4. RELATIONAL DATABASE LANGUAGES

This chapter describes relational database languages. Sections 1 and 2 show examples of how the languages of Chapter 2 (the fourth-generation and the logic-based languages) apply to the relational databases. The principal example of Section 1 discusses the principles of writing a transaction-processing program for an application. The optional Section 3 defines the Relational Algebra. Section 4 describes SQL, a popular commercial language related to the Relational Predicate Calculus of Section 2. The expressive power and equivalence of relational languages is discussed in Section 5.

4.1. Fourth-generation Programming

The structured extension of Pascal for relational databases is the same as the extension of Pascal for semantic databases but is used for the relational schema. (Every relational schema is a binary schema. Thus, every language used for the Binary Model can be used for the Relational Model.)

Example 4-1.

The university has decided to expel all the students whose average grade is below 60 (out of 100). To prevent this wrong-doing to computer science students, the department offered a fictitious course, Computer-Pass, by Prof. Good, in which all computer science students are to receive a sufficient grade so as to not to be expelled, if possible.

The following program fabricates Prof. Good and the Computer-Pass course, enrolls students in this course, grades them accordingly, and prints the names of those computer science students whom this measure cannot help.

```
program Pass (Input, Output, UNIVERSITY-DB, UNIVERSITY-MASTER-VIEW);
var
  Computer-Pass-Course, Prof-Good, Good-Offer, computer-science-name, comp-science,
  cs-student, her-enrollment, fictitious-enrollment: ABSTRACT;
  the-grade, desired-grade, number-of-grades, total-of-grades, current-year: INTEGER;
begin
(* Get the current year from the standard input file. *)
  read (current-year);
```
transaction begin

(* Fabricate the course. *)

create new Computer-Pass-Course in COURSE;
Computer-Pass-Course.NAME-key := ‘Computer Pass’;

(* Fabricate the Prof. Good. *)

create new Prof-Good in INSTRUCTOR;
Prof-Good.LAST-NAME := ‘Good’;
Prof-Good.ID-key := 1234; (* Let’s hope that this fabricated ID number does not already belong to a legitimate instructor. Otherwise, an error will result. *)

(* Fabricate the offering. *)

create new Good-Offer in COURSE-OFFERING;
Good-Offer.COURSE-NAME-in-key := ‘Computer Pass’;
Good-Offer.INSTRUCTOR-ID-in-key := 1234;
Good-Offer.YEAR := current-year;
Good-Offer.SEASON := ‘Winter’
end;

(* The following two nested loops will be performed only once. Inside the body of the second loop, the variable comp-science will refer to the Computer Science Department. *)

for computer-science-name in DEPARTMENT-NAMING

where (computer-science-name.NAME-key = ‘COMPUTER SCIENCE’) do

for comp-science in DEPARTMENT

where (comp-science.MAIN-NAME-key = comp-science-name.MAIN-NAME)

begin

(* Make believe that Prof. Good works in Computer Science. *)

(* In terms of the binary schema, relate: Prof-Good WORKS-IN comp-science *)

transaction begin

create new work in WORK;
work.INSTRUCTOR-ID-in-key := 1234;
work.DEPARTMENT-MAIN-NAME-in-key := comp-science.MAIN-NAME-key

end;

Rishe-92-DDS p. 4.2
for cs-student in STUDENT
  where (cs-student.MAJOR-DEPARTMENT-MAIN-NAME
          = comp-science.MAIN-NAME-key) do
begin (* the current computer science student *)
(* calculate this student’s current statistics: number-of-grades and total-of-grades *)
  number-of-grades := 0;
  total-of-grades := 0;
for her-enrollment in COURSE-ENROLLMENT where (her-enrollment.-
  STUDENT-ID-in-key = cs-student.ID-key and not her-enrollment
  FINAL-GRADE null) do begin
  the-grade := her-enrollment.FINAL-GRADE;
  number-of-grades := number-of-grades + 1;
  total-of-grades := total-of-grades + the-grade
end;
(* calculate the minimal desired grade in computer-pass course, solving the equation
  \[(total+x)/(number+1)=60\] *)
  desired-grade := 60 * (number-of-grades + 1) — total-of-grades;
if desired-grade > 100 then
  (* the student cannot be helped. Print a message *)
    writeln (`The student ', cs-student.LAST-NAME, ` cannot be helped. Sorry!')
else if desired-grade > 60 then
  transaction begin
    create new fictitious-enrollment in COURSE-ENROLLMENT;
    fictitious-enrollment.STUDENT-ID-in-key := cs-student.ID-key;
    fictitious-enrollment.COURSE-NAME-in-key := `Computer Pass’;
    fictitious-enrollment.INSTRUCTOR-ID-in-key := 1234;
    fictitious-enrollment.YEAR := 1990;
    fictitious-enrollment.SEASON := `Winter’;
    fictitious-enrollment.FINAL-GRADE := desired-grade
  end (* transaction *)
end (* current student *)
4.1.1. Transaction processing

Many application programs process a continuous sequence of end-user requests for transactions. In each cycle, such a program accepts a request from an end-user, translates it from the data-entry form into the terms of the database, determines the validity of the request, and performs it.

Most relational DBMS are unable to automatically enforce the integrity constraints, particularly referential integrity. In this case, the busy-work of manual integrity validation must be performed by the application programs processing the user transactions.

Example 4-2.
The following program reads from the standard input a series of requests for enrolling students in classes. For every request, the program checks its integrity against the database. If the request is integral, the program performs the update. Otherwise, an error message is printed.

It is assumed that the DBMS is unable to automatically enforce the referential integrity.

program Enroll (Input, Output, University-database, University-master-userview);
var
    student-id, instructor-id, year: Integer;
    course-name, season: String;
    student, instructor, course, quarter, offering, enrollment: ABSTRACT;
    student-ok, instructor-ok, course-ok, quarter-ok, offering-ok, enrollment-ok: Boolean;
procedure Erroneous-transaction (explanation: String);
    (* This procedure is called
        • from Transaction-error-handler, when an error is detected by the DBMS
        • from the program, when an error is detected by the program. *)
begin
    writeln (`The enrollment request listed in the following line could not be granted for the reason: ', explanation);
    writeln (student-id, instructor-id, course-name, year, season)
end
procedure Transaction-error-handler (error-description: String);
begin
Erroneous-transaction (concatenate(‘System error: ’, error-description);
end

begin
while not eof(Input) do
transaction begin
(* Get a request for the next enrollment transaction *)
readln (student-id, instructor-id, course-name, year, season)
(* Check student ID *)
student-ok := false;
for student in STUDENT
   where (student.ID-key = student-id)
do student-ok := true;
if not student-ok then
   Erroneous-transaction ('No student known by the student-id submitted in
   the first field of the enrollment request');

(* Check instructor ID *)
instructor-ok := false;
for instructor in INSTRUCTOR
   where (instructor.ID-key = instructor-id)
do instructor-ok := true;
instructor-ok := false;
for instructor in INSTRUCTOR
   where (instructor.ID-key = instructor-id)
do instructor-ok := true;
if not instructor-ok then
   Erroneous-transaction ('No instructor known by the instructor-id submitted in
   the second field of the enrollment request');

(* Check course name *)
course-ok := false;
for course in COURSE
   where (course.NAME-key = course-name)
do course-ok := true;
course-ok := false;
for course in COURSE
   where (course.NAME-key = course-name)
do course-ok := true;
if not course-ok then
   Erroneous-transaction ('No course known by the course-name submitted in
   the third field of the enrollment request');
(* Check quarter *)
quarter-ok := false;

for quarter in QUARTER
    where (quarter.YEAR-in-key = year and quarter.SEASON-in-key = season)
    do quarter-ok := true;
if not quarter-ok then
    Erroneous-transaction ('No quarter known by the year and season submitted in the fourth and fifth fields of the enrollment request');

(* Check the offering *)

offering-ok := false;

for offering in COURSE-OFFERING
    where (offering.INSTRUCTOR-ID-in-key = instructor-id and offering.COURSE-NAME-in-key = course-name and offering.YEAR-in-key = year and offering.SEASON-in-key = season)
    do offering-ok := true;
if instructor-ok and course-ok and quarter-ok and not offering-ok then
    Erroneous-transaction ('No offering known by the instructor-id, course, year and season submitted in the second through fifth fields of the enrollment request');

(* The following check of nonduplicate enrollment is not strictly necessary, since it can be performed automatically by the DBMS, which knows to enforce the uniqueness of the keys. Thus, the only practical reason for this test is to produce a better message than what would be produced by the system by default. *)

(* Check that student is not already enrolled in the offering *)

enrollment-ok := true;

for enrollment in COURSE-ENROLLMENT
    where (enrollment.STUDENT-ID-in-key = student-id and enrollment.INSTRUCTOR-ID-in-key = instructor-id and enrollment.COURSE-NAME-in-key = course-name and enrollment.YEAR-in-key = year and enrollment.SEASON-in-key = season)
    do enrollment-ok := false;
if not enrollment-ok then
    Erroneous-transaction ('The requested enrollment of the student already exists')

if student-ok and instructor-ok and course-ok and quarter-ok and offering-ok and enrollment-ok
then

Rishe-92-DDS p. 4.6
begin
create new enrollment in COURSE-ENROLLMENT;
enrollment.STUDENT-ID-in-key := student-id;
enrollment.INSTRUCTOR-ID-in-key := instructor-id;
enrollment.COURSE-NAME-in-key := course-name;
enrollment.YEAR-in-key := year;
enrollment.SEASON-in-key := season
end

end (* transaction *)

end.

4.2. Logic for Relational Databases

Relational Calculus — Predicate Calculus, when used for the relational schema. (Since every relational schema is a binary schema, we already know Relational Calculus.)

Example 4-3.
What are the last names of all the students?
get s.LAST-NAME
where s is a STUDENT

Example 4-4.
What are the distinct last names of the students? (No name may be printed twice.)
get n
where
    exists s in STUDENT:
    n = s.LAST-NAME

Tuple-oriented Relational Calculus — Relational calculus with the following restriction: the quantification is done only on abstract categories (i.e., tables).

Among the languages of the Relational Model, more languages are based on the tuple-oriented Predicate Calculus than on the more general form of Predicate Calculus.
Example 4-5.
The previous example was not in the tuple-oriented form because the variable $n$ was implicitly quantified over the concrete category $String$.
The following examples are in tuple-oriented form.

Example 4-6.
Has every student enrolled in at least one course in 1990?

for every $st$ in $STUDENT$:
exists $enrl$ in $COURSE-ENROLLMENT$:

((enrl.$STUDENT-ID-in-key$ = $st$.ID-key) and (enrl.YEAR=1990))

Example 4-7.
Who took Prof. Smith’s courses?

get $student$.LAST-NAME where

(student is a $STUDENT$ and
exists $enrl$ in $COURSE-ENROLLMENT$:

(enrl.$STUDENT-ID-in-key$=student.ID-key and
exists $inst$ in $INSTRUCTOR$:

(inst.LAST-NAME = ‘Smith’ and
enrl.INSTRUCTOR-ID-in-key=inst.ID-key)))

Example 4-8.
Print the average grade of every computer science student.

get $student$.LAST-NAME, (average enrollment.FINAL GRADE

where

enrollment.$STUDENT-ID-in-key$ = student.ID-key)

where

student is a $STUDENT$ and
Example 4-9.
How many students are there in the university?
\[
\text{get} \ (\text{count std where std is a STUDENT})
\]

Example 4-10.
What students have their average grade below 60?
\[
\text{get std.LAST-NAME}
\quad \text{where std is a STUDENT and}
\quad 60 >
\quad \text{average enrl.FINAL-GRADE}
\quad \text{where}
\quad \text{enrl is a COURSE-ENROLLMENT and}
\quad \text{enrl.STUDENT-ID-in-key = std.ID-key}
\]

4.3. *Relational Algebra

Relational Algebra is an algebraic language in which new tables are defined by applying operators to other tables.

This is a language of expressions. In it, a new table is defined as an expression involving original tables and operators.

The most important operators are:

- **Projection operator** creates a new table containing some of the columns of another table.
- **Join operator** combines the rows of the first table with "related" rows of the second table.
- **Selection operator** extracts some rows from a table according to a given condition on the values of the row.

Example 4-11.
\[
\{\text{The last names of the students born in 1975} \} =
\quad (\text{project-the-column-LAST-NAME}
\quad (\text{select-the-rows-where-the-BIRTH-YEAR-is-1975})
\]

Rishe-92-DDS p. 4.9
In this section, the operators of Relational Algebra are defined by inference laws in Predicate Calculus.

Consider two tables:

- table \( T \), whose attributes are \( A_1, \ldots, A_n \)
- table \( T' \), whose attributes are \( A'_1, \ldots, A'_{n'} \)

1. The projection operator \( \ldots \{ \text{attributes} \} \) creates a new table containing some of the columns of another table.

   Let \( F_1, \ldots, F_k \) be some of the attributes (columns) of table \( T \). Then

   \[
   T \{ F_1, \ldots, F_k \} = \text{get} \ F_1 : v_1, \ldots, F_k : v_k \,
   \text{where exists } x \text{ in } T:
   \]

   \[ x.F_1 = v_1 \text{ and } \cdots \text{ and } x.F_k = v_k \]

   \textbf{Note:} When several tuples of } T \text{ have the same values in the columns being projected, only one row will appear in the result. Thus, the resulting table may have fewer tuples than } T.

\[ \]

\textbf{Example 4-12.}

A list of the distinct last names of the students =

\[
\text{STUDENT} \{ \text{LAST-NAME} \} = \text{get} \ \text{LAST-NAME} : \text{name}
\text{where}
\]

\[ \text{exists } x \text{ in } \text{STUDENT}:
\]

\[ x.\text{LAST-NAME} = \text{name} \]

\[ \]

\textbf{Example 4-13.}

Last names and majors of all the students =

\[
\text{STUDENT} \{ \text{LAST-NAME, MAJOR-DEPARTMENT-MAIN-NAME} \} = \text{get} \ \text{LAST-NAME} : \text{name, MAJOR-DEPARTMENT-MAIN-NAME} : \text{major}
\text{where}
\]

\[ \text{exists } x \text{ in } \text{STUDENT} : \]

\[ \]

Rishe-92-DDS p. 4.10
Example 4-14.

We can define an inferred table \textit{STUDENT-BASIC} containing all the information from the table \textit{STUDENT} except the departments:

\[
\text{STUDENT-BASIC} = \text{STUDENT} [\text{ID-key, LAST-NAME, FIRST-NAME, BIRTH-YEAR, ADDRESS}]
\]

2. The \textbf{renaming operator} (\ldots [\text{attribute/new-name}]) changes the name of a column in a table.

\[
T / A_i, A_i =
\]

(* Copy the attributes \(A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n\); rename the attribute \(A_i\). *)

\[
\text{get } A_1: x.A_1, \ldots, A_{i-1}: x.A_{i-1}, \overline{A_i}: x.A_i, A_{i+1}: x.A_{i+1}, \ldots, A_n: x.A_n
\]

where \textit{x} is a \textit{T}

Example 4-15.

A table just like \textit{STUDENT}, with ‘FAMILY-NAME’ column title instead of ‘LAST-NAME’:

\[
\text{STUDENT [LAST-NAME/FAMILY-NAME]}
\]

3. The \textbf{cartesian product operator} \(\times\)

For every row of the first operand and for every row of the second operand, the product operator produces the concatenation of the two rows.

The number of rows in the result =

\[
(\text{the number of rows in the first operand} \times \text{the number of rows in the second operand})
\]

The number of columns in the result =

\[
(\text{the number of columns in the first operand} + \text{the number of columns in the second operand})
\]

This operation is syntactically erroneous when the two tables have a common attribute.

Cartesian product =

\[
T \times T' =
\]

Rishe-92-DDS p. 4.11
get $A_1:x.A_1, \ldots, A_n:x.A_n$, $A_1':y.A_1', \ldots, A_n':y.A_n'$

where $x$ is a $T$ and $y$ is a $T'$

4. Set operators

The following operators are defined only when the two tables have the same attributes.

a. **Union of tables** produces all the rows of the first table and all the rows of the second table.

$$T \cup T' =$$

get $A_1:v_1, \ldots, A_n:v_n$

where

exists $x$ in $T$:

$x.A_1=v_1$ and $\cdots$ and $x.A_n=v_n$

or exists $x$ in $T'$:

$x.A_1=v_1$ and $\cdots$ and $x.A_n=v_n$

Example 4-16.

All the persons =

$$\text{STUDENT-BASIC} \cup \text{INSTRUCTOR} =$$

get ID-key : id, $\ldots$, ADDRESS : addr

where

exists $x$ in STUDENT-BASIC:

$x.ID-key = \text{id}$ and $\cdots$ and $x.ADDRESS = \text{addr}$

or exists $x$ in INSTRUCTOR:

$x.ID-key = \text{id}$ and $\cdots$ and $x.ADDRESS = \text{addr}$

b. **Intersection of tables** produces the rows which appear in both tables.

$$T \cap T' =$$

get $A_1:v_1, \ldots, A_n:v_n$

where

exists $x$ in $T$:

$x.A_1=v_1$ and $\cdots$ and $x.A_n=v_n$

and exists $x$ in $T'$:
Instructors who are students = 
\[ \text{INSTRUCTOR} \cap \text{STUDENT-BASIC} \]

Example 4-17.

The difference of tables operator \((-\)) produces the rows of the first table which do not appear in the second table.

\[ T - T' = \]
\[
\text{get } A_1 : v_1, \ldots, A_n : v_n \\
\text{where} \\
\text{exists } x \text{ in } T: \\
x.A_1 = v_1 \text{ and } \cdots \text{ and } x.A_n = v_n \\
\text{and not exists } x \text{ in } T': \\
x.A_1 = v_1 \text{ and } \cdots \text{ and } x.A_n = v_n
\]

Example 4-18.

Instructors who are not students = 
\[ \text{INSTRUCTOR} - \text{STUDENT-BASIC} \]

5. The selection operator \((...[\text{condition}])\) extracts some rows from a table according to a given condition for the values of the row.

Let \( F_1, \ldots, F_k \) be some of the attributes of \( T \).

Let \( \text{boolexp} (v_1, \ldots, v_k) \) be a Boolean expression with \( k \) variables.

Then
\[ T [\text{boolexp} (F_1, \ldots, F_k)] = \]
\[
\text{get } x.A_1, \ldots, x.A_n \\
\text{where} \\
x \text{ is a } T \text{ and} \\
\text{boolexp} (x.F_1, \ldots, x.F_k)
\]

Rishe-92-DDS p. 4.13
Example 4-19.
The student whose first name is Mary =

\[\text{STUDENT}\left[\text{FIRST-NAME}='\text{Mary}'\right]\]

Example 4-20.
The instructor whose name is Chung and who is not a student (as distinguished from another Chung who is both an instructor and a student.)

All the instructors whose name is ‘Chung’ =

\[\text{INSTRUCTOR}\left[\text{LAST-NAME}='\text{Chung}'\right]\]

The nonstudent instructor(s) whose name is ‘Chung’ =

\[\text{INSTRUCTOR}\left[\text{LAST-NAME}='\text{Chung}'\right] - \text{STUDENT-BASIC}\]

Example 4-21.
Names of the instructors teaching databases.

All the combinations of instructors and offerings (including the unrelated ones) =

\[(\text{INSTRUCTOR} \times \text{COURSE-OFFERING})\] (*product*)

All combinations of instructors and their offerings =

\[(\text{INSTRUCTOR} \times \text{COURSE-OFFERING})\]
\[\left[\text{ID-key}=\text{INSTRUCTOR-ID-in-key}\right]\] (*selection*)

All combinations of instructors and their offerings of Databases =

\[(\text{INSTRUCTOR} \times \text{COURSE-OFFERING})\]
\[\left[\text{ID-key}=\text{INSTRUCTOR-ID-in-key}\right]\]
\[\left[\text{COURSE-NAME-in-key}='\text{Databases}'\right]\] (*selection*)

The last names of the instructors offering Databases =

\[(\text{INSTRUCTOR} \times \text{COURSE-OFFERING})\]
\[\left[\text{ID-key}=\text{INSTRUCTOR-ID-in-key}\right]\]
\[\left[\text{COURSE-NAME-in-key}='\text{Databases}'\right]\]
\[\left[\text{LAST-NAME}\right]\] (*projection*)
6. The **join operator** combines the rows of the first table with "related" rows of the second table. It is equivalent to a selection from the cartesian product.

\[
T[\text{boolexp (attributes )}]T' = (T \times T')[\text{boolexp (attributes )}]
\]

**Example 4-22.**
Names of instructors teaching databases.

All combinations of instructors and their offerings =

\[
(INSTRUCTOR [ID-key=INSTRUCTOR-ID-in-key] COURSE-OFFERING)
\]

(*join*)

The last names of the instructors offering databases =

\[
(INSTRUCTOR [ID-key=INSTRUCTOR-ID-in-key] COURSE-OFFERING)
\]

[\text{COURSE-NAME-in-key=‘Databases’}]

[\text{LAST-NAME}]

7. The **natural join operator** combines two tables according to the equal values of the common attributes (column names) of the two tables.

Let the table \(T\) have \(k\) attributes with names identical to the names of \(k\) attributes of the table \(T'\), that is:

- the attributes of \(T\) are: \(A_1, \ldots, A_k, A_{k+1}, \ldots, A_n\)
- the attributes of \(T'\) are: \(A_1, \ldots, A_k, A_{k+1}, \ldots, A_n'\)

Then

\[
T \bowtie T' = \text{get } A_1:v_1, \ldots, A_n:v_n, A_{k+1}:w_{k+1}, \ldots, A_n':w_n'
\]

where

**Example 4-23.**
Names of instructors teaching databases.

The table \textit{INSTRUCTOR} with the column \textit{ID-key} renamed in order to be naturally
joinable with the table $\textit{COURSE-OFFERING} =$

$\textit{(INSTRUCTOR [ID-key/INSTRUCTOR-ID-in-key]}$

(*rename*)

All combinations of instructors and their offerings $=\textit{(INSTRUCTOR [ID-key/INSTRUCTOR-ID-in-key]} \Box$

$\textit{COURSE-OFFERING})$

(*natural join*)

The last names of the instructors offering databases $=\textit{(INSTRUCTOR [ID-key/INSTRUCTOR-ID-in-key]} \Box$

$\textit{COURSE-OFFERING})$

$[\textit{COURSE-NAME-in-key}=\textquote{Databases'}]$

$[\textit{LAST-NAME}]$

**Uses of the Relational Algebra:**


2. Specification of queries. Albeit, the language is not friendly enough to be used for specification of complex queries.

3. An intermediate language, because it is easy to implement Relational Algebra. Other, more friendly languages, can be translated into Relational Algebra.

4. A tool to evaluate and compare different languages. We can estimate the expressive power of an arbitrary language by checking whether it is
   
   a. Able to specify every query expressible in Relational Algebra.
   
   b. Able to specify every query expressible in Relational Algebra and more.
   
   c. Able to specify a subset of the queries expressible in Relational Algebra, where the subset is defined by weakening the Algebra through eliminating some of its operators. The list of the eliminated operators shows the weakness of the language. The list of the remaining operators shows the power of the language:
      
      - Many simple query languages can express *projection* and *selection* but not *join* or *difference*.
      - More powerful languages can express *join*.
      - Languages which are even more powerful can also express *difference*.

4.4. **SQL**
4.4.1. Preview
SQL has become a very popular language of commercial relational database management systems.
The acronym SQL stands for Structured Query Language.
A basic query in SQL selects the values of some attributes from some rows of a table or tables where the rows satisfy a condition

Example 4-24.
When was student Russel born?
```sql
select BIRTH-YEAR
from STUDENT
where LAST-NAME=‘Russel’
```

Example 4-25.
List the names of all students.
```sql
select FIRST-NAME, LAST-NAME
from STUDENT
where true
```

Example 4-26.
What courses has Prof. Graham taught?
```sql
select COURSE-NAME
from COURSE-OFFERING, INSTRUCTOR
where INSTRUCTOR.NAME = ‘Graham’ and INSTRUCTOR.ID = COURSE-OFFERING. INSTRUCTOR-ID
```

4.4.2. Basic queries
Syntax:
```sql
select expression_1, . . . , expression_n
from table_1 var_1, . . . , table_n var_n
```
where condition

The condition is a Boolean expression without quantifiers. It may depend on the variables \( \text{var}_1, \ldots, \text{var}_n \).

Meaning:

get \( \text{expression}_1, \ldots, \text{expression}_n \)

where

\[
\text{var}_1 \text{ is a table}_1 \text{ and } \cdots \text{ var}_n \text{ is a table}_n \text{ and } \\
\text{condition}
\]

Example 4-27.

Print the names of the pairs of students who live together.

```
select s_1.LAST-NAME, s_2.LAST-NAME 
from STUDENT s_1, STUDENT s_2 
where s_1.ADDRESS = s_2.ADDRESS
```

Abbreviation:

If a table \( T_i \) appears exactly once in the \text{from} list, then it does not have to be explicitly accompanied by a variable. Implicitly, the name of the variable is identical to the name of the table.

```
select \text{expression}_1, \ldots, \text{expression}_n 
from table_1, \ldots, table_n 
where condition
```

Example 4-28.

Print the names of the pairs of a student and an instructor who live together. (This includes an instructor who is also a student and lives alone.)

```
select STUDENT.LAST-NAME, INSTRUCTOR.LAST-NAME 
from STUDENT, INSTRUCTOR 
where STUDENT.ADDRESS = INSTRUCTOR.ADDRESS
```

Abbreviation: When there is only one table in the \text{from} list, then whenever \( T.\text{attribute} \) appears in the query, it may be shortened to \text{attribute} without the prefix \( T \).
Example 4-29.
Print the names and the addresses of all computer science students.

\[
\text{select } \text{LAST-NAME}, \text{ADDRESS} \\
\text{from STUDENT} \\
\text{where MAJOR-DEPARTMENT-MAIN-NAME} = \text{‘Computer Science’}
\]

Abbreviation: When the select list consists of all the attributes of the from tables, the select list may be abbreviated by "select *".

Example 4-30.
Print the names (last and first), the IDs, the birth-years, the major and minor departments, and the addresses of all computer science students.

\[
\text{select } * \\
\text{from STUDENT} \\
\text{where MAJOR-DEPARTMENT-MAIN-NAME} = \text{‘Computer Science’}
\]

Note:
- The output of a query is a partial instantaneous binary database. It can be printed as a table (in the common sense of the word table).
- Often, but not always, the output of a query is an instantaneous table in the sense of the Relational Model. This is not always true since the output of a query may contain identical rows, while a relational instantaneous table may not contain identical rows.

4.4.3. Basic aggregates
Basic aggregates are predefined functions which are applied to the whole output of a query. Syntactically, the functions are applied to the expression(s) in the select list.

1. count — when this function is applied to the select list, it replaces the output of the query by the number of rows in the output.

Example 4-31.
How many computer science students are there?

\[
\text{select count(*)} \\
\text{from STUDENT} \\
\text{where MAJOR-DEPARTMENT-MAIN-NAME} = \text{‘Computer Science’}
\]
2. **avg** — when the *select* list consists of only one expression, and the expression produces numerical values, the function **avg** replaces the output by the average of the values in the output.

   ```sql
   Example 4-32.
   What is the average grade in the Databases course?
   
   select avg(FINAL-GRADE)
   from COURSE-ENROLLMENT
   where COURSE-NAME-IN-KEY = ‘Databases’
   ```

3. **sum** — when the *select* list consists of only one expression, and the expression produces numerical values, the function **sum** replaces the output by the sum of the values in the output.

4. **max** — when the *select* list consists of only one expression, and the expression produces numerical values, the function **max** replaces the output by the maximum of the values in the output.

5. **min** — when the *select* list consists of only one expression, and the expression produces numerical values, the function **min** replaces the output by the minimum of the values in the output.

6. **distinct** — eliminates duplicate rows in the output. This function must be applied to the whole *select* list. (In many implementations, this function is called "**unique.**")

   ```sql
   Example 4-33.
   List the distinct addresses of the students. (Do not list the same address twice.)
   
   select distinct(ADDRESS)
   from STUDENT
   where TRUE
   ```

7. The function **distinct** can be combined with any other aggregate function. The function **distinct** is applied first, and then another function is applied to the result.

   ```sql
   Example 4-34.
   How many departments have minor students?
   
   select count (distinct MINOR-DEPARTMENT-MAIN-NAME)
   from STUDENT
   where TRUE
   ```
4.4.4. Nested queries

Query forms are represented in SQL by allowing some expressions to contain variables which are not defined (either explicitly or implicitly) in the from list. A query form would become a query if the expressions with undefined variables were replaced by constants. Query forms are used in SQL primarily in order to construct nested queries.

Example 4-35.
The following is not a query because it contains an undefined variable s. It is a query form, which would become a query if the expression s.ID-KEY were replaced by a constant, such as 345466.

```sql
select *
from COURSE-ENROLLMENT
where s.ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY
```

The nested queries are obtained in SQL by extending the syntax of the where condition by allowing the following subconditions within the condition:

1. exists query-form

This subcondition gives TRUE when the result of the query form is not empty — when it contains at least one row. (This subcondition is evaluated when all the variables on which the query form depends are interpreted.)

Example 4-36.
Find the names of the students who never took a course.

```sql
select LAST-NAME
from STUDENT
where not exists
    (select *
     from COURSE-ENROLLMENT
     where STUDENT.ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY)
```

Example 4-37.
List the instructor IDs and course names such that the instructor is the exclusive teacher of the course (i.e. no other instructors have offered the course).

```sql
select INSTRUCTOR-ID-in-key, COURSE-NAME-in-key
from COURSE-OFFERING co
```
where not exists

    select INSTRUCTOR-ID-in-key
    from COURSE-OFFERING co1
    where co.COURSE-NAME-in-key=co1.COURSE-NAME-in-key and
    not co.INSTRUCTOR-ID-in-key=co1.INSTRUCTOR-ID-in-key

2. expression in query-form-producing-only-one-value-per-row

This subcondition gives TRUE when the value of the expression constitutes a row in the output of the query-form

Example 4-38.
Find the names of the students who took at least one course.

    select LAST-NAME
    from STUDENT
    where
      (ID-KEY in
        select STUDENT-ID-IN-KEY
        from COURSE-ENROLLMENT
        where TRUE)

3. expression not in query-form-producing-only-one-value-per-row

This subcondition gives TRUE when the value of the expression does not constitute a row in the output of the query-form

Example 4-39.
Find the names of the students who never took a course.

    select LAST-NAME
    from STUDENT
    where
      (ID-KEY not in
        select STUDENT-ID-IN-KEY
        from COURSE-ENROLLMENT
        where TRUE)
4. \(<\text{expressions}>\text{ in } \text{query-form}\)

This subcondition gives TRUE when the values of the \textit{expressions} constitute a row in the output of the \textit{query-form}

\begin{verbatim}
Example 4-40.
Find the names of the students who may be spouses of instructors — those who have the same last name and address as an instructor.

\textbf{select} \textit{LAST-NAME, FIRST-NAME}
\textbf{from} \textit{STUDENT}
\textbf{where}
\textit{<LAST-NAME, ADDRESS> in}
\textbf{select} \textit{LAST-NAME, ADDRESS}
\textbf{from} \textit{INSTRUCTOR}
\textbf{where} \textit{TRUE}
\end{verbatim}

5. \(<\text{expressions}>\text{ not in } \text{query-form}\)

This subcondition gives TRUE when the values of the \textit{expressions} do not constitute a row in the output of the \textit{query-form}

\begin{verbatim}
Example 4-41.
Find the names of some students who are certainly not spouses of instructors.

\textbf{select} \textit{LAST-NAME, FIRST-NAME}
\textbf{from} \textit{STUDENT}
\textbf{where}
\textit{<LAST-NAME, ADDRESS> not in}
\textbf{select} \textit{LAST-NAME, ADDRESS}
\textbf{from} \textit{INSTRUCTOR}
\textbf{where} \textit{TRUE}
\end{verbatim}

6. \textit{query-form contains} \textit{query-form}

This subcondition gives TRUE when every row produced by the right query form is also produced by the left query form.

\begin{verbatim}
Example 4-42.
Find the names of the students who took all the courses.

\textbf{select} \textit{LAST-NAME, FIRST-NAME}
\end{verbatim}
from STUDENT
where
    select COURSE-NAME-IN-KEY
from COURSE-ENROLLMENT
where STUDENT.ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY
contains
    select NAME-KEY
from COURSE
where TRUE

7. expression comparison query-form-producing-only-one-value

The allowed comparisons are: =, <, >, >=, <=, <>.

Example 4-43.
Find the names of the students who took more than 1000 course offerings.

    select LAST-NAME, FIRST-NAME
    from STUDENT
where
    1000 <
    select count (*)
    from COURSE-ENROLLMENT
where STUDENT.ID-KEY = COURSE-ENROLLMENT.STUDENT-ID-IN-KEY

8. expression comparison any query-form

This subcondition is TRUE if the comparison with at least one row of the query-form’s output is TRUE.

Example 4-44.
Find the names of the students who studied something in the first 20 calendar years of their life.

    select LAST-NAME, FIRST-NAME
    from STUDENT
where
9. expression comparison all query-form

This subcondition is TRUE if the comparison with every one of the query-form’s output rows is TRUE.

Example 4-45.
Find the names of the students who studied nothing (as far as the database knows) in the first 20 calendar years of their life.

```sql
select LAST-NAME, FIRST-NAME
from STUDENT
where
  BIRTH-YEAR + 20 <= all
```

Example 4-46.
List the instructor IDs and course names such that the instructor is the exclusive teacher of the course (i.e. no other instructors have offered the course).

```sql
select INSTRUCTOR-ID-in-key, COURSE-NAME-in-key
from COURSE-OFFERING co
where INSTRUCTOR-ID-in-key = all
```
Example 4-47.

An alternative code for the above query:

```
select INSTRUCTOR-ID-in-key, COURSE-NAME-in-key
from COURSE-OFFERING co
where 1 =
    select count (distinct INSTRUCTOR-ID-in-key)
from COURSE-OFFERING co1
where co.COURSE-NAME-in-key=co1.COURSE-NAME-in-key
```

4.4.5. Grouping of rows

The aggregate functions can be applied to subsets of rows produced by `select`. For this purpose, the rows resulting from `select` can be partitioned into groups according to the values of some attributes.

Syntax:

```
select expression_1, . . . , expression_n
from table_1 var_1, . . . , table_n var_n
where condition
    group by attribute_1, . . . , attribute_k
```

Each `attribute_i` has the form

```
variable.attribute-name
```

When no ambiguity arises, the table-name can be used instead of the variable:

```
table-name.attribute-name
```

When no ambiguity further arises, the table-name can be omitted:

```
attribute-name
```

Meaning:

The rows satisfying the `condition` are combined into groups so that in each group the attributes of the grouping have constant values; that is, two rows \( r_1 \) and \( r_2 \) are in the same group if and only if

\[
\begin{align*}
    r_1.attribute_1 &= r_2.attribute_1 \\
    r_1.attribute_2 &= r_2.attribute_2 \\
    & \quad \text{and} \\
    r_1.attribute_k &= r_2.attribute_k
\end{align*}
\]

For every group, only one cumulative row is produced in the result. The resulting cumulative row is obtained by evaluation of the expressions of the `select` clause. The aggregates in those expressions are interpreted as applying not to the whole output but
only to the rows comprising one group.

Example 4-48.
For every department, list the number of instructors it employs.

```sql
select DEPARTMENT-MAIN-NAME-in-key, count(INSTRUCTOR-ID-in-key)
from WORK
group by DEPARTMENT-MAIN-NAME-in-key
```

Example 4-49.
For every student who took classes in a summer, for every instructor who gave grades to the student in a summer, print the average of the summer grades given by the instructor to the student.

```sql
select STUDENT-ID-in-key, INSTRUCTOR-ID-in-key, avg(FINAL-GRADE)
from COURSE-ENROLLMENT
where SEASON = 'Summer'
group by STUDENT-ID-in-key, INSTRUCTOR-ID-in-key
```

Some of the groups produced by `group by` can be screened out according to the values of aggregate functions applied to the group. A group screening condition can be specified in a `having` clause as follows.

```sql
select expressions
from tables
where condition-on-the-source-rows-of-the-tables
group by attributes
having condition-on-the-groups
```

The `condition-on-the-groups` is a Boolean expression. The aggregate functions appearing in this condition apply to the rows comprising one group.

Example 4-50.
What departments employ more than 100 instructors each?

```sql
select DEPARTMENT-MAIN-NAME-in-key
from WORK
where true
group by DEPARTMENT-MAIN-NAME-in-key
```
Example 4-51.
For every student who took classes in a summer, for every instructor who gave grades to the student in a summer, so that the average of the summer grades given by the instructor to the student is greater than 60, print the average of the summer grades given by the instructor to the student.

```sql
select STUDENT-ID-in-key, INSTRUCTOR-ID-in-key, avg(FINAL_GRADE)
from COURSE_ENROLLMENT
where SEASON = 'Summer'
group by STUDENT-ID-in-key, INSTRUCTOR-ID-in-key
having avg(FINAL_GRADE) > 60
```

Example 4-52.
What students have taken classes with every instructor?

```sql
select STUDENT-ID-in-key
from COURSE_ENROLLMENT
group by STUDENT-ID-in-key
having count(*) =
(select count(*) from INSTRUCTOR )
```

4.4.6. Sorting
The output of an SQL query can be sorted for the purpose of printing in any desired order or for delivery to an application program in a desired order. This is accomplished by an `order by` clause specifying one or more attributes to sort by:

```
query
    order by attributes
```
When more than one sorting attribute is given, the output’s primary order is according to the first attribute, then according to the second, and so on.

Example 4-53.
List all computer science majoring students sorted by their minors.

```sql
select *
from STUDENT
```
where MAJOR-DEPARTMENT-MAIN-NAME = ‘Computer Science’

order by MINOR-DEPARTMENT-MAIN-NAME

4.4.7. Update transactions

Update transactions can be specified in SQL.

1. Deleting a set of rows from a table.
   
   `delete from table
   where condition-on-rows-to-be-deleted`

   **Example 4-54.**
   Delete the student whose ID is 11111.
   
   `delete from STUDENT
   where ID-key = 11111`

   **Example 4-55.**
   Delete all music majors.
   
   `delete from STUDENT
   where MAJOR-DEPARTMENT-MAIN-NAME = ‘Music’`

2. Inserting a row into a table.
   
   `insert into table
   attribute_1, . . . , attribute_n
   values
   value_1, . . . , value_n`

   The attributes of the table which are not specified in the `insert` command are set to `null`
   values for the row being inserted.

   **Example 4-56.**
   Let the instructor whose ID is 22222 work in the department whose main name is Arts.
   
   `insert into WORK
   INSTRUCTOR-ID-in-key, DEPARTMENT-MAIN-NAME-in-key
   values`
3. Inserting a set of rows into a table. A set of rows to be inserted can be defined as the result of a query.

\[
\text{insert into } \text{table} \quad \text{attribute}_1, \ldots, \text{attribute}_n \\
\text{query}
\]

Example 4-57.
Let all physics instructors work also in Arts.

\[
\text{insert into WORK} \\
\text{INSTRUCTOR-ID-in-key, DEPARTMENT-MAIN-NAMe-in-key} \\
\text{select INSTRUCTOR-ID-in-key, 'Arts'} \\
\text{from WORK} \\
\text{where DEPARTMENT-MAIN-NAME-in-key = 'Physics'}
\]

4. Modifying the values of some attributes in a set of rows of a table. The set of rows can be specified by a \textit{where} condition. The new values can be specified as constants or as expressions using the old values of the row being updated.

\[
\text{update table} \\
\text{set} \quad \text{attribute}_i = \text{expression}_i, \ldots, \text{attribute}_n = \text{expression}_n \\
\text{where condition}
\]

Example 4-58.
Decrease by 10 percent all grades above 90.

\[
\text{update COURSE-ENROLLMENT} \\
\text{set FINAL-GRADE = FINAL-GRADE * 0.9} \\
\text{where FINAL-GRADE > 90}
\]

Risque-92-DDS p. 4.30
4.4.8. DDL

SQL has a data definition capability.

**Specification of a table**

```sql
create table table-name
    attribute_1 data-type_1, ... , attribute_n data-type_n
```

*Example 4-59.*

```sql
create table QUARTER
YEARN-in-key Integer
SEASON-in-key String
```

**Specification of a userview table**

```sql
create view new-table-name
    attribute_1, ... , attribute_n
as select-command
```

*Example 4-60.*

```sql
create view TAUGHT
STUDENT-ID HIS-TEACHER-ID
as select STUDENT-ID-in-key, INSTRUCTOR-ID-in-key
from COURSE-ENROLLMENT
```

4.4.9. SQL extension of Pascal

SQL can be used not only interactively but also as a DML extension of a programming language. This section shows how SQL can be embedded in Pascal. The embedding in other programming languages is similar.

**Host variables in SQL statements**

Wherever a constant can appear in SQL, a host program variable can appear instead. Before the SQL statement is performed, the variable is evaluated to give a value. To distinguish between host program variables and SQL variables, the host variables are preceded by a colon (:).

*Example 4-61.*

Create a new course whose name is in the Pascal variable `course-name`.

```sql
insert into COURSE
```
Example 4-62.
For each standard input line create a new course whose name is the string appearing in that line.

```
var course-name: String;
begin
  while not eof(Input) do begin
    readln (course-name);
    insert into COURSE
      NAME-key values :course-name
  end
end.
```

Retrieving a row of values from the database

If we anticipate that a select command will retrieve exactly one row of data, we can have this data placed into variables of the host program by the following command.

```
select-command into host-variables
```

Example 4-63.
(* Get the total number students born in 1980 into the variable myvar. *)

```
select count (*)
from STUDENT
where BIRTH-YEAR = 1980
into :myvar
```

Processing of multirow output of a query

The program can retrieve a set of rows of data from the database.

- Such a set of rows can be defined as the output of a select command. The program would then process the retrieved rows, one row at a time.

Example 4-64.
```
select BIRTH-YEAR
```
from STUDENT
where MAJOR-DEPARTMENT-MAIN-NAME = ‘Management’;

- To scan such rows in a program, SQL defines a cursor, a logical pointer to the current row. The declaration of a cursor defines a query.

Example 4-65.

declare current-student cursor for
    select BIRTH-YEAR
    from STUDENT
    where MAJOR-DEPARTMENT-MAIN-NAME = ‘Management’;

- The opening of the cursor performs the query.

Example 4-66.

open current-student;

- The fetch command then brings to the program one row each time and advances the cursor to point to the next row.

Example 4-67.

fetch current-student into :birth-year;

- When a fetch is attempted beyond the last row in the output of a query, the special variable sqlstatus is set to the value of the special constant not-found.

Example 4-68.

(* Print the logarithm of the birth year of every management student (major) *)

declare current-student cursor for
    select BIRTH-YEAR
    from STUDENT
    where MAJOR-DEPARTMENT-MAIN-NAME = ‘Management’;

open current-student;

repeat
    fetch current-student into :birth-year;

Risque-92-DDS p. 4.33
if sqlstatus ≠ not-found then writeln(log(birth-year))
until sqlstatus = not-found

The relevant commands are:

a. declare cursor-name cursor for select-command
b. open cursor-name
c. fetch cursor-name into host-variables

Update and delete of fetched rows

After a row has been fetched, it can be updated or deleted provided the row belongs to one table of the schema (and not to a join of tables).

a. update table-name
   set
   
   \text{attribute}_1 = \text{expression}_1, \ldots, \text{attribute}_n = \text{expression}_n

   \text{where current of cursor-name}

Example 4-69.

(* Replace the birth year of every management student (major) by the logarithm of the birth year. *)

\begin{verbatim}
declare current-student cursor for
   select BIRTH-YEAR
   from STUDENT
   where MAJOR-DEPARTMENT-MAIN-NAME = 'Management';

open current-student;
repeat
   fetch current-student into :birth-year;
   birth-year := log (birth-year);
   if sqlstatus ≠ not-found then
      update STUDENT
      set BIRTH-YEAR = :birth-year
      where current of current-student
   until sqlstatus = not-found
\end{verbatim}
**Example 4-70.**

(* Display the name of every student; prompt the user whether the student should be deleted; if the user replies ‘yes’ — delete the student. *)

```
declare current-student cursor for
    select LAST-NAME, FIRST-NAME
    from STUDENT

open current-student;
repeat
    fetch current-student into :last-name, :first-name;
    if sqlstatus ≠ not-found then begin
        writeln (‘Would you like to delete ’, last-name, ‘ ’, first-name, ‘?’);
        readln (answer);
        if answer='yes' then
            delete from STUDENT
            where current of current-student
        end
    until sqlstatus = not-found
```

### 4.5. Expressive Power of Relational Query Languages

Ignoring minor differences in expressiveness, such as the output of identical rows, the following languages have approximately equal power. A query that can be expressed in one language can also be expressed in the others:

- Tuple-oriented Relational Calculus without aggregate functions
- Relational Algebra
- SQL without aggregate functions

The aggregate extension of the Predicate Calculus and SQL with aggregate functions have a higher power.

The structured extension of Pascal, being a general-purpose programming language, has a much higher power of expressiveness.
Reference Schemas

The following are the semantic and relational schemas for the university case study application. These schemas are referred to in most of the examples in this text.

Figure Ref-1. A semantic schema for a university application.

Rishe-92-DDS p. 4.36
Figure Ref-2. A relational schema for the university application.