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

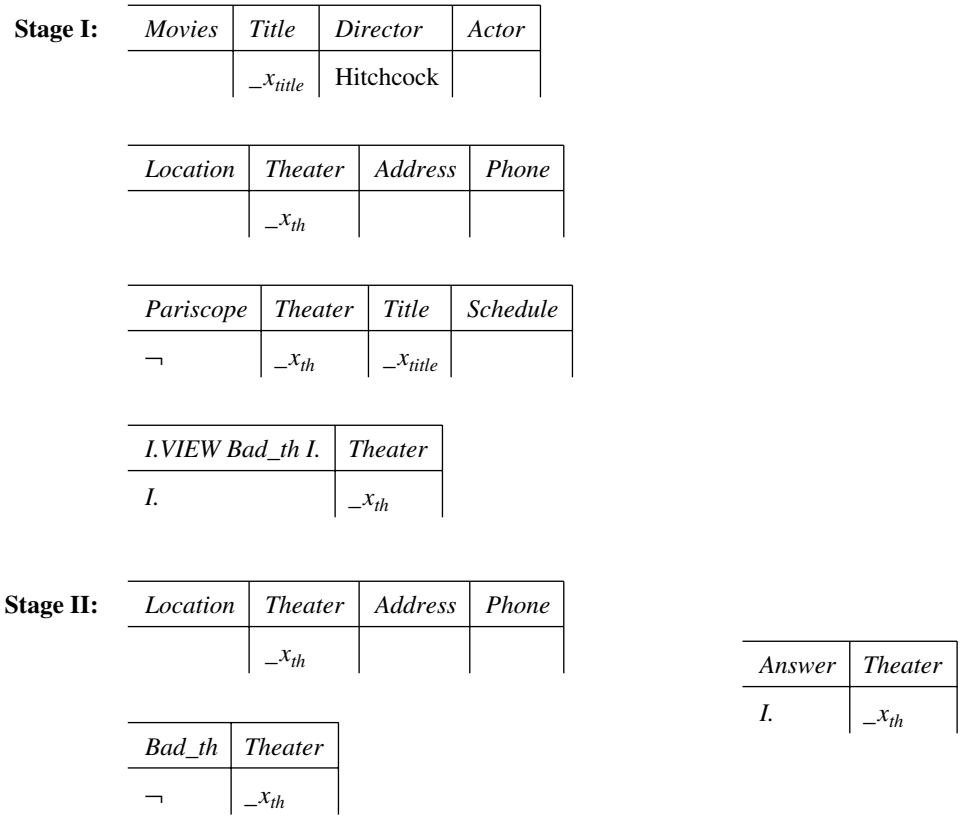

Figure 7.3: Illustration of relational completeness of QBE

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.

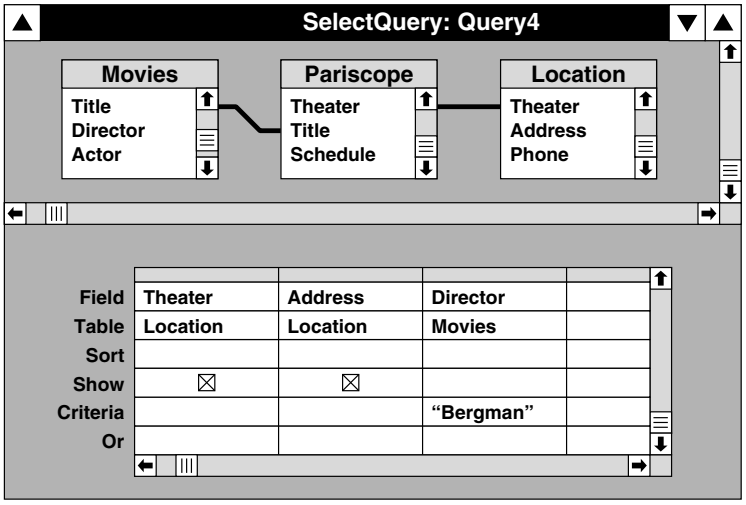

Figure 7.4: Example query in Access

(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?