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$ and $\forall x\varphi \equiv \neg\exists x\neg\varphi$.

The formal definition of the syntax of the relational calculus is a straightforward extension of that for the conjunctive calculus given in the previous chapter. We include the full definition here for the reader's convenience. A *term* is a constant or a variable. For a given input schema **R**, the *base formulas* include, as before, atoms over **R** and equality (inequality) atoms of the form $e = e'$ ($e \neq e'$) for terms e, e'. The (*well-formed*) *formulas* of the relational calculus over **R** include the base formulas and formulas of the form

(a) $(\varphi \wedge \psi)$, where φ and ψ are formulas over **R**;

(b) $(\varphi \vee \psi)$, where φ and ψ are formulas over **R**;

(c) $\neg\varphi$, where φ is a formula over **R**;

(d) $\exists x\varphi$, where x is a variable and φ a formula over **R**;

(e) $\forall x\varphi$, where x is a variable and φ a formula over **R**.

As with conjunctive calculus,

$\exists x_1, x_2, \ldots, x_m \varphi$ abbreviates $\exists x_1 \exists x_2 \ldots \exists x_m \varphi$, and
$\forall x_1, x_2, \ldots, x_m \varphi$ abbreviates $\forall x_1 \forall x_2 \ldots \forall x_m \varphi$.

It is sometimes convenient to view the binary connectives $\wedge$ and $\vee$ as polyadic connectives. In some contexts, $e \neq e'$ is viewed as an abbreviation of $\neg(e = e')$.

It is often convenient to include two additional logical connectives, *implies* ($\rightarrow$) and *is equivalent to* ($\leftrightarrow$). We view these as syntactic abbreviations as follows:

$$\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$$
$$\varphi \leftrightarrow \psi \equiv (\varphi \wedge \psi) \vee (\neg\varphi \wedge \neg\psi).$$

The notions of *free* and *bound* occurrences of variables in a formula, and of *free*(φ) for formula φ, are defined analogously to their definition for the conjunctive calculus. In addition, the notion of *relational calculus query* is defined, in analogy to the notion of conjunctive calculus query, to be an expression of the form

$\{\langle e_1, \ldots, e_m \rangle : A_1, \ldots, A_m \mid \varphi\}$, in the named perspective,
$\{e_1, \ldots, e_m \mid \varphi\}$, in the unnamed perspective,
or if the sort is understood from the context,

where $e_1, \ldots, e_m$ are terms, repeats permitted, and where the set of variables occurring in $e_1, \ldots, e_m$ is exactly *free*(φ).

Example 5.3.1 Suppose that each movie has just one director. Query (5.1) can be expressed in the relational calculus as

$$\{x_t \mid \exists x_a \mathit{Movies}(x_t, \text{“Hitchcock”}, x_a) \wedge \neg \mathit{Movies}(x_t, \text{“Hitchcock”}, \text{“Hitchcock”})\}.$$

Query (5.3) is expressed by

$$\{x_t \mid \exists x_d, x_a\ \mathit{Movies}(x_t, x_d, x_a) \wedge \forall y_a\ (\exists y_d \mathit{Movies}(x_t, y_d, y_a) \rightarrow \exists z_t\ \mathit{Movies}(z_t, \text{“Hitchock”}, y_a))\}.$$

The first conjunct ensures that the variable x_t ranges over titles in the current value of *Movies*, and the second conjunct enforces the condition on actors of the movie identified by x_t.

“Unsafe” Queries

Before presenting the alternative semantics for the relational calculus, we present an intuitive indication of the kinds of problems that arise if the conventional definitions from predicate calculus are adapted directly to the current context.

The fundamental problems of using the calculus are illustrated by the following expressions:

$$(\textit{unsafe-1}) \quad \{x \mid \neg \textit{Movies}(\text{“Cries and Whispers”}, \text{“Bergman”}, x)\}$$
$$(\textit{unsafe-2}) \quad \{x, y \mid \textit{Movies}(\text{“Cries and Whispers”}, \text{“Bergman”}, x) \vee \textit{Movies}(y, \text{“Bergman”}, \text{“Ullman”})\}.$$

If the usual semantics of predicate calculus are adapted directly to this context, then the query (*unsafe-1*) produces all tuples $\langle a \rangle$ where $a \in$ **dom** and $\langle$“Cries and Whispers”, “Bergman”, $a\rangle$ is not in the input. Because all input instances are by definition finite, the query yields an infinite set on all input instances. The same is true of query (*unsafe-2*), even though it does not use explicit negation.

An intuitively appealing approach to resolving this problem is to view the different relation columns as typed and to insist that variables occurring in a given column range over only values of the appropriate type. For example, this would imply that the answer to query (*unsafe-1*) is restricted to the set of actors. This approach is not entirely satisfactory because query answers now depend on the domains of the types. For example, different answers are obtained if the type *Actor* includes all and only the current actors [i.e., persons occurring in $\pi_{Actor}(\textit{Movies})$] or includes all current *and potential* actors. This illustrates that query (*unsafe-1*) is not independent of the underlying domain within which the query is interpreted (i.e., it is not “domain independent”). The same is true of query (*unsafe-2*).

Even if the underlying domain is finite, users will typically not know the exact contents of the domains used for each variable. In this case it would be disturbing to have the result of a user query depend on information not directly under the user's control. This is another argument for permitting only domain-independent queries.

A related but more subtle problem arises with regard to the interpretation of quantified variables. Consider the query

$$(\textit{unsafe-3}) \quad \{x \mid \forall y R(x, y)\}.$$

The answer to this query is necessarily finite because it is a subset of $\pi_1(R)$. However, the query is not domain independent. To see why, note that if y is assumed to range over all of **dom**, then the answer is always the empty relation. On the other hand, if the underlying domain of interpretation is finite, it is possible that the answer will be nonempty. (This occurs, for example, if the domain is $\{1, \ldots, 5\}$, and the input for R is $\{\langle 3, 1\rangle, \ldots \langle 3, 5\rangle\}$.) So again, this query depends on the underlying domain(s) being used (for the different variables) and is not under the user's control.

There is a further difficulty of a more practical nature raised by query (*unsafe-3*). Specifically, if the intuitively appealing semantics of the predicate calculus are used, then the naive approach to evaluating quantifiers leads to the execution of potentially infinite procedures. Although the proper answer to such queries can be computed in a finite manner (see Theorem 5.6.1), this is technically intricate.

The following example indicates how easy it is to form an unsafe query mistakenly in practice.

Example 5.3.2 Recall the calculus query answering query (5.3) in Example 5.3.1. Suppose that the first conjunct of that query is omitted to obtain the following:

$$\{x_t \mid \forall y_a(\exists y_d \mathit{Movies}(x_t, y_d, y_a) \rightarrow \exists z_t \mathit{Movies}(z_t, \text{“Hitchcock”}, y_a))\}.$$

This query returns all titles of movies that have the specified property and also all elements of **dom** not occurring in $\pi_{\mathit{Title}}(\mathit{Movies})$. Even if x_t were restricted to range over the set of actual and potential movie titles, it would not be domain independent.

Relativized Interpretations

We now return to the formal development. As the first step, we present a definition that will permit us to talk about calculus queries in connection with different underlying domains.

Under the conventional semantics associated with predicate calculus, quantified variables range over all elements of the underlying domain, in our case, **dom**. For our purposes, however, we generalize this notion to permit explicit specification of the underlying domain to use (i.e., over which variables may range).

A *relativized instance* over schema $\mathbf{R}$ is a pair $(\mathbf{d}, \mathbf{I})$, where $\mathbf{I}$ is an instance over $\mathbf{R}$ and $\mathit{adom}(\mathbf{I}) \subseteq \mathbf{d} \subseteq \mathbf{dom}$. A calculus formula φ is *interpretable* over $(\mathbf{d},\mathbf{I})$ if $\mathit{adom}(\varphi) \subseteq \mathbf{d}$. In this case, if ν is a valuation over $\mathit{free}(\varphi)$ with range contained in $\mathbf{d}$, then $\mathbf{I}$ *satisfies* φ for ν *relative* to $\mathbf{d}$, denoted $\mathbf{I} \models_{\mathbf{d}} \varphi[\nu]$, if

(a) $\varphi = R(u)$ is an atom and $\nu(u) \in \mathbf{I}(R)$;
(b) $\varphi = (s = s')$ is an equality atom and $\nu(s) = \nu(s')$;
(c) $\varphi = (\psi \wedge \xi)$ and[1] $\mathbf{I} \models_{\mathbf{d}} \psi[\nu|_{\mathit{free}(\psi)}]$ and $\mathbf{I} \models_{\mathbf{d}} \xi[\nu|_{\mathit{free}(\xi)}]$;
(d) $\varphi = (\psi \vee \xi)$ and $\mathbf{I} \models_{\mathbf{d}} \psi[\nu|_{\mathit{free}(\psi)}]$ or $\mathbf{I} \models_{\mathbf{d}} \xi[\nu|_{\mathit{free}(\xi)}]$;
(e) $\varphi = \neg\psi$ and $\mathbf{I} \not\models_{\mathbf{d}} \psi[\nu]$ (i.e., $\mathbf{I} \models_{\mathbf{d}} \psi[\nu]$ does not hold);
(f) $\varphi = \exists x \psi$ and for some $c \in \mathbf{d}$, $\mathbf{I} \models_{\mathbf{d}} \psi[\nu \cup \{x/c\}]$; or
(g) $\varphi = \forall x \psi$ and for each $c \in \mathbf{d}$, $\mathbf{I} \models_{\mathbf{d}} \psi[\nu \cup \{x/c\}]$.

The notion of “satisfies ... relative to” just presented is equivalent to the usual notion of satisfaction found in first-order logic, where the set $\mathbf{d}$ plays the role of the universe of discourse in first-order logic. In practical database settings it is most natural to assume that the underlying universe is **dom**; for this reason we use specialized terminology here.

Recall that for a query q and input instance $\mathbf{I}$, we denote $\mathit{adom}(q) \cup \mathit{adom}(\mathbf{I})$ by $\mathit{adom}(q, \mathbf{I})$, and the notation $\mathit{adom}(\varphi, \mathbf{I})$ for formula φ is defined analogously.

We can now define the relativized semantics for the calculus. Let $\mathbf{R}$ be a schema, $q = \{e_1, \ldots, e_n \mid \varphi\}$ a calculus query over $\mathbf{R}$, and $(\mathbf{d}, \mathbf{I})$ a relativized instance over $\mathbf{R}$. Then

[1] $\nu|_V$ for variable set V denotes the restriction of ν to V.

the *image* of $\mathbf{I}$ under q *relative to* $\mathbf{d}$ is

$$q_{\mathbf{d}}(\mathbf{I}) = \{\nu(\langle e_1, \ldots, e_n\rangle) \mid \mathbf{I} \models_{\mathbf{d}} \varphi[\nu], \\ \nu \text{ is a valuation over } \mathit{free}(\varphi) \text{ with range} \subseteq \mathbf{d}\}.$$

Note that if $\mathbf{d}$ is infinite, then this image may be an infinite set of tuples.

As a minor generalization, for arbitrary $\mathbf{d} \subseteq \mathbf{dom}$, the *image* of q on $\mathbf{I}$ relative to $\mathbf{d}$ is defined by[2]

$$q_{\mathbf{d}}(\mathbf{I}) = q_{\mathbf{d} \cup adom(q,\mathbf{I})}(\mathbf{I}).$$

Example 5.3.3 Consider the query

$$q = \{x \mid R(x) \wedge \exists y(\neg R(y) \wedge \forall z(R(z) \vee z = y))\}$$

Then

$$q_{\mathbf{dom}}(I) = \{\} \text{ for any instance } I \text{ over } R$$
$$q_{\{1,2,3,4\}}(J_1) = \{\} \text{ for } J_1 = \{\langle 1\rangle, \langle 2\rangle\} \text{ over } R$$
$$q_{\{1,2,3,4\}}(J_2) = J_2 \text{ for } J_2 = \{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle\} \text{ over } R$$
$$q_{\{1,2,3,4\}}(J_3) = \{\} \text{ for } J_3 = \{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle\} \text{ over } R$$
$$q_{\{1,2,3,4\}}(J_4) = J_4 \text{ for } J_4 = \{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 5\rangle\} \text{ over } R.$$

This illustrates that under an interpretation relative to a set $\mathbf{d}$, a calculus query q on input $\mathbf{I}$ may be affected by $|\mathbf{d} - adom(q, \mathbf{I})|$.

It is important to note that the semantics of algebra and datalog$^\neg$ queries q evaluated on instance $\mathbf{I}$ are independent of whether $\mathbf{dom}$ or some subset $\mathbf{d}$ satisfying $adom(q, \mathbf{I}) \subseteq \mathbf{d} \subseteq \mathbf{dom}$ is used as the underlying domain.

The Natural and Active Domain Semantics for Calculus Queries

The relativized semantics for calculus formulas immediately yields two important semantics for calculus queries. The first of these corresponds most closely to the conventional interpretation of predicate calculus and is thus perhaps the intuitively most natural semantics for the calculus.

Definition 5.3.4 For calculus query q and input instance $\mathbf{I}$, the *natural* (or *unrestricted*) interpretation of q on $\mathbf{I}$, denoted $q_{nat}(\mathbf{I})$, is $q_{\mathbf{dom}}(\mathbf{I})$ if this is finite and is undefined otherwise.

---

[2] Unlike the convention of first-order logic, interpretations over an empty underlying domain are permitted; this arises only with empty instances.

The second interpretation is based on restricting quantified variables to range over the active domain of the query and the input. Although this interpretation is unnatural from the practical perspective, it has the advantage that the output is always defined (i.e., finite). It is also a convenient semantics for certain theoretical developments.

DEFINITION 5.3.5 For calculus query q and input instance $\mathbf{I}$, the *active domain* interpretation of q on $\mathbf{I}$, denoted $q_{adom}(\mathbf{I})$, is $q_{adom(q,\mathbf{I})}(\mathbf{I})$. The family of mappings obtained from calculus queries under the active domain interpretation is denoted CALC_{adom}.

EXAMPLE 5.3.6 Recall query (*unsafe-2*). Under the natural interpretation on input the instance $\mathbf{I}$ shown in Chapter 3, this query yields the undefined result. On the other hand, under the active domain interpretation this yields as output (written informally) ({actors in "Cries and Whispers"} $\times$ $adom(\mathbf{I})$) $\cup$ ($adom(\mathbf{I})$ $\times$ {movies by Bergman featuring Ullman}), which is finite and defined.

Domain Independence

As noted earlier, there are two difficulties with the natural interpretation of the calculus from a practical point of view: (1) it is easy to write queries with undefined output, and (2) even if the output is defined, the naive approach to computing it may involve consideration of quantifiers ranging over an infinite set. The active domain interpretation solves these problems but generally makes the answer dependent on information (the active domain) not readily available to users. One approach to resolving this situation is to restrict attention to the class of queries that yield the same output on all possible underlying domains.

DEFINITION 5.3.7 A calculus query q is *domain independent* if for each input instance $\mathbf{I}$, and each pair $\mathbf{d}$, $\mathbf{d}' \subseteq \mathbf{dom}$, $q_{\mathbf{d}}(\mathbf{I}) = q_{\mathbf{d}'}(\mathbf{I})$. If q is domain independent, then the *image* of q on input instance $\mathbf{I}$, denoted simply $q(\mathbf{I})$, is $q_{\mathbf{dom}}(\mathbf{I})$ [or equivalently, $q_{adom}(\mathbf{I})$]. The family of mappings obtained from domain-independent calculus queries is denoted CALC_{di}.

In particular, if q is domain independent, then the output according to the natural interpretation can be obtained by computing the active domain interpretation. Thus,

LEMMA 5.3.8 $\text{CALC}_{di} \sqsubseteq \text{CALC}_{adom}$.

EXAMPLE 5.3.9 The two calculus queries of Example 5.3.1 are domain independent, and the query of Example 5.3.2 is not (see Exercise 5.15).

Equivalence of Algebra and Calculus

We now demonstrate the equivalence of the various languages introduced so far in this chapter.

THEOREM 5.3.10 (Equivalence Theorem) The domain-independent calculus, the calculus under active domain semantics, the relational algebras, and the family of nr-datalog$^\neg$ programs that have single-relation output have equivalent expressive power.

Proposition 5.2.2 shows that nr-datalog$^\neg$ and the algebras have equivalent expressive power. In addition, Lemma 5.3.8 shows that $\text{CALC}_{di} \sqsubseteq \text{CALC}_{adom}$. To complete the proof, we demonstrate that

(i) algebra $\sqsubseteq \text{CALC}_{di}$ (Lemma 5.3.11)
(ii) $\text{CALC}_{adom} \sqsubseteq$ algebra (Lemma 5.3.12).

LEMMA 5.3.11 For each unnamed algebra query, there is an equivalent domain-independent calculus query.

Proof Let q be an unnamed algebra query with arity n. We construct a domain-independent query $q' = \{x_1, \ldots, x_n \mid \varphi_q\}$ that is equivalent to q. The formula φ_q is constructed using an induction on subexpressions of q. In particular, for subexpression E of q, we define φ_E according to the following cases:

(a) E is R for some $R \in \mathbf{R}$: φ_E is $R(x_1, \ldots, x_{arity(R)})$.
(b) E is $\{u_1, \ldots, u_m\}$, where each u_j is a tuple of arity α: φ_E is

$$(x_1 = u_1(1) \wedge \cdots \wedge x_\alpha = u_1(\alpha)) \vee \cdots \vee (x_1 = u_m(1) \wedge \cdots \wedge x_\alpha = u_m(\alpha)).$$

(c) E is $\sigma_F(E_1)$: φ_E is $\varphi_{E_1} \wedge \psi_F$, where ψ_F is the formula obtained from F by replacing each coordinate identifier i by variable x_i.
(d) E is $\pi_{i_1,\ldots,i_n}(E_1)$: φ_E is

$$\exists y_{i_1}, \ldots, y_{i_n}((x_1 = y_{i_1} \wedge \cdots \wedge x_n = y_{i_n}) \wedge \exists y_{j_1} \ldots \exists y_{j_l} \varphi_{E_1}(y_1, \ldots, y_{arity(E_1)})),$$

where $j_1, \ldots, j_l$ is a listing of $[1, arity(E_1)] - \{i_1, \ldots, i_n\}$.
(e) E is $E_1 \times E_2$: φ_E is $\varphi_{E_1} \wedge \varphi_{E_2}(x_{arity(E_1)+1}, \ldots, x_{arity(E_1)+arity(E_2)})$.
(f) E is $E_1 \cup E_2$: φ_E is $\varphi_{E_1} \vee \varphi_{E_2}$.
(g) E is $E_1 - E_2$: φ_E is $\varphi_{E_1} \wedge \neg\varphi_{E_2}$.

We leave verification of this construction and the properties of q' to the reader (see Exercise 5.13a). ■

LEMMA 5.3.12 For each calculus query q, there is a query in the unnamed algebra that is equivalent to q under the active domain interpretation.

Crux Let $q = \{x_1, \ldots, x_n \mid \varphi\}$ be a calculus query over $\mathbf{R}$. It is straightforward to develop a unary algebra query E_{adom} such that for each input instance $\mathbf{I}$,

$$E_{adom}(\mathbf{I}) = \{\langle a \rangle \mid a \in adom(q, \mathbf{I})\}.$$

Next an inductive construction is performed. To each subformula $\psi(y_1, \ldots, y_m)$ of φ this associates an algebra expression E_ψ with the property that (abusing notation slightly)

$$\{y_1, \ldots, y_m \mid \psi\}_{adom(q,\mathbf{I})}(\mathbf{I}) = E_\psi(\mathbf{I}) \cap (adom(q, \mathbf{I}))^m.$$

[This may be different from using the active domain semantics on ψ, because we may have $adom(\psi, \mathbf{I}) \subset adom(q, \mathbf{I})$.] It is clear that E_φ is equivalent to q under the active domain semantics.

We now illustrate a few cases of the construction of expressions E_ψ and leave the rest for the reader (see Exercise 5.13b). Suppose that ψ is a subformula of φ. Then E_ψ is constructed in the following manner:

(a) $\psi(y_1, \ldots, y_m)$ is $R(t_1, \ldots, t_l)$, where each t_i is a constant or in $\vec{y}$: Then $E_\psi \equiv \pi_{\vec{k}}(\sigma_F(R))$, where $\vec{k}$ and F are chosen in accordance with $\vec{y}$ and $\vec{t}$.

(b) $\psi(y_1, y_2)$ is $y_1 \neq y_2$: E_ψ is $\sigma_{1\neq 2}(E_{adom} \times E_{adom})$.

(c) $\psi(y_1, y_2, y_3)$ is $\psi'(y_1, y_2) \vee \psi''(y_2, y_3)$: E_ψ is $(E_{\psi'} \times E_{adom}) \cup (E_{adom} \times E_{\psi''})$.

(d) $\psi(y_1, \ldots, y_m)$ is $\neg\psi'(y_1, \ldots, y_m)$: E_ψ is $(E_{adom} \times \cdots \times E_{adom}) - E_{\psi'}$. ■

5.4 Syntactic Restrictions for Domain Independence

As seen in the preceding section, to obtain the natural semantics for calculus queries, it is desirable to focus on domain independent queries. However, as will be seen in the following chapter (Section 6.3), it is undecidable whether a given calculus query is domain independent. This has led researchers to develop syntactic conditions that ensure domain independence, and many such conditions have been proposed.

Several criteria affect the development of these conditions, including their generality, their simplicity, and the ease with which queries satisfying the conditions can be translated into the relational algebra or other lower-level representations. We present one such condition here, called "safe range," that is relatively simple but that illustrates the flavor and theoretical properties of many of these conditions. It will serve as a vehicle to illustrate one approach to translating these restricted queries into the algebra. Other examples are explored in Exercises 5.25 and 5.26; translations of these into the algebra are considerably more involved.

This section begins with a brief digression concerning equivalence preserving rewrite rules for the calculus. Next the family CALC_{sr} of safe-range queries is introduced. It is shown easily that the algebra $\sqsubseteq \mathrm{CALC}_{sr}$. A rather involved construction is then presented for transforming safe-range queries into the algebra. The section concludes by defining a variant of the calculus that is equivalent to the conjunctive queries with union.

1	$\varphi \wedge \psi$	$\leftrightarrow$	$\psi \wedge \varphi$
2	$\psi_1 \wedge \cdots \wedge \psi_n \wedge (\psi_{n+1} \wedge \psi_{n+2})$	$\leftrightarrow$	$\psi_1 \wedge \cdots \wedge \psi_n \wedge \psi_{n+1} \wedge \psi_{n+2}$
3	$\varphi \vee \psi$	$\leftrightarrow$	$\psi \vee \varphi$
4	$\psi_1 \vee \cdots \vee \psi_n \vee (\psi_{n+1} \vee \psi_{n+2})$	$\leftrightarrow$	$\psi_1 \vee \cdots \vee \psi_n \vee \psi_{n+1} \vee \psi_{n+2}$
5	$\neg(\varphi \wedge \psi)$	$\leftrightarrow$	$(\neg\varphi) \vee (\neg\psi)$
6	$\neg(\varphi \vee \psi)$	$\leftrightarrow$	$(\neg\varphi) \wedge (\neg\psi)$
7	$\neg(\neg\varphi)$	$\leftrightarrow$	φ
8	$\exists x\varphi$	$\leftrightarrow$	$\neg\forall x\neg\varphi$
9	$\forall x\varphi$	$\leftrightarrow$	$\neg\exists x\neg\varphi$
10	$\neg\exists x\varphi$	$\leftrightarrow$	$\forall x\neg\varphi$
11	$\neg\forall x\varphi$	$\leftrightarrow$	$\exists x\neg\varphi$
12	$\exists x\varphi \wedge \psi$	$\leftrightarrow$	$\exists x(\varphi \wedge \psi)$ (x not free in ψ)
13	$\forall x\varphi \wedge \psi$	$\leftrightarrow$	$\forall x(\varphi \wedge \psi)$ (x not free in ψ)
14	$\exists x\varphi \vee \psi$	$\leftrightarrow$	$\exists x(\varphi \vee \psi)$ (x not free in ψ)
15	$\forall x\varphi \vee \psi$	$\leftrightarrow$	$\forall x(\varphi \vee \psi)$ (x not free in ψ)
16	$\exists x\varphi$	$\leftrightarrow$	$\exists y\varphi_y^x$ (y not free in φ)
17	$\forall x\varphi$	$\leftrightarrow$	$\forall y\varphi_y^x$ (y not free in φ)

Figure 5.1: Equivalence-preserving rewrite rules for calculus formulas

Equivalence-Preserving Rewrite Rules

We now digress for a moment to present a family of rewrite rules for the calculus. These preserve equivalence regardless of the underlying domain used to evaluate calculus queries. Several of these rules will be used in the transformation of safe-range queries into the algebra.

Calculus formulas φ, ψ over schema $\mathbf{R}$ are *equivalent*, denoted $\varphi \equiv \psi$, if for each $\mathbf{I}$ over $\mathbf{R}$, $\mathbf{d} \subseteq \mathbf{dom}$, and valuation ν with range $\subseteq \mathbf{d}$

$$\mathbf{I} \models_{\mathbf{d} \cup adom(\varphi, \mathbf{I})} \varphi[\nu] \text{ if and only if } \mathbf{I} \models_{\mathbf{d} \cup adom(\psi, \mathbf{I})} \psi[\nu].$$

(It is verified easily that this generalizes the notion of equivalence for conjunctive calculus formulas.)

Figure 5.1 shows a number of equivalence-preserving rewrite rules for calculus formulas. It is straightforward to verify that if ψ transforms to ψ' by a rewrite rule and if φ' is the result of replacing an occurrence of subformula ψ of φ by formula ψ', then $\varphi' \equiv \varphi$ (see Exercise 5.14).

Note that, assuming $x \notin free(\psi)$ and $y \notin free(\varphi)$,

$$\exists x\varphi \wedge \forall y\psi \equiv \exists x\forall y(\varphi \wedge \psi) \equiv \forall y\exists x(\varphi \wedge \psi).$$

Example 5.4.1 Recall from Chapter 2 that a formula φ is in *prenex normal form* (PNF) if it has the form $\%_1 x_1 \ldots \%_n x_n \psi$, where each $\%_i$ is either $\forall$ or $\exists$, and no quantifiers occur in ψ. In this case, ψ is called the *matrix* of formula φ.

A formula ψ without quantifiers or connectives $\rightarrow$ or $\leftrightarrow$ is in *conjunctive normal form* (CNF) if it has the form $\xi_1 \wedge \cdots \wedge \xi_m$ ($m \geq 1$), where each conjunct ξ_j has the form $L_1 \vee \cdots \vee L_k$ ($k \geq 1$) and where each L_l is a literal (i.e., atom or negated atom). Similarly, a formula ψ without quantifiers or connectives $\rightarrow$ or $\leftrightarrow$ is in *disjunctive normal form* (DNF) if it has the form $\xi_1 \vee \cdots \vee \xi_m$, where each disjunct ξ_j has the form $L_1 \wedge \cdots \wedge L_k$ where each L_l is a literal (i.e., atom or negated atom).

It is easily verified (see Exercise 5.14) that the rewrite rules can be used to transform an arbitrary calculus formula into an equivalent formula that is in PNF with a CNF matrix, and into an equivalent formula that is in PNF with a DNF matrix.

Safe-Range Queries

The notion of safe range is presented now in three stages, involving (1) a normal form called SRNF, (2) a mechanism for determining how variables are "range restricted" by subformulas, and (3) specification of a required global property of the formula.

During this development, it is sometimes useful to speak of calculus formulas in terms of their parse trees. For example, we will say that the formula $(R(x) \wedge \exists y(S(y, z)) \wedge \neg T(x, z))$ has 'and' or $\wedge$ as a root (which has an atom, an $\exists$, and a $\neg$ as children).

The normalization of formulas puts them into a form more easily analyzed for safety without substantially changing their syntactic structure. The following equivalence-preserving rewrite rules are used to place a formula into *safe-range normal form* (SRNF):

Variable substitution: This is from Section 4.2. It is applied until no distinct pair of quantifiers binds the same variable and no variable occurs both free and bound.

Remove universal quantifiers: Replace subformula $\forall\vec{x}\psi$ by $\neg\exists\vec{x}\neg\psi$. (This and the next condition can be relaxed; see Example 5.4.5.)

Remove implications: Replace $\psi \rightarrow \xi$ by $\neg\psi \vee \xi$, and similarly for $\leftrightarrow$.

Push negations: Replace

(i) $\neg\neg\psi$ by ψ

(ii) $\neg(\psi_1 \vee \cdots \vee \psi_n)$ by $(\neg\psi_1 \wedge \cdots \wedge \neg\psi_n)$

(iii) $\neg(\psi_1 \wedge \cdots \wedge \psi_n)$ by $(\neg\psi_1 \vee \cdots \vee \neg\psi_n)$

so that the child of each negation is either an atom or an existentially quantified formula.

Flatten 'and's, 'or's, and existential quantifiers: This is done so that no child of an 'and' is an 'and,' and similarly for 'or' and existential quantifiers.

The SRNF formula resulting from applying these rules to φ is denoted $\texttt{SRNF}(\varphi)$. A formula φ (query $\{\vec{e} \mid \varphi\}$) is in SRNF if $\texttt{SRNF}(\varphi) = \varphi$.

Example 5.4.2 The first calculus query of Example 5.3.1 is in SRNF. The second calculus query is not in SRNF; the corresponding SRNF query is

$$\{x_t \mid \exists x_d, x_a \mathit{Movies}(x_t, x_d, x_a) \wedge \neg\exists y_a(\exists y_d \mathit{Movies}(x_t, y_d, y_a) \wedge \neg\exists z_t \mathit{Movies}(z_t, \text{“Hitchcock”}, y_a))\}.$$

Transforming the query of Example 5.3.2 into SRNF yields

$$\{x_t \mid \neg\exists y_a(\exists y_d \mathit{Movies}(x_t, y_d, y_a) \wedge \neg\exists z_t \mathit{Movies}(z_t, \text{“Hitchcock”}, y_a))\}.$$

We now present a syntactic condition on SRNF formulas that ensures that each variable is "range restricted," in the sense that its possible values all lie within the active domain of the formula or the input. If a quantified variable is not range restricted, or if one of the free variables is not range restricted, then the associated query is rejected. To make the definition, we first define the set of *range-restricted variables* of an SRNF formula using the following procedure, which returns either the symbol $\perp$, indicating that some quantified variable is not range restricted, or the set of free variables that is range restricted.

ALGORITHM 5.4.3 (Range restriction (rr))

Input: a calculus formula φ in SRNF

Output: a subset of the free variables of φ or[3] $\perp$

begin
case φ **of**

$$\begin{array}{rcl}
R(e_1, \ldots, e_n) & : & rr(\varphi) = \text{the set of variables in } \{e_1, \ldots, e_n\}; \\
x = a \text{ or } a = x & : & rr(\varphi) = \{x\}; \\
\varphi_1 \wedge \varphi_2 & : & rr(\varphi) = rr(\varphi_1) \cup rr(\varphi_2); \\
\varphi_1 \wedge x = y & : & rr(\varphi) = \begin{cases} rr(\psi) & \text{if } \{x, y\} \cap rr(\psi) = \emptyset, \\ rr(\psi) \cup \{x, y\} & \text{otherwise;} \end{cases} \\
\varphi_1 \vee \varphi_2 & : & rr(\varphi) = rr(\varphi_1) \cap rr(\varphi_2); \\
\neg\varphi_1 & : & rr(\varphi) = \emptyset; \\
\exists\vec{x}\varphi_1 & : & \textbf{if } \vec{x} \subseteq rr(\varphi_1) \\
 & & \quad \textbf{then } rr(\varphi) = rr(\varphi_1) - \vec{x} \\
 & & \quad \textbf{else return } \perp
\end{array}$$

end case
end ■

---

[3] In the following, for each Z, $\perp \cup Z = \perp \cap Z = \perp - Z = Z - \perp = \perp$. In addition, we show the case of binary 'and's, etc., but we mean this to include polyadic 'and's, etc. Furthermore, we sometimes use '$\vec{x}$' to denote the set of variables occurring in $\vec{x}$.

Intuitively, the occurrence of a variable x in a base relation or in an atom of the form $x = a$ restricts that variable. This restriction is propagated through $\wedge$, possibly lost in $\vee$, and always lost in $\neg$. In addition, each quantified variable must be restricted by the subformula it occurs in.

A calculus query $\{u \mid \varphi\}$ is *safe range* if $rr(\mathtt{SRNF}(\varphi)) = free(\varphi)$. The family of safe-range queries is denoted by CALC_{sr}.

Example 5.4.4 Recall Examples 5.3.1 and 5.4.2. The first query of Example 5.3.1 is safe range. The first query of Example 5.4.2 is also safe range. However, the second query of Example 5.4.2 is not because the free variable x_t is not range restricted by the formula.

Before continuing, we explore a generalization of the notion of safe range to permit universal quantification.

Example 5.4.5 Suppose that formula φ has a subformula of the form

$$\psi \equiv \forall \vec{x}(\psi_1(\vec{x}) \rightarrow \psi_2(\vec{y})),$$

where $\vec{x}$ and $\vec{y}$ might overlap. Transforming into SRNF (and assuming that the parent of ψ is not $\neg$), we obtain

$$\psi' \equiv \neg\exists \vec{x}(\psi_1(\vec{x}) \wedge \neg\psi_2(\vec{y})).$$

Now $rr(\psi')$ is defined iff

(a) $rr(\psi_1) = \vec{x}$, and

(b) $rr(\psi_2)$ is defined.

In this case, $rr(\psi') = \emptyset$. This is illustrated by the second query of Example 5.3.1, that was transformed into SRNF in Example 5.4.2.

Thus SRNF can be extended to permit subformulas that have the form of ψ without materially affecting the development.

The calculus query constructed in the proof of Lemma 5.3.11 is in fact safe range. It thus follows that the algebra $\sqsubseteq \mathrm{CALC}_{sr}$.

As shown in the following each safe range query is domain independent (Theorem 5.4.6). For this reason, if q is safe range we generally use the natural interpretation to evaluate it; we may also use the active domain interpretation.

The development here implies that all of CALC_{sr}, CALC_{di}, and CALC_{adom} are equivalent. When the particular choice is irrelevant to the discussion, we use the term *relational calculus* to refer to any of these three equivalent query languages.

From Safe Range to the Algebra

We now present the main result of this section (namely, the translation of safe-range queries into the named algebra). Speaking loosely, this translation is relatively direct in the sense that the algebra query E constructed for calculus query q largely follows the structure of q. As a result, evaluation of E will in most cases be more efficient than using the algebra query that is constructed for q by the proof of Lemma 5.3.12.

Examples of the construction used are presented after the formal argument.

THEOREM 5.4.6 $\text{CALC}_{sr} \equiv$ the relational algebra. Furthermore, each safe-range query is domain independent.

The proof of this theorem involves several steps. As seen earlier, the algebra $\sqsubseteq \text{CALC}_{sr}$. To prove the other direction, we develop a translation from safe-range queries into the named algebra. Because the algebra is domain independent, this will also imply the second sentence of the theorem.

To begin, let φ be a safe-range formula in SRNF. An occurrence of a subformula ψ in φ is *self-contained* if its root is $\wedge$ or if

(i) $\psi = \psi_1 \vee \cdots \vee \psi_n$ and $rr(\psi) = rr(\psi_1) = \cdots = rr(\psi_n) = free(\psi)$;
(ii) $\psi = \exists \vec{x} \psi_1$ and $rr(\psi) = free(\psi_1)$; or
(iii) $\psi = \neg \psi_1$ and $rr(\psi) = free(\psi_1)$.

A safe-range, SRNF formula φ is in[4] *relational algebra normal form* (RANF) if each subformula of φ is self-contained.

Intuitively, if ψ is a self-contained subformula of φ that does not have $\wedge$ as a root, then all free variables in ψ are range restricted within ψ. As we shall see, if φ is in RANF, this permits construction of an equivalent relational algebra query E_φ using an induction from leaf to root.

We now develop an algorithm RANF-ALG that transforms safe-range SRNF formulas into RANF. It is based on the following rewrite rules:

(R1) Push-into-or: Consider the subformula

$$\psi = \psi_1 \wedge \cdots \wedge \psi_n \wedge \xi,$$

where

$$\xi = \xi_1 \vee \cdots \vee \xi_m.$$

Suppose that $rr(\psi) = free(\psi)$, but $rr(\xi_1 \vee \cdots \vee \xi_m) \neq free(\xi_1 \vee \cdots \vee \xi_m)$. Nondeterministically choose a subset $i_1, \ldots, i_k$ of $1, \ldots, n$ such that

$$\xi' = (\xi_1 \wedge \psi_{i_1} \wedge \cdots \wedge \psi_{i_k}) \vee \cdots \vee (\xi_m \wedge \psi_{i_1} \wedge \cdots \wedge \psi_{i_k})$$

[4] This is a variation of the notion of RANF used elsewhere in the literature; see Bibliographic Notes.

satisfies $rr(\xi') = free(\xi')$. (One choice of $i_1, \ldots, i_k$ is to use all of $1, \ldots, n$; this necessarily yields a formula ξ' with this property.) Letting $\{j_1, \ldots, j_l\} = \{1, \ldots, n\} - \{i_1, \ldots, i_k\}$, set

$$\psi' = \mathtt{SRNF}(\psi_{j_1} \wedge \cdots \wedge \psi_{j_l} \wedge \xi').$$

The application of SRNF to ξ' only has the effect of possibly renaming quantified variables[5] and of flattening the roots of subformulas $\xi_p \wedge \psi_{i_1} \wedge \cdots \wedge \psi_{i_k}$, where ξ_p has root $\wedge$; analogous remarks apply. The rewrite rule is to replace subformula ψ by ψ' and possibly apply SRNF to flatten an $\vee$, if both $l = 0$ and the parent of ψ is $\vee$.

(R2) Push-into-quantifier: Suppose that

$$\psi = \psi_1 \wedge \cdots \wedge \psi_n \wedge \exists \vec{x} \xi,$$

where $rr(\psi) = free(\psi)$, but $rr(\xi) \neq free(\xi)$. Then replace ψ by

$$\psi' = \mathtt{SRNF}(\psi_{j_1} \wedge \cdots \wedge \psi_{j_l} \wedge \exists \vec{x} \xi'),$$

where

$$\xi' = \psi_{i_1} \wedge \cdots \wedge \psi_{i_k} \wedge \xi$$

and where $rr(\xi') = free(\xi')$ and $\{j_1, \ldots, j_l\} = \{1, \ldots, n\} - \{i_1, \ldots, i_k\}$. The rewrite rule is to replace ψ by ψ' and possibly apply SRNF to flatten an $\exists$.

(R3) Push-into-negated-quantifier: Suppose that

$$\psi = \psi_1 \wedge \cdots \wedge \psi_n \wedge \neg \exists \vec{x} \xi,$$

where $rr(\psi) = free(\psi)$, but $rr(\xi) \neq free(\xi)$. Then replace ψ by

$$\psi' = \mathtt{SRNF}(\psi_1 \wedge \cdots \wedge \psi_n \wedge \neg \exists \vec{x} \xi'),$$

where

$$\xi' = \psi_{i_1} \wedge \cdots \wedge \psi_{i_k} \wedge \xi$$

and where $rr(\xi') = free(\xi')$ and $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$. That ψ' is equivalent to ψ follows from the observation that the propositional formulas $p \wedge q \wedge \neg r$ and $p \wedge q \wedge \neg(p \wedge r)$ are equivalent. The rewrite rule is to replace ψ by ψ'.

The algorithm RANF-ALG for applying these rewrite rules is essentially top-down and recursive. We sketch the algorithm now (see Exercise 5.19).

[5] It is assumed that under SRNF renamed variables are chosen so that they do not occur in the full formula under consideration.

ALGORITHM 5.4.7 (Relational Algebra Normal Form (RANF-ALG))

Input: a safe-range calculus formula φ in SRNF

Output: a RANF formula $\varphi' = \texttt{RANF}(\varphi)$ equivalent to φ

begin
  while some subformula ψ (with its conjuncts possibly reordered) of φ satisfies the premise of R1, R2, or R3
  do
    case R1: (left as exercise)
         R2: (left as exercise)
         R3: Let $\psi = \psi_1 \wedge \cdots \wedge \psi_n \wedge \neg\exists\vec{x}\xi$
             and $\psi_{i_1}, \ldots, \psi_{i_k}$ satisfy the conditions of R3;
             $\alpha := \texttt{RANF}(\psi_1 \wedge \cdots \wedge \psi_n)$;
             $\beta := \texttt{RANF}(\texttt{SRNF}(\psi_{i_1} \wedge \cdots \wedge \psi_{i_k} \wedge \xi))$;
             $\psi' := \alpha \wedge \neg\exists\vec{x}\beta$;
             $\varphi :=$ result of replacing ψ by ψ' in φ;
    end case
  end while
end

The proof that these rewrite rules can be used to transform a safe-range SRNF formula into a RANF formula has two steps (see Exercise 5.19). First, a case analysis can be used to show that if safe-range φ in SRNF is not in RANF, then one of the rewrite rules (R1, R2, R3) can be applied. Second, it is shown that Algorithm 5.4.7 terminates. This is accomplished by showing that (1) each successfully completed call to RANF-ALG reduces the number of non-self-contained subformulas, and (2) if a call to RANF-ALG on ψ invokes other calls to RANF-ALG, the input to these recursive calls has fewer non-self-contained subformulas than does ψ.

We now turn to the transformation of RANF formulas into equivalent relational algebra queries. We abuse notation somewhat and assume that each variable is also an attribute. (Alternatively, a one-one mapping *var-to-att* : **var** $\rightarrow$ **att** could be used.) In general, given a RANF formula φ with free variables $x_1, \ldots, x_n$, we shall construct a named algebra expression E_φ over attributes $x_1, \ldots, x_n$ such that for each input instance $\mathbf{I}$, $E_\varphi(\mathbf{I}) = \{x_1, \ldots, x_n \mid \varphi\}(\mathbf{I})$. (The special case of queries $\{e_1, \ldots, e_n \mid \varphi\}$, where some of the e_i are constants, is handled by performing a join with the constants at the end of the construction.)

A formula φ is in *modified relational algebra normal form (modified RANF)* if it is RANF, except that each polyadic 'and' is ordered and transformed into binary 'and's, so that atoms $x = y$ ($x \neq y$) are after conjuncts that restrict one (both) of the variables involved and so that each free variable in a conjunct of the form $\neg\xi$ occurs in some preceding conjunct. It is straightforward to verify that each RANF formula can be placed into modified RANF. Note that each subformula of a modified RANF formula is self-contained.

Let RANF formula φ be fixed. The construction of E_φ is inductive, from leaf to root, and is sketched in the following algorithm. The special operator **diff**, on inputs R and S where $att(S) \subset att(R)$, is defined by

$$R \textbf{ diff } S = R - (R \bowtie S).$$

(Many details of this transformation, such as the construction of renaming function f, projection list $\vec{k}$, and selection formula F in the first entry of the case statement, are left to the reader; see Example 5.4.9 and Exercise 5.19.)

ALGORITHM 5.4.8 (Translation into the Algebra)

Input: a formula φ in modified RANF

Output: an algebra query E_φ equivalent to φ

begin
case φ **of**

$R(\vec{e})$ $\quad \delta_f(\pi_{\vec{k}}(\sigma_F(R)))$

$x = a$ $\quad \{\langle x : a \rangle\}$

$\psi \wedge \xi$ $\quad$ if ξ is $x = x$, then E_ψ
if ξ is $x = y$ (with x, y distinct), then
$\sigma_{x=y}(E_\psi)$, if $\{x, y\} \subseteq free(\psi)$
$\sigma_{x=y}(E_\psi \bowtie \delta_{x\to y} E_\psi)$, if $x \in free(\psi)$ and $y \notin free(\psi)$
$\sigma_{x=y}(E_\psi \bowtie \delta_{y\to x} E_\psi)$, if $y \in free(\psi)$ and $x \notin free(\psi)$
if ξ is $x \neq y$, then $\sigma_{x\neq y}(E_\psi)$
if $\xi = \neg\xi'$, then
E_ψ **diff** $E_{\xi'}$, if $free(\xi') \subset free(\psi)$
$E_\psi - E_{\xi'}$, if $free(\xi') = free(\psi)$
otherwise, $E_\psi \bowtie E_\xi$

$\neg\psi$ $\quad \{\langle\rangle\} - E_\psi$
(in the case that $\neg\psi$ does not have 'and' as parent)

$\psi_1 \vee \cdots \vee \psi_n$ $\quad E_{\psi_1} \cup \cdots \cup E_{\psi_n}$

$\exists x_1, \ldots, x_n \psi(x_1, \ldots, x_n, y_1, \ldots, y_m)$
$\pi_{y_1,\ldots,y_m}(E_\psi)$

end case
end

Finally, let $q = \{x_1, \ldots, x_n \mid \varphi\}$ be safe range. Because the transformations used for SRNF and RANF are equivalence preserving, without loss of generality we can assume that φ is in modified RANF. To conclude the proof of Theorem 5.4.6, it must be shown that q and E_φ are equivalent. In fact, it can be shown that for each instance $\mathbf{I}$ and each $\mathbf{d}$ satisfying $adom(q, \mathbf{I}) \subseteq \mathbf{d} \subseteq \mathbf{dom}$,

$$q_{\mathbf{d}}(\mathbf{I}) = E_\varphi(\mathbf{I}).$$

This will also yield that q is domain independent.

Let $\mathbf{I}$ and $\mathbf{d}$ be fixed. A straightforward induction can be used to show that for each subformula $\psi(y_1, \ldots, y_m)$ of φ and each variable assignment ν with range $\mathbf{d}$,

$$\mathbf{I} \models_{\mathbf{d}} \psi[\nu] \Leftrightarrow \langle \nu(y_1), \ldots, \nu(y_m) \rangle \in E_\psi(\mathbf{I})$$

(see Exercise 5.19.) This completes the proof of Theorem 5.4.6.

Example 5.4.9 (a) Consider the query

$$q_1 = \{\langle a, x, y\rangle : A_1A_2A_3 \mid \exists z(P(x, y, z) \vee [R(x, y) \wedge ([S(z) \wedge \neg T(x, z)] \vee [T(y, z)])])\}.$$

The formula of q_1 is in SRNF. Transformation into RANF yields

$$\exists z(P(x, y, z) \vee [R(x, y) \wedge S(z) \wedge \neg T(x, z)] \vee [R(x, y) \wedge T(y, z)]).$$

Assuming the schemas $P[B_1B_2B_3]$, $R[C_1C_2]$, $S[D]$, and $T[F_1F_2]$, transformation of this into the algebra yields

$$\begin{aligned} E = \pi_{x,y}(&\delta_{B_1B_2B_3 \to xyz}(P) \\ &\cup ((\delta_{C_1C_2 \to xy}(R) \bowtie \delta_{D \to z}(S)) \textbf{ diff } \delta_{F_1F_2 \to yz}(T)) \\ &\cup (\delta_{C_1C_2 \to xy}(R) \bowtie \delta_{F_1F_2 \to yz}(T))). \end{aligned}$$

Finally, an algebra query equivalent to q_1 is

$$\{\langle A_1 : a\rangle\} \bowtie \delta_{xy \to A_2A_3}(E).$$

(b) Consider the query

$$q_2 = \{x \mid \exists y[R(x, y) \wedge \forall z(S(z, a) \to T(y, z)) \wedge \exists v, w(\neg T(v, w) \wedge w = b \wedge v = x)]\}.$$

Transforming to SRNF, we have

$$\exists y[R(x, y) \wedge \neg\exists z(S(z, a) \wedge \neg T(y, z)) \wedge \exists v, w(\neg T(v, w) \wedge w = b \wedge v = x)].$$

Transforming to RANF and reordering the conjunctions, we obtain

$$\exists y[\exists v, w(R(x, y) \wedge w = b \wedge v = x \wedge \neg T(v, w)) \wedge \neg\exists z(R(x, y) \wedge S(z, a) \wedge \neg T(y, z))].$$

Assuming schemas $R[A_1, A_2]$, $S[B_1, B_2]$, and $T[C_1, C_2]$, the equivalent algebra query is obtained using the program

$$E_1 := (\delta_{A_1A_2 \to xy}(R) \bowtie \{\langle w : b \rangle\});$$
$$E_2 := (\sigma_{v=x}(E_1 \bowtie \delta_{x \to v}(E_1))) \textbf{ diff } \delta_{C_1C_2 \to vw}(T);$$
$$E_3 := \pi_{x,y}(E_2);$$
$$E_4 := \pi_{x,y}(\delta_{A_1A_2 \to xy}(R) \bowtie \delta_{B_1 \to z}(\pi_{B_1}(\sigma_{B_2=a}(S))) \textbf{ diff } \delta_{C_1C_2 \to yz}(T));$$
$$E_5 := \pi_x(E_3 - E_4).$$

The Positive Existential Calculus

In Chapter 4, disjunction was incorporated into the rule-based conjunctive queries, and union was incorporated into the tableau, SPC, and SPJR queries. Incorporating disjunction into the conjunctive calculus was more troublesome because of the possibility of infinite "answers." We now apply the tools developed earlier in this chapter to remedy this situation.

A *positive existential* (*calculus*) *query* is a domain-independent calculus query $q = \{e_1, \ldots, e_n \mid \varphi\}$, possibly with equality, in which the only logical connectives are $\wedge$, $\vee$, and $\exists$. It is decidable whether a query q with these logical connectives is domain independent; and if so, q is equivalent to a safe-range query using only these connectives (see Exercise 5.16). The following is easily verified.

THEOREM 5.4.10 The positive existential calculus is equivalent to the family of conjunctive queries with union.

5.5 Aggregate Functions

In practical query languages, the underlying domain is many-sorted, with sorts such as boolean, string, integer, or real. These languages allow the use of comparators such as $\leq$ between database entries in an ordered sort and "aggregate" functions such as **sum**, **count**, or **average** on numeric sorts. In this section, aggregate operators are briefly considered. In the next section, a novel approach for incorporating arithmetic constraints into the relational model will be addressed.

Aggregate operators operate on collections of domain elements. The next example illustrates how these are used.

EXAMPLE 5.5.1 Consider a relation *Sales*[*Theater*, *Title*, *Date*, *Attendance*], where a tuple $\langle th, ti, d, a \rangle$ indicates that on date d a total of a people attended showings of movie *ti* at theater *th*. We assume that {*Theater*, *Title*, *Date*} is a key, i.e., that two distinct tuples cannot share the same values on these three attributes. Two queries involving aggregate functions are

(5.4) For each theater, list the total number of movies that have been shown there.

(5.5) For each theater and movie, list the total attendance.

Informally, the first query might be expressed in a pidgin language as

$$\{\langle th,c\rangle \mid th \text{ is a theater occurring in } Sales \text{ and } c = |\pi_{Title}(\sigma_{Theater=th}(Sales))|\}$$

and the second as

$$\{\langle th, ti, s\rangle \mid \langle th, ti\rangle \text{ is a theater-title pair appearing in } Sales \text{ and } s \text{ is the sum that includes each occurrence of each } a\text{-value in } \sigma_{Theater=th \wedge Title=ti}(Sales)\}$$

A subtlety here is that this second query cannot be expressed simply as

$$\{\langle th, ti, s\rangle \mid \langle th, ti\rangle \text{ is a theater-title pair appearing in } Sales \text{ and } s = \Sigma\{a \in \pi_{Attendance}(\sigma_{Theater=th \wedge Title=ti}(Sales))\}\}$$

since a value a has to be counted as many times as it occurs in the selection. This suggests that a more natural setting for studying aggregate functions would explicitly include *bags* (or multisets, i.e., collections in which duplicates are permitted) and not just sets, a somewhat radical departure from the model we have used so far.

The two queries can be expressed as follows using aggregate functions in an algebraic language:

$$\pi_{Theater;\ \mathbf{count}(Title)}(Sales)$$
$$\pi_{Theater,Title;\ \mathbf{sum}(Attendance)}(Sales).$$

We now briefly present a more formal development. To simplify, the formalism is based on the unnamed perspective, and we assume that $\mathbf{dom} = \mathbf{N}$, i.e., the set of non-negative integers. We stay within the relational model although as noted in the preceding example, a richer data model with bags would be more natural. Indeed, the complex value model that will be studied in Chapter 20 provides a more appropriate context for considering aggregate functions.

We shall adopt a somewhat abstract view of aggregate operators. An *aggregate function* f is defined to be a family of functions $f_1, f_2, \ldots$ such that for each $j \geq 1$ and each relation schema S with $arity(S) \geq j$, $f_j : Inst(S) \to \mathbf{N}$. For instance, for the **sum** aggregate function, we will have $\mathbf{sum}_1$ to sum the first column and, in general, $\mathbf{sum}_i$ to sum the i^{th} one. As in the case of **sum**, we want the f_i to depend only on the content of the column to which they are applied, where the "content" includes not only the set of elements in the column, but also the number of their occurrences (so, columns are viewed as bags). This requirement is captured by the following *uniformity property* imposed on each aggregate function f:

> Suppose that the i^{th} column of I and the j^{th} of J are identical, i.e., for each a, there are as many occurrences of a in the i^{th} column of I and in the j^{th} column of J. Then $f_i(I) = f_j(J)$.

All of the commonly arising aggregate functions satisfy this uniformity property. The uniformity condition is also used when translating calculus queries with aggregates into the algebra with aggregates.

We next illustrate how aggregate functions can be incorporated into the algebra and calculus (we do not discuss how this is done for nr-datalog$^\neg$, since it is similar to the algebra.) Aggregate functions are added to the algebra using an extended projection operation. Specifically, the projection function for aggregate function f on relation instance I is defined as follows:

$$\pi_{j_1,\ldots,j_m;f(k)}(I) = \{\langle a_{j_1},\ldots,a_{j_m}, f_k(\sigma_{j_1=a_{j_1}\wedge\cdots\wedge j_m=a_{j_m}}(I))\rangle \mid \langle a_1,\ldots,a_n\rangle \in I\}.$$

Note that the aggregate function f_k is applied separately to each group of tuples in I corresponding to a different possible value for the columns $j_1,\ldots,j_m$.

Turning to the calculus, we begin with an example. Query (5.5) can be expressed in the extended calculus as

$$\{th, ti, s \mid \exists d_1, a_1(Sales(th, ti, d_1, a_1) \wedge s = \mathbf{sum}_2\{d_2, a_2 \mid Sales(th, ti, d_2, a_2)\})\}$$

where $\mathbf{sum}_2$ is the aggregate function summing the second column of a relation. Note that the subexpression $\{d_2, a_2 \mid Sales(th, ti, d_2, a_2)\}$ has free variables *th* and *ti* that do not occur in the target of the subexpression. Intuitively, different assignments for these variables will yield different values for the subexpression.

More formally, aggregate functions are incorporated into the calculus by permitting *aggregate terms* that have the form $f_j\{\vec{x} \mid \psi\}$, where f is an aggregate function, $j \leq arity(\vec{x})$ and ψ is a calculus formula (possibly with aggregate terms). When defining the extended calculus, care must be taken to guarantee that aggregate terms do not recursively depend on each other. This can be accomplished with a suitable generalization of safe range. This generalization will also ensure that free variables occurring in an aggregate term are range restricted by a subformula containing it. It is straightforward to define the semantics of the generalized safe-range calculus with aggregate functions. One can then show that the extensions of the algebra and safe-range calculus with the same set of aggregate functions have the same expressive power.

5.6 Digression: Finite Representations of Infinite Databases

Until now we have considered only finite instances of relational databases. As we have seen, this introduced significant difficulty in connection with domain independence of calculus queries. It is also restrictive in connection with some application areas that involve temporal or geometric data. For example, it would be convenient to think of a rectangle in the real plane as an infinite set of points, even if it can be represented easily in some finite manner.

In this short section we briefly describe some recent and advanced material that uses logic to permit the finite representation of infinite databases. We begin by presenting an alternative approach to resolving the problem of safety, that permits queries to have answers

that are infinite but finitely representable. We then introduce a promising generalization of the relational model that uses constraints to represent infinite databases, and we describe how query processing can be performed against these in an efficient manner.

An Alternative Resolution to the Problem of Safety

As indicated earlier, much of the research on safety has been directed at syntactic restrictions to ensure domain independence. An alternative approach is to use the natural interpretation, even if the resulting answer is infinite. As it turns out, the answers to such queries are recursive and have a finite representation.

For this result, we shall use a finite set $\mathbf{d} \subset \mathbf{dom}$, which corresponds intuitively to the active domain of a query and input database; and a set $C = \{c_1, \ldots, c_m\}$ of m distinct "new" symbols, which will serve as placeholders for elements of $\mathbf{dom} - \mathbf{d}$. Speaking intuitively, the elements of C sometimes act as elements of **dom**, and so it is not appropriate to view them as simple variables.

A *tuple with placeholders* is a tuple $t = \langle t_1, \ldots, t_n \rangle$, where each t_i is in $\mathbf{d} \cup C$. The *semantics* of such t relative to $\mathbf{d}$ are

$$sem_{\mathbf{d}}(t) = \{\rho(t) \mid \rho \text{ is a one-one mapping from } \mathbf{d} \cup C \text{ that leaves } \mathbf{d} \text{ fixed and maps } C \text{ into } \mathbf{dom} - \mathbf{d}\}.$$

The following theorem, stated without proof, characterizes the result of applying an arbitrary calculus query using the natural semantics.

THEOREM 5.6.1 Let $q = \{e_1, \ldots, e_n \mid \varphi\}$ be an arbitrary calculus query, such that each quantifier in φ quantifies a distinct variable that is not free in φ. Let $C = \{c_1, \ldots, c_m\}$ be a set of m distinct "new" symbols not occurring in **dom**, but viewed as domain elements, where m is the number of distinct variables in φ. Then for each input instance $\mathbf{I}$,

$$q_{\mathbf{dom}}(\mathbf{I}) = \cup\{sem_{adom(q,\mathbf{I})}(t) \mid t \in q_{adom(q,\mathbf{I})\cup C}(\mathbf{I})\}.$$

This shows that if we apply a calculus query (under the natural semantics) to a finite database, then the result is recursive, even if infinite. But is the set of infinite databases described in this manner closed under the application of calculus queries? The affirmative answer is provided by an elegant generalization of the relational model presented next (see Exercise 5.31).

Constraint Query Languages

The following generalization of the relational model seems useful to a variety of new applications. The starting point is to consider infinite databases with finite representations based on the use of constraints. To begin we define a generalized n-tuple as a conjunction of constraints over n variables. The constraints typically include $=$, $\neq$, $\leq$, etc. In some sense, such a constraint can be viewed as a finite representation of a (possibly infinite) set of (normal) n-tuples (i.e., the valuations of the variables that satisfy the constraint).

Example 5.6.2 Consider the representation of rectangles in the plane. Suppose first that rectangles are given using 5-tuples (n, x_1, y_1, x_2, y_2), where n is the name of the rectangle, (x_1, y_1) are the coordinates of the lower left corner, and (x_2, y_2) are the coordinates of the upper right. The set of points $\langle u, v\rangle$ in such a rectangle delimited by x_1, y_1, x_2, y_2 is given by the constraint

$$x_1 \leq u \leq x_2 \wedge y_1 \leq v \leq y_2.$$

Now the names of intersecting rectangles from a relation R are given by

$$\begin{aligned}\{\langle n_1, n_2\rangle \mid \exists\, & x_1, y_1, x_2, y_2, x_1', y_1', x_2', y_2', u, v \\ & (R(n_1, x_1, y_1, x_2, y_2) \wedge (x_1 \leq u \leq x_2 \wedge y_1 \leq v \leq y_2) \wedge \\ & R(n_2, x_1', y_1', x_2', y_2') \wedge (x_1' \leq u \leq x_2' \wedge y_1' \leq v \leq y_2'))\}.\end{aligned}$$

This is essentially within the framework of the relational model presented so far, except that we are using an infinite base relation $\leq$. There is a level of indirection between the representation of a rectangle (a, x_1, y_1, x_2, y_2) and the actual set of points that it contains.

In the following constraint formalism, a named rectangle can be represented by a "generalized tuple" (i.e., a constraint). For instance, the rectangle of name a with corners (0.5, 1.0) and (1.5, 5.5) is represented by the constraint

$$z_1 = a \wedge 0.5 \leq z_2 \wedge z_2 \leq 1.5 \wedge 1.0 \leq z_3 \wedge z_3 \leq 5.5.$$

This should be viewed as a finite syntactic representation of an infinite set of triples. A triple $\langle z_1, z_2, z_3\rangle$ satisfying this constraint indicates that the point of coordinates (z_2, z_3) is in a rectangle with name z_1.

One can see a number of uses in allowing constraints in the language. First, constraints arise naturally for domains concerning measures (price, distance, time, etc.). The introduction of time has already been studied in the active area of temporal databases (see Section 22.6). In other applications such as spatial databases, geometry plays an essential role and fits nicely in the realm of constraint query languages.

One can clearly obtain different languages by considering various domains and various forms of constraints. Relational calculus, relational algebra, or some other relational languages can be extended with, for instance, the theory of real closed fields or the theory of dense orders without endpoints. Of course, a requirement is the decidability of the resulting language.

Assuming some notion of constraints (to be formalized soon), we now define somewhat more precisely the constraint languages and then illustrate them with two examples.

Definition 5.6.3 A *generalized n-tuple* is a finite conjunction of constraints over variables $x_1, \ldots, x_n$. A *generalized instance* of arity n is a finite set of generalized n-tuples (the corresponding formula is the disjunction of the constraints).

Suppose that I is a generalized instance. We refer to I as a *syntactic* database and to the set of conventional tuples represented by I as the *semantic* database.

We now present two applications of this approach, one in connection with the reals and the other with the rationals.

We assume now that the constants are interpreted over a real closed field (e.g., the reals). The constraints are polynomial inequality constraints [i.e., inequalities of the form $p(x_1, \ldots, x_n) \geq 0$, where p is a polynomial]. Two 3-tuples in this context are

$$(3.56 \times x_1^2 + 4.0 \times x_2 \geq 0) \wedge (x_3 - x_1 \geq 0)$$
$$(x_1 + x_2 + x_3 \geq 0).$$

One can evaluate queries algebraically bottom-up (i.e., at each step of the computation, the result is still a generalized instance). This is a straightforward consequence of Tarski's decision procedure for the theory of real closed fields. A difficulty resides in projection (i.e., quantifier elimination). The procedure for projection is extremely costly in the size of the query. However, for a fixed query, the complexity *in the size of the syntactic database* is reasonable (in NC).

We assume now that the constants are interpreted over a countably infinite set with a binary relation $\leq$ that is a dense order (e.g., the rationals). The constraints are of the form $x\theta y$ or $x\theta c$, where x, y are variables, c is a constant, and θ is among $\leq, <, =$. An example of a 3-tuple is

$$(x_1 \leq x_2) \wedge (x_2 < x_3).$$

Here again, a bottom-up algebraic evaluation is feasible. Indeed, evaluation is in AC_0 in the size of the syntactic database (for a fixed query).

In the remainder of this book, we consider standard databases and not generalized ones.

Bibliographic Notes

One of the first investigations of using predicate calculus to query relational database structures is [Kuh67], although the work by Codd [Cod70, Cod72b] played a tremendous role in bringing attention to the relational model and to the relational algebra and calculus. In particular, [Cod72b] introduced the equivalence of the calculus and algebra to the database community. That paper coined the phrase *relational completeness* to describe the expressive power of relational query languages: Any language able to simulate the algebra is called relationally complete. We have not emphasized that phrase here because subsequent research has suggested that a more natural notion of completeness can be described in terms of Turing computability (see Chapter 16).

Actually, a version of the result on the equivalence of the calculus and algebra was known much earlier to the logic community (see [CT48, TT52]) and is used to show Tarski's algebraization theorem (e.g., see [HMT71]). The relation between relational algebras and cylindric algebras is studied in [IL84] (see Exercise 5.36). The development of algebras equivalent to various calculus languages has been a fertile area for database theory. One such result presented in this chapter is the equivalence of the positive existential cal-

culus and the SPCU algebra [CH82]; analogous results have also been developed for the relational calculus extended with aggregate operators [Klu82], the complex value model [AB88] (studied in Chapter 20), the Logical Data Model [KV84], the directory model [DM86a, DM92], and formalizations of the hierarchy and network models using database logic [Jac82].

Notions related to domain independence are found as early as [Low15] in the logic community; in the database community the first paper on this topic appears to be [Kuh67], which introduced the notion of *definite* queries. The notion of domain independence used here is from [Fag82b, Mak81]; the notions of definite and domain independent were proved equivalent in [ND82]. A large number of classes of domain-independent formulas have been investigated. These include the safe [Ull82b], safe DRC [Ull88], range separable [Cod72b], allowed [Top87] (Exercise 5.25), range restricted [Nic82] (Exercise 5.26), and evaluable [Dem82] formulas. An additional investigation of domain independence for the calculus is found in [ND82]. Surveys on domain independence and various examples of these classes can be found in [Kif88, VanGT91]. The focus of [VanGT91] is the general problem of practical translations from calculus to algebra queries; in particular, it provides a translation from the set of evaluable formulas into the algebra. It is also shown there that the notions of evaluable and range restricted are equivalent. These are the most general syntactic conditions in the literature that ensure domain independence. The fact that domain independence is undecidable was first observed in [DiP69]; this is detailed in Chapter 6.

Domain independence also arises in the context of dependencies [Fag82b, Mak81] and datalog [Dec86, Top87, RBS87, TS88]. The issue of extending domain independence to incorporate functions (e.g., arithmetic functions, or user-defined functions) is considered in [AB88, Top91, EHJ93]. The issue of extending domain independence to incorporate freely interpreted functions (such as arise in logic programming) is addressed in [Kif88]. Syntactic conditions on (recursive) datalog programs with arithmetic that ensure safety are developed in [RBS87, KRS88a, KRS88b, SV89]. Issues of safety in the presence of function or order symbols are also considered in [AH91]. Aggregate functions were first incorporated into the relational algebra and calculus in [Klu82]; see also [AB88].

The notion of safe range presented here is richer than safe, safe DRC, and range separable and weaker than allowed, evaluable, and range restricted. It follows the spirit of the definition of allowed presented in [VanGT91] and safe range in [AB88]. The transformations of the safe-range calculus to the algebra presented here follows the more general transformations in [VanGT91, EHJ93]. The notion of "relational algebra normal form" used in those works is more general than the notion by that name used here.

Query languages have mostly been considered for finite databases. An exception is [HH93]. Theorem 5.6.1 is due to [AGSS86]. An alternative proof and extension of this result is developed in [HS94].

Programming with constraints has been studied for some time in topic areas ranging from linear programming to AI to logic programming. Although the declarative spirit of both constraint programming and database query languages leads to a natural marriage, it is only recently that the combination of the two paradigms has been studied seriously [KKR90]. This was probably a consequence of the success of constraints in the field of logic programming (see, e.g., [JL87] and [Lel87, Coh90] for surveys). Our presentation was influenced by [KKR90] (calculii for closed real fields with polynomial inequalities and for dense order with inequalities are studied there) as well as by [KG94] (an algebra

for constraint databases with dense order and inequalities is featured there). Recent works on constraint database languages can be found in [Kup93, GS94].

Exercises

Exercise 5.1 Express queries (5.2 and 5.3) in (1) the relational algebras, (2) nonrecursive datalog$^\neg$, and (3) domain-independent relational calculus.

Exercise 5.2 Express the following queries against the *CINEMA* database in (1) the relational algebras, (2) nonrecursive datalog$^\neg$, and (3) domain-independent relational calculus.

(a) Find the actors cast in at least one movie by Kurosawa.

(b) Find the actors cast in every movie by Kurosawa.

(c) Find the actors cast only in movies by Kurosawa.

(d) Find all pairs of actors who act together in at least one movie.

(e) Find all pairs of actors cast in exactly the same movies.

(f) Find the directors such that every actor is cast in one of his or her films.

(Assume that each film has exactly one director.)

Exercise 5.3 Prove or disprove (assuming $X \subseteq sort(P) = sort(Q)$):

(a) $\pi_X(P \cup Q) = \pi_X(P) \cup \pi_X(Q)$;

(b) $\pi_X(P \cap Q) = \pi_X(P) \cap \pi_X(Q)$.

Exercise 5.4

(a) Give formal definitions for the syntax and semantics of the unnamed and named relational algebras.

(b) Show that in the unnamed algebra $\cap$ can be simulated using (1) the difference operator $-$; (2) the operators $\times, \pi, \sigma$.

(c) Give a formal definition for the syntax and semantics of selection operators in the unnamed algebra that permit conjunction, disjunction, and negation in their formulas. Show that these selection operators can be simulated using atomic selection operators, union, intersect, and difference.

★(d) Show that the SPCU algebra, in which selection operators with negation in the formulas are permitted, cannot simulate the difference operator.

★(e) Formulate and prove results analogous to those of parts (b), (c), and (d) for the named algebra.

Exercise 5.5

(a) Prove that the unnamed algebra operators $\{\sigma, \pi, \times, \cup, -\}$ are nonredundant.

(b) State and prove the analogous result for the named algebra.

Exercise 5.6

(a) Exhibit a relational algebra query that is not monotonic.

(b) Exhibit a relational algebra query that is not satisfiable.

Exercise 5.7 Prove Proposition 5.1.2 (i.e., that the unnamed and named relational algebras have equivalent expressive power).

Exercise 5.8 (Division) The *division* operator, denoted $\div$, is added to the named algebra as follows. For instances I and J with $sort(J) \subseteq sort(I)$, the value of $I \div J$ is the set of tuples $r \in \pi_{sort(I)-sort(J)}(I)$ such that $(\{r\} \bowtie J) \subseteq I$. Use the division to express algebraically the query, "Which theater is featuring all of Hitchcock's movies?". Describe how nr-datalog$^\neg$ can be used to simulate division. Describe how the named algebra can simulate division. Is division a monotonic operation?

Exercise 5.9 Show that the semantics of each nr-datalog$^\neg$ rule can be described as a difference $q_1 - q_2$, where q_1 is an SPJR query and q_2 is an SPJRU query.

Exercise 5.10 Verify that each nr-datalog$^\neg$ program with equality can be simulated by one without equality.

Exercise 5.11 Prove Proposition 5.2.2. *Hint:* Use the proof of Theorem 4.4.8 and the fact that the relational algebra is closed under composition.

★ **Exercise 5.12** Prove that the domain-independent relational calculus without equality is strictly weaker than the domain-independent relational calculus. *Hint:* Suppose that calculus query q without equality is equivalent to $\{x \mid R(x) \wedge x \neq a\}$. Show that q can be translated into an algebra query q' that is constructed without using a constant base relation and such that all selections are on base relation expressions. Argue that on each input relation I over R, each subexpression of q' evaluates to either I^n for some $n \geq 0$, or to the empty relation for some $n \geq 0$.

Exercise 5.13

(a) Complete the proof of Lemma 5.3.11.

(b) Complete the proof of Lemma 5.3.12.

Exercise 5.14

(a) Prove that the rewrite rules of Figure 5.1 preserve equivalence.

(b) Prove that these rewrite rules can be used to transform an arbitrary calculus formula into an equivalent formula in PNF with CNF matrix. State which rewrite rules are needed.

(c) Do the same as (b), but for DNF matrix.

(d) Prove that the rewrite rules of Figure 5.1 are not complete in the sense that there are calculus formulas φ and ψ such that (1) $\varphi \equiv \psi$, but (2) there is no sequence of applications of the rewrite rules that transforms φ into ψ.

Exercise 5.15 Verify the claims of Example 5.3.9.

Exercise 5.16

(a) Show that each positive existential query is equivalent to one whose formula is in PNF with either CNF or DNF matrix and that they can be expressed in the form $\{e_1, \ldots, e_n \mid \psi_1 \vee \cdots \vee \psi_m\}$, where each ψ_j is a conjunctive calculus formula with $free(\psi_j)$ = the set of variables occurring in $e_1, \ldots, e_n$. Note that this formula is safe range.

(b) Show that it is decidable, given a relational calculus query q (possibly with equality) whose only logical connectives are $\wedge$, $\vee$, and $\exists$, whether q is domain independent.

(c) Prove Theorem 5.4.10.

Exercise 5.17 Use the construction of the proof of Theorem 5.4.6 to transform the following into the algebra.

(a) $\{\langle\,\rangle \mid \exists x(R(x) \wedge \exists y(S(x, y) \wedge \neg\exists z(T(x, y, a))))\}$

(b) $\{w, x, y, z \mid (R(w, x, y) \vee R(w, x, z)) \wedge (R(y, z, w) \vee R(y, z, x))\}$

Exercise 5.18 For each of the following queries, indicate whether it is domain independent and/or safe range. If it is not domain independent, give examples of different domains yielding different answers on the same input; and if it is safe range, translate it into the algebra.

(a) $\{x, y \mid \exists z[T(x, z) \wedge \exists w T(w, x, y)] \wedge x = y\}$

(b) $\{x, y \mid [x = a \vee \exists z(R(y, z))] \wedge S(y)\}$

(c) $\{x, y \mid [x = a \vee \exists z(R(y, z))] \wedge S(y) \wedge T(x)\}$

(d) $\{x \mid \forall y(R(y) \rightarrow S(x, y))\}$

(e) $\{\langle\rangle \mid \exists x \forall y(R(y) \rightarrow S(x, y))\}$

(f) $\{x, y \mid \exists z T(x, y, z) \wedge \exists u, v([(R(u) \vee S(u, v)) \wedge R(v)]$
$\rightarrow [\exists w(T(x, w, v) \wedge T(u, v, y))])\}$

★ **Exercise 5.19** Consider the proof of Theorem 5.4.6.

(a) Give the missing parts of Algorithm 5.4.7.

(b) Show that Algorithm 5.4.7 is correct and terminates on all input.

(c) Give the missing parts of Algorithm 5.4.8 and verify its correctness.

(d) Given $q = \{\langle x_1, \ldots, x_n\rangle \mid \varphi\}$ with φ in modified RANF, show for each instance $\mathbf{I}$ and each $\mathbf{d}$ satisfying $adom(q, \mathbf{I}) \subseteq \mathbf{d} \subseteq \mathbf{dom}$ that $q_{\mathbf{d}}(\mathbf{I}) = E_\varphi(\mathbf{I})$.

Exercise 5.20 Consider the proof of Theorem 5.4.6.

(a) Present examples illustrating how the nondeterministic choices in these rewrite rules can be used to help optimize the algebra query finally produced by the construction of the proof of this lemma. (Refer to Chapter 6 for a general discussion of optimization.)

(b) Consider a generalization of rules (R1) and (R2) that permits using a set of indexes $\{j_1, \ldots, j_l\} \subseteq \{1, \ldots, n\} - \{i_1, \ldots, i_k\}$. What are the advantages of this generalization? What restrictions must be imposed to ensure that Algorithm 5.4.8 remains correct?

Exercise 5.21 Develop a direct proof that $\mathrm{CALC}_{adom} \sqsubseteq \mathrm{CALC}_{sr}$. *Hint:* Given calculus query q, first build a formula $\xi_{adom}(x)$ such that $\mathbf{I} \models \xi_{adom}(x)[\nu]$ iff $\nu(x) \in adom(q, \mathbf{I})$. Now perform an induction on subformulas.

★ **Exercise 5.22** [Coh86] Let R have arity n. Define the *gen*(*erator*) operator so that for instance I of R, indexes $1 \leq i_1 < \cdots < i_k \leq n$, and constants $a_1, \ldots, a_k$,

$$gen_{i_1:a_1,\ldots,i_k:a_k}(I) = \pi_{j_1,\ldots,j_l}(\sigma_{i_1=a_1 \wedge \cdots \wedge i_k=a_k}(I)),$$

where $\{j_1, \ldots, j_l\}$ is a listing in order of (some or all) indexes in $\{1, \ldots, n\} - \{i_1, \ldots, i_k\}$. Note that the special case of $gen_{1:b_1,\ldots,n:b_n}(I)$ can be viewed as a test of $\langle b_1, \ldots, b_n\rangle \in I$; and $gen_{[\,]}(I)$

is a test of whether I is nonempty. In some research in AI, the primitive mechanism for accessing relations is based on generators that are viewed as producing a stream of tuples as output. For example, the query $\{\langle x, y, z\rangle \mid R(x, y) \wedge S(y, z)\}$ can be computed using the algorithm

for each tuple $\langle x, y\rangle$ generated by $gen_{1:x,2:y}(R)$
  for each value $\langle z\rangle$ generated by $gen_{1:y}(S)$
    output $\langle x, y, z\rangle$
  end for each
end for each

Develop an algorithm for translating calculus queries into programs using generators. Describe syntactic restrictions on the calculus that ensure that your algorithm succeeds.

♠ **Exercise 5.23** [Cod72b] (Tuple calculus.) We use a set **tvar** of sorted tuple variables. The *tuple calculus* is defined as follows. If t is a tuple variable and A is an attribute in the sort of t, $t.A$ is a *term*. A constant is also a *term*. The atomic formulas are either of the form $R(t)$ with the appropriate constraint on sorts, or $e = e'$, where e, e' are terms. Formulas are constructed as in the standard relational calculus. For example, query (5.1) is expressed by the tuple calculus query

$$\{t\colon \mathit{title} \mid \exists s\colon \mathit{title}, \mathit{director}, \mathit{actor}[\mathit{Movie}(s) \wedge t.\mathit{title} = s.\mathit{title} \wedge s.\mathit{director} = \text{“Hitchcock”}] \wedge \neg\exists u\colon \mathit{title}, \mathit{director}, \mathit{actor}[\mathit{Movie}(u) \wedge u.\mathit{title} = s.\mathit{title} \wedge u.\mathit{actor} = \text{“Hitchcock”}]\}.$$

Give a formal definition for the syntax of the tuple calculus and for the relativized interpretation, active domain, and domain-independent semantics. Develop an analog of safe range. Prove the equivalence of conventional calculus and tuple calculus under all of these semantics.

Exercise 5.24 Prove that the relational calculus and the family of nr-datalog$^\neg$ programs with single-relation output have equivalent expressive power by using direct simulations between the two families.

♠ **Exercise 5.25** [Top87] Let **R** be a database schema, and define the binary relation *gen*(*erates*) on variables and formulas as follows:

$gen(x, \varphi)$	if $\varphi = R(u)$ for some $R \in \mathbf{R}$ and $x \in free(\varphi)$
$gen(x, \neg\varphi)$	if $gen(x, pushnot(\neg\varphi))$
$gen(x, \exists y\varphi)$	if x, y are distinct and $gen(x, \varphi)$
$gen(x, \forall y\varphi)$	if x, y are distinct and $gen(x, \varphi)$
$gen(x, \varphi \vee \psi)$	if $gen(x, \varphi)$ and $gen(x, \psi)$
$gen(x, \varphi \wedge \psi)$	if $gen(x, \varphi)$ or $gen(x, \psi)$,

where $pushnot(\neg\varphi)$ is defined in the natural manner to be the result of pushing the negation into the next highest level logical connective (with consecutive negations cancelling each other) unless φ is an atom (using the rewrite rules 5, 6, 7, 10, and 11 from Fig. 5.1). A formula φ is *allowed*

(i) if $x \in free(\varphi)$ then $gen(x, \varphi)$;

(ii) if for each subformula $\exists y\psi$ of φ, $gen(y, \psi)$ holds; and

(iii) if for each subformula $\forall y\psi$ of φ, $gen(y, \neg\psi)$ holds.

A calculus query is *allowed* if its formula is allowed.

(a) Exhibit a query that is allowed but not safe range.

★(b) Prove that each allowed query is domain independent.

In [VanGT91, EHJ93] a translation of allowed formulas into the algebra is presented.)

★ **Exercise 5.26** [Nic82] The notion of "range-restricted" queries, which ensures domain independence, is based on properties of the normal form equivalents of queries. Let $q = \{\vec{x} \mid \varphi\}$ be a calculus query, and let $\varphi_{DNF} = \vec{\exists} y(D_1 \vee \cdots \vee D_n)$ be the result of transforming φ into PNF with DNF matrix using the rewrite rules of Fig. 5.1; and similarly let $\varphi_{CNF} = \vec{\exists} z(C_1 \wedge \cdots \wedge C_m)$ be the result of transforming φ into PNF with CNF matrix. The query q is *range restricted* if

(i) each free variable x in φ occurs in a positive literal (other than $x = y$) in every D_i;

(ii) each existentially quantified variable x in φ_{DNF} occurs in a positive literal (other than $x = y$) in every D_i where x occurs; and

(iii) each universally quantified variable x in φ_{CNF} occurs in a negative literal (other than $x \neq y$) in every C_j where x occurs.

Prove that range-restricted queries are domain independent. (In [VanGT91] a translation of the range-restricted queries into the algebra is presented.)

Exercise 5.27 [VanGT91] Suppose that R[*Product*, *Part*] holds project numbers and the parts that are used to make them, and S[*Supplier*, *Part*] holds supplier names and the parts that they supply. Consider the queries

$$q_1 = \{x \mid \forall y(R(100, y) \rightarrow S(x, y))\}$$
$$q_2 = \{\langle\rangle \mid \exists x \forall y(R(100, y) \rightarrow S(x, y))\}$$

(a) Prove that q_1 is not domain independent.

(b) Prove that q_2 is not allowed (Exercise 5.25) but it is range restricted (Exercise 5.26) and hence domain independent.

(c) Find an algebra query q' equivalent to q_2.

Exercise 5.28 [Klu82] Consider a database schema with relations *Dept*[*Name*, *Head*, *College*], *Faculty*[*Name*, *Dname*], and *Grad*[*Name*, *MajorProf*, *GrantAmt*], and the query

> For each department in the Letters and Science College, compute the total graduate student support for each of the department's faculty members, and produce as output a relation that includes all pairs $\langle d, a\rangle$ where d is a department in the Letters and Science College, and a is the average graduate student support per faculty member in d.

Write algebra and calculus queries that express this query.

Exercise 5.29 We consider constraint databases involving polynomial inequalities over the reals. Let $I_1 = \{(9x_1^2 + 4x_2 \geq 0)\}$ be a generalized instance over AB, where x_1 ranges over A and x_2 ranges over B, and let $I_2 = \{(x_3 - x_1 \geq 0)\}$ over AC. Express $\pi_{BC}(I_1 \bowtie I_2)$ as a generalized instance.

★ **Exercise 5.30** Recall Theorem 5.6.1.

(a) Let finite $\mathbf{d} \subset \mathbf{dom}$ be fixed, C be a set of new symbols, and t be a tuple with placeholders. Describe a generalized tuple (in the sense of constraint databases) t' whose semantics are equal to $sem_{\mathbf{d}}(t)$.

(b) Show that the family of databases representable by sets of tuples with placeholders is closed under the relational calculus.

♠ **Exercise 5.31** Prove Theorem 5.6.1.

Exercise 5.32 [Mai80] (Unrestricted algebra) For this exercise we permit relations to be finite or infinite. Consider the *complement* operator ^{c} defined on instances I of arity n by $I^c = \mathbf{dom}^n - I$. (The analogous operator is defined for the named algebra.) Prove that the calculus under the natural interpretation is equivalent to the algebra with operators $\{\sigma, \pi, \times, \cup, ^c\}$.

★ **Exercise 5.33** A total mapping τ from instances over $\mathbf{R}$ to instances over S is *C-generic* for $C \subseteq \mathbf{dom}$, iff for each bijection ρ over $\mathbf{dom}$ that is the identity on C, τ and ρ commute. That is, $\tau(\rho(\mathbf{I})) = \rho(\tau(\mathbf{I}))$ for each instance $\mathbf{I}$ of $\mathbf{R}$. The mapping τ is *generic* if it is C-generic for some finite $C \subseteq \mathbf{dom}$. Prove that each relational algebra query is generic—in particular, that each algebra query q is $adom(q)$-generic.

♠ **Exercise 5.34** Let R be a unary relation name. A *hyperplane query* over R is a query of the form $\sigma_F(R \times \cdots \times R)$ (with 0 or more occurrences of R), where F is a conjunction of atoms of the form $i = j$, $i \neq j$, $i = a$, or $i \neq a$ (for indexes i, j and constant a). A formula F of this form is called a *hyperplane formula*. A *hyperplane-union query* over R is a query of the form $\sigma_F(R \times \cdots \times R)$, where F is a disjunction of hyperplane formulas; a formula of this form is called a *hyperplane-union* formula.

(a) Show that if q is an algebra query over R, then there is an $n \geq 0$ and a hyperplane-union query q' such that for all instances I over R, if $|I| \geq n$ and $adom(I) \cap adom(q) = \emptyset$, then $q(I) = q'(I)$.

The query *even* is defined over R as follows: $even(I) = \{\langle\rangle\}$ (i.e., yes) if $|I|$ is even; and $even(I) = \{\}$ (i.e., no) otherwise.

(b) Prove that there is no algebra query q over R such that $q \equiv even$.

Exercise 5.35 [CH80b] (Unsorted algebra) An *h-relation* (for heterogeneous relation) is a finite set of tuples not necessarily of the same arity.

(a) Design an algebra for h-relations that is at least as expressive as relational algebra.

★ (b) Show that the algebra in (a) can be chosen to have the additional property that if q is a query in this algebra taking standard relational input and producing standard relational output, then there is a standard algebra query q' such that $q' \equiv q$.

♠ **Exercise 5.36** [IL84] (Cylindric algebra) Let n be a positive integer, $R[A_1, \ldots, A_n]$ a relation schema, and C a (possibly infinite) set of constants. Recall that a *Boolean algebra* is a 6-tuple $(\mathcal{B}, \vee, \wedge, \overline{\ }, \bot, \top)$, where $\mathcal{B}$ is a set containing $\bot$ and $\top$; $\vee$, $\wedge$ are binary operations on $\mathcal{B}$; and $\overline{\ }$ is a unary operation on $\mathcal{B}$ such that for all $x, y, z \in \mathcal{B}$:

(a) $x \vee y = y \vee x$;

(b) $x \wedge y = y \wedge x$;

(c) $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$;

(d) $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$;

(e) $x \wedge \bot = \bot$;
(f) $x \vee \top = \top$;
(g) $x \wedge \overline{x} = \bot$;
(h) $x \vee \overline{x} = \top$; and
(i) $\bot \neq \top$.

For a Boolean algebra, define $x \leq y$ to mean $x \wedge y = x$.

(a) Show that $\langle \mathcal{R}_C, \cup, \cap, ^c, \emptyset, C^n \rangle$ is a Boolean algebra where $\mathcal{R}_C$ is the set of all (possibly infinite) R-relations over constants in C and ^{c} denotes the unary *complement* operator, defined so that $I^c = C^n - I$. In addition, show that $I \leq J$ iff $I \subseteq J$.

Let the diagonals d_{ij} be defined by the statement, "for each i, j, $d_{ij} = \sigma_{A_i = A_j}(C^n)$"; and let the i^{th} cylinder C_i be defined for each I by the statement, "$C_i I$ is the relation over $\mathcal{R}_C$ defined by

$$C_i I = \{t \mid \pi_{A_1 \ldots A_{i-1} A_{i+1} \ldots A_n}(t) \in \pi_{A_1 \ldots A_{i-1} A_{i+1} \ldots A_n}(I) \text{ and } t(A_i) \in C\}."$$

(b) Show the following properties of cylindric algebras: (1) $C_i \emptyset = \emptyset$; (2) $x \leq C_i x$; (3) $C_i(x \cap C_i y) = C_i x \cap C_i y$; (4) $C_i C_j x = C_j C_i x$; (5) $d_{ii} = C^n$; (6) if $i \neq j$ and $i \neq k$, then $d_{jk} = C_i(d_{ji} \cap d_{ik})$; (7) if $i \neq j$, then $C_i(d_{ij} \cap x) \cap C_i(d_{ij} \cap \overline{x}) = \emptyset$.

(c) Let h be the mapping from any (possibly infinite) relation S with $sort(S) \subset A_1 \ldots A_n$ with entries in C to a relation over R obtained by extending each tuple in S to $A_1 \ldots A_n$ in all possible ways with values in C. Prove that (1) $h(R_1 \bowtie R_2) = h(R_1) \cap h(R_2)$ and (2) if $A_1 \in sort(R)$, then $h(\pi_{A_1}(R)) = C_1 h(R_1)$.

6 Static Analysis and Optimization

Alice: *Do you guys mean* real *optimization?*
Riccardo: *Well, most of the time it's local maneuvering.*
Vittorio: *But sometimes we go beyond incremental reform . . .*
Sergio: *. . . with provably global results.*

This chapter examines the conjunctive and first-order queries from the perspective of static analysis (in the sense of programming languages). It is shown that many properties of conjunctive queries (e.g., equivalence, containment) are decidable although they are not decidable for first-order queries. Static analysis techniques are also applied here in connection with query optimization (i.e., transforming queries expressed in a high-level, largely declarative language into equivalent queries or machine instruction programs that are arguably more efficient than a naive execution of the initial query).

To provide background, this chapter begins with a survey of practical optimization techniques for the conjunctive queries. The majority of practically oriented research and development on query optimization has been focused on variants of the conjunctive queries, possibly extended with arithmetic operators and comparators. Because of the myriad factors that play a role in query evaluation, most practically successful techniques rely heavily on heuristics.

Next the chapter presents the elegant and important *Homomorphism Theorem*, which characterizes containment and equivalence between conjunctive queries. This leads to the notion of tableau "minimization": For each tableau query there is a unique (up to isomorphism) equivalent tableau query with the smallest number of rows. This provides a theoretical notion of true optimality for conjunctive queries. It is also shown that deciding these properties and minimizing conjunctive queries is NP-complete in the size of the input queries.

Undecidability results are then presented for the first-order queries. Although related to undecidability results for conventional first-order logic, the proof techniques used here are necessarily different because all instances considered are finite by definition. The undecidability results imply that there is no hope of developing an algorithm that performs optimization of first-order queries that is complete. Only limited optimization of first-order queries involving difference is provided in most systems.

The chapter closes by returning to a specialized subset of the conjunctive queries based on *acyclic joins*. These have been shown to enjoy several interesting properties, some yielding insight into more efficient query processing.

Chapter 13 in Part D examines techniques for optimizing datalog queries.

6.1 Issues in Practical Query Optimization

Query optimization is one of the central topics of database systems. A myriad of factors play a role in this area, including storage and indexing techniques, page sizes and paging protocols, the underlying operating system, statistical properties of the stored data, statistical properties of anticipated queries and updates, implementations of specific operators, and the expressive power of the query languages used, to name a few. Query optimization can be performed at all levels of the three-level database architecture. At the physical level, this work focuses on, for example, access techniques, statistical properties of stored data, and buffer management. At a more logical level, algebraic equivalences are used to rewrite queries into forms that can be implemented more efficiently.

We begin now with a discussion of rudimentary considerations that affect query processing (including the usual cost measurements) and basic methods for accessing relations and implementing algebraic operators. Next an optimization approach based on algebraic equivalences is described; this is used to replace a given algebraic expression by an equivalent one that can typically be computed more quickly. This leads to the important notion of query evaluation plans and how they are used in modern systems to represent and choose among many alternative implementations of a query. We then examine intricate techniques for implementing multiway joins based on different orderings of binary joins and on join decomposition.

The discussion presented in this section only scratches the surface of the rich body of systems-oriented research and development on query optimizers, indicating only a handful of the most important factors that are involved. Nothing will be said about several factors, such as the impact of negation in queries, main-memory buffering strategies, and the implications of different environments (such as distributed, object oriented, real time, large main memory, and secondary memories other than conventional disks). In part due to the intricacy and number of interrelated factors involved, little of the fundamental theoretical research on query optimization has found its way into practice. As the field is maturing, salient aspects of query optimization are becoming isolated; this may provide some of the foothold needed for significant theoretical work to emerge and be applied.

The Physical Model

The usual assumption of relational databases is that the current database state is so large that it must be stored in secondary memory (e.g., on disk). Manipulation of the stored data, including the application of algebraic operators, requires making copies in primary memory of portions of the stored data and storing intermediate and final results again in secondary memory. By far the major time expense in query processing, for a single-processor system, is the number of disk pages that must be swapped in and out of primary memory. In the case of distributed systems, the communication costs typically dominate all others and become an important focus of optimization.

Viewed a little more abstractly, the physical level of relational query implementation involves three basic activities: (1) generating streams of tuples, (2) manipulating streams

of tuples (e.g., to perform projections), and (3) combining streams of tuples (e.g., to perform joins, unions, and intersections). Indexing methods, including primarily B-trees and hash indexes, can be used to reduce significantly the size of some streams. Although not discussed here, it is important to consider the cost of maintaining indexes and clusterings as updates to the database occur.

Main-memory buffering techniques (including the partitioning of main memory into segments and paging policies such as deleting pages based on policies of least recent use and most recent use) can significantly impact the number of page I/Os used.

Speaking broadly, an *evaluation plan* (or *access plan*) for a query, a stored database state, and a collection of existing indexes and other data structures is a specification of a sequence of operations that will compute the answer to the query. The term evaluation plan is used most often to refer to specifications that are at a low physical level but may sometimes be used for higher-level specifications. As we shall see, query optimizers typically develop several evaluation plans and then choose one for execution.

Implementation of Algebraic Operators

To illustrate the basic building blocks from which evaluation plans are constructed, we now describe basic implementation techniques for some of the relational operators.

Selection can be realized in a straightforward manner by a scan of the argument relation and can thus be achieved in linear time. Access structures such as B-tree indexes or hash tables can be used to reduce the search time needed to find the selected tuples. In the case of selections with single tuple output, this permits evaluation within essentially constant time (e.g., two or three page fetches). For larger outputs, the selection may take two or three page fetches per output tuple; this can be improved significantly if the input relation is *clustered* (i.e., stored so that all tuples with a given attribute value are on the same or contiguous disk pages).

Projection is a bit more complex because it actually calls for two essentially different operations: *tuple rewriting* and *duplicate elimination*. The tuple rewriting is typically accomplished by bringing tuples into primary memory and then rewriting them with coordinate values permuted and removed as called for. This may yield a listing of tuples that contains duplicates. If a pure relational algebra projection is to be implemented, then these duplicates must be removed. One strategy for this involves sorting the list of tuples and then removing duplicates; this takes time on the order of $n \log n$. Another approach that is faster in some cases uses a hash function that incorporates all coordinate values of a tuple.

Because of the potential expense incurred by duplicate elimination, most practical relational languages permit duplicates in intermediate and final results. An explicit command (e.g., *distinct*) that calls for duplicate elimination is typically provided. Even in languages that support a pure algebra, it may be more efficient to leave duplicates in intermediate results and perform duplicate elimination once as a final step.

The equi-join is typically much more expensive than selection or projection because two relations are involved. The following naive *nested loop* implementation of $\bowtie_F$ will take time on the order of the product $n_1 \times n_2$ of the sizes of the input relations I_1, I_2:

$J := \emptyset$;
for each u **in** I_1
  for each v **in** I_2
    if u and v are joinable **then** $J := J \cup \{u \bowtie_F v\}$.

Typically this can be improved by using the *sort-merge* algorithm, which independently sorts both inputs according to the join attributes and then performs a simultaneous scan of both relations, outputting join tuples as discovered. This reduces the running time to the order of $\max(n_1 \log n_1 + n_2 \log n_2,$ size of output).

In many cases a more efficient implementation of join can be accomplished by a variant of the foregoing nested loop algorithm that uses indexes. In particular, replace the inner loop by indexed retrievals to tuples of I_2 that match the tuple of I_1 under consideration. Assuming that a small number of tuples of I_2 match a given tuple of I_1, this computes the join in time proportional to the size of I_1. We shall consider implementations of multiway joins later in this section and again in Section 6.4. Additional techniques have been developed for implementing richer joins that include testing, e.g., relationships based on order ($\leq$).

Cross-product in isolation is perhaps the most expensive algebra operation: The output necessarily has size the product of the sizes of the two inputs. In practice this arises only rarely; it is much more common that selection conditions on the cross-product can be used to transform it into some form of join.

Query Trees and Query Rewriting

Alternative query evaluation plans are usually generated by rewriting (i.e., by local transformation rules). This can be viewed as a specialized case of program transformation. Two kinds of transformations are typically used in query optimization: one that maps from the higher-level language (e.g., the algebra) into the physical language, and others that stay within the same language but lead to alternative, equivalent implementations of a given construct.

We present shortly a family of rewriting rules that illustrates the general flavor of this component of query optimizers (see Fig. 6.2). Unlike true optimizers, however, the rules presented here focus exclusively on the algebra. Later we examine the larger issue of how rules such as these are used to find optimal and near-optimal evaluation plans.

We shall use the SPC algebra, generalized by permitting positive conjunctive selection and equi-join. A central concept used is that of *query tree*, which is essentially the parse tree of an algebraic expression. Consider again Query (4.4), expressed here as a rule:

$$ans(x_{th}, x_{ad}) \leftarrow Movies(x_{ti}, \text{“Bergman”}, x_{ac}),\ Pariscope(x_{th}, x_{ti}, x_s),\ Location(x_{th}, x_{ad}, x_p).$$

A naive translation into the generalized SPC algebra yields

$$q_1 = \pi_{4,8}\sigma_{2=\text{“Bergman”}}((Movies \bowtie_{1=2} Pariscope) \bowtie_{4=1} Location).$$

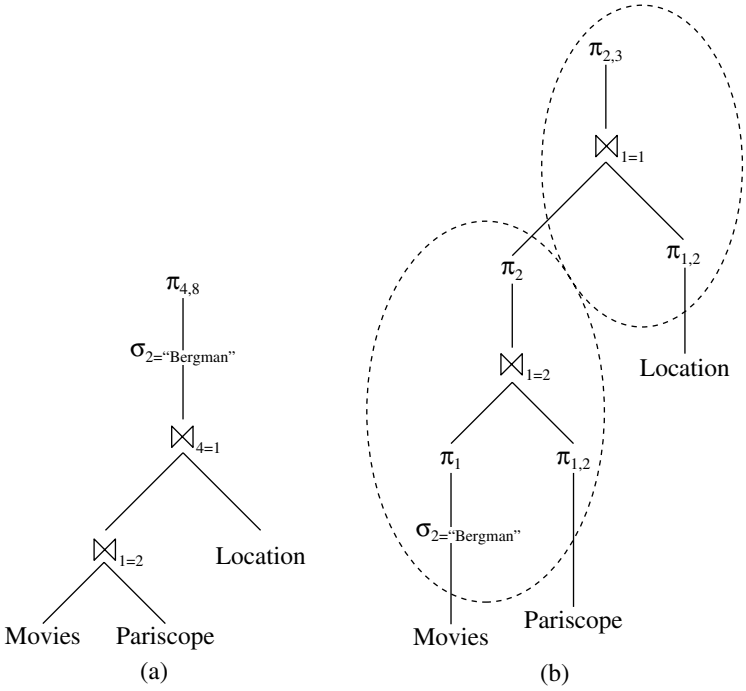


Figure 6.1: Two query trees for Query (4.4) from Chapter 4

The query tree of this expression is shown in Fig. 6.1(a).

To provide a rough idea of how evaluation costs might be estimated, suppose now that *Movies* has 10,000 tuples, with about 5 tuples per movie; *Pariscope* has about 200 tuples, and *Location* has about 100 tuples. Suppose further that in each relation there are about 50 tuples per page and that no indexes are available.

Under a naive evaluation of q_1, an intermediate result would be produced for each internal node of q_1's query tree. In this example, then, the join of *Movies* and *Pariscope* would produce about $200 \times 5 = 1000$ tuples, which (being about twice as wide as the input tuples) will occupy about 40 pages. The second equi-join will yield about 1000 tuples that fit 18 to a page, thus occupying about 55 pages. Assuming that there are four Bergman films playing in one or two theaters each, the final answer will contain about six tuples. The total number of page fetches performed here is about 206 for reading the input relations (assuming that no indexes are available) and 95 for working with the intermediate relations. Additional page fetches might be required by the join operations performed.

Consider now the query q_2 whose query tree is illustrated in Fig. 6.1(b). It is easily verified that this is equivalent to q_1. Intuitively, q_2 was formed from q_1 by "pushing" selections and projections as far "down" the tree as possible; this generally reduces the size of intermediate results and thus of computing with them.

In this example, assuming that all (i.e., about 20) of Bergman's films are in *Movies*, the selection on *Movies* will yield about 100 tuples; when projected these will fit onto a single page. Joining with *Pariscope* will yield about six tuples, and the final join with *Location* will again yield six tuples. Thus only one page is needed to hold the intermediate results constructed during this evaluation, a considerable savings over the 95 pages needed by the previous one.

It is often beneficial to combine several algebraic operators into a single implemented operation. As a general rule of thumb, it is typical to materialize the inputs of each equi-join. The equi-join itself and all unary operations directly above it in the query tree are performed before output. The dashed ovals of Fig. 6.1(b) illustrate a natural grouping that can be used for this tree. In practical systems, the implementation and grouping of operators is typically considered in much finer detail.

The use of different query trees and, more generally, different evaluation plans can yield dramatically different costs in the evaluation of equivalent queries. Does this mean that the user will have to be extremely careful in expressing queries? The beauty of query optimization is that the answer is a resounding no. The user may choose any representation of a query, and the system will be responsible for generating several equivalent evaluation plans and choosing the least expensive one. For this reason, even though the relational algebra is conceptually procedural, it is implemented as an essentially declarative language.

In the case of the algebra, the generation of evaluation plans is typically based on the existence of rules for transforming algebraic expressions into equivalent ones. We have already seen rewrite rules in the context of transforming SPC and SPJR expressions into normal form (see Propositions 4.4.2 and 4.4.6). A different set of rules is useful in the present context due to the focus on optimizing the execution time and space requirements.

In Fig. 6.2 we present a family of representative rewrite rules (three with inverses) that can be used for performing the transformations needed for optimization at the logical level. In these rules we view cross-product as a special case of equi-join in which the selection formula is empty. Because of their similarity to the rules used for the normal form results, several of the rules are shown only in abstract form; detailed formulation of these, as well as verification of their soundness, is left for the reader (see Exercise 6.1). We also include the following rule:

Simplify-identities: replace $\pi_{1,\ldots,arity(q)}q$ by q; replace $\sigma_{i=i}q$ by q; replace $q \times \{\langle\rangle\}$ by q; replace $q \times \{\}$ by $\{\}$; and replace $q \bowtie_{1=1 \wedge \cdots \wedge arity(q)=arity(q)} q$ by q.

Generating and Choosing between Evaluation Plans

As suggested in Fig. 6.2, in most cases the transformations should be performed in a certain direction. For example, the fifth rule suggests that it is always desirable to push selections through joins. However, situations can arise in which pushing a selection through a join is in fact much more costly than performing it second (see Exercise 6.2). The broad variety of factors that influence the time needed to execute a given query evaluation plan make it virtually impossible to find an optimal one using purely analytic techniques. For this reason, modern optimizers typically adopt the following pragmatic strategy: (1) generate a possibly large number of alternative evaluation plans; (2) estimate the costs of executing

$$
\begin{array}{lcl}
\sigma_F(\sigma_{F'}(q)) & \leftrightarrow & \sigma_{F \wedge F'}(q) \\
\pi_{\bar{j}}(\pi_{\bar{k}}(q)) & \leftrightarrow & \pi_{\bar{l}}(q) \\
\sigma_F(\pi_{\bar{l}}(q)) & \leftrightarrow & \pi_{\bar{l}}(\sigma_{F'}(q)) \\
q_1 \bowtie q_2 & \leftrightarrow & q_2 \bowtie q_1 \\
\sigma_F(q_1 \bowtie_G q_2) & \rightarrow & \sigma_F(q_1) \bowtie_G q_2 \\
\sigma_F(q_1 \bowtie_G q_2) & \rightarrow & q_1 \bowtie_G \sigma_{F'}(q_2) \\
\sigma_F(q_1 \bowtie_G q_2) & \rightarrow & q_1 \bowtie_{G'} q_2 \\
\pi_{\bar{l}}(q_1 \bowtie_G q_2) & \rightarrow & \pi_{\bar{l}}(q_1) \bowtie_{G'} q_2 \\
\pi_{\bar{l}}(q_1 \bowtie_G q_2) & \rightarrow & q_1 \bowtie_{G'} \pi_{\bar{k}}(q_2)
\end{array}
$$

Figure 6.2: Rewriting rules for SPC algebra

them; and (3) select the one of lowest cost. The database system then executes the selected evaluation plan.

In early work, the transformation rules used and the method for evaluation plan generation were essentially intermixed. Motivated in part by the desire to make database systems extensible, more recent proposals have isolated the transformation rules from the algorithms for generating evaluation plans. This has the advantages of exposing the semantics of evaluation plan generation and making it easier to incorporate new kinds of information into the framework.

A representative system for generating evaluation plans was developed in connection with the Exodus database toolkit. In this system, techniques from AI are used and, a set of transformation rules is assumed. During processing, a set of partial evaluation plans is maintained along with a set of possible locations where rules can be applied. Heuristics are used to determine which transformation to apply next, so that an exhaustive search through all possible evaluation plans can be avoided while still having a good chance of finding an optimal or near-optimal evaluation plan. Several of the heuristics include weighting factors that can be tuned, either automatically or by the dba, to reflect experience gained while using the optimizer.

Early work on estimating the cost of evaluation plans was based essentially on "thought experiments" similar to those used earlier in this chapter. These analyses use factors including the size of relations, their expected statistical properties, selectivity factors of joins and selections, and existing indexes. In the context of large queries involving multiple joins, however, it is difficult if not impossible to predict the sizes of intermediate results based only on statistical properties. This provides one motivation for recent research on using random and background sampling to estimate the size of subquery answers, which can provide more reliable estimates of the overall cost of an evaluation plan.

Sideways Information Passing

We close this section by considering two practical approaches to implementing multiway joins as they arise in practical query languages.

Much of the early research on practical query optimization was performed in connection with the System R and INGRES systems. The basic building block of the query

languages used in these systems (SQL and Quel, respectively) takes the form of "select-from-where" clauses or blocks. For example, as detailed further in Chapter 7, Query (4.4) can be expressed in SQL as

select *Theater, Address*
from *Movies*, *Location*, *Pariscope*
where *Director* = "Bergman"
and *Movies.Title* = *Pariscope.Title*
and *Pariscope.Theater* = *Location.Theater*.

This can be translated into the algebra as a join between the three relations of the **from** part, using join condition given by the **where** and projecting onto the columns mentioned in the **select**. Thus a typical select-from-where block can be expressed by an SPC query as

$$\pi_{\bar{j}}(\sigma_F(R_1 \times \cdots \times R_n)).$$

With such expressions, the System R query optimizer pushes selections that affect a single relation into the join and then considers evaluation plans based on *left-to-right* joins that have the form

$$(\ldots(R_{i_1} \bowtie R_{i_2}) \bowtie \cdots \bowtie R_{i_n})$$

using different orderings $R_{i_1}, \ldots, R_{i_n}$. We now present a heuristic based on "sideways information passing," which is used in the System R optimizer for eliminating some possible orderings from consideration. Interestingly, this heuristic has also played an important role in developing evaluation techniques for recursive datalog queries, as discussed in Chapter 13.

To describe the heuristic, we rewrite the preceding SPC query as a (generalized) rule that has the form

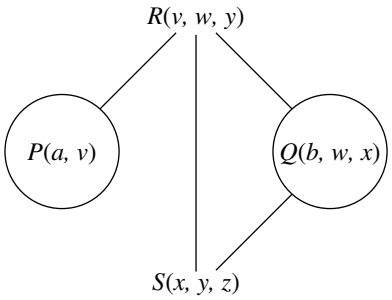
$$(*) \qquad ans(u) \leftarrow R_1(u_1), \ldots, R_n(u_n), C_1, \ldots, C_m,$$

where all equalities of the selection condition F are incorporated by using constants and equating variables in the free tuples $u_1, \ldots, u_n$, and the expressions $C_1, \ldots, C_m$ are conditions in the selection condition F not captured in that way. (This might include, e.g., inequalities and conditions based on order.) We shall call the $R_i(u_i)$'s *relation atoms* and the C_j's *constraint atoms*.

Example 6.1.1 Consider the rule

$$ans(z) \leftarrow P(a, v), Q(b, w, x), R(v, w, y), S(x, y, z), v \leq x,$$

where a, b denote constants. A common assumption in this case is that there are few values for v such that $P(a, v)$ is satisfied. This in turn suggests that there will be few triples (v, w, y) satisfying $P(a, v) \wedge R(v, w, y)$. Continuing by transitivity, then, we also expect there to be few 5-tuples (v, w, y, x, z) that satisfy the join of this with $S(x, y, z)$.



6-3.eps

Figure 6.3: A sip graph

More generally, the *sideways information passing graph*, or *sip graph*, of a rule ρ that has the form $(*)$ just shown has vertexes the set of relation atoms of a rule, and includes an undirected edge between atoms $R_i(u_i)$, $R_j(u_j)$ if u_i and u_j have at least one variable in common. Furthermore, each node with a constant appearing is specially marked. The sip graph for the rule of Example 6.1.1 is shown in Fig. 6.3.

Let us assume that the sip graph for a rule ρ is connected. In this case, a sideways information passing *strategy* (*sip strategy*) for ρ is an ordering $A_1, \ldots, A_n$ of the atoms in the rule, such that for each $j > 1$, either

(a) a constant occurs in A_j;

(b) A_j is a relational atom and there is at least one $i < j$ such that $\{A_i, A_j\}$ is an edge of the sip graph of (ρ); or

(c) A_j is a constraint atom and each variable occurring in A_j occurs in some atom A_i for $i < j$.

Example 6.1.2 A representative sample of the several sip strategies for the rule of Example 6.1.1 is as follows:

$$P(a, v), Q(b, w, x), v \leq x, R(v, w, y), S(x, y, z)$$
$$P(a, v), R(v, w, y), S(x, y, z), v \leq x, Q(b, w, x)$$
$$Q(b, w, x), R(v, w, y), P(a, v), S(x, y, z), v \leq x.$$

A sip strategy for the case in which the sip graph of rule ρ is not connected is a set of sip strategies, one for each connected component of the sip graph. (Incorporation of constraint atoms whose variables lie in distinct components is left for the reader.) The System R optimizer focuses primarily on joins that have connected sip graphs, and it considers only those join orderings that correspond to sip strategies. In some cases a more efficient evaluation plan can be obtained if an arbitrary tree of binary joins is permitted; see Exercise 6.5. While generating sip strategies the System R optimizer also considers

alternative implementations for the binary joins involved and records information about the orderings that the partial results would have if computed. An additional logical-level technique used in System R is illustrated in the following example.

EXAMPLE 6.1.3 Let us consider again the rule

$$ans(z) \leftarrow P(a, v), R(v, w, y), S(x, y, z), v \leq x, Q(b, w, x).$$

Suppose that a left-to-right join is performed according to the sip strategy shown. At different intermediate stages certain variables can be "forgotten," because they are not used in the answer, nor are they used in subsequent joins. In particular, after the third atom the variable y can be projected out, after the fourth atom v can be projected out, and after the fifth atom w and x can be projected out. It is straightforward to formulate a general policy for when to project out unneeded variables (see Exercise 6.4).

Query Decomposition: Join Detachment and Tuple Substitution

We now briefly discuss the two main techniques used in the original INGRES system for evaluating join expressions. Both are based on decomposing multiway joins into smaller ones.

While again focusing on SPC queries of the form

$$\pi_{\bar{j}}(\sigma_F(R_1 \times \cdots \times R_n))$$

for this discussion, we use a slightly different notation. In particular, tuple variables rather than domain variables are used. We consider expressions of the form

$$(**) \qquad ans(s) \leftarrow R_1(s_1), \ldots, R_n(s_n), C_1, \ldots, C_m, T,$$

where $s, s_1, \ldots, s_n$ are tuple variables; $C_1, \ldots, C_n$ are Boolean conditions referring to coordinates of the variables $s_1, \ldots, s_n$ (e.g., $s_1.3 = s_4.1 \vee s_2.4 = a$); and T is a *target condition* that gives a value for each coordinate of the target variable s. It is generally assumed that none of $C_1, \ldots, C_n$ has $\wedge$ as its parent connective.

A condition C_j is called *single variable* if it refers to only one of the variables s_i. At any point in the processing it is possible to apply one or more single-variable conditions to some R_i, thereby constructing an intermediate relation R_i' that can be used in place of R_i. In the INGRES optimizer, this is typically combined with other steps.

Join detachment is useful for separating a query into two separate queries, where the second refers to the first. Consider a query that has the specialized form

$$(\dagger) \qquad \begin{aligned} ans(t) \leftarrow\ & P_1(p_1), \ldots, P_m(p_m), C_1, \ldots, C_k, T, \\ & Q(q), \\ & R_1(r_1), \ldots, R_n(r_n), D_1, \ldots, D_l, \end{aligned}$$

where conditions $C_1, \ldots, C_k, T$ refer only to variables $t, p_1, \ldots, p_m, q$ and $D_1, \ldots, D_l$ refer only to $q, r_1, \ldots, r_n$. It is easily verified that this is equivalent to the sequence

$$temp(q) \leftarrow Q(q), R_1(r_1), \ldots, R_n(r_n), D_1, \ldots, D_l$$
$$ans(t) \leftarrow P_1(p_1), \ldots, P_m(p_m), temp(q), C_1, \ldots, C_k, T.$$

In this example, variable q acts as a "pivot" around which the detachment is performed. More general forms of join detachment can be developed in which a set of variables serves as the pivot (see Exercise 6.6).

Tuple substitution chooses one of the underlying relations R_j and breaks the n-variable join into a set of $(n-1)$-variable joins, one for each tuple in R_j. Consider again a query of form $(**)$ just shown. The tuple substitution of this on R_i is given by the "program"

for each r **in** R_i **do**
$$ans(s) \ +\!\leftarrow R_1(s_1), \ldots, R_{i-1}(s_{i-1}), R_{i+1}(s_{i+1}), \ldots, R_n(s_n), (C_1, \ldots, C_m, T)[s_i/r].$$

Here we use $+\!\leftarrow$ to indicate that *ans* is to accumulate the values stemming from all tuples r in (the value of) R_i; furthermore, r is substituted for s_i in all of the conditions.

There is an obvious trade-off here between reducing the number of variables in the join and the number of tuples in R_i. In the INGRES optimizer, each of the R_i's is considered as a candidate for forming the tuple substitution. During this process single-variable conditions may be applied to the R_i's to decrease their size.

6.2 Global Optimization

The techniques for creating evaluation plans presented in the previous section are essentially *local* in their operation: They focus on clusters of contiguous nodes in a query tree. In this section we develop an approach to the *global* optimization of conjunctive queries. This allows a transformation of an algebra query that removes several joins in a single step, a capability not provided by the techniques of the previous section. The global optimization technique is based on an elegant Homomorphism Theorem.

The Homomorphism Theorem

For two queries q_1, q_2 over the same schema $\mathbf{R}$, q_1 is *contained* in q_2, denoted $q_1 \subseteq q_2$, if for each $\mathbf{I}$ over $\mathbf{R}$, $q_1(\mathbf{I}) \subseteq q_2(\mathbf{I})$. Clearly, $q_1 \equiv q_2$ iff $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$. The Homomorphism Theorem provides a characterization for containment and equivalence of conjunctive queries.

We focus here on the tableau formalism for conjunctive queries, although the rule-based formalism could be used equally well. In addition, although the results hold for tableau queries over database schemas involving more than one relation, the examples presented focus on queries over a single relation.

Recall the notion of *valuation*—a mapping from variables to constants extended to be the identity on constants and generalized to free tuples and tableaux in the natural fashion.

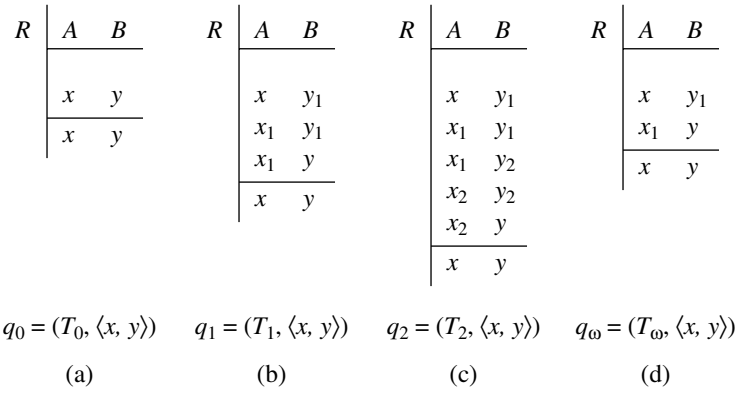


Figure 6.4: Tableau queries used to illustrate the Homomorphism Theorem

Valuations are used in the definition of the semantics of tableau queries. More generally, a *substitution* is a mapping from variables to variables and constants, which is extended to be the identity on constants and generalized to free tuples and tableaux in the natural fashion. As will be seen, substitutions play a central role in the Homomorphism Theorem.

We begin the discussion with two examples. The first presents several simple examples of the Homomorphism Theorem in action.

Example 6.2.1 Consider the four tableau queries shown in Fig. 6.4. By using the Homomorphism Theorem, it can be shown that $q_0 \subseteq q_1 \subseteq q_2 \subseteq q_\omega$.

To illustrate the flavor of the proof of the Homomorphism Theorem, we argue informally that $q_1 \subseteq q_2$. Note that there is substitution θ such that $\theta(T_2) \subseteq T_1$ and $\theta(\langle x, y\rangle) = \langle x, y\rangle$ [e.g., let $\theta(x_1) = \theta(x_2) = x_1$ and $\theta(y_1) = \theta(y_2) = y_1$]. Now suppose that I is an instance over AB and that $t \in q_1(I)$. Then there is a valuation ν such that $\nu(T_1) \subseteq I$ and $\nu(\langle x, y\rangle) = t$. It follows that $\theta \circ \nu$ is a valuation that embeds T_2 into I with $\theta \circ \nu(\langle x, y\rangle) = t$, whence $t \in q_2(I)$.

Intuitively, the existence of a substitution embedding the tableau of q_2 into the tableau of q_1 and mapping the summary of q_2 to the summary of q_1 implies that q_1 is *more restrictive* than q_2 (or more correctly, *no less restrictive* than q_2.) Surprisingly, the Homomorphism Theorem states that this is also a necessary condition for containment (i.e., if $q \subseteq q'$, then q is more restrictive than q' in this sense).

The second example illustrates a limitation of the techniques discussed in the previous section.

Example 6.2.2 Consider the two tableau queries shown in Fig. 6.5. It can be shown that $q \equiv q'$ but that q' cannot be obtained from q using the rewrite rules of the previous section (see Exercise 6.3) or the other optimization techniques presented there.

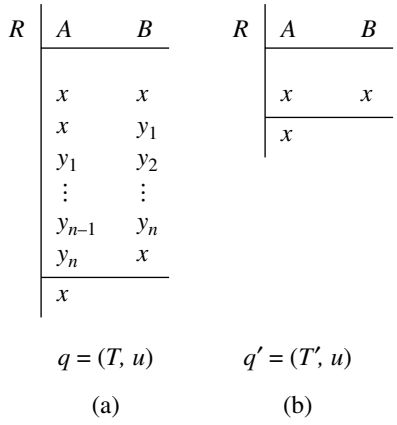


Figure 6.5: Pair of equivalent tableau queries

Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be two tableau queries over the same schema $\mathbf{R}$. A *homomorphism* from q' to q is a substitution θ such that $\theta(\mathbf{T}') \subseteq \mathbf{T}$ and $\theta(u') = u$.

THEOREM 6.2.3 (Homomorphism Theorem) Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be tableau queries over the same schema $\mathbf{R}$. Then $q \subseteq q'$ iff there exists a homomorphism from $(\mathbf{T}', u')$ to $(\mathbf{T}, u)$.

Proof Suppose first that there exists a homomorphism θ from q' to q. Let $\mathbf{I}$ be an instance over $\mathbf{R}$. To see that $q(\mathbf{I}) \subseteq q'(\mathbf{I})$, suppose that $w \in q(\mathbf{I})$. Then there is a valuation ν that embeds $\mathbf{T}$ into $\mathbf{I}$ such that $\nu(u) = w$. It is clear that $\theta \circ \nu$ embeds $\mathbf{T}'$ into $\mathbf{I}$ and $\theta \circ \nu(u') = w$, whence $w \in q'(\mathbf{I})$ as desired.

For the opposite inclusion, suppose that $q \subseteq q'$ [i.e., that $(\mathbf{T}, u) \subseteq (\mathbf{T}', u')$]. Speaking intuitively, we complete the proof by applying both q and q' to the "instance" $\mathbf{T}$. Because q will yield the free tuple u, q' also yields u (i.e., there is an embedding θ of $\mathbf{T}'$ into $\mathbf{T}$ that maps u' to u). To make this argument formal, we construct an instance $\mathbf{I}_\mathbf{T}$ that is isomorphic to $\mathbf{T}$.

Let V be the set of variables occurring in $\mathbf{T}$. For each $x \in V$, let a_x be a new distinct constant not occurring in $\mathbf{T}$ or $\mathbf{T}'$. Let μ be the valuation mapping each x to a_x, and let $\mathbf{I}_\mathbf{T} = \mu(\mathbf{T})$. Because μ is a bijection from V to $\mu(V)$, and because $\mu(V)$ has empty intersection with the constants occurring in $\mathbf{T}$, the inverse μ^{-1} of μ is well defined on $adom(\mathbf{I}_\mathbf{T})$.

It is clear that $\mu(u) \in q(\mathbf{I}_\mathbf{T})$, and so by assumption, $\mu(u) \in q'(\mathbf{I}_\mathbf{T})$. Thus there is a valuation ν that embeds $\mathbf{T}'$ into $\mathbf{I}_\mathbf{T}$ such that $\nu(u') = \mu(u)$. It is now easily verified that $\nu \circ \mu^{-1}$ is a homomorphism from q' to q. ■

Permitting a slight abuse of notation, we have the following (see Exercise 6.8).

COROLLARY 6.2.4 For tableau queries $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$, $q \subseteq q'$ iff $u \in q'(\mathbf{T})$.

We also have

COROLLARY 6.2.5 Tableau queries q, q' over schema $\mathbf{R}$ are equivalent iff there are homomorphisms from q to q' and from q' to q.

In particular, if $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ are equivalent, then u and u' are identical up to one-one renaming of variables.

Only one direction of the preceding characterization holds if the underlying domain is finite (see Exercise 6.12). In addition, the direct generalization of the theorem to tableau queries with equality does not hold (see Exercise 6.9).

Query Optimization by Tableau Minimization

Although the Homomorphism Theorem yields a decision procedure for containment and equivalence between conjunctive queries, it does not immediately provide a mechanism, given a query q, to find an "optimal" query equivalent to q. The theorem is now applied to obtain just such a mechanism.

We note first that there are simple algorithms for translating tableau queries into (satisfiable) SPC queries and vice versa. More specifically, given a tableau query, the corresponding generalized SPC query has the form $\pi_{\bar{j}}(\sigma_F(R_1 \times \cdots \times R_k))$, where each component R_i corresponds to a distinct row of the tableau. For the opposite direction, one algorithm for translating SPC queries into tableau queries is first to translate into the normal form for generalized SPC queries and then into a tableau query. A more direct approach that inductively builds tableau queries corresponding to subexpressions of an SPC query can also be developed (see Exercise 4.18). Analogous remarks apply to SPJR queries.

The goal of the optimization presented here is to minimize the number of rows in the tableau. Because the number of rows in a tableau query is one more than the number of joins in the SPC (SPJR) query corresponding to that tableau (see Exercise 4.18c), the tableau minimization procedure provides a way to minimize the number of joins in SPC and SPJR queries.

Surprisingly, we show that an optimal tableau query equivalent to tableau query q can be obtained simply by eliminating some rows from the tableau of q.

We say that a tableau query $(\mathbf{T}, u)$ is *minimal* if there is no query $(\mathbf{S}, v)$ equivalent to $(\mathbf{T}, u)$ with $|\mathbf{S}| < |\mathbf{T}|$ (i.e., where $\mathbf{S}$ has strictly fewer rows than $\mathbf{T}$).

We can now demonstrate the following.

THEOREM 6.2.6 Let $q = (\mathbf{T}, u)$ be a tableau query. Then there is a subset $\mathbf{T}'$ of $\mathbf{T}$ such that $q' = (\mathbf{T}', u)$ is a minimal tableau query and $q' \equiv q$.

Proof Let $(\mathbf{S}, v)$ be a minimal tableau that is equivalent to q. By Corollary 6.2.5, there are homomorphisms θ from q to $(\mathbf{S}, v)$ and λ from $(\mathbf{S}, v)$ to q. Let $\mathbf{T}' = \theta \circ \lambda(\mathbf{S})$. It is straightforward to verify that $(\mathbf{T}', u) \equiv q$ and $|\mathbf{T}'| \leq |\mathbf{S}|$. By minimality of $(\mathbf{S}, v)$, it follows that $|\mathbf{T}'| = |\mathbf{S}|$, and $(\mathbf{T}', u)$ is minimal. ■

Example 6.2.7 illustrates how one might minimize a tableau by hand.

R	A	B	C
u_1	x_2	y_1	z
u_2	x	y_1	z_1
u_3	x_1	y	z_1
u_4	x	y_2	z_2
u_5	x_2	y_2	z
u	x	y	z

Figure 6.6: The tableau (T, u)

Example 6.2.7 Let R be a relation schema of sort ABC and (T, u) the tableau over R in Fig. 6.6. To minimize (T, u), we wish to detect which rows of T can be eliminated. Consider u_1. Suppose there is a homomorphism θ from (T, u) onto itself that eliminates u_1 [i.e., $u_1 \notin \theta(T)$]. Because any homomorphism on (T, u) is the identity on u, $\theta(z) = z$. Thus $\theta(u_1)$ must be u_5. But then $\theta(y_1) = y_2$, and $\theta(u_2) \in \{u_4, u_5\}$. In particular, $\theta(z_1) \in \{z_2, z\}$. Because u_3 involves z_1, it follows that $\theta(u_3) \neq u_3$ and $\theta(y) \neq y$. But the last inequality is impossible because y is in u so $\theta(y) = y$. It follows that row u_1 cannot be eliminated and is in the minimal tableau. Similar arguments show that u_2 and u_3 cannot be eliminated. However, u_4 and u_5 can be eliminated using $\theta(y_2) = y_1, \theta(z_2) = z_1$ (and identity everywhere else). The preceding argument emphasizes the global nature of tableau minimization.

The preceding theorem suggests an improvement over the optimization strategies described in Section 6.1. Specifically, given a (satisfiable) conjunctive query q, the following steps can be used:

1. Translate q into a tableau query.
2. Minimize the number of rows in the tableau of this query.
3. Translate the result into a generalized SPC expression.
4. Apply the optimization techniques of Section 6.1.

As illustrated by Examples 6.2.2, 6.2.7, and 6.2.8, this approach has the advantage of performing global optimizations that typical query rewriting systems cannot achieve.

Example 6.2.8 Consider the relation schema R of sort ABC and the SPJR query q over R:

$$\pi_{AB}(\sigma_{B=5}(R)) \bowtie \pi_{BC}(\pi_{AB}(R) \bowtie \pi_{AC}(\sigma_{B=5}(R))).$$

R	A	B	C
	x	5	z_1
	x_1	5	z_2
	x_1	5	z
u	x	5	z

Figure 6.7: Tableau equivalent to q

The tableau (T, u) corresponding to it is that of Fig. 6.7. To minimize (T, u), we wish to find a homomorphism that "folds" T onto a subtableau with minimal number of rows. (If desired, this can be done in several stages, each of which eliminates one or more rows.) Note that the first row cannot be eliminated because every homomorphism is the identity on u and therefore on x. A similar observation holds for the third row. However, the second row can be eliminated using the homomorphism that maps z_2 to z and is the identity everywhere else. Thus the minimal tableau equivalent to (T, u) consists of the first and third rows of T. An SPJR query equivalent to the minimized tableau is

$$\pi_{AB}(\sigma_{B=5}(R)) \bowtie \pi_{BC}(\sigma_{B=5}(R)).$$

Thus the optimization procedure resulted in saving one join operation.

Before leaving minimal tableau queries, we present a result that describes a strong correspondence between equivalent minimal tableau queries. Two tableau queries $(\mathbf{T}, u)$, $(\mathbf{T}', u')$ are *isomorphic* if there is a one-one substitution θ that maps variables to variables such that $\theta((\mathbf{T}, u)) = (\mathbf{T}', u')$. In other words, (T, u) and (T', u') are the same up to renaming of variables. The proof of this result is left to the reader (see Exercise 6.11).

PROPOSITION 6.2.9 Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be minimal and equivalent. Then q and q' are isomorphic.

Complexity of Tableau Decision Problems

The following theorem shows that determining containment and equivalence between tableau queries is NP-complete and tableau query minimization is NP-hard.

THEOREM 6.2.10 The following problems, given tableau queries q, q', are NP-complete:

(a) Is $q \subseteq q'$?

(b) Is $q \equiv q'$?

(c) Suppose that the tableau of q is obtained by deleting free tuples of the tableau of q'. Is $q \equiv q'$ in this case?

These results remain true if q, q' are restricted to be single-relation typed tableau queries that have no constants.

Proof The proof is based on a reduction from the "exact cover" problem to the different tableau problems. The *exact cover* problem is to decide, given a set $X = \{x_1, \ldots, x_n\}$ and a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of X such that $\cup S = X$, whether there is an exact cover of X by $\mathcal{S}$ (i.e., a subset $\mathcal{S}'$ of $\mathcal{S}$ such that each member of X occurs in exactly one member of $\mathcal{S}'$). The exact cover problem is known to be NP-complete.

We now sketch a polynomial transformation from instances $\mathcal{E} = (X, \mathcal{S})$ of the exact cover problem to pairs $q_\mathcal{E}, q'_\mathcal{E}$ of typed tableau queries. This construction is then applied in various ways to obtain the NP-completeness results. The construction is illustrated in Fig. 6.8.

Let $\mathcal{E} = (X, \mathcal{S})$ be an instance of the exact cover problem, where $X = \{x_1, \ldots, x_n\}$ and $\mathcal{S} = \{S_1, \ldots, S_m\}$. Let $A_1, \ldots, A_n, B_1, \ldots, B_m$ be a listing of distinct attributes, and let R be chosen to have this set as its sort. Both $q_\mathcal{E}$ and $q'_\mathcal{E}$ are over relation R, and both queries have as summary $t = \langle A_1 : a_1, \ldots, A_n : a_n \rangle$, where $a_1, \ldots, a_n$ are distinct variables.

Let $b_1, \ldots, b_m$ be an additional set of m distinct variables. The tableau $T_\mathcal{E}$ of $q_\mathcal{E}$ has n tuples, each corresponding to a different element of X. The tuple for x_i has a_i for attribute A_i; b_j for attribute B_j for each j such that $x_i \in S_j$; and a new, distinct variable for all other attributes.

Let $c_1, \ldots, c_m$ be an additional set of m distinct variables. The tableau $T'_\mathcal{E}$ of $q'_\mathcal{E}$ has m tuples, each corresponding to a different element of $\mathcal{S}$. The tuple for S_j has a_i for attribute A_i for each i such that $x_i \in S_j$; $c_{j'}$ for attribute $B_{j'}$ for each j' such that $j' \neq j$; and a new, distinct variable for all other attributes.

To illustrate the construction, let $\mathcal{E} = (X, \mathcal{S})$ be an instance of the exact cover problem, where $X = \{x_1, x_2, x_3, x_4\}$ and $\mathcal{S} = \{S_1, S_2, S_3\}$ where

$$S_1 = \{x_1, x_3\}$$
$$S_2 = \{x_2, x_3, x_4\}$$
$$S_3 = \{x_2, x_4\}.$$

The tableau queries q_ξ and q'_ξ corresponding to $(X, \mathcal{S})$ are shown in Fig. 6.8. (Here the blank entries indicate distinct, new variables.) Note that $\xi = (X, \mathcal{S})$ is satisfiable, and $q'_\xi \subseteq q_\xi$.

More generally, it is straightforward to verify that for a given instance $\xi = (X, \mathcal{S})$ of the exact cover problem, X has an exact cover in $\mathcal{S}$ iff $q'_\xi \subseteq q_\xi$. Verification of this, and of parts (b) and (c) of the theorem, is left for Exercise 6.16. ■

A subclass of the typed tableau queries for which containment and equivalence is decidable in polynomial time is considered in Exercise 6.21.

Although an NP-completeness result often suggests intractability, this conclusion may not be warranted in connection with the aforementioned result. The complexity there is measured relative to the size of the *query* rather than in terms of the underlying stored

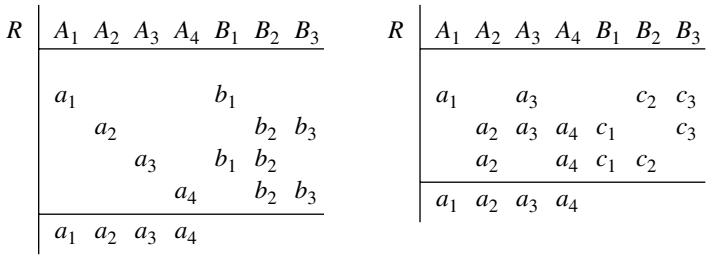


Figure 6.8: Tableau queries corresponding to an exact cover

data. Given an n-way join, the System R optimizer may potentially consider $n!$ evaluation strategies based on different orderings of the n relations; this may be exponential in the size of the query. In many cases, the search for a minimal tableau (or optimal left-to-right join) may be justified because the data is so much larger than the initial query. More generally, in Part D we shall examine both "data complexity" and "expression complexity," where the former focuses on complexity relative to the size of the data and the latter relative to the size of queries.

6.3 Static Analysis of the Relational Calculus

We now demonstrate that the decidability results for conjunctive queries demonstrated in the previous section do not hold when negation is incorporated (i.e., do not hold for the first-order queries). In particular, we present a general technique for proving the undecidability of problems involving static analysis of first-order queries and demonstrate the undecidability of three such problems.

We begin by focusing on the basic property of satisfiability. Recall that a query q is *satisfiable* if there is some input $\mathbf{I}$ such that $q(\mathbf{I})$ is nonempty. All conjunctive queries are satisfiable (Proposition 4.2.2), and if equality is incorporated then satisfiability is not guaranteed but it is decidable (Exercise 4.5). This no longer holds for the calculus.

To prove this result, we use a reduction of the Post Correspondence Problem (PCP) (see Chapter 2) to the satisfiability problem. The reduction is most easily described in terms of the calculus; of course, it can also be established using the algebras or nr-datalog$^\neg$.

At first glance, it would appear that the result follows trivially from the analogous result for first-order logic (i.e., the undecidability of satisfiability of first-order sentences). There is, however, an important difference. In conventional first-order logic (see Chapter 2), both finite and infinite interpretations are considered. Satisfiability of first-order sentences is co-recursively enumerable (co-r.e.) but not recursive. This follows from Gödel's Completeness Theorem. In contrast, in the context of first-order queries, only finite instances are considered legal. This brings us into the realm of finite model theory. As will

be shown, satisfiability of first-order queries is recursively enumerable (r.e.) but not recursive. (We shall revisit the contrast between conventional first-order logic and the database perspective, i.e., finite model theory, in Chapters 9 and 10.)

THEOREM 6.3.1 Satisfiability of relational calculus queries is r.e. but not recursive.

Proof To see that the problem is r.e., imagine a procedure that, when given query q over $\mathbf{R}$ as input, generates all instances $\mathbf{I}$ over $\mathbf{R}$ and tests $q(\mathbf{I}) = \emptyset$ until a nonempty answer is found.

To show that satisfiability is not recursive, we reduce the PCP to the satisfiability problem. In particular, we show that if there were an algorithm for solving satisfiability, then it could be used to construct an algorithm that solves the PCP.

Let $\mathcal{P} = (u_1, \ldots, u_n; v_1, \ldots, v_n)$ be an instance of the PCP (i.e., a pair of sequences of nonempty words over alphabet $\{0,1\}$). We describe now a (domain independent) calculus query $q_\mathcal{P} = \{\langle\rangle \mid \varphi_\mathcal{P}\}$ with the property that $q_\mathcal{P}$ is satisfiable iff $\mathcal{P}$ has a solution.

We shall use a relation schema $\mathbf{R}$ having relations *ENC(ODING)* with sort $[A, B, C, D, E]$ and *SYNCH(RONIZATION)* with sort $[F, G]$. The query $q_\mathcal{P}$ shall use constants $\{0, 1, \$, c_1, \ldots, c_n, d_1, \ldots, d_n\}$. (The use of multiple relations and constants is largely a convenience; the result can be demonstrated using a single ternary relation and no constants. See Exercise 6.19.)

To illustrate the construction of the algorithm, consider the following instance of the PCP:

$$u_1 = 011,\ u_2 = 011,\ u_3 = 0; \quad v_1 = 0,\ v_2 = 11,\ v_3 = 01100.$$

Note that $s = (1, 2, 3, 2)$ is a solution of this instance. That is,

$$u_1u_2u_3u_2 = 0110110011 = v_1v_2v_3v_2.$$

Figure 6.9 shows an input instance $\mathbf{I}_s$ over $\mathbf{R}$ which encodes this solution and satisfies the query $q_\mathcal{P}$ constructed shortly.

In the relation *ENC* of this figure, the first two columns form a *cycle*, so that the 10 tuples can be viewed as a sequence rather than a set. The third column holds a listing of the word $w = 0110110011$ that witnesses the solution to P; the fourth column describes which words of sequence $(u_1, \ldots, u_n)$ are used to obtain w; and the fifth column describes which words of sequence $(v_1, \ldots, v_n)$ are used. The relation *SYNCH* is used to synchronize the two representations of w by listing the pairs corresponding to the beginnings of new u-words and v-words.

The formula $\varphi_\mathcal{P}$ constructed now includes subformulas to test whether the various conditions just enumerated hold on an input instance. In particular,

$$\varphi = \varphi_{ENC\text{-}key} \wedge \varphi_{cycle} \wedge \varphi_{SYNCH\text{-}keys} \wedge \varphi_{u\text{-}encode} \wedge \varphi_{v\text{-}encode} \wedge \varphi_{u\text{-}v\text{-}synch},$$

where, speaking informally,

ENC	*A*	*B*	*C*	*D*	*E*
	\$	a_1	0	c_1	d_1
	a_1	a_2	1	c_1	d_2
	a_2	a_3	1	c_1	d_2
	a_3	a_4	0	c_2	d_3
	a_4	a_5	1	c_2	d_3
	a_5	a_6	1	c_2	d_3
	a_6	a_7	0	c_3	d_3
	a_7	a_8	0	c_2	d_3
	a_8	a_9	1	c_2	d_2
	a_9	\$	1	c_2	d_2

SYNCH	*F*	*G*
	\$	\$
	a_3	a_1
	a_6	a_3
	a_7	a_8

Figure 6.9: Encoding of a solution to PCP

$\varphi_{ENC\text{-}key}$*:* states that the first column of *ENC* is a *key*; that is, each value occurring in the *A* column occurs in exactly one tuple of *ENC*.

φ_{cycle}*:* states that constant \$ occurs in a cycle with length > 1 in the first two columns of *ENC*. (There may be other cycles, which can be ignored.)

$\varphi_{SYNCH\text{-}keys}$*:* states that both the first and second columns of *SYNCH* are keys.

$\varphi_{u\text{-}encode}$*:* states that for each value x occurring in the first column of *SYNCH*, if tuple $\langle x_1, y_1, z_1, c_i, d_{j_1}\rangle$ is in *ENC*, then there are at least $|u_i| - 1$ additional tuples in *ENC* "after" this tuple, all with value c_i in the fourth coordinate, and if these tuples are

$$\langle x_2, y_2, z_2, c_i, d_{j_2}\rangle, \ldots, \langle x_k, y_k, z_k, c_i, d_{j_k}\rangle$$

then $z_1 \ldots z_k = u_i$; none of $x_2, \ldots, x_k$ occurs in the first column of *SYNCH*; and if $y_k \neq \$$, then the *A* value "after" x_k occurs in the first column of *SYNCH*.

$\varphi_{v\text{-}encode}$*:* is analogous to $\varphi_{u\text{-}encode}$.

$\varphi_{u\text{-}v\text{-}synch}$*:* states that (1) $\langle \$, \$\rangle$ is in *SYNCH*; (2) if a tuple $\langle x, y\rangle$ is in *SYNCH*, then the associated u-word and v-word have the same index; and (3) if a tuple $\langle x, y\rangle$ is in *SYNCH*, and either x or y are not the "maximum" *A* value occurring in *F* or *G*, then there exists a tuple $\langle x', y'\rangle$ in *SYNCH*, where x' is the first *A* value "after" x occurring in *F* and y' is the first *A* value "after" y occurring in *G*. Finding the *A* values "after" x and y is done as in $\varphi_{u\text{-}encode}$.

The constructions of these formulas are relatively straightforward; we give two of them here and leave the others for the reader (see Exercise 6.19). In particular, we let

$$\psi(x, y) = \exists p, q, r\ ENC(x, y, p, q, r)$$

and set

$$\begin{aligned}\varphi_{cycle} = {} & \exists x(\psi(x, \$) \wedge \neg(x = \$)) \wedge \exists y(\psi(\$, y) \wedge \neg(y = \$)) \wedge \\ & \forall x((\exists y \psi(x, y)) \rightarrow (\exists z \psi(z, x))) \wedge \\ & \forall x((\exists y \psi(y, x)) \rightarrow (\exists z \psi(x, z))) \wedge \\ & \forall x, y_1, y_2(\psi(y_1, x) \wedge \psi(y_2, x) \rightarrow y_1 = y_2).\end{aligned}$$

If *ENC* satisfies $\varphi_{ENC-key} \wedge \varphi_{cycle}$, then the first two coordinates of *ENC* hold one or more disjoint cycles, exactly one of which contains the value \$.

Parts (1) and (2) of $\varphi_{u\text{-}v\text{-}synch}$ are realized by the formula

$$\begin{aligned}& SYNCH(\$, \$) \wedge \\ & \forall x, y(SYNCH(x, y) \rightarrow \\ & \quad \exists s, p, r, t, p', q((ENC(x, s, p, c_1, r) \wedge ENC(y, t, p', q, d_1)) \vee \\ & \qquad\qquad (ENC(x, s, p, c_2, r) \wedge ENC(y, t, p', q, d_2)) \vee \\ & \qquad\qquad \vdots \\ & \qquad\qquad (ENC(x, s, p, c_n, r) \wedge ENC(y, t, p', q, d_n)))).\end{aligned}$$

Verifying that the query $q_{\mathcal{P}}$ is satisfiable if and only if P has a solution is left to the reader (see Exercise 6.19). ■

The preceding theorem can be applied to derive other important undecidability results.

COROLLARY 6.3.2

(a) Equivalence and containment of relational calculus queries are co-r.e. and not recursive.

(b) Domain independence of a relational calculus query is co-r.e. and not recursive.

Proof It is easily verified that the two problems of part (a) and the problem of part (b) are co-r.e. (see Exercise 6.20). The proofs of undecidability are by reduction from the satisfiability problem. For equivalence, suppose that there were an algorithm for deciding equivalence between relational calculus queries. Then the satisfiability problem can be solved as follows: For each query $q = \{x_1, \ldots, x_n \mid \varphi\}$, this is unsatisfiable if and only if it is equivalent to the empty query $q^{\emptyset}$. This demonstrates that equivalence is not decidable. The undecidability of containment also follows from this.

For domain independence, let ψ be a sentence whose truth value depends on the underlying domain. Then $\{x_1, \ldots, x_n \mid \varphi \wedge \psi\}$ is domain independent if and only if φ is unsatisfiable. ■

The preceding techniques can also be used to show that "true" optimization cannot be performed for the first-order queries (see Exercise 6.20d).

6.4 Computing with Acyclic Joins

We now present a family of interesting theoretical results on the problem of computing the projection of a join. In the general case, if both the data set and the join expression are allowed to vary, then the time needed to evaluate such expressions appears to be exponential. The measure of complexity here is a combination of both "data" and "expression" complexity, and is somewhat non-standard; see Part D. Interestingly, there is a special class of joins, called *acyclic*, for which this evaluation is polynomial. A number of interesting properties of acyclic joins are also presented.

For this section we use the named perspective and focus exclusively on *flat project-join* queries of the form

$$q = \pi_X(R_1 \bowtie \cdots \bowtie R_n)$$

involving projection and natural join. For this discussion we assume that $\mathbf{R} = R_1, \ldots, R_n$ is a fixed database schema, and we use $\mathbf{I} = (I_1, \ldots, I_n)$ to refer to instances over it.

One of the historical motivations for studying this problem stems from the *pure universal relation assumption* (*pure URA*). An instance $\mathbf{I} = (I_1, \ldots, I_n)$ over schema $\mathbf{R}$ satisfies the pure URA if $\mathbf{I} = (\pi_{R_1}(I), \ldots, \pi_{R_n}(I))$ for some "universal" instance I over $\cup_{j=1}^{n} R_j$. If $\mathbf{I}$ satisfies the pure URA, then $\mathbf{I}$ can be stored, and queries against the corresponding instance I can be answered using joins of components in $\mathbf{I}$. The URA will be considered in more depth in Chapter 11.

Worst-Case Results

We begin with an example.

Example 6.4.1 Let $n > 0$ and consider the relations $R_i[A_i A_{i+1}]$, $i \in [1, n-1]$, as shown in Fig. 6.10(a). It is easily seen that the natural join of $R_1, \ldots, R_{n-1}$ is exponential in n and thus exponential in the size of the input query and data.

Now suppose that n is odd. Let R_n be as in Fig. 6.10(b), and consider the natural join of $R_1, \ldots, R_n$. This is empty. On the other hand, the join of any i of these for $i < n$ has size exponential in i. It follows that the algorithms of the System R and INGRES optimizers take time exponential in the size of the input and output to evaluate this query.

The following result implies that it is unlikely that there is an algorithm for computing projections of joins in time polynomial in the size of the query and the data.

Theorem 6.4.2 It is NP-complete to decide, given project-join expression q_0 over $\mathbf{R}$, instance $\mathbf{I}$ of $\mathbf{R}$, and tuple t, whether $t \in q_0(\mathbf{I})$. This remains true if q_0 and $\mathbf{I}$ are restricted so that $|q_0(\mathbf{I})| \leq 1$.

Proof The problem is easily seen to be in NP. For the converse, recall from Theorem 6.2.10(a) that the problem of tableau containment is NP-complete, even for single-

R_i

A_i	A_{i+1}
0	a
0	b
1	a
1	b
a	0
a	1
b	0
b	1

(a)

R_n

A_n	A_1
0	a
0	b
1	a
1	b
a	0
a	1
b	0
b	1

(b)

Figure 6.10: Relations to illustrate join sizes

relation typed tableaux having no constants. We reduce this to the current problem. Let $q = (T, u)$ and $q' = (T', u')$ be two typed constant-free tableau queries over the same relation schema. Recall from the Homomorphism Theorem that $q \subseteq q'$ iff there is a homomorphism of q' to q, which holds iff $u \in q'(T)$.

Assume that the sets of variables occurring in q and in q' are disjoint. Without loss of generality, we view each variable occurring in q to be a constant. For each variable x occurring in q', let A_x be a distinct attribute. For free tuple $v = (x_1, \ldots, x_n)$ in T', let I_v over $A_{x_1}, \ldots, A_{x_n}$ be a copy of T, where the i^{th} attribute is renamed to A_{x_i}. Letting $u' = \langle u'_1, \ldots, u'_m \rangle$, it is straightforward to verify that

$$q'(T) = \pi_{A_{u'_1}, \ldots, A_{u'_m}} (\bowtie\{I_v \mid v \in T'\}).$$

In particular, $u \in q'(T)$ iff u is in this projected join.

To see the last sentence of the theorem, let $u = \langle u_1, \ldots, u_m \rangle$ and use the query

$$\pi_{A_{u'_1}, \ldots, A_{u'_m}} (\bowtie\{I_v \mid v \in T'\} \bowtie \{\langle A_{u'_1} : u_1, \ldots, A_{u'_m} : u_m \rangle\}). \blacksquare$$

Theorem 6.2.10(a) considers complexity relative to the size of queries. As applied in the foregoing result, however, the queries of Theorem 6.2.10(a) form the basis for constructing a database instance $\{I_v \mid v \in T'\}$. In contrast with the earlier theorem, the preceding result suggests that computing projections of joins is intractable relative to the size of the query, the stored data, and the output.

Acyclic Joins

In Example 6.4.1, we may ask what is the fundamental difference between $R_1 \bowtie \cdots \bowtie R_{n-1}$ and $R_1 \bowtie \cdots \bowtie R_n$? One answer is that the relation schemas of the latter join form a cycle, whereas the relation schemas of the former do not.

We now develop a formal notion of acyclicity for joins and four properties equivalent

to it. All of these are expressed most naturally in the context of the named perspective for the relational model. In addition, the notion of acyclicity is sometimes applied to database schemas $\mathbf{R}=\{R_1, \ldots, R_n\}$ because of the natural correspondence between the schema $\mathbf{R}$ and the join $R_1 \bowtie \cdots \bowtie R_n$.

We begin by describing four interesting properties that are equivalent to acyclicity. Let $\mathbf{R} = \{R_1, \ldots, R_n\}$ be a database schema, where each relation schema has a different sort. An instance $\mathbf{I}$ of $\mathbf{R}$ is said to be *pairwise consistent* if for each pair $j, k \in [1, n]$, $\pi_{R_j}(I_j \bowtie I_k) = I_j$. Intuitively, this means that no tuple of I_j is "dangling" or "lost" after joining with I_k. Instance $\mathbf{I}$ is *globally consistent* if for each $j \in [1, n]$, $\pi_{R_j}(\bowtie\mathbf{I}) = I_j$ (i.e., no tuple of I_j is dangling relative to the full join). Pairwise consistency can be checked in PTIME, but checking global consistency is NP-complete (Exercise 6.25). The first property that is equivalent to acyclicity is:

Property (1): Each instance $\mathbf{I}$ that is pairwise consistent is globally consistent.

Note that the instance for schema $\{R_1, \ldots, R_{n-1}\}$ of Example 6.4.1 is both pairwise and globally consistent, whereas the instance for $\{R_1, \ldots, R_n\}$ is pairwise but not globally consistent.

The second property we consider is motivated by query processing in a distributed environment. Suppose that each relation of $\mathbf{I}$ is stored at a different site, that the join $\bowtie\mathbf{I}$ is to be computed, and that communication costs are to be minimized. A very naive algorithm to compute the join is to send each of the I_j to a specific site and then form the join. In the general case this may cause the shipment of many unneeded tuples because they are dangling in the full join.

The *semi-join* operator can be used to alleviate this problem. Given instances I, J over R, S, then semi-join of I and J is

$$I \ltimes J = \pi_R(I \bowtie J).$$

It is easily verified that $I \bowtie J = (I \ltimes J) \bowtie J = (J \ltimes I) \bowtie I$. Furthermore there are many cases in which computing the join in one of these ways can reduce data transmission costs if I and J are at different nodes of a distributed database (see Exercise 6.24).

Suppose now that $\mathbf{R}$ satisfies Property (1). Given an instance $\mathbf{I}$ distributed across the network, one can imagine replacing each relation I_j by its semi-join with other relations of $\mathbf{I}$. If done cleverly, this might be done with communication cost polynomial in the size of $\mathbf{I}$, with the result of the replacements satisfying pairwise consistency. Given Property (1), all relations can now be shipped to a common site, safe in the knowledge that no dangling tuples have been shipped.

More generally, a *semi-join program* for $\mathbf{R}$ is a sequence of commands

$$\begin{aligned} R_{i_1} &:= R_{i_1} \ltimes R_{j_1}; \\ R_{i_2} &:= R_{i_2} \ltimes R_{j_2}; \\ &\vdots \\ R_{i_p} &:= R_{i_p} \ltimes R_{j_p}; \end{aligned}$$

R_1

A	B	C
0	3	2
0	1	2
3	1	2
1	1	3

R_2

B	C	D	E
3	2	1	0
1	2	3	0
1	3	1	0

R_3

B	C	D	G
3	2	1	4
1	2	3	2
1	3	1	0
1	3	1	1

R_4

C	D	E	F
2	1	1	4
2	3	0	1
3	1	0	2
3	1	0	3

Figure 6.11: Instance for Example 6.4.3

(In practice, the original values of R_{i_j} would not be overwritten; rather, a scratch copy would be made.) This is a *full reducer* for $\mathbf{R}$ if for each instance $\mathbf{I}$ over $\mathbf{R}$, applying this program yields an instance $\mathbf{I}'$ that is globally consistent.

Example 6.4.3 Let $\mathbf{R} = \{ABC, BCDE, BCDG, CDEF\} = \{R_1, R_2, R_3, R_4\}$ and consider the instance $\mathbf{I}$ of $\mathbf{R}$ shown in Fig. 6.11. $\mathbf{I}$ is not globally consistent; nor is it pairwise consistent.

A full reducer for this schema is

$$
\begin{aligned}
R_2 &:= R_2 \ltimes R_1;\\
R_2 &:= R_2 \ltimes R_4;\\
R_3 &:= R_3 \ltimes R_2;\\
R_2 &:= R_2 \ltimes R_3;\\
R_4 &:= R_4 \ltimes R_2;\\
R_1 &:= R_1 \ltimes R_2;
\end{aligned}
$$

Note that application of this program to $\mathbf{I}$ has the effect of removing the first tuple from each relation.

We can now state the second property:

Property (2): $\mathbf{R}$ has a full reducer.

It can be shown that the schema $\{R_1, \ldots, R_{n-1}\}$ of Example 6.4.1 has a full reducer, but $\{R_1, \ldots, R_n\}$ does not (see Exercise 6.26).

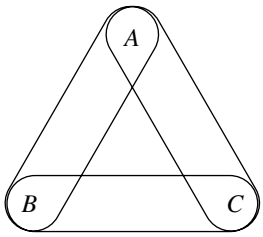
The next property provides a way to view a schema as a tree with certain properties. A *join tree* of a schema $\mathbf{R}$ is an undirected tree $T = (\mathbf{R}, E)$ such that

(i) each edge (R, R') is labeled by the set of attributes $R \cap R'$; and

(ii) for every pair R, R' of distinct nodes, for each $A \in R \cap R'$, each edge along the unique path between R and R' includes label A.

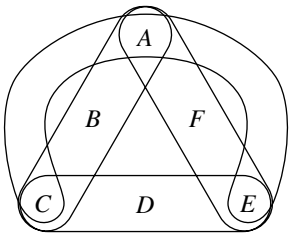
Property (3): $\mathbf{R}$ has a join tree.

For example, two join trees of the schema $\mathbf{R}$ of Figure 6.11 are $T_1 = (\mathbf{R}, \{(R_1, R_2), (R_2, R_3), (R_2, R_4)\})$ and $T_2 = (\mathbf{R}, \{(R_1, R_3), (R_3, R_2), (R_2, R_4)\})$. (The edge labels are not shown.)



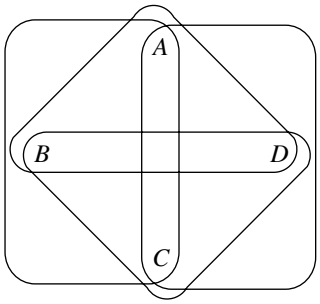
$R_1[AB]$, $R_2[BC]$, $R_3[AC]$

(a)



$S_1[ABC]$, $S_2[CDE]$, $S_3[AFE]$, $S_4[ACE]$

(b)



$T_1[ABC]$, $T_2[BCD]$, $T_3[ABD]$, $T_4[ACD]$

(c)

Figure 6.12: Three schemas and their hypergraphs

The fourth property we consider focuses entirely on the database schema **R** and is based on a simple algorithm, called the *GYO algorithm*.[1] This is most easily described in terms of the hypergraph corresponding to **R**. A *hypergraph* is a pair $\mathcal{F} = (V, F)$, where V is a set of vertexes and F is family of distinct nonempty subsets of V, called *edges* (or *hyperedges*). The *hypergraph* of schema **R** is the pair $(U, \mathbf{R})$, where $U = \cup\mathbf{R}$. In what follows, we often refer to a database schema **R** as a hypergraph. Three schemas and their hypergraphs are shown in Fig. 6.12.

A hypergraph is *reduced* if there is no pair f, f' of distinct edges with f a proper subset of f'. The *reduction* of $\mathcal{F} = (V, F)$ is $(V, F - \{f \in F \mid \exists f' \in F \text{ with } f \subset f'\})$. Suppose that **R** is a schema and **I** over **R** satisfies the pure URA. If $R_j \subset R_k$, then $I_j =$

[1] This is so named in honor of M. Graham and the team C. T. Yu and M. Z. Ozsoyoglu, who independently came to essentially this algorithm.

$\pi_{R_j}(I_k)$, and thus I_j holds redundant information. It is thus natural in this context to assume that **R**, viewed as a hypergraph, is reduced.

An *ear* of hypergraph $\mathcal{F} = (V, F)$ is an edge $f \in F$ such that for some distinct $f' \in F$, no vertex of $f - f'$ is in any other edge or, equivalently, such that $f \cap (\cup(F - \{f\})) \subseteq f'$. In this case, f' is called a *witness* that f is an ear. As a special case, if there is an edge f of $\mathcal{F}$ that intersects no other edge, then f is also considered an ear.

For example, in the hypergraph of Fig. 6.12(b), edge ABC is an ear, with witness ACE. On the other hand, the hypergraph of Fig. 6.12(a) has no ears.

We now have

ALGORITHM 6.4.4 (GYO Algorithm)

Input: Hypergraph $\mathcal{F} = (V, F)$

Output: A hypergraph involving a subset of edges of $\mathcal{F}$

Do until $\mathcal{F}$ has no ears:

1. Nondeterministically choose an ear f of $\mathcal{F}$.
2. Set $\mathcal{F} := (V', F - \{f\})$, where $V' = \cup(F - \{f\})$. ■

The output of the GYO algorithm is always reduced.

A hypergraph is *empty* if it is $(\emptyset, \emptyset)$. In Fig. 6.12, it is easily verified that the output of the GYO algorithm is empty for part (b), but that parts (a) and (c) have no ears and so equal their output under the algorithm. The output of the GYO algorithm is independent of the order of steps taken (see Exercise 6.28).

We now state the following:

Property (4): The output of the GYO algorithm on **R** is empty.

Speaking informally, Example 6.4.1 suggests that an absence of cycles yields Properties (1) to (4), whereas the presence of a cycle makes these properties fail. This led researchers in the late 1970s to search for a notion of acyclicity for hypergraphs that both generalized the usual notion of acyclicity for conventional undirected graphs and was equivalent to one or more of the aforementioned properties. For example, the conventional notion of hypergraph acyclicity from graph theory is due to C. Berge; but it turns out that this condition is necessary but not sufficient for the four properties (see Exercise 6.32).

We now define the notion of acyclicity that was found to be equivalent to the four aforementioned properties. Let $\mathcal{F} = (V, F)$ be a hypergraph. A *path* in $\mathcal{F}$ from vertex v to vertex v' is a sequence of $k \geq 1$ edges $f_1, \ldots, f_k$ such that

(i) $v \in f_1$;

(ii) $v' \in f_k$;

(iii) $f_i \cap f_{i+1} \neq \emptyset$ for $i \in [1, k-1]$.

Two vertexes are *connected* in $\mathcal{F}$ if there is a path between them. The notions of *connected* pair of edges, *connected component*, and *connected hypergraph* are now defined in the usual manner.

Now let $\mathcal{F} = (V, F)$ be a hypergraph, and $U \subseteq V$. The *restriction* of $\mathcal{F}$ to U, denoted $\mathcal{F}|_U$, is the result of forming the reduction of $(U, \{f \cap U \mid f \in F\} - \{\emptyset\})$.

Let $\mathcal{F} = (V, F)$ be a reduced hypergraph, let f, f' be distinct edges, and let $g = f \cap f'$. Then g is an *articulation set* of $\mathcal{F}$ if the number of connected components of $\mathcal{F}|_{V-g}$ is greater than the number of connected components of $\mathcal{F}$. (This generalizes the notion of articulation point for ordinary graphs.)

Finally, a reduced hypergraph $\mathcal{F} = (V, F)$ is *acyclic* if for each $U \subseteq V$, if $\mathcal{F}|_U$ is connected and has more than one edge then it has an articulation set; it is *cyclic* otherwise. A hypergraph is *acyclic* if its reduction is.

Note that if $\mathcal{F} = (V, F)$ is an acyclic hypergraph, then so is $\mathcal{F}|_U$ for each $U \subseteq V$.

Property (5): The hypergraph corresponding to **R** is acyclic.

We now present the theorem stating the equivalence of these five properties. Additional equivalent properties are presented in Exercise 6.31 and in Chapter 8, where the relationship of acyclicity with dependencies is explored.

THEOREM 6.4.5 Properties (1) through (5) are equivalent.

Proof We sketch here arguments that (4) $\Rightarrow$ (2) $\Rightarrow$ (1) $\Rightarrow$ (5) $\Rightarrow$ (4). The equivalence of (3) and (4) is left as Exercise 6.30(a).

We assume in this proof that the hypergraphs considered are connected; generalization to the disconnected case is straightforward.

(4) $\Rightarrow$ (2): Suppose now that the output of the GYO algorithm on $\mathbf{R} = \{R_1, \ldots, R_n\}$ is empty. Let $S_1, \ldots, S_n$ be an ordering of **R** corresponding to a sequence of ear removals stemming from an execution of the GYO algorithm, and let T_i be a witness for S_i for $i \in [1, n-1]$. An induction on n ("from the inside out") shows that the following is a full reducer (see Exercise 6.30a):

$$
\begin{aligned}
T_1 &:= T_1 \ltimes S_1;\\
T_2 &:= T_2 \ltimes S_2;\\
&\vdots\\
T_{n-1} &:= T_{n-1} \ltimes S_{n-1};\\
S_{n-1} &:= S_{n-1} \ltimes T_{n-1};\\
&\vdots\\
S_2 &:= S_2 \ltimes T_2;\\
S_1 &:= S_1 \ltimes T_1;
\end{aligned}
$$

(2) $\Rightarrow$ (1): Suppose that **R** has a full reducer, and let **I** be a pairwise consistent instance of **R**. Application of the full reducer to **I** yields an instance $\mathbf{I}'$ that is globally consistent. But by pairwise consistency, each step of the full reducer leaves **I** unchanged. It follows that $\mathbf{I} = \mathbf{I}'$ is globally consistent.

(1) $\Rightarrow$ (5): This is proved by contradiction. Suppose that there is a hypergraph that satisfies Property (1) but violates the definition of *acyclic*. Let $\mathbf{R} = \{R_1, \ldots, R_n\}$ be such a hypergraph where n is minimal among such hypergraphs and where the size of $U = \cup\mathbf{R}$ is minimal among such hypergraphs with n edges.

I	A_1	A_2	...	A_p	B_1	...	B_q
	1	0	...	0	1	...	1
	0	1	...	0	2	...	2
	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	0	0	...	1	p	...	p

Figure 6.13: Instance for proof of Theorem 6.4.5

It follows easily from the minimality conditions that $\mathbf{R}$ is reduced. In addition, by minimality no vertex (attribute) in U is in only one edge (relation schema).

Consider now the schema $\mathbf{R}' = \{R_2 - R_1, \ldots, R_n - R_1\}$. Two cases arise:

Case 1: $\mathbf{R}'$ is connected. Suppose that $R_1 = \{A_1, \ldots, A_p\}$ and $U - R_1 = \{B_1 \ldots, B_q\}$. Consider the instance I over U shown in Fig. 6.13. Define $\mathbf{I} = \{I_1, \ldots, I_n\}$ so that

$$I_j = \pi_{R_j}(I) \text{ for } j \in [2, n], \text{ and}$$
$$I_1 = \pi_{R_1}(I) \cup \{\langle 0, 0, \ldots, 0\rangle\}.$$

Using the facts that $\mathbf{R}'$ is connected and that each vertex of $\mathbf{R}$ occurs in at least two edges, it is straightforward to verify that $\mathbf{I}$ is pairwise consistent but not globally consistent, which is a contradiction (see Exercise 6.30b).

Case 2: $\mathbf{R}'$ is not connected. Choose a connected component of $\mathbf{R}'$ and let $\{S_1, \ldots, S_k\}$ be the set of edges of $\mathbf{R} - \{R_1\}$ involved in that connected component. Let $S = \cup_{i=1}^{k} S_i$ and let $R_1' = R_1 \cap S$. Two subcases arise:

Subcase 2.a: $R_1' \subseteq S_j$ for some $j \in [1, k]$. If this holds, then $R_1' \cap S_j$ is an articulation set for $\mathbf{R}$, which is a contradiction (see Exercise 6.30b).

Subcase 2.b: $R_1' \not\subseteq S_j$ for each $j \in [1, k]$. In this case $\mathbf{R}'' = \{S_1, \ldots, S_k, R_1'\}$ is a reduced hypergraph with fewer edges than $\mathbf{R}$. In addition, it can be verified that this hypergraph satisfies Property (1) (see Exercise 6.30b). By minimality of n, this implies that $\mathbf{R}''$ is acyclic. Because it is connected and has at least two edges, it has an articulation set. Two nested subcases arise:

Subcase 2.b.i: $S_i \cap S_j$ is an articulation pair for some i, j. We argue in this case that $S_i \cap S_j$ is an articulation pair for $\mathbf{R}$. To see this, let $x \in R_1' - (S_i \cap S_j)$ and let y be a vertex in some other component of $\mathbf{R}''|_{S-\{S_i \cap S_j\}}$. Suppose that $R_{i_1}, \ldots, R_{i_l}$ is a path in $\mathbf{R}$ from y to x. Let R_{i_p} be the first edge in this path that is not in $\{S_1, \ldots, S_k\}$. By the choice of $\{S_1, \ldots, S_k\}$, $R_{i_p} = R_1$. It follows that there is a path from y to x in $\mathbf{R}''|_{S-\{S_i \cap S_j\}}$, which is a contradiction. We conclude that $\mathbf{R}$ has an articulation pair, contradicting the initial assumption in this proof.

Subcase 2.b.ii: $R_1' \cap S_i$ is an articulation pair for some i. In this case $R_1 \cap S_i$ is an articulation pair for $\mathbf{R}$ (see Exercise 6.30b), again yielding a contradiction to the initial assumption of the proof.

(5) ⇒ (4): We first show inductively that each connected reduced acyclic hypergraph $\mathcal{F}$ with at least two edges has at least two ears. For the case in which $\mathcal{F}$ has two edges, this result is immediate. Suppose now that $\mathcal{F} = (V, F)$ is connected, reduced, and acyclic, with $|F| > 2$. Let $h = f \cap f'$ be an articulation set of $\mathcal{F}$. Let $\mathcal{G}$ be a connected component of $\mathcal{F}|_{V-h}$. By the inductive hypothesis, this has at least two ears. Let g be an ear of $\mathcal{G}$ that is different from $f - h$ and different from $f' - h$. Let g' be an edge of $\mathcal{F}$ such that $g = g' - h$. It is easily verified that g' is an ear of $\mathcal{F}$ (see Exercise 6.30b). Because $\mathcal{F}|_{V-h}$ has more than two connected components, it follows that $\mathcal{F}$ has at least two ears.

Finally, suppose that $\mathcal{F} = (V, F)$ is acyclic. If there is only one edge, then the GYO algorithm yields the empty hypergraph. Suppose that it has more than one edge. If $\mathcal{F}$ is not reduced, the GYO algorithm can be applied to reduce it. If $\mathcal{F}$ is reduced, then by the preceding argument $\mathcal{F}$ has an ear, say f. Then a step of the algorithm can be applied to yield $\mathcal{F}|_{\cup(\mathcal{F}-\{f\})}$. This is again acyclic. An easy induction now yields the result. ■

Recall from Theorem 6.4.2 that computing projections of arbitrary joins is probably intractable if both query and data size are considered. The following shows that this is not the case when the join is acyclic.

COROLLARY 6.4.6 If $\mathbf{R}$ is acyclic, then for each instance $\mathbf{I}$ over $\mathbf{R}$, the expression $\pi_X(\bowtie\mathbf{I})$ can be computed in time polynomial in the size of $\mathbf{IR}$, the input, and the output.

Proof Because the computation for each connected component of $\mathbf{R}$ can be performed separately, we assume without loss of generality that $\mathbf{R}$ is connected. Let $\mathbf{R}=(R_1, \ldots, R_n)$ and $\mathbf{I} = (I_1, \ldots, I_n)$. First apply a full reducer to $\mathbf{I}$ to obtain $\mathbf{I}' = (I'_1, \ldots, I'_n)$. This takes time polynomial in the size of the query and the input; the result is globally consistent; and $\bowtie\mathbf{II} = \bowtie\mathbf{II}'$.

Because $\mathbf{R}$ is acyclic, by Theorem 6.4.5 there is a join tree T for $\mathbf{R}$. Choose a root for T, say R_1. For each subtree T_k of T with root $R_k \neq R_1$, let $X_k = X \cap (\cup\{R \mid R \in T_k\})$, and $Z_k = R_k \cap$ (the parent of R_k). Let $J_k = I'_k$ for $k \in [1, n]$. Inductively remove nodes R_k and replace instances J_k from leaf to root of T as follows: Delete node R_k with parent R_m by replacing J_m with $J_m \bowtie \pi_{X_k Z_k} J_k$. A straightforward induction shows that immediately before nonleaf node R_k is deleted, then $J_k = \pi_{X_k R_k}(\bowtie_{R_l \in T_k} I'_l)$. It follows that at the end of this process the answer is $\pi_X J_1$ and that at each intermediate stage each instance J_k has size bounded by $|I'_k| \cdot |\pi_X(\bowtie\mathbf{II}_k)|$ (see Exercise 6.33). ■

Bibliographic Notes

An extensive discussion of issues in query optimization is presented in [Gra93]. Other references include [JK84a, KS91, Ull89b]. Query optimization for distributed databases is surveyed in [YC84]. Algorithms for binary joins are surveyed in [ME92].

The paper [SAC+79] describes query optimization in System/R, including a discussion of generating and analyzing multiple evaluation plans and a thorough discussion of accessing tuples from a single relation, as from a projection and selection. System/R is the precursor of IBM's DB2 database management system. The optimizer for INGRES introduces query decomposition, including both join detachment and tuple substitution [WY76, SWKH76].

The use of semi-joins in query optimization was first introduced in INGRES [WY76, SWKH76] and used for distributed databases in [BC81, BG81]. Research on optimizing buffer management policies includes [FNS91, INSS92, NCS91, Sto81]. Other system optimizers include those for Exodus [GD87], distributed INGRES [ESW78], SDD-1 [BGW+81], and the TI Open Object-Oriented Data Base [BMG93].

[B+88] presents the unifying perspective that physical query implementation can be viewed as generation, manipulation, and merging of streams of tuples and develops a very flexible toolkit for constructing dbms's. A formal model incorporating streams, sets, and parallelism is presented in [PSV92].

The recent work [IK90] focuses on finding optimal and near-optimal evaluation plans for n-way joins, where n is in the hundreds, using simulated annealing and other techniques. Perhaps most interesting about this work are characterizations of the space of evaluation plans (e.g., properties of evaluation plan cost in relation to natural metrics on this space).

Early research on generation and selection of query evaluation plans is found in [SAC+79, SWKH76]. Treatments that separate plan generation from transformation rules include [Fre87, GD87, Loh88]. More recent research has proposed mechanisms for generating *parameterized* evaluation plans; these can be generated at compile time but permit the incorporation of run-time information [GW89, INSS92]. An extensive listing of references to the literature on estimating the costs of evaluation plans is presented in [SLRD93]. This article introduces an estimation technique based on the computation of series functions that approximates the distribution of values and uses regression analysis to estimate the output sizes of select-join queries.

Many forward-chaining expert systems in AI also face the problem of evaluating what amounts to conjunctive queries. The most common technique for evaluating conjunctive queries in this context is based on a sequential generate-and-test algorithm. The paper [SG85] presents algorithms that yield optimal and near-optimal orderings under this approach to evaluation.

The technique of tableau query minimization was first developed in connection with database queries in [CM77], including the Homomorphism Theorem (Theorem 6.2.3) and Theorem 6.2.6. Theorem 6.2.10 is also due to [CM77]; the proofs sketched in the exercises are due to [SY80] and [ASU79b]. Refinements of this result (e.g., to subclasses of typed tableau queries) are presented in [ASU79b, ASU79a].

The notion of tableau homomorphism is a special case of the notion of *subsumption* used in resolution theorem proving [CL73]. That work focuses on clauses (i.e., disjunctions of positive and negative literals), and permits function symbols. A clause $C = (L_1 \vee \cdots \vee L_n)$ *subsumes* a clause $D = (M_1 \vee \cdots \vee M_k)$ if there is a substitution σ such that $C\sigma$ is a subclause of D. A generalized version of tableau minimization, called *condensation*, also arises in this connection. A condensation of a clause $C = (L_1 \vee \cdots \vee L_n)$ is a clause $C' = (L_{i_1} \vee \cdots \vee L_{i_m})$ with m minimal such that $C' = C\theta$ for some substitution θ. As observed in [Joy76], condensations are unique up to variable substitution.

Reference [SY80] studies restricted usage of difference with SPCU queries, for which several positive results can be obtained (e.g., decidability of containment; see Exercise 6.22).

The undecidability results for the relational calculus derive from results in [DiP69] (see also [Var81]). The assumption in this chapter that relations be finite is essential. For

instance, the test for containment is co-r.e. in our context whereas it is r.e. when possibly infinite structures are considered. (This is by reduction to the validity of a formula in first-order predicate logic with equality using the Gödel Completeness Theorem.

The complexity of query languages is studied in [CH82, Var82a] and is considered in Part E of this volume.

As discussed in Chapter 7, practical query languages typically produce *bags* (also called multisets; i.e., collections whose members may occur more than once). The problem of containment and equivalence of conjunctive queries under the bag semantics is considered in [CV93]. It remains open whether containment is decidable, but it is Π_2^p-hard. On the other hand, two conjunctive queries are equivalent under the bag semantics iff they are isomorphic.

Acyclic joins enjoyed a flurry of activity in the database research community in the late 1970s and early 1980s. As noted in [Mal86], the same concept has been studied in the field of statistics, beginning with [Goo70, Hab70]. An early motivation for their study in databases stemmed from distributed query processing; the notions of join tree and full reducers are from [BC81, BG81]; see also [GS82, GS84, SS86]. The original GYO algorithm was developed in [YO79] and [Gra79]; we use here a variant due to [FMU82]. The notion of globally consistent is studied in [BR80, HLY80, Ris82, Var82b]; see also [Hul83]. Example 6.4.1 is taken from [Ull89b]. The paper [BFM$^+$81] introduced the notion of acyclicity presented here and observed the equivalence to acyclicity of several previously studied properties, including those of having a full reducer and pairwise consistency implying global consistency; this work is reported in journal form in [BFMY83].

A linear-time test for acyclicity is developed in [TY84]. Theorem 6.4.2 and Corollary 6.4.6 are due to [Yan81].

The notion of Berge acyclic is due to [Ber76a]. [Fag83] investigates several notions of acyclicity, including the notion studied in this chapter and Berge acyclicity. Further investigation of these alternative notions of acyclicity is presented in [ADM85, DM86b, GR86]. Early attempts to develop a notion of acyclic that captured desirable database characteristics include [Zan76, Gra79].

The relationship of acyclicity with dependencies is considered in Chapter 8.

Many variations of the universal relation assumption arose in the late 1970s and early 1980s. We return to this topic in Chapter 11; surveys of these notions include [AP82, Ull82a, MRW86].

Exercises

Exercise 6.1

(a) Give detailed definitions for the rewrite rules proposed in Section 6.1. In other words, provide the conditions under which they preserve equivalence.

(b) Give the step-by-step description of how the query tree of Fig. 6.1(a) can be transformed into the query tree of Fig. 6.1(b) using these rewrite rules.

Exercise 6.2 Consider the transformation $\sigma_F(q_1 \bowtie_G q_2) \to \sigma_F(q_1) \bowtie_G q_2$ of Fig. 6.2. Describe a query q and database instance for which applying this transformation yields a query whose direct implementation is dramatically more expensive than that of q.

Exercise 6.3

(a) Write generalized SPC queries equivalent to the two tableau queries of Example 6.2.2.

(b) Show that the optimization of this example cannot be achieved using the rewrite rules or multiway join techniques of System/R or INGRES discussed in Section 6.1.

(c) Generate an example analogous to that of Example 6.2.2 that shows that even for typed tableau queries, the rewrite rules of Section 6.1 cannot achieve the optimizations of the Homomorphism Theorem.

Exercise 6.4 Present an algorithm that identifies when variables can be projected out during a left-to-right join of a sip strategy.

Exercise 6.5 Describe a generalization of sip strategies that permits evaluation of multiway joins according to an arbitrary binary tree rather than using only left-to-right join processing. Give an example in which this yields an evaluation plan more efficient than any left-to-right join.

Exercise 6.6 Consider query expressions that have the form (†) mentioned in the discussion of join detachment in Section 6.1.

(a) Describe how the possibility of applying join detachment depends on how equalities are expressed in the conditions (e.g., Is there a difference between using conditions '$x.1 = y.1$, $y.1 = z.1$' versus '$x.1 = z.1$, $z.1 = y.1$'?). Describe a technique for eliminating this dependence.

(b) Develop a generalization of join detachment in which a set of variables serves as the pivot.

Exercise 6.7 [WY76]

(a) Describe some heuristics for choosing the atom $R_i(s_i)$ for forming a tuple substitution. These may be in the context of using tuple substitution and join detachment for the resulting subqueries, or they may be in a more general context.

(b) Develop a query optimization algorithm based on applying single-variable conditions, join detachment, and tuple substitution.

Exercise 6.8 Prove Corollary 6.2.4.

Exercise 6.9

(a) State the direct generalization of Theorem 6.2.3 for tableau queries with equality, and show that it does not hold.

(b) State and prove a correct generalization of Theorem 6.2.3 that handles tableau queries with equality.

Exercise 6.10 For queries q, q', write $q \subset q'$ to denote that $q \subseteq q'$ and $q \not\equiv q'$. The meaning of $q \supset q'$ is defined analogously.

(a) Exhibit an infinite set $\{q_0, q_1, q_2, \ldots\}$ of typed tableau queries involving no constants over a single relation with the property that $q_0 \subset q_1 \subset q_2 \subset \ldots$.

(b) Exhibit an infinite set $\{q_0', q_1', q_2', \ldots\}$ of (possibly nontyped) tableau queries involving no constants over a single relation such that $q_i' \not\subseteq q_j'$ and $q_j' \not\subseteq q_i'$ for each pair $i \neq j$.

(c) Exhibit an infinite set $\{q_0'', q_1'', q_2'', \ldots\}$ of (possibly nontyped) tableau queries involving no constants over a single relation with the property that $q_0'' \supset q_1'' \supset q_2'' \supset \ldots$.

(d) Do parts (b) and (c) for typed tableau queries that may contain constants.

★(e) [FUMY83] Do parts (b) and (c) for typed tableau queries that contain no constants.

Exercise 6.11 [CM77] Prove Proposition 6.2.9.

Exercise 6.12

(a) Prove that if the underlying domain **dom** is finite, then only one direction of the statement of Theorem 6.2.3 holds.

(b) Let $n > 1$ be arbitrary. Exhibit a pair of tableau queries q, q' such that under the assumption that **dom** has n elements, $q \subseteq q'$, but there is no homomorphism from q' to q. In addition, do this using typed tableau queries.

(c) Show for arbitrary $n > 1$ that Theorem 6.2.6 and Proposition 6.2.9 do not hold if **dom** has n elements.

Exercise 6.13 Let R be a relation schema of sort ABC. For each of the following SPJR queries over R, construct an equivalent tableau (see Exercise 4.19), minimize the tableau, and construct from the minimized tableau an equivalent SPJR query with minimal number of joins.

(a) $\pi_{AC}[\pi_{AB}(R) \bowtie \pi_{BC}(R)] \bowtie \pi_A[\pi_{AC}(R) \bowtie \pi_{CB}(R)]$

(b) $\pi_{AC}[\pi_{AB}(R) \bowtie \pi_{BC}(R)] \bowtie \pi_{AB}(\sigma_{B=8}(R)) \bowtie \pi_{BC}(\sigma_{A=5}(R))$

(c) $\pi_{AB}(\sigma_{C=1}(R)) \bowtie \pi_{BC}(R) \bowtie \pi_{AB}[\sigma_{C=1}(\pi_{AC}(R)) \bowtie \pi_{CB}(R)]$

♠**Exercise 6.14** [SY80]

(a) Give a decision procedure for determining whether one union of tableaux query is contained in another one. *Hint:* Let the queries be $q = (\{\mathbf{T}_1, \ldots, \mathbf{T}_n\}, u)$ and $q' = (\{\mathbf{S}_1, \ldots, \mathbf{S}_m\}, v)$; and prove that $q \subseteq q'$ iff for each $i \in [1, n]$ there is some $j \in [1, m]$ such that $(\mathbf{T}_i, u) \subseteq (\mathbf{S}_j, v)$. (The case of queries equivalent to $q^{\emptyset}$ must be handled separately.)

A union of tableaux query $(\{\mathbf{T}_1, \ldots, \mathbf{T}_n\}, u)$ is *nonredundant* if there is no distinct pair i, j such that $(\mathbf{T}_i, u) \subseteq (\mathbf{T}_j, u)$.

(b) Prove that if $(\{\mathbf{T}_1, \ldots, \mathbf{T}_n\}, u)$ and $(\{\mathbf{S}_1, \ldots, \mathbf{S}_m\}, v)$ are nonredundant and equivalent, then $n = m$; for each $i \in [1, n]$ there is a $j \in [1, n]$ such that $(\mathbf{T}_i, u) \equiv (\mathbf{S}_j, v)$; and for each $j \in [1, n]$ there is a $i \in [1, n]$ such that $(\mathbf{S}_j, v) \equiv (\mathbf{T}_i, u)$.

(c) Prove that for each union of tableaux query q there is a unique (up to renaming) equivalent union of tableaux query that has a minimal total number of atoms.

Exercise 6.15 Exhibit a pair of typed restricted SPJ algebra queries q_1, q_2 over a relation R and having no constants, such that there is no conjunctive query equivalent to $q_1 \cup q_2$. *Hint:* Use tableau techniques.

♠ **Exercise 6.16** [SY80]

(a) Complete the proof of part (a) of Theorem 6.2.10.

(b) Prove parts (b) and (c) of that theorem. *Hint:* Given ξ and $q_\xi = (T_\xi, t)$ and $q'_\xi = (T'_\xi, t)$ as in the proof of part (a), set $q''_\xi = (T_\xi \cup T'_\xi, t)$. Show that ξ is satisfiable iff $q''_\xi \equiv q'_\xi$.

(c) Prove that it is NP-hard to determine, given a pair q, q' of typed tableau queries over the same relation schema, whether q is minimal and equivalent to q'. Conclude that optimizing conjunctive queries, in the sense of finding an equivalent with minimal number of atoms, is NP-hard.

Exercise 6.17 [ASU79b] Prove Theorem 6.2.10 using a reduction from 3-SAT (see Chapter 2) rather than from the exact cover problem.

Exercise 6.18 [ASU79b]

(a) Prove that determining containment between two typed SPJ queries of the form $\pi_X(\bowtie_{i=1}^n (\pi_{X_i} R))$ is NP-complete. *Hint:* Use Exercise 6.16.

(b) Prove that the problem of finding, given an SPJ query q of the form $\pi_X(\bowtie_{i=1}^n (\pi_{X_i} R))$, an SPJ query q' equivalent to q that has the minimal number of join operations among all such queries is NP-hard.

Exercise 6.19

(a) Complete the proof of Theorem 6.3.1.

(b) Describe how to modify that proof so that $q_\mathcal{P}$ uses no constants.

(c) Describe how to modify the proof so that no constants and only one ternary relation is used. *Hint:* Speaking intuitively, a tuple $t = \langle a_1, \ldots, a_5 \rangle$ of *ENC* can be simulated as a set of tuples $\{\langle b_t, b_1, a_1 \rangle, \ldots, \langle b_t, b_5, a_5 \rangle\}$, where b_t is a value not used elsewhere and $b_1, \ldots, b_5$ are values established to serve as integers $1, \ldots, 5$.

(d) Describe how, given instance $\mathcal{P}$ of the PCP, to construct an nr-datalog$^\neg$ program that is satisfiable iff $\mathcal{P}$ has a solution.

Exercise 6.20 This exercise develops further undecidability results for the relational calculus.

(a) Prove that containment and equivalence of range-safe calculus queries are co-r.e.

(b) Prove that domain independence of calculus queries is co-r.e. *Hint:* Theorem 5.6.1 is useful here.

(c) Prove that containment of safe-range calculus queries is undecidable.

(d) Show that there is no algorithm that always halts and on input calculus query q gives an equivalent query q' of minimum length. Conclude that "complete" optimization of the relational calculus is impossible. *Hint:* If there were such an algorithm, then it would map each unsatisfiable query to a query with formula (of form) $\neg(a = b)$.

♠ **Exercise 6.21** [ASU79a, ASU79b] In a typed tableau query (T, u), a *summary variable* is a variable occurring in u. A *repeated nonsummary variable* for attribute A is a nonsummary variable in $\pi_A(T)$ that occurs more than once in T. A typed tableau query is *simple* if for each attribute A, there is a repeated nonsummary variable in $\pi_A(T)$, then no other constant or variable in $\pi_A(T)$ occurs more than once $\pi_A(T)$. Many natural typed restricted SPJ queries translate into simple tableau queries.

(a) Show that the tableau query over $R[ABCD]$ corresponding to

$$\pi_{AC}(\pi_{AB}(R) \bowtie \pi_{BC}(R)) \bowtie (\pi_{AB}(R) \bowtie \pi_{BD}(R))$$

is *not* simple.

(b) Exhibit a simple tableau query that is not the result of transforming a typed restricted SPJ query under the algorithm of Exercise 4.19.

(c) Prove that if (T, u) is simple, $T' \subseteq T$, and (T', u) is a tableau query, then (T', u) is simple.

(d) Develop an $O(n^4)$ algorithm that, on input a simple tableau query q, produces a minimal tableau query equivalent to q.

(e) Develop an $O(n^3)$ algorithm that, given simple tableau queries q, q', determines whether $q \equiv q'$.

(f) Prove that testing containment for simple tableau queries is NP-complete.

♠ **Exercise 6.22** [SY80] Characterize containment and equivalence between queries of the form $q_1 - q_2$, where q_1, q_2 are SPCU queries. *Hint:* First develop characterizations for the case in which q_1, q_2 are SPC queries.

Exercise 6.23 Recall from Exercise 5.9 that an arbitrary nonrecursive datalog$^\neg$ rule can be described as a difference $q_1 - q_2$, where q_1 is an SPC query and q_2 is an SPCU query.

(a) Show that Exercise 5.9 cannot be strengthened so that q_2 is an SPC query.

(b) Show that containment between pairs of nonrecursive datalog$^\neg$ rules is decidable. *Hint:* Use Exercise 6.22.

(c) Recall that for each nr-datalog program P with a single-relation target there is an equivalent nr-datalog program P' such that all rule heads have the same relation name (see Exercise 4.24). Prove that the analogous result does not hold for nr-datalog$^\neg$ programs.

Exercise 6.24

(a) Verify that $I \bowtie J = (I \ltimes J) \bowtie J$.

(b) Analyze the transmission costs incurred by the left-hand and right-hand sides of this equation, and describe conditions under which one is more efficient than the other.

Exercise 6.25 [HLY80] Prove that the problem of deciding, given instance $\mathbf{I}$ of database schema $\mathbf{R}$, whether $\mathbf{I}$ is globally consistent is NP-complete.

Exercise 6.26 Prove the following without using Theorem 6.4.5.

(a) The database schema $\mathbf{R} = \{AB, BC, CA\}$ has no full reducer.

(b) For arbitrary $n > 1$, the schema $\{R_1, \ldots, R_{n-1}\}$ of Example 6.4.1 has a full reducer.

(c) For arbitrary (odd or even) $n > 1$, the schema $\{R_1, \ldots, R_n\}$ of Example 6.4.1 has no full reducer.

Exercise 6.27

(a) Draw the hypergraph of the schema of Example 6.4.3.

(b) Draw the hypergraph of Fig. 6.12(b) in a fashion that suggests it to be acyclic.

Exercise 6.28 Prove that the output of Algorithm 6.4.4 is independent of the nondeterministic choices.

Exercise 6.29 As originally introduced, the GYO algorithm involved the following steps:

Nondeterministically perform either step,
until neither can be applied

1. If $v \in V$ is in exactly one edge $f \in F$
   then $\mathcal{F} := (V - \{v\}, (F - \{f\} \cup \{f - \{v\}\}) - \{\emptyset\})$.
2. If $f \subseteq f'$ for distinct $f, f' \in F$,
   then $\mathcal{F} := (V, F - \{f\})$.

The result of applying the original GYO algorithm to a schema $\mathbf{R}$ is the *GYO reduction* of $\mathbf{R}$.

(a) Prove that the original GYO algorithm yields the same output independent of the nondeterministic choices.

(b) [FMU82] Prove that Algorithm 6.4.4 given in the text yields the empty hypergraph on $\mathbf{R}$ iff the GYO reduction of $\mathbf{R}$ is the empty hypergraph.

Exercise 6.30 This exercise completes the proof of Theorem 6.4.5.

(a) [BG81] Prove that (3) $\Leftrightarrow$ (4).

(b) Complete the other parts of the proof.

Exercise 6.31 [BFMY83] $\mathbf{R}$ has the *running intersection property* if there is an ordering $R_1, \ldots, R_n$ of $\mathbf{R}$ such that for $2 \leq i \leq n$ there exists $j_i < i$ such that $R_i \cap (R_1 \cup \cdots \cup R_{i-1}) \subseteq R_{j_i}$. In other words, the intersection of each R_i with the union of the previous R_j's is contained in one of these. Prove that $\mathbf{R}$ has the running intersection property iff $\mathbf{R}$ is acyclic.

Exercise 6.32 [BFMY83] A *Berge cycle* in a hypergraph $\mathcal{F}$ is a sequence $(f_1, v_1, f_2, v_2, \ldots, f_n, v_n, f_{n+1})$ such that

(i) $v_1, \ldots, v_n$ are distinct vertexes of $\mathcal{F}$;

(ii) $f_1, \ldots, f_n$ are distinct edges of $\mathcal{F}$, and $f_{n+1} = f_1$;

(iii) $n \geq 2$; and

(iv) $v_i \in f_i \cap f_{i+1}$ for $i \in [1, n]$.

A hypergraph is *Berge cyclic* if it has a Berge cycle, and it is *Berge acyclic* otherwise.

(a) Prove that Berge acyclicity is necessary but not sufficient for acyclicity.

(b) Show that any hypergraph in which two edges have two nodes in common is Berge cyclic.

Exercise 6.33 [Yan81] Complete the proof of Corollary 6.4.6.

7 Notes on Practical Languages

Alice: *What do you mean by practical languages?*
Riccardo: **select from where**.
Alice: *That's it?*
Vittorio: *Well, there are of course lots of bells and whistles.*
Sergio: *But basically, this forms the core of most practical languages.*

In this chapter we discuss the relationship of the abstract query languages discussed so far to three representative commercial relational query languages: Structured Query Language (SQL), Query-By-Example (QBE), and Microsoft Access. SQL is by far the dominant relational query language and provides the basis for languages in extensions of the relational model as well. Although QBE is less widespread, it illustrates nicely the basic capabilities and problems of graphic query languages. Access is a popular database management system for personal computers (PCs) and uses many elements of QBE.

Our discussion of the practical languages is not intended to provide a complete description of them, but rather to indicate some of the similarities and differences between theory and practice. We focus here on the central aspects of these languages. Many features, such as string-comparison operators, iteration, and embeddings into a host language, are not mentioned or are touched on only briefly.

We first present highlights of the three languages and then discuss considerations that arise from their use in the real world.

7.1 SQL: The Structured Query Language

SQL has emerged as *the* preeminent query language for mainframe and client-server relational dbms's. This language combines the flavors of both the algebra and the calculus and is well suited for the specification of conjunctive queries.

This section begins by describing how conjunctive queries are expressed using SQL. We then progress to additional features, including nested queries and various forms of negation.

Conjunctive Queries in SQL

Although there are numerous variants of SQL, it has become the standard for relational query languages and indeed for most aspects of relational database access, including data definition, data modification, and view definition. SQL was originally developed under the

name Sequel at the IBM San Jose Research Laboratory. It is currently supported by most of the dominant mainframe commercial relational systems, and increasingly by relational dbms's for PCs.

The basic building block of SQL queries is the *select-from-where* clause. Speaking loosely, these have the form

select <*list of fields to select*>
from <*list of relation names*>
where <*condition*>

For example, queries (4.1) and (4.4) of Chapter 4 are expressed by

select *Director*
from *Movies*
where *Title* = 'Cries and Whispers';

select *Location.Theater, Address*
from *Movies*, *Location*, *Pariscope*
where *Director* = 'Bergman'
and *Movies.Title* = *Pariscope.Title*
and *Pariscope.Theater* = *Location.Theater*;

In these queries, relation names themselves are used to denote variables ranging over tuples occurring in the corresponding relation. For example, in the preceding queries, the identifier *Movies* can be viewed as ranging over tuples in relation *Movies*. Relation name and attribute name pairs, such as *Location.Theater*, are used to refer to tuple components; and the relation name can be dropped if the attribute occurs in only one of the relations in the **from** clause.

The **select** keyword has the effect of the relational algebra *projection* operator, the **from** keyword has the effect of the *cross-product* operator, and the **where** keyword has the effect of the *selection* operator (see Exercise 7.3). For example, the second query translates to (using abbreviated attribute names)

$$\pi_{L.Th,A}(\,\sigma_{D=\text{'Bergman'}\wedge M.Ti=P.Ti\wedge P.Th=L.Th}(Movies \times Location \times Pariscope)).$$

If all of the attributes mentioned in the **from** clause are to be output, then * can be used in place of an attribute list in the **select** clause. In general, the **where** condition may include conjunction, disjunction, negation, and (as will be seen shortly) nesting of select-from-where blocks. If the **where** clause is omitted, then it is viewed as having value *true* for all tuples of the cross-product. In implementations, as suggested in Chapter 6, optimizations will be used; for example, the **from** and **where** clauses will typically be merged to have the effect of an *equi-join* operator.

In SQL, as with most practical languages, duplicates may occur in a query answer.

Technically, then, the output of an SQL query may be a *bag* (also called "multiset")—a collection whose members may occur more than once. This is a pragmatic compromise with the pure relational model because duplicate removal is rather expensive. The user may request that duplicates be removed by inserting the keyword **distinct** after the keyword **select**.

If more than one variable ranging over the same relation is needed, then variables can be introduced in the **from** clause. For example, query (4.7), which asks for pairs of persons such that the first directed the second and the second directed the first, can be expressed as

select *M1.Director*, *M1.Actor*
from *Movies M1*, *Movies M2*
where *M1.Director* = *M2.Actor*
and *M1.Actor* = *M2.Director*;

In the preceding example, the *Director* coordinate of *M1* is compared with the *Actor* coordinate of *M2*. This is permitted because both coordinates are (presumably) of type **character string**. Relations are declared in SQL by specifying a relation name, the attribute names, and the scalar types associated with them. For example, the schema for *Movies* might be declared as

create table *Movies*
(*Title* **character**[60]
Director **character**[30]
Actor **character**[30]);

In this case, *Title* and *Director* values would be comparable, even though they are character strings of different lengths. Other scalar types supported in SQL include integer, small integer, float, and date.

Although the select-from-where block of SQL has a syntactic flavor close to the relational calculus (but using tuple variables rather than domain variables), from a technical perspective the SQL semantics are firmly rooted in the algebra, as illustrated by the following example.

EXAMPLE 7.1.1 Let $\{R[A], S[B], T[C]\}$ be a database schema, and consider the following query:

select A
from $R,\ S,\ T$
where $R.A = S.B$ **or** $R.A = T.C$;

A direct translation of this into the SPJR algebra extended to permit disjunction in selection formulas (see Exercise 4.22) yields

$$\pi_A(\sigma_{A=B \vee A=C}(R \times S \times T)),$$

which yields the empty answer if S is empty or if T is empty. Thus the foregoing SQL query is not equivalent to the calculus query:

$$\{x \mid R(x) \wedge (S(x) \vee T(x))\}.$$

A correct translation into the conjunctive calculus (with disjunction) query is

$$\{w \mid \exists x, y, z(R(x) \wedge S(y) \wedge T(z) \wedge x = w \wedge (x = y \vee x = z))\}.$$

Adding Set Operators

The select-from-where blocks of SQL can be combined in a variety of ways. We describe first the incorporation of the set operators (**union**, **intersect**, and **difference**). For example, the query

(4.14) List all actors and director of the movie "Apocalypse Now."

can be expressed as

(**select** *Actor Participant*
from *Movies*
where *Title* = 'Apocalypse Now')
union
(**select** *Director Participant*
from *Movies*
where *Title* = 'Apocalypse Now');

In the first subquery the output relation uses attribute *Participant* in place of *Actor*. This illustrates renaming of attributes, analogous to relation variable renaming. This is needed here so that the two relations that are unioned have compatible sort.

Although **union**, **intersect**, and **difference** were all included in the original SQL, only **union** is in the current SQL2 standard developed by the American National Standards Institute (ANSI). The two left out can be simulated by other mechanisms, as will be seen later in this chapter.

SQL also includes a keyword **contains**, which can be used in a selection condition to test containment between the output of two nested select-from-where expressions.

Nested SQL Queries

Nesting permits the use of one SQL query within the **where** clause of another. A simple illustration of nesting is given by this alternative formulation of query (4.4):

```
select   Theater
from     Pariscope
where    Title      in
                    (select   Title
                    from      Movies
                    where     Director = 'Bergman');
```

The preceding example tests membership of a unary tuple in a unary relation. The keyword **in** can also be used to test membership for arbitrary arities. The symbols $<$ and $>$ are used to construct tuples from attribute expressions. In addition, because negation is permitted in the **where** clause, set difference can be expressed. Consider the query

List title and theater for movies being shown in only one theater.

This can be expressed in SQL by

```
select   Title, Theater
from     Pariscope
where    ⟨Title, Theater⟩   not in
                            (select   P1.Title, P1.Theater
                            from      Pariscope P1, Pariscope P2
                            where     P1.Title = P2.Title
                                      and not (P1.Theater = P2.Theater));
```

Expressing First-Order Queries in SQL

We now discuss the important result that SQL is relationally "complete," in the sense that it can express all relational queries expressible in the calculus. Recall from Chapter 5 that the family of nr-datalog$^\neg$ programs is equivalent to the calculus and algebra. We shall show how to simulate nr-datalog$^\neg$ using SQL. Intuitively, the result follows from the facts that

(a) each rule can be simulated using the *select-from-where* construct;

(b) multiple rules defining the same predicate can be simulated using **union**; and

(c) negation in rule bodies can be simulated using **not in**.

We present an example here and leave the formal proof for Exercise 7.4.

Example 7.1.2 Consider the following query against the **CINEMA** database:

Find the theaters showing every movie directed by Hitchcock.

An nr-datalog$^\neg$ program expressing the query is

$$
\begin{aligned}
Pariscope'(x_{th}, x_{title}) &\leftarrow Pariscope(x_{th}, x_{title}, x_{sch}) \\
Bad\_th(x_{th}) &\leftarrow Movies(x_{title}, Hitchcock, x_{act}), \\
&\quad\ Location(x_{th}, x_{loc}, x_{ph}), \\
&\quad\ \neg Pariscope'(x_{th}, x_{title}) \\
Answer(x_{th}) &\leftarrow Location(x_{th}, x_{loc}, x_{ph}), \neg Bad\_th(x_{th}).
\end{aligned}
$$

In the program, *Bad_th* holds the list of "bad" theaters, for which one can find a movie by Hitchcock that the theater is not showing. The last rule takes the complement of *Bad_th* with respect to the list of theaters provided by *Location*.

An SQL query expressing an nr-datalog$^{\neg}$ program such as this one can be constructed in two steps. The first is to write SQL queries for each rule separately. In this example, we have

Pariscope′:
select *Theater*, *Title*
from *Pariscope*;

Bad_th:
select *Theater*
from *Movies*, *Location*
where *Director* = 'Hitchcock'
and ⟨*Theater, Title*⟩ **not in**
(**select** *
from *Pariscope′*);

Answer:
select *Theater*
from *Location*
where *Theater* **not in**
(**select** *
from *Bad_th*);

The second step is to combine the queries. In general, this involves replacing nested queries by their definitions, starting from the *answer* relation and working backward. In this example, we have

select *Theater*
from *Location*
where *Theater* **not in**
(**select** *Theater*
from *Movies*, *Location*
where *Director* = 'Hitchcock'
and ⟨*Theater, Title*⟩ **not in**
(**select** *Theater*, *Title*
from *Pariscope*));

In this example, each *idb* (see Section 4.3) relation that occurs in a rule body occurs

negatively. As a result, all variables that occur in the rule are bound by *edb* relations, and so the *from* part of the (possibly nested) query corresponding to the rule refers only to *edb* relations. In general, however, variables in rule bodies might be bound by positively occurring *idb* relations, which cannot be used in any *from* clause in the final SQL query. To resolve this problem, the nr-datalog$^\neg$ program should be rewritten so that all positively occurring relations in rule bodies are *edb* relations (see Exercise 7.4a).

View Creation and Updates

We conclude our consideration of SQL by noting that it supports both view creation and updates.

SQL includes an explicit mechanism for view creation. The relation *Champo-info* from Example 4.3.4 is created in SQL by

create view *Le Champo* **as**
    select *Pariscope.Title, Schedule, Phone*
    from *Pariscope, Location*
    where *Pariscope.Theater* = 'Le Champo'
        and *Location.Theater* = 'Le Champo.'

Views in SQL can be accessed as can normal relations and are useful in building up complex queries.

As a practical database language, SQL provides commands for updating the database. We briefly illustrate these here; some theoretical aspects concerning updates are presented in Chapter 22.

SQL provides three primitive commands for modifying the contents of a database—**insert**, **delete**, and **update** (in the sense of modifying individual tuples of a relation).

The following can be used to insert a new tuple into the *Movies* database:

insert into *Movies*
    values ('Apocalypse Now,' 'Coppola,' 'Duvall');

A set of tuples can be deleted simultaneously:

delete *Movies*
    where *Director* = 'Hitchcock';

Tuple update can also operate on sets of tuples (as illustrated by the following) that might be used to correct a typographical error:

update *Movies*
    set *Director* = 'Hitchcock'
    where *Director* = 'Hickcook';

The ability to insert and delete tuples provides an alternative approach to demonstrating the relational completeness of SQL. In particular, subexpressions of an algebra expression can be computed in intermediate, temporary relations (see Exercise 7.6). This approach does not allow the same degree optimization as the one based on views because the SQL interpreter is required to materialize each of the intermediate relations.

7.2 Query-by-Example and Microsoft Access

We now turn to two query languages that have a more visual presentation. The first, Query-by-Example (QBE), presents a visual display for expressing conjunctive queries that is close to the perspective of tableau queries. The second language, Access, is available on personal computers; it uses elements of QBE, but with a more graphical presentation of join relationships.

QBE

The language Query-By-Example (QBE) was originally developed at the IBM T. J. Watson Research Center and is currently supported as part of IBM's Query Management Facility. As illustrated at the beginning of Chapter 4, the basic format of QBE queries is fundamentally two-dimensional and visually close to the tableau queries. Importantly, a variety of features are incorporated into QBE to give more expressive power than the tableau queries and to provide data manipulation capabilities. We now indicate some features that can be incorporated into a QBE-like visual framework. The semantics presented here are a slight variation of the semantics supported for QBE in IBM's product line.

As seen in Fig. 4.2, which expresses query (4.4), QBE uses strings with prefix _ to denote variables and other strings to denote constants. If the string is preceded by P., then the associated coordinate value forms part of the query output. QBE framework can provide a partial union capability by permitting the inclusion in a query of multiple tuples having a P. prefix in a single relation. For example, Fig. 7.1 expresses the query

(4.12) What films with Allen as actor or director are currently featured at the Concorde?

Under one natural semantics for QBE queries, which parallels the semantics of conjunctive queries and of SQL, this query will yield the empty answer if either $\sigma_{Director=\text{“Allen”}}Movies$ or $\sigma_{Actor=\text{“Allen”}}Movies$ is empty (see Example 7.1.1).

QBE also includes a capability of *condition boxes*, which can be viewed as an extension of the incorporation of equality atoms into tableau queries.

QBE does not provide a mechanism analogous to SQL for nesting of queries. It is hard to develop an appropriate visual representation of such nesting within the QBE framework, in part due to the lack of scoping rules. More recent extensions of QBE address this issue by incorporating, for example, hierarchical windows. QBE also provides mechanisms for both view definition and database update.

Negation can be incorporated into QBE queries in a variety of ways. The use of database update is an obvious mechanism, although not especially efficient. Two restricted

Movies	*Title*	*Director*	*Actor*
	_X	Allen	
	_Y		Allen

Pariscope	*Theater*	*Title*	*Schedule*
	Concorde	P._X	
	Concorde	P._Y	

Figure 7.1: One form of union in QBE

Movies	*Title*	*Director*	*Actor*
¬	_Z	Bergman	

Pariscope	*Theater*	*Title*	*Schedule*
	P._champio ¬Concorde	_Z	

Figure 7.2: A query with negation in QBE

forms of negation are illustrated in Fig. 7.2, which expresses the following query: (assuming that each film has only one director) what theaters, other than the Concorde, feature a film *not* directed by Bergman? The ¬ in the *Pariscope* relation restricts attention to those tuples with *Theater* coordinate not equal to Concorde, and the ¬ preceding the tuple in the *Movies* relation is analogous to a negative literal in a datalog rule and captures a limited form of ¬∃ from the calculus; in this case it excludes all films directed by Bergman. When such negation is used, it is required that all variables that occur in a row preceded by ¬ also appear in positive rows. Other restricted forms of negation in QBE include using negative literals in condition boxes and supporting an operator analogous to relational division (as defined in Exercise 5.8).

The following example shows more generally how view definition can be used to obtain relational completeness.

EXAMPLE 7.2.1 Recall the query and nr-datalog¬ program of Example 7.1.2. As with SQL, the QBE query corresponding to an nr-datalog¬ will involve one or more views for each rule (see Exercise 7.5). For this example, however, it turns out that we can compute the effect of the first two rules with a single QBE query. Thus the two stages of the full query are shown in Fig. 7.3, where the symbol *I*. indicates that the associated tuples are to be inserted into the answer. The creation of the view *Bad_th* is accomplished using the

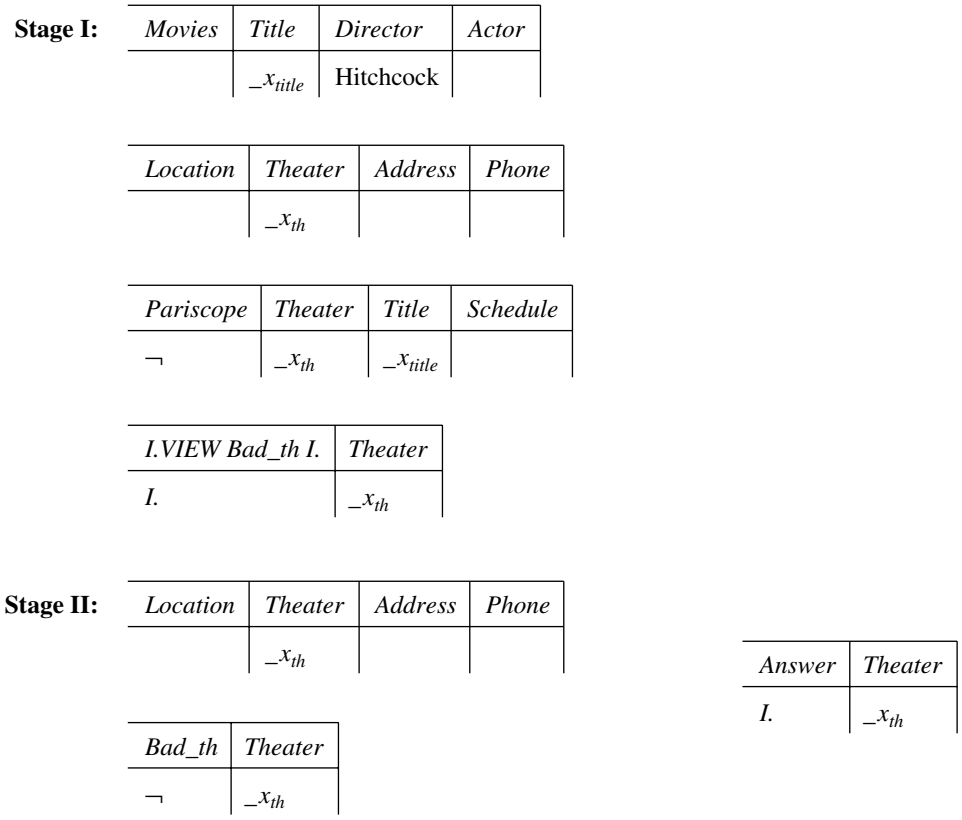


Figure 7.3: Illustration of relational completeness of QBE

expression *I.VIEWBad_th I.*, which both creates the view and establishes the attribute names for the view relation.

Microsoft Access: A Query Language for PCs

A number of dbms's for personal computers have become available over the past few years, such as DBASE IV, Microsoft Access, Foxpro, and Paradox. Several of these support a version of SQL and a more visual query language. The visual languages have a flavor somewhat different from QBE. We illustrate this here by presenting an example of a query from the Microsoft Access dbm's.

Access provides an elegant graphical mechanism for constructing conjunctive queries. This includes a tabular display to indicate the form and content of desired output tuples, the use of single-attribute conditions within this display (in the rows named "Criteria" and "or"), and a graphical presentation of join relationships that are to hold between relations used to form the output. Fig. 7.4 shows how query (4.4) can be expressed using Access.

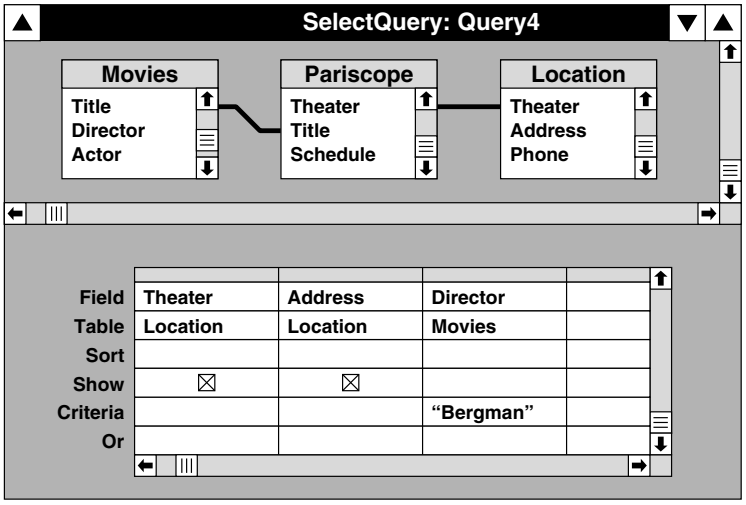


Figure 7.4: Example query in Access

(Although not shown in the figure, join conditions can also be expressed using single-attribute conditions represented as text.)

Limited forms of negation and union can be incorporated into the condition part of an Access query. For more general forms of negation and union, however, the technique of building views to serve as intermediate relations can be used.

7.3 Confronting the Real World

Because they are to be used in practical situations, the languages presented in this chapter incorporate a number of features not included in their formal counterparts. In this section we touch on some of these extensions and on fundamental issues raised by them. These include domain independence, the implications of incorporating many-sorted atomic objects, the use of arithmetic, and the incorporation of aggregate operators.

Queries from all of the practical languages described in this chapter are domain independent. This is easily verified from the form of queries in these languages: Whenever a variable is introduced, the relation it ranges over is also specified. Furthermore, the specific semantics associated with **or**'s occurring in **where** clauses (see Example 7.1.1) prevent the kind of safety problem illustrated by query *unsafe-2* of Section 5.3.

Most practical languages permit the underlying domain of values to be many-sorted—for example, including distinct scalar domains for the types integer, real, character string, etc., and some constructed types, such as date, in some languages. (More recent systems, such as POSTGRES, permit the user to incorporate abstract data types as well.) For most of the theoretical treatment, we assumed that there was one underlying domain of values, **dom**, which was shared equally by all relational attributes. As noted in the discussion of

SQL, the typing of attributes can be used to ensure that comparisons make sense, in that they compare values of comparable type. Much of the theory developed here for a single underlying domain can be generalized to the case of a *many-sorted* underlying domain (see Exercise 7.8).

Another fundamental feature of practical query languages is that they offer value comparators other than equality. Typically most of the base sorts are totally ordered. This is the case for the integers or the strings (under the lexicographical ordering). It is therefore natural to introduce $\leq, \geq, <, >$ as comparators. For example, to ask the query, "What can we see at the Le Champo after 21:00," we can use

$$ans(x_t) \leftarrow Pariscope(\text{“Le Champo,”}x_t, x_s),\ x_s > \text{“21:00”};$$

and, in the algebra, as

$$\pi_{Title}(\sigma_{Theater=\text{“Le Champo”} \wedge Schedule>\text{“21:00”}} Pariscope).$$

Exercise 4.30 explores the impact of incorporating comparators into the conjunctive queries. Many languages also incorporate string-comparison operators.

Given the presence of integers and reals, it is natural to incorporate arithmetic operators. This yields a fundamental increase in expressive power: Even simple counting is beyond the power of the calculus (see Exercise 5.34).

Another extension concerns the incorporation of *aggregate* operators into the practical languages (see Section 5.5). Consider, for example, the query, "How many films did Hitchcock direct?". In SQL, this can be expressed using the query

select **count**(**distinct** *Title*)
from *Movies*
where *Director* = 'Hitchcock';

(The keyword **distinct** is needed here, because otherwise SQL will not remove duplicates from the projection onto *Title*.) Other aggregate operators typically supported in practical languages include **sum**, **average**, **minimum**, and **maximum**.

In the preceding example, the aggregate operator was applied to an entire relation. By using the **group by** command, aggregate operators can be applied to clusters of tuples, each common values on a specified set of attributes. For example, the following SQL query determines the number of movies directed by each director:

select *Director*, **count**(**distinct** *Title*)
from *Movies*
group by *Director*;

The semantics of *group by* in SQL are most easily understood when we study an extension of the relational model, called the complex object (or nested relation) model, which models grouping in a natural fashion (see Chapter 20).

Bibliographic Notes

General descriptions of SQL and QBE may be found in [EN89, KS91, Ull88]; more details on SQL can be found in [C+76], and on QBE in [Zlo77]. Another language similar in spirit to SQL is Quel, which was provided with the original INGRES system. A description of Quel can be found in [SWKH76]. Reference [OW93] presents a survey of QBE langauges and extensions. A reference on Microsoft Access is [Cam92]. In Unix, the command *awk* provides a basic relational tool.

The formal semantics for SQL are presented in [NPS91]. Example 7.1.1 is from [VanGT91]. Other proofs that SQL can simulate the relational calculus are presented in [PBGG89, Ull88]. Motivated by the fact that SQL outputs bags rather than sets, [CV93] studies containment and equivalence of conjunctive queries under the bag semantics (see "Bibliographic Notes" in Chapter 6).

Aggregate operators in query languages are studied in [Klu82].

SQL has become the standard relational query language [57391, 69392]; reference [GW90] presents the original ANSI standard for SQL, along with commentary about particular products and some history. SQL is available on most main-frame relational dbms's, including, for example, IBM's DB2, Oracle, Informix, INGRES, and Sybase, and in some more recent database products for personal computers (e.g.,Microsoft Access, dBASE IV). QBE is available as part of IBM's product QMF (Query Management Facility). Some personal computer products support more restricted graphical query languages, including Microsoft Access and Paradox (which supports a form-based language).

Exercises

Exercise 7.1 Write SQL, QBE, and Access queries expressing queries (4.1 to 4.14) from Chapter 4. Start by expressing them as nr-datalog$^\neg$ programs.

Exercise 7.2 Consider again the queries (5.2 and 5.3) of Chapter 5. Express these in SQL, QBE, and Access.

Exercise 7.3 Describe formally the mapping of SQL select-from-where blocks into the SPJR algebra.

♠ **Exercise 7.4**

(a) Let P be an nr-datalog$^\neg$ program. Describe how to construct an equivalent program P' such that each predicate that occurs positively in a rule body is an *edb* predicate.

(b) Develop a formal proof that SQL can simulate nr-datalog$^\neg$.

Exercise 7.5 Following Example 7.2.1, show that QBE is relationally complete.

Exercise 7.6

(a) Assuming that R and S have compatible sorts, show how to compute in SQL the value of $R - S$ into the relation T using **insert** and **delete**.

(b) Generalize this to show that SQL is relationally complete.

Exercise 7.7 In a manner analogous to Exercise 7.6, show that Access is relationally complete.

★ **Exercise 7.8** The intuition behind the typed restricted PSJ algebra is that each attribute has a distinct type whose elements are incomparable with the types of other attributes. As motivated by the practical query languages, propose and study a restriction of the SPJR algebra analogous to the typed restricted PSJ algebra, but permitting more than one attribute with the same type. Does the equivalence of the various versions of the conjunctive queries still hold? Can Exercise 6.21 be generalized to this framework?