structure, and so on. For each attribute of the relation, the system may maintain a tuple recording the relation identifier, attribute name, type, size, and so forth. Different DBMSs keep different amounts of information in the directory relations. However, because the implementation is usually as relations, the same data manipulation language that the DBMS supports can be used to query these relations.

In this section we briefly examined some implementation issues. Implementors of databases and DBMSs must be aware that there exists much more detail than that contained in the model.

4.7 Summary

In this chapter we studied the relational data model, consisting of the relational data structure, relational operations, and the relational integrity rules. This model borrows heavily from set theory and is based on sound fundamental principles. Relational operations are applied to relations, and the result is a relation.

Conceptually, a relation can be represented as a table; each column of the table represents an attribute of the relation and each row represents a tuple of the relation. Mathematically a relation is a correspondence between a number of sets and is a subset of the cartesian product of these sets. The sets are the domains of the attributes of the relation.

Duplicate tuples are not permitted in a relation. Each tuple can be identified uniquely using a subset of the attributes of the relation. Such a minimum subset is called a key (primary) of the relation. The unique identification property of the keys is used to capture relationships between entities. Such a relationship is represented by a relation that contains a key for each entity involved in the relationship.

Relational algebra is a procedural manipulation language. It specifies the operations and the order in which they are to be performed on tuples of relations. The result of these operations is also a relation. The relational algebraic operations are union, difference, cartesian product, intersection, projection, selection, join, and division.

Relational calculus consists of two distinct calculi, tuple calculus and domain calculus. In relational calculus queries are expressed using variables, a formula involving these variables, and compatible constants. The query expression specifies the result relation to be obtained without specifying the mechanism and the order used to evaluate the formula. It is up to the underlying database system to transform these nonprocedural queries into equivalent, efficient, procedural queries. In relational tuple calculus the variables represent tuples from specific relations; in domain calculus the variables represent values from specific domains.

Since relational calculus specifies queries as formulas, it is important that these formulas generate result relations of finite cardinality in an acceptable period of time. This in turn requires that the formulas be defined on a finite domain and the result be within that domain. The domain consists of relations and constants appearing in the formulas. Such formulas are called safe. With a safe formula, it is possible to convert a query expression from one representation to another.

In the next chapter we consider a number of commercial query languages based on relational algebra and calculus.

Key Terms

cardinality
degree
arity
projecting
join
set
members
intension
extension
union
intersection
cartesian product
difference
atomic domain
application-independent domain
application-dependent domain
structured domain
composite domain
n-tuple
projection
relation scheme
unique identification
nonredundancy
prime attribute
associative relation
foreign key
target
domino deletion
cascading deletion
union compatible
set-theoretic union
restriction operation
theta join
equi-join
natural join
relational calculus
predicate calculus
predicate
one-place predicate
monadic predicate
two-place predicate
atomic formula
well-formed formula (wff)
bound variable
free variable
closed
open
tuple calculus
atom
domain calculus
safe
fragmentation
tuple identifier

Exercises

4.1 For the relations P and Q shown in Figure N, perform the following operations and show the resulting relations.

(a) Find the projection of Q on the attributes (B,C).
(b) Find the natural join of P and Q on the common attributes.
(c) Divide P by the relation that is obtained by first selecting those tuples of Q where the value of B is either b_1 or b_2 and then projecting Q on the attributes (C,D).

Figure N For Exercise 4.1.

P

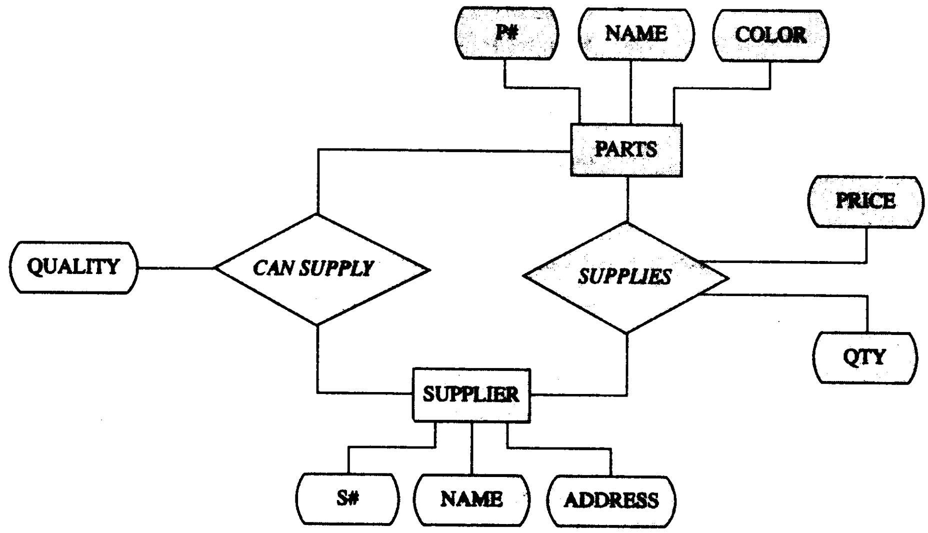
A	B	C	D
a_1	b_2	c_2	d_2
a_2	b_1	c_1	d_2
a_1	b_1	c_2	d_1
a_2	b_1	c_2	d_2
a_1	b_2	c_1	d_2
a_3	b_1	c_2	d_1
a_1	b_2	c_2	d_2
a_2	b_1	c_1	d_2
a_1	b_3	c_2	d_2

Q

B	C	D
b_1	c_1	d_2
b_3	c_1	d_2
b_2	c_2	d_1
b_1	c_1	d_2
b_3	c_2	d_2

4.2 Given the E-R diagram in Figure O, give the most suitable relational database scheme to implement this database. For each relation, choose a suitable name and list corresponding attributes, underlining the primary key. For each relation, also identify the foreign keys. Could any problems result as a consequence of tuple additions, deletions, or updates?

Figure O For Exercise 4.2.



4.3 For the database of Figure O, write relational algebra and calculus expressions to pose the following queries:

(a) Get the supplier details and the price of bolts for all suppliers who supply 'bolts'.
(b) Find details of parts that suppliers who supply 'bolts' costing less than $0.01 are capable of supplying, with the parts being of a quality better than 'x'.

4.4 Given the relational schemes:

ENROLL *(S#, C#, Section)*—*S#* represents student number
TEACH *(Prof, C#, Section)*—*C#* represents course number
ADVISE *(Prof, S#)*—*Prof* is thesis advisor of S#
PRE_REQ *(C#, Pre_C#)*—*Pre_C#* is prerequisite course
GRADES *(S#, C#, Grade, Year)*
STUDENT *(S#, Sname)*—*Sname* is student name

Give queries expressed in relational algebra, tuple calculus, and domain calculus for the following queries:

(a) List all students taking courses with Smith or Jones.
(b) List all students taking at least one course that their advisor teaches.
(c) List those professors who teach more than one section of the same course.
(d) List the courses that student "John Doe" can enroll in, i.e., has passed the necessary prerequisite courses but not the course itself.

4.5 An orchestra database consists of the following relations:

CONDUCTS *(Conductor, Composition)*
REQUIRES *(Composition, Instrument)*
PLAYS *(Player, Instrument)*
LIKES *(Player, Composition)*

Give relational algebra, tuple calculus, and domain calculus queries for the following queries?

(a) List the players and their instruments who can be part of the orchestra when Letitia Melody conducts.
(b) From the above list of players, identify those who would like the composition they are to play.

4.6 Give the equivalent

(a) English statement,
(b) domain calculus, and
(c) algebra

expressions for the following tuple calculus query:

$$\{t \mid t \in rel_1 \wedge \exists s(s \in rel_2 \wedge (s.c = t.b))\}$$

given the relations $rel_1(A,B)$ and $rel_2(C,D)$.

4.7 Convert the following domain calculus query

$$\{<A,B> \mid <A,B> \in rel_1 \wedge B='B_1' \vee B='B_2'\}$$

into

(a) an English statement
(b) relational algebra
(c) tuple calculus.

4.8 Investigate the physical implementation details of a relational DBMS with which you are familiar. Under what circumstances would any file organization not supported by the system be beneficial?

4.9 An inverted file management system allows for the definition of inverted files and supports queries of the form "List records (or tuples) where the attribute_name has value x," and a Boolean combination of such queries. Discuss how the relational algebra operations can be handled using such a system.

4.10 Consider the queries in Examples 4.44 through 4.49. Rewrite the queries in tuple calculus; however, use the quantifier $\forall$ instead of $\exists$ and vice versa.

4.11 Consider the queries in Examples 4.52 through 4.57. Rewrite the queries in domain calculus; however, use the quantifier $\forall$ instead of $\exists$ and vice versa.

4.12 Using the relations ASSIGNED_TO, EMPLOYEE, and PROJECT given in the text, generate the following queries in relational algebra.

(a) Acquire details of the projects for each employee by name.
(b) Compile the names of projects to which employee 107 is assigned.
(c) Access all employees assigned to projects whose chief architect is employee 109.
(d) Derive the list of employees who are assigned to all projects on which employee 109 is the chief architect.
(e) Get all project names to which employee 107 is not assigned.
(f) Get complete details of employees who are assigned to projects not assigned to employee 107.

4.13 Repeat Exercise 4.12 using tuple calculus.

4.14 Repeat Exercise 4.12 using domain calculus.

4.15 Give the tuple calculus expressions for the relational algebraic operation of (a) the union of two relations P and Q, (b) the difference $P - Q$, (c) the projection of relation P on the attribute X, (d) the selection $\sigma_B(P)$, (e) the division of relation P by Q, i.e., $P \div Q$.

4.16 Consider the following relations concerning a driving school. The primary key of each relation is in boldface.

STUDENT : ***(St_Name, Class#***, *Th_Mark, Dr_Mark)*
STUDENT_DRIVING_TEACHER : ***(St_Name, Dr_T_Name)***
TEACHER_THEORY_CLASS : ***(Class#***, *Th_T_Name)*
TEACHER_VEHICLE : ***(Dr_T_Name, License#)***
VEHICLE : ***(License#***, *Make, Model, Year)*

A student takes one theory class as well as driving lessons and at the end of the session receives marks for theory and driving. A teacher may teach theory, driving, or both. Write the following queries in relational algebra, domain calculus, and tuple calculus.

(a) Find the list of teachers who teach theory and give driving lessons on all the vehicles.
(b) Find the pairs of students satisfying the following conditions.

They have the same theory mark and
They have different theory teachers and
They have the same driving mark and
They have different driving teachers

(c) Find the list of students who are taught neither theory lessons nor driving lessons by "Johnson" (teacher).
(d) Find the list of students who have better marks than "John" in both theory and driving.
(f) Find the list of students who have more marks than the average theory mark of class 8 (Class#).
(g) Find the list of teachers who can drive all the vehicles.

4.17 Comment on the correctness of the following relational calculus solutions to the query: "Get employee numbers of employees who do not work on project COMP453."

(a)
$$\{t[Emp\#] \mid t \in ASSIGNED\_TO \wedge \forall u(u \in ASSIGNED\_TO \wedge t[Emp\#] = u[Emp\#] \wedge u[Project\#] \neq 'COMP453')\}$$

(b)
$$\{e \mid \exists p(<p,e> \in ASSIGNED\_TO \wedge \forall p_1,e_1(<p_1,e_1> \in ASSIGNED\_TO \wedge p_1 = 'COMP453' \wedge e \neq e_1)))\}$$

4.18 Comment on the correctness of the following relational calculus solutions to the query: "Compile a list of employee numbers of employees who work on all projects."

(a)
$$\{t[Emp\#] \mid t \in ASSIGNED\_TO \wedge \exists p,u(p \in PROJECT \wedge u \in ASSIGNED\_TO \wedge p[Project\#] = u[Project\#] \wedge t[Emp\#] = u[Emp\#]$$

(b)
$$\{e \mid \forall p_2(<p_2,n_2,c_2> \in PROJECT \wedge <p,e> \in ASSIGNED\_TO$$

$$\rightarrow \exists\, p_1,e_1\ (\langle p_1,e_1\rangle \in \text{ASSIGNED\_TO}$$
$$\wedge\ p_1 = p_2 \wedge e = e_1))\}$$

(c) $\{e \mid \exists p\ (\langle p,e\rangle \in \text{ASSIGNED\_TO}$
$$\wedge\ \forall\, p_1,e_1\ (\langle p_1,e_1\rangle \notin \text{ASSIGNED\_TO}$$
$$\vee\ p_1 \neq \text{COMP453} \vee e_1 \neq e))\}$$

(d) $\{e \mid \exists p\ (\langle p,e\rangle \in \text{ASSIGNED\_TO}$
$$\wedge\ \forall\, p_1(\langle p_1,n_1,c_1\rangle \in \text{PROJECT}$$
$$\rightarrow \langle p_1,e\rangle \in \text{ASSIGNED\_TO}))\}$$

4.19 Comment on the correctness of the following relational calculus solution to the query: "Acquire the employee numbers of employees, other than employee 107, who work on at least one project that employee 107 works on."

$$\{e \mid \exists p,p_1,e_1\ (\langle p,e\rangle \in \text{ASSIGNED\_TO}$$
$$\wedge\ \langle p_1,e_1\rangle \in \text{ASSIGNED\_TO}$$
$$\wedge\ p_1 = p \wedge e \neq e_1 \wedge e_1 = 107)\}$$

Bibliographic Notes

The original concept of the use of relations to represent data was presented by Levien and Maron (Levi 67). The formal relational model as we know it today, however, was first proposed by E. F. Codd (Codd 70). Relational algebra was defined by Codd in his original paper and relational calculi in a subsequent paper (Codd 72). Since Codd's original article, the relational model has been extensively studied and is covered in most database texts, including Date (Date 86), Korth and Silberschatz (Kort 86), Maier (Maie 83), and Ullman (Ullm 82). Maier's text gives a comprehensive theoretical treatment of the relational model.

Bibliography

(Beer 77) C. Beeri, R. Fagin, & J. H. Howard, "A Complete Axiomisation for Functional and Multivalued Dependencies," Proc. ACM SIGMOD Record Conference, Toronto, Aug. 1977, pp. 47–61.

(Beer 78) C. Beeri, P. A. Bernstein, & N. Goodman, "A Sophisticate's Introduction to Database Normalization Theory," Proc. 4th International Conference on Very Large Data Bases, Berlin, 1978, pp. 113–123.

(Bern 76) P. A. Bernstein, "Synthesizing Third Normal Form Relations from Functional Dependencies," *ACM Transactions on Database Systems* 1(4), 1976, pp. 277–298.

(Brod 82) M. L. Brodie, & J. W. Schmidt, eds., "Final Report of the ANSI/X3/SPARC DBS-SG Relational Database Task Group," SPARC-81-690, *ACM SIGMOD Record* 12(4), 1982, pp. 1–62.

(Buss 83) V. Bussolati, S. Ceri, V. De Antenollis, & B. Zonta, "Views Conceptual Design," in S. Ceri, ed., *Methodology and Tools for Data Base Design*. North Holland, Amsterdam 1983, pp. 25–55.

(Codd 70) E. F. Codd, "A Relational Model for Large Shared Data Banks," *Communications of the ACM* 13(6), 1970, 377–387.

(Codd 72) E. F. Codd, "Relational Completeness of Data Base Sublanguages," in R. Randall, ed., *Data Base Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1972, pp. 65–98.

(Codd 81) E. F. Codd, "Data Models in Database Management," *ACM SIGMOD Record* 11(2), 1981.

(Codd 82) E. F. Codd, "Relational Database: A Practical Foundation for Productivity," 1981 ACM Turing Award Lecture, *Communications of the ACM* 25(2), 1982, pp. 109–117.

(Date 86) C. J. Date, "An Introduction to Database Systems," 4th ed. Reading, Mass: Addison Wesley, 1986.

(Fagi 77) R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases," *ACM Transactions of Database Systems* 2(3), 1977, pp. 262–278.

(Gall 78) H. Gallaire & J. Minker, *Logic and Databases*. New York: Plenum Press, 1978.

(Kort 86) H. F. Korth & A. Silberschatz, *Database System Concepts*, New York: McGraw-Hill, 1986.

(Kowa 79) R. Kowalski, *Logic for Problem Solving*, New York: North-Holland, 1979.

(Lacr 77) M. Lacroix & A. Pirotte, "Domain-Oriented Relational Languages," Proc. 3rd International Conference on Very Large Data Bases, October 6-8, 1977. Tokyo, IEEE, New York, pp. 370–378.

(Levi 67) R. Levien, & M. E. Maron, "A Computer System for Inference Execution and Data Retrieval, *Communications of the ACM* 10(11), 1967, pp. 715–721.

(Lum 79) V. Lum et al., 1978 New Orleans Data Base Design Workshop Report, IBM Yorktown Heights (RJ 2554), 1979.

(Maie 83) D. Maier, *The Theory of Relational Databases*," Rockville, MD: Computer Science Press, 1983.

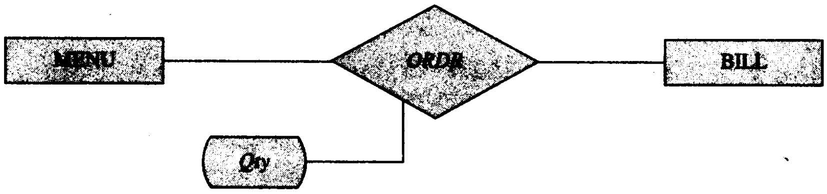
(Niem 84) T. Niemi & K. Jarvelin, "A Straightforward Formalization of the Relational Model," *ACM SIGMOD Record* 14(1), 1984, pp. 15–38.

(Piro 82) A. Pirotte, "A Precise Definition of Basic Relational Notions and of the Relational Algebra," *ACM SIGMOD Record* 13(1), 1982, pp. 30–45.

(Ullmn 82) J. D. Ullman, *Principles of Database Systems*, 2nd ed. Rockville, Md: Computer Science Press, 1982.

(Yang 86) C. C. Yang, *Relational Databases*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

Figure 5.1 The *ORDR* relationship.



BILL *(Bill#, Day, Table#, Waiter#, Total, Tip)*

Bill#: integer—unique bill identifier
Day: date—in yyyymmdd unsigned decimal digits format
Table#: integer—table number
Waiter#: integer—employee identifier
Total: real—total amount
Tip: real

ORDR *(Bill#, Dish#, Qty)*

Bill#: integer—bill identifier
Dish#: integer—dish identifier
Qty: integer—number of dish ordered by client

The *DUTY_ALLOCATION* relationship (Figure 5.3) between various positions (POSITION) and employees (EMPLOYEE) in a restaurant can be described by the attributes *Day* and *Shift*. Each position in the restaurant is defined by a unique *Posting_No* and requires a (minimum) skill specified by *Skill*. The structure of the tables for these entities and the relationship is given below. Some tuples from these relations are given in Figure 5.4.

Figure 5.2 Some tuples from the MENU, BILL, and ORDR relations.

MENU

Dish#	*Dish_Description*	*Price*
50	Coffee	2.50
100	Scrambled eggs	7.50
200	Special du jour	19.50
250	Club sandwich	10.50
300	Pizza	14.50

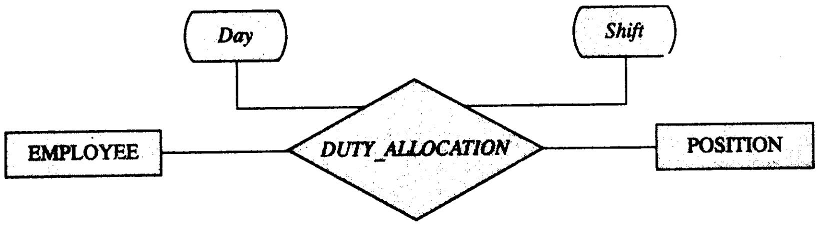
ORDR

Bill#	*Dish#*	*Qty*
9234	50	2
9234	250	2
9235	300	1

BILL

Bill#	*Table#*	*Day*	*Waiter#*	*Total*	*Tip*
9234	12	19860419	123456	26.00	3.90
9235	17	19860420	123461	14.50	2.20

Figure 5.3 The *DUTY_ALLOCATION* relationship.



EMPLOYEE *(Empl_No, Name, Skill, Pay_Rate)*

Empl_No: integer—unique identifier
Name: string—employee's name
Skill: string—employee's skill
Pay_Rate: real—hourly pay rate

POSITION *(Posting_No, Skill)*

Posting_No: integer—unique position identifier
Skill: string—skill required for the position

Figure 5.4 Some Tuples from *EMPLOYEE, POSITION, DUTY_ALLOCATION relations.*

EMPLOYEE

Empl_No	*Name*	*Skill*	*Pay_Rate*
123456	Ron	waiter	7.50
123457	Jon	bartender	8.79
123458	Don	busboy	4.70
123459	Pam	hostess	4.90
123460	Pat	bellboy	4.70
123461	Ian	maître d'	9.00
123471	Pierre	chef	14.00
123472	Julie	chef	14.50

POSITION

Posting_No	*Skill*
321	waiter
322	bartender
323	busboy
324	hostess
325	maître d'
326	waiter
350	chef
351	chef

DUTY_ALLOCATION

Posting_No	*Empl_No*	*Day*	*Shift*
321	123456	19860419	1
322	123457	19860418	2
323	123458	19860418	1
321	123461	19860420	2
321	123461	19860419	2
350	123471	19860418	1
323	123458	19860420	3
351	123471	19860419	1

alter table existing-table-name
    add column-name data-type [. . . .]

alter table EMPLOYEE
    add *Phone_Number* **decimal** (10)

The **create index** statement allows the creation of an index for an already existing relation. The columns to be used in the generation of the index are also specified. The index is named and the ordering for each column used in the index can be specified as either ascending or descending. The **cluster** option could be specified to indicate that the records are to be placed in physical proximity to each other. The **unique** option specifies that only one record could exist at any time with a given value for the column(s) specified in the statement to create the index. (Even though this is just an access aid and a wrong place to declare the primary key.) Such columns, for instance, could form the primary key of the relation and hence duplicate tuples are not allowed. One case is the ORDR relation where the key is the combination of the attribute *Bill#*, *Dish#*. In the case of an existing relation, an attempt to create an index with the unique option will not succeed if the relation does not satisfy this uniqueness criterion. The syntax of the create index statement is shown below:

create [unique] index name-of-index
    on existing-table-name
        (column-name [**ascending** or **descending**]
        [,column-name[order] . . .])
    [cluster]

The following statement causes an index called *empindex* to be built on the columns *Name* and *Pay_Rate*. The entries in the index are ascending by *Name* value and descending by *Pay_Rate*. In this example there are no restrictions on the number of records with the same *Name* and *Pay_Rate*.

create index *empindex*
    on EMPLOYEE (*Name* **asc**, *Pay_Rate* **desc**);

An existing relation or index could be deleted from the database by the **drop** SQL statement. The syntax of the drop statement is as follows:

drop table existing-table-name;
drop index existing-index-name;

5.3 Data Manipulation: SQL

In this section we present the data manipulation statements supported in SQL. Examples of their usage are given in subsequent sections. SQL provides the following basic data manipulation statements: select, update, delete, and insert.

Select Statement

The **select** statement, the only data retrieval statement in SQL, specifies the method of selecting the tuples of the relation(s). The tuples processed are from one or more relations specified by the from clause of the select statement; the selection predicates are specified by the where clause. The select statement could also specify the projection of the target tuples. Do not confuse the select verb of SQL with σ, the select operation of relational algebra. The difference is that the select statement entails selection, joins, and projection, whereas σ is a simple selection.

The syntax of the select statement is as follows:

select [distinct] <target list>
from <relation list>
[**where** <predicate>]

The **distinct** option is used in the select statement to eliminate duplicate tuples in the result. Without the distinct option duplicate tuples may appear in the result.

The <target list> is a method of specifying a projection operation of the result relation. It takes the form:

<target list> : : = <attribute name> [,<target list>]

The **from** clause specifies the relations to be used in the evaluation of the statement. It includes a relation list:

<relation list> : : = <relation name> [<tuple variable>]
[,<relation list>]

A tuple variable is an identifier; the domain of the tuple variable is the relation preceding it.

The **where** clause is used to specify the predicates involving the attributes of the relation appearing in the from clause.

An example of the use of a simple form of select to find the values for the attribute *Name* in the employee relation is given below:

select *Name*
from EMPLOYEE

The result of this select operation is a projection of the EMPLOYEE relation on the attribute *Name*. Unlike the theoretical version of projection, this projection contains duplicate tuples. The reason for not eliminating these duplicates is the large amount of processing time required to do so. If the theoretical equivalent is desired, however, the distinct clause is added to the select statement, as shown below:

select distinct *Name*
from EMPLOYEE

The predicates used to specify selection are added to a select statement by the use of the where clause. Additional features and examples of the select statement will be discussed in following sections.

Update Statement

The **update** statement is used to modify one or more records in a specified relation. The records to be modified are specified by a predicate in a where clause and the new value of the column(s) to be modified is specified by a **set** clause. The syntax of the update statement is shown on the next page.

update <relation> **set** <target_value_list>
    [**where** <predicate>]

where the <target value list> is of the form:

<target value list> : : = <attribute name> = <value expression>
    [,<target value list>]

The following statement changes the *Pay_Rate* of the employee Ron in the EMPLOYEE relation of Figure 5.4:

update EMPLOYEE
    set *Pay_Rate* = 7.85
    where *Name* = 'Ron'

Delete Statement

The **delete** statement is used to delete one or more records from a relation. The records to be deleted are specified by the predicate in the where clause. The syntax of the delete statement is given below:

delete <relation> [**where** <predicate>]

The following statement deletes the tuple for employee Ron in the EMPLOYEE relation of Figure 5.4.

delete EMPLOYEE
    where *Name* = *'Ron'*

If the where clause is left out, all the tuples in the relation are deleted. In this case, the relation is still known to the database although it is an empty relation. A relation along with its tuples could be deleted by the drop statement.

Insert Statement

The **insert** statement is used to insert a new tuple into a specified relation. The value of each field of the record to be inserted is either specified by an expression or could come from selected records of existing relations. The format of the insert statement is given below:

insert into <relation>
values (<value list>)

where the <value list> takes the form:

<value list> : : = <value expression> [,<value list>]

In another form of the insert statement, a list of attribute names whose values are included in the <value list> are specified:

insert into <relation> (<target list>)
    values (<value list>)

and the <target list> takes the form:

<target list> : : = <attribute name> [,<target list>]

The **value** clause can be replaced by a select statement, which is evaluated, and the result is inserted into the relation specified in the insert statement.

The following statement reinserts a tuple for the employee Ron in the EMPLOYEE relation of Figure 5.4:

```
insert into EMPLOEE
    values (123456, 'Ron', 'waiter', 7.50)
```


5.3.1 Basic Data Retrieval

The SQL mapping operation basically consists of a selection and join followed by a projection. The select verb of SQL is used to represent this mapping operation.

Example 5.1

Here we give two simple examples of the data retrieval operation.

(a) The *Posting_No* and *Empl_No* values from the DUTY_ALLOCATION relation can be retrieved by the SQL statement shown below. For the DUTY_ALLOCATION table of Figure 5.4, the statement produces the result shown in part i of Figure A.

select *Posting_No, Empl_No*
from DUTY_ALLOCATION

The above query resembles the relational algebra projection operation. This is not strictly a projection because duplicates are not removed, as shown in part i of Figure A. Duplicates may be removed by using the distinct option in the select statement, as indicated on page 218. The distinct option is applied to the entire result relation *(Posting_No, Empl_No)*. The result of this statement is shown in part ii of Figure A.

Figure A (i) A simple projection via select with duplicates tuples; (ii) Eliminating duplicate tuple by the distinct clause in the select statement.

Posting_No	*Empl_No*
321	123456
322	123457
323	123458
321	123461
321	123461
350	123471
351	123471

(i)

Posting_No	*Empl_No*
321	123456
322	123457
323	123458
321	123461
350	123471
351	123471

(ii)

```
select distinct Posting_No, Empl_No
from DUTY_ALLOCATION
```

(b) "Get complete details from DUTY_ALLOCATION."

```
select *
from DUTY_ALLOCATION
```

The asterisk character is used as shorthand for the full attribute list. The result of this statement is the entire DUTY_ALLOCATION relation shown in Figure 5.4. ■

5.3.2 Condition Specification

SQL supports the following Boolean and comparison operators: **and, or, not,** =, ≠ (not equal), >, ≥, >, ≤. These operators allow the formulation of more complex predicates, which are attached to the select statement by the where clause. Such predicates in the where clause specify the selection of specific tuples and/or a join of tuples from two relations (i.e., they provide the capability of the selection and join operations of relational algebra). If more than one of the Boolean operators appear together, not has the highest priority while **or** has the lowest. Parentheses may be used to indicate the desired order of evaluation.

Example 5.2 "Get DUTY_ALLOCATION details for *Empl_No* 123461 for the month of April 1986." This query is given on page 219. The result of the query is shown in part i of Figure B.

Figure B (i) Selecting specified tuples followed by projection; (ii) Ordering the result; (iii) Selecting tuples specified by disjunctive predicates.

Posting_No	*Shift*	*Day*
321	2	19860420
321	2	19860419

(i)

Posting_No	*Shift*	*Day*
321	2	19860419
321	2	19860420

(ii)

Posting_No	*Empl_No*	*Day*	*Shift*
321	123461	19860420	2
321	123461	19860419	2
323	123458	19860420	3

(iii)

```
select Posting_No, Shift, Day
from DUTY_ALLOCATION
where Empl_No = 123461 and
      Day >19860401 and
      Day ≤19860430
```

If the result had to be rearranged, the order clause could be specified as shown below. The result of this statement on our sample database is shown in part ii of Figure B.

```
select Posting_No, Shift, Day
from DUTY_ALLOCATION
where Empl_No = 123461
order by Day asc
```

The following statement selects the posting information about employee 123461 for the month of April 1986, as well as for all employees for shift 3 regardless of dates. The result of this statement on our sample database is shown in part iii of Figure B.

```
select *
from DUTY_ALLOCATION
where (Empl_No = 123461 and
      Day >19860401 and
      Day ≤9860430) or
      (Shift = 3) ■
```


5.3.3 Arithmetic and Aggregate Operators

SQL provides a full complement of arithmetic operators and functions. This includes functions to find the average, minimum, maximum, sum, and to count the number of occurrences.

Let us first consider the SQL facility to specify arithmetic operations on attribute values.

Example 5.3

Consider the relation SALARY(*Empl_No, Pay_Rate, Hours*), used for computing the weekly salary in our sample database. Part of this relation is shown in part i of Figure C. Consider the evaluation of the weekly salary (gross). This operation can be expressed in SQL as shown below. The result of this statement is shown in part ii of Figure C.

```
select Empl_No, Pay_Rate*Hours
from SALARY
where Hours > 0.0
```

This statement is evaluated[1] by performing a cartesian product of the tables T_1, T_2, and thence the tuples satisfying the where clause are selected. These tuples are then projected on the attributes $T_1.a_{11}, \ldots T_1.a_{1n}, T_2.a_{21}, \ldots T_2.a_{2m}$. The relational algebraic form of this statement is

$$\pi_{a_{11}, \ldots, a_{1n}, a_{21}, \ldots, a_{2m}} (T_1 \underset{a_{1j} = a_{2k} \ldots}{\bowtie} T_2)$$

In general the select statement represents the following relational algebraic operations where X is the cartesian product of the relations represented by the **from** list.

$$\pi_{(\text{represented by the target list})} \sigma_{(\text{represented by the where clause})}(X))$$

Joins involving more than two relations can be similarly encoded in SQL. Queries of this form need data from more than one relation. In the case where the join involves a relation with itself, the query needs data from more than one record of the same relation.

Example 5.5

The following SQL query is used to retrieve the shift details for employee Ron:

select *Posting_No, Day, Shift*
from DUTY_ALLOCATION, EMPLOYEE
where DUTY_ALLOCATION.*Empl_No* = EMPLOYEE.*Empl_no*
and *Name* = 'Ron'

Note that attributes *Empl_No* have been qualified, since the names of these attributes are identical. The result of the query on the DUTY_ALLOCATION, EMPLOYEE tables of Figure 5.4 is the triple (321, 19860419, 1). ■

SQL uses the concept of tuple variable from relational calculus. In SQL a tuple variable is defined in the from clause of the select statement. The syntax of the declaration requires that the name of the tuple variable be declared after the relation name in the from clause, as shown below:

from relation_name$_1$ tv$_1$ [,relation_name$_2$ tv$_2$, . . .]

We use tuple variables in Example 5.6 to compare two tuples of the relation EMPLOYEE. The two tuple variables e_1, and e_2 are defined on the same relation.

Example 5.6

"Get employees whose rate of pay is more than or equal to the rate of pay of employee Pierre."

select e_1.*Name*, e_1.Pay_Rate
from EMPLOYEE e_1, EMPLOYEE e_2

[1]This is a conceptual explanation. The actual evaluation of the query may be optimized.

where e_1.*Pay_Rate* > e_2.*Pay_Rate*
and e_2.*Name* = 'Pierre'

The result of this query for the EMPLOYEE table shown in Figure 5.4 is the tuple (Julie, 14.50). ■

Now we turn to an example of a join involving one relation.

Example 5.7

"Compile all pairs of *Posting_Nos* requiring the same Skill."

select p_1.*Posting_No*, p_2.*Posting_No*
from POSITION p_1 POSITION p_2
where p_1.*Skill* = p_2.*Skill*
and p_1.*Posting_No* < p_2.*Posting_No*

p_1. *Posting_No*	p_2. *Posting_No*
321	326
350	351

For the POSITION table of Figure 5.4, this SQL statement generates the result shown above. *Posting_Nos* 321 and 326 require a skill of waiter and *Posting_Nos* 350 and 351 require a skill of chef. The predicate p_1.*Posting_No* < p_2.*Posting_No* is used to avoid including tuples such as (326, 321), (350,350), (351,350), etc., in the result. ■

The following is an example that requires joining two relations.

Example 5.8

Consider the requirement to generate the eligibility of employees to fill a given position. Each position *(Posting_No)* requires a skill and only those employees who have this skill are eligible to fill that position. Thus to generate the position eligibility relation, we are required to join the relations EMPLOYEE and POSITION for equal values of the common attribute *Skill*. The following SQL statement implements the join. The result of the join is shown on the next page.

select EMPLOYEE.*Empl_No*,POSITION.*Posting_No*,POSITION.*Skill*
from EMPLOYEE, POSITION
where EMPLOYEE.*Skill* = POSITION.*Skill*

EMPLOYEE. *Empl_No*	POSITION. *Posting_No*	POSITION. *Skill*
123456	321	waiter
123456	326	waiter
123457	322	bartender
123458	323	busboy
123459	324	hostess
123461	325	maître d'
123471	350	chef
123471	351	chef
123472	350	chef
123472	351	chef

■

The following is an example of joining three relations.

Example 5.9 Consider the requirement to generate the itemized bill for table 12 for the date 19860419. This requires details from three relations, BILL, ORDR, and MENU. The itemized bill can be generated using the following query. The result is shown in Figure D.

Figure D Itemized bill

result

Bill#	*Dish_Description*	*Price*	*Qty*	*Price**Qty
9234	Coffee	2.50	2	5.00
9234	Club sandwich	10.50	2	21.00

```
select BILL.Bill#, MENU.Dish_Description, MENU.Price,
        ORDR.Qty, MENU.Price*ORDR.Qty
from BILL, MENU, ORDR
where BILL.Bill# = ORDR.Bill#
   and ORDR.Dish# = MENU.Dish
   and BILL.Table# = 12
   and BILL.Day = 19860419
```

A select statement can be nested in another select statement. The result of the nested select statement is a relation that can be used by the outer select statement. An alternate method of generating this itemized bill is by using the nested select statement (which forms a sub-query) as shown below:

```
select ORDR.Bill#, MENU.Dish_Description, MENU.Price,
        ORDR.Qty, MENU.Price*ORDR.Qty
```

```
from MENU, ORDR
where ORDR.Dish# = MENU.Dish#
  and ORDR.Bill# =
    (select BILL.Bill#
     from BILL
     where BILL.Table# = 12
     and BILL.Day = 19860419) ■
```


5.3.5 Set Manipulation

SQL provides a number of set operators: any, in, all, exists, not exists, union, minus, intersect, and contains. These constructs, based on the operations used in relational calculus and relational algebra, are used for testing the membership of a value in a set of values, or the membership of a tuple in a set of tuples, or the membership of one set of values in another set of values. When using these operators, remember that the SQL statement "select . . ." returns a set of tuples (which is a set of values in cases where the target list is a single attribute). We describe these set manipulation operators below and illustrate them with a number of examples.

Any

The operator **any** allows the testing of a value against a set of values. The comparisons can be one of {<, ≤, >, ≥, =, ≠}, and are specified in SQL as the operators, <any, ≤any, >any, ≥any, =any, and ≠any (not equal to any). We refer to any one of these operators by the notation θany.

In general, the condition

c θ**any** (**select** X **from** . . .)

evaluates to true if and only if the comparison "c θany {at least one value from the result of the select X from . . . }"is true.[2] Let us illustrate this condition with the following example:

Example 5.10

Let the result of

```
select X
from rel
where P
```

be the set of values {'30', '40', '60', '70'}. Then the following statements, which compare the two sets on both sides of the θany operators, are valid and give the result indicated on the next page.

[2]The implementation of any and all leads to some confusion since ≠any actually is implemented, in some systems, to be not equal to some (any one of the set of values). For example {'50'} ≠any ({'30', '40', '50', '70'}) is evaluated to true since 50 ≠ 30. To justify this implementation, some is used as an alias for any in these systems! Such an implementation tends to give results that do not agree with the interpretation given here.

```
(select p.Posting_No
from POSITION p
where p.Skill = 'chef')
```

Here the first nested subquery finds the positions where an employee is assigned. The second nested subquery finds the set of positions requiring a chef's skill. The main select statement considers each employee and for that employee finds all the positions and tests if this is a superset of the positions requiring a chef's skill. If this test evaluates to a true value, the attribute *Name* is output. For our sample database, the result of this query is (Pierre). ■

All

The set operator **all** is used, in general, to show that the condition

c θall (select X from . . .)

evaluates to true. This is so, if and only if the comparison "c θ all the values from the result of (**select** X **from** . . .)" is true. We illustrate the various format of this condition in the following example:

Example 5.14 Let the result of:

```
select X
from rel
where P
```

be the set of values {'30', '40', '60', '70'}. Then each of the following statements is valid and produces the results indicated:

'50' =**all** ({'30', '40', '60', '70'}) is false
'29' <**all** ({'30', '40', '60', '70'}) is true
'50' ≠**all** ({'30', '40', '60', '70'}) is true
'70' >**all** ({'30', '40', '60', '70'}) is false
'70' ≥**all** ({'30', '40', '60', '70'}) is true ■

Example 5.15 below uses the all condition to find the employee with the lowest pay rate from the EMPLOYEE relation.

Example 5.15 "Find the employees with the lowest pay rate."

```
select Empl_No, Name, Pay_Rate
from EMPLOYEE
where Pay_Rate ≤all
      (select Pay_Rate
      from EMPLOYEE)
```