SQL

STRUCTURED QUERY
LANGUAGE

4.1 Introduction

Originally, SQL was called SEQUEL (for Structured English QUery Language) and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R. SQL is now the language for IBM’s DB2 and SQL/DS commercial relational DBMSs. Variations of SQL have been implemented by most commercial DBMSs vendors.

A joint effort under the way by ANSI (the American National Standards Institute) and ISO (the International Standards Organization) has led to a standard version of SQL (ANSI 1986), called SQL1. A revised and much expanded standard called SQL2 (also referred to as SQL-92) has also been developed. Plans are already underway for SQL3, which will further extend SQL with object-oriented and other recent database concepts.

SQL has the following general features:

• SQL is a comprehensive database language, it has its own DDL (Data Definition Language) component including statements for data definition and DML (Data Manipulation Language) component including statements for query and update operations.

• It has facilities for defining views on the database, for creating and dropping indexes on the files that represent relations, and for embedding SQL statements into a high-level general-purpose programming language such as C.

• It has catalog and dictionary facilities.

• It maintains several level of authentication and data security.
4.2 Data Definition in SQL

SQL uses the terms *table*, *row*, and *column* for *relation*, *tuple*, and *attribute*, respectively.

The SQL commands for data definition are:
- CREATE
- ALTER
- DROP
4.2.1 The CREATE SCHEMA Command

An SQL schema is identified by a schema name, and includes an authorization identifier to indicate the user or account who owns the schema, as well as descriptors for each element in the schema. Schema elements include the tables, views, domains, and other constructs (such as authorization grants and assertions) that describe the schema. A schema is created via the CREATE SCHEMA statement, which can include all the schema elements’ definitions.

Example:

CREATE SCHEMA COMPANY AUTHORIZATION KHALIL;

In addition to using the concept of schema, SQL2 uses the concept of catalog - a named collection of schemas in an SQL environment. A catalog always contains a special schema called INFORMATION_SCHEMA, which provides information on all the element descriptors of all the schemas in the catalog to authorized users.

4.2.2 The CREATE TABLE Command

The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and constraints. The attributes are specified first; and each attribute is given a name, a data type to specify the domain of values, and possibly some constraints. The key, entity integrity, and referential constraints are then specified.

Figure 4.1 shows sample data definition statements in SQL for the relational schema for the COMPANY database - described in previous chapter. Figure 4.1 also shows some data types that are supported by SQL2.

SQL2 provides the following facilities:
- It is possible to specify the data type of each attribute directly, as in Figure 4.1; alternatively a domain can be declared, and the domain name used. This makes it easier to change the data type for a domain that is used by numerous attributes in a schema, and improves schema readability. For example, we can create a domain SSN_TYPE by the following statement:

CREATE DOMAIN SSN_TYPE AS CHAR(9);
• We can use SSN_TYPE in place of CHAR(9) in Figure 4.1 for all attributes referring to SSN (Figure 4.2). A domain can also have an optional default specification via a DEFAULT clause.

• SQL2 allows NULLS for attribute values and a constraint NOT NULL may be specified if NULL is not permitted for that attribute.

• It is possible to define a default value for an attribute by appending the clause \texttt{DEFAULT < value >} to an attribute definition. Figure 4.3 illustrates an example of specifying a default manager for a new department and a default department for a new employee.
CREATE TABLE EMPLOYEE
    (FNAME VARCHAR(15) NOT NULL,
     MINIT CHAR,
     LNAME VARCHAR(15) NOT NULL,
     SSN CHAR(9) NOT NULL,
     BDATE DATE,
     ADDRESS VARCHAR(30),
     SEX CHAR,
     SALARY DECIMAL(10,2),
     SUPERSSN CHAR(9),
     DNO INT NOT NULL,
     PRIMARY KEY (SSN),
     FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN),
     FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER));

CREATE TABLE DEPARTMENT
    (DNAME VARCHAR(15) NOT NULL,
     DNUMBER INT NOT NULL,
     MGRSSN CHAR(9) NOT NULL,
     MGRSTARTDATE DATE
     PRIMARY KEY (DNUMBER),
     UNIQUE (DNAME)
     FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN));

CREATE TABLE DEPT_LOCATIONS
    (DNUMBER INT NOT NULL,
     DLOCATION VARCHAR(15)
     PRIMARY KEY (DNUMBER, DLOCATION),
     FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER));

Figure 4.1 SQL2 definitions for the COMPANY schema.
(to be continued)
Figure 4.1 SQL2 definitions for the COMPANY schema.
CREATE TABLE EMPLOYEE
  (FNAME VARCHAR(15) NOT NULL,
   MINIT CHAR,
   LNAME VARCHAR(15) NOT NULL,
   SSN SSN_TYPE NOT NULL,
   BDATE DATE,
   ADDRESS VARCHAR(30),
   SEX CHAR,
   SALARY DECIMAL(10,2),
   SUPERSSN SSN_TYPE,
   DNO INT NOT NULL,
   PRIMARY KEY (SSN),
   FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN),
   FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER));

CREATE TABLE DEPARTMENT
  (DNAME VARCHAR(15) NOT NULL,
   DNUMBER INT NOT NULL,
   MGRSSN SSN_TYPE NOT NULL,
   MGRSTARTDATE DATE,
   PRIMARY KEY (DNUMBER),
   UNIQUE (DNAME),
   FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN));

CREATE TABLE DEPT_LOCATIONS
  (DNUMBER INT NOT NULL,
   DLOCATION VARCHAR(15) NOT NULL,
   PRIMARY KEY (DNUMBER, DLOCATION),
   FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER));

Figure 4.2 SQL2 definitions for the COMPANY schema using the domain SSN_TYPE (to be continued).
CREATE TABLE PROJECT
  (PNAME VARCHAR(15) NOT NULL,
   PNUMBER INT NOT NULL,
   PLOCATION VARCHAR(15) NOT NULL,
   DNUM INT NOT NULL,
   PRIMARY KEY (PNUMBER),
   UNIQUE (PNAME),
   FOREIGN KEY (DNUM) REFERENCES DEPARTMENT(DNUMBER));

CREATE TABLE WORKS_ON
  (ESSN SSN_TYPE NOT NULL,
   PNO INT NOT NULL,
   HOURS DECIMAL(3,1) NOT NULL,
   PRIMARY KEY (ESSN, PNO),
   FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN),
   FOREIGN KEY (PNO) REFERENCES PROJECT(PNUMBER));

CREATE TABLE DEPENDENT
  (ESSN SSN_TYPE NOT NULL,
   DEPENDENT_NAME VARCHAR(15) NOT NULL,
   SEX CHAR,
   BDATE DATE,
   RELATIONSHIP VARCHAR(8),
   PRIMARY KEY (ESSN, DEPENDENT_NAME),
   FOREIGN KEY (ESSN) REFERENCES EMPLOYEE(SSN));

Figure 4.2 SQL2 definitions for the COMPANY schema using the domain SSN_TYPE.
CREATE TABLE EMPLOYEE
(.....,
  DNO INT NOT NULL DEFAULT 1,
CONSTRAINT EMPPK PRIMARY KEY (SSN)
CONSTRAINT EMPSUPERFK FOREIGN KEY (SUPERSSN) REFERENCES EMPLOYEE(SSN) ON DELETE SET NULL ON UPDATE CASCADE,
CONSTRAINT EMPDEPTFK FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER) ON DELETE SET DEFAULT ON UPDATE CASCADE);

CREATE TABLE DEPARTMENT
(.....,
SQL

MGRSSN CHAR(9) NOT NULL DEFAULT ‘888665555’

CONSTRAINT DEPTPK
  PRIMARY KEY (DNUMBER)
CONSTRAINT DEPTSK
  UNIQUE (DNAME)
CONSTRAINT DEPTMGRFK
  FOREIGN KEY (MGRSSN) REFERENCES EMPLOYEE(SSN) ON DELETE SET DEFAULT ON UPDATE CASCADE);

CREATE TABLE DEPT_LOCATIONS
  (....
  PRIMARY KEY (DNUMBER, DLOCATION),
  FOREIGN KEY (DNUMBER) REFERENCES DEPARTMENT(DNUMBER)
  ON DELETE CASCADE ON UPDATE CASCADE);

Figure 4.4 Naming constraints and using referential triggered actions
4.2.3 The DROP SCHEMA Command

If a whole schema is not needed any more, the DROP SCHEMA command can be used. There are two drop behavior options:

- CASCADE, and
- RESTRICT.

For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

```
DROP SCHEMA COMPANY CASCADE;
```

If the RESTRICT option in chosen in place of CASCADE, the schema is dropped only if it has no elements in it; otherwise the drop command will not be executed.

4.2.4 The DROP TABLE Command

If a base relation within a schema is not needed any longer, the relation and its definition can be deleted by using the DROP TABLE command. For example, if we no longer wish to keep track of dependents of employees in the COMPANY database, we can rid of the DEPENDENT relation by issuing the command:

```
DROP TABLE DEPENDENT CASCADE;
```

If the RESTRICT option in chosen in place of CASCADE, a table is dropped only if it is not referenced in any constraints (such as by foreign key definitions in another relation) or views. With the CASCADE option, all such constraints and views that reference the table are dropped automatically from the schema along with the table itself.

4.2.5 The ALTER TABLE Command

The definition of a base table can be changed by using the ALTER TABLE command. The possible alter table actions include:

- Adding an attribute,
- Dropping an attribute,
- Changing an attribute definition,
- Adding a constraint, and
- Dropping a constraint.

For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relation in the COMPANY database schema, we can use the command:

```
ALTER TABLE EMPLOYEE ADD JOB VARCHAR(12);
```

To drop an attribute, one must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the attribute are dropped automatically from the schema, along with the attribute. If RESTRICT is chosen, the command is successful only if no views or constraints reference the attribute. For example, the following command removes the attribute ADDRESS from the EMPLOYEE base table:

```
ALTER TABLE EMPLOYEE DROP ADDRESS CASCADE;
```

It is also possible to alter an attribute definition by dropping existing default clause or by defining a new default clause. The following example illustrates this facility:

```
ALTER TABLE EMPLOYEE ALTER DNO DROP DEFAULT;
ALTER TABLE EMPLOYEE ALTER DNO SET DEFAULT 2;
```

Finally, one can change the constraints specified on a table by adding or dropping a constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 4.4 from the EMPLOYEE relation, we write

```
ALTER TABLE EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;
```
4.3 Queries in SQL

SQL has one basic statement for retrieving information from a database: the SELECT statement. That statement is used, for example, to select tuples from individual relations and to combine related tuples from several relations for the purpose of specifying a query - a retrieval request - on the database. The result of each query is a new relation, which can be further manipulated.

In the following sections, the basic features and facilities of the SELECT statement are introduced. Examples of queries in SQL are given using the sample instance for a simplified schema of the COMPANY database (Figure 4.5).
SQL

**EMPLOYEE**

<table>
<thead>
<tr>
<th>SSN</th>
<th>ENAME</th>
<th>SEX</th>
<th>DNO</th>
<th>SUPERSSN</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Khalil</td>
<td>M</td>
<td>1</td>
<td>NULL</td>
<td>3000</td>
</tr>
<tr>
<td>123</td>
<td>Badran</td>
<td>M</td>
<td>1</td>
<td>111</td>
<td>1000</td>
</tr>
<tr>
<td>444</td>
<td>Dalia</td>
<td>F</td>
<td>4</td>
<td>111</td>
<td>1000</td>
</tr>
<tr>
<td>525</td>
<td>Saleh</td>
<td>M</td>
<td>5</td>
<td>343</td>
<td>2000</td>
</tr>
<tr>
<td>343</td>
<td>Samr</td>
<td>F</td>
<td>5</td>
<td>686</td>
<td>3000</td>
</tr>
<tr>
<td>686</td>
<td>Ahmed</td>
<td>M</td>
<td>4</td>
<td>NULL</td>
<td>2500</td>
</tr>
</tbody>
</table>

**DEPARTMENT**

<table>
<thead>
<tr>
<th>DNUMBER</th>
<th>DNAME</th>
<th>MGRSSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Management</td>
<td>111</td>
</tr>
<tr>
<td>4</td>
<td>Research</td>
<td>686</td>
</tr>
<tr>
<td>5</td>
<td>Production</td>
<td>525</td>
</tr>
</tbody>
</table>

**PROJECT**

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>DNUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProjectX</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>ProjectY</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>ProjectZ</td>
<td>4</td>
</tr>
</tbody>
</table>

**WORKS_ON**

<table>
<thead>
<tr>
<th>ESSN</th>
<th>PNO</th>
<th>HOURS</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>123</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>123</td>
<td>4</td>
<td>60</td>
</tr>
<tr>
<td>444</td>
<td>2</td>
<td>40</td>
</tr>
<tr>
<td>444</td>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>525</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>525</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 4.5  A Sample instance for a simplified company schema
4.3.1 Basic SQL Queries

The basic form of the SELECT statement is formed of the three clauses SELECT, FROM, and WHERE and has the following form:

```
SELECT  < Attribute list >
FROM     < table list >
[WHERE   < condition >];
```

Where

- `< attribute list >` is a list of attribute names whose values are to be retrieved by the query.
- `< table list >` is a list of the relation names required to process the query.
- `< condition >` is a conditional (Boolean) search expression that identifies the tuples to be retrieved by the query. The WHERE clause is optional (That is why it is included between []).
**Examples:**

*Query (1):* Retrieve all information for all employees in the company.

```sql
SELECT * FROM EMPLOYEE;
```

- A missing `WHERE` clause indicates no condition on tuple selection; hence, all tuples of the `EMPLOYEE` relation are selected.

- The asterisk (*) stands for all attributes of the specified relation `EMPLOYEE`.

- The result of that query is an exact image of relation `EMPLOYEE` (Figure 4.6).

**RESULT (1):**

<table>
<thead>
<tr>
<th>SSN</th>
<th>ENAME</th>
<th>SEX</th>
<th>DNO</th>
<th>SUPERSSN</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Khalil</td>
<td>M</td>
<td>1</td>
<td>NULL</td>
<td>3000</td>
</tr>
<tr>
<td>123</td>
<td>Badran</td>
<td>M</td>
<td>1</td>
<td>111</td>
<td>1000</td>
</tr>
<tr>
<td>444</td>
<td>Dalia</td>
<td>F</td>
<td>4</td>
<td>111</td>
<td>1000</td>
</tr>
<tr>
<td>525</td>
<td>Saleh</td>
<td>M</td>
<td>5</td>
<td>343</td>
<td>2000</td>
</tr>
<tr>
<td>343</td>
<td>Samr</td>
<td>F</td>
<td>5</td>
<td>686</td>
<td>3000</td>
</tr>
<tr>
<td>686</td>
<td>Ahmed</td>
<td>M</td>
<td>4</td>
<td>NULL</td>
<td>2500</td>
</tr>
</tbody>
</table>

Figure 4.6 Result of Query (1)
Query (2):
Retrieve all information for all employees who work in department number 5.

```
SELECT * 
FROM EMPLOYEE;
WHERE DNO = 5;
```

- That query produces a relation of the same structure as the base relation EMPLOYEE but containing only the tuples representing the employees working for department number 5 (Figure 4.7).

RESULT (2):

<table>
<thead>
<tr>
<th>SSN</th>
<th>ENAME</th>
<th>SEX</th>
<th>DNO</th>
<th>SUPERSSN</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>525</td>
<td>Saleh</td>
<td>M</td>
<td>5</td>
<td>343</td>
<td>2000</td>
</tr>
<tr>
<td>343</td>
<td>Samr</td>
<td>F</td>
<td>5</td>
<td>686</td>
<td>3000</td>
</tr>
</tbody>
</table>

Figure 4.7 Result of Query (2)
**Query (3):**
Retrieve all information for all employees who work in department number 5 and their salary greater than 1000.

```
SELECT * FROM EMPLOYEE;
WHERE DNO = 5 AND SALARY > 2000;
```

- This query involves only the EMPLOYEE relation listed in the FROM clause. The query selects the EMPLOYEE tuples that satisfy the condition of the WHERE clause. The result of that query is a relation of the same schema structure as relation EMPLOYEE and it will contain the values of all attributes for all tuples representing employees satisfying the given condition (working for the department number 5 and having salary greater than 2000). The result of Query (3) is shown in Figure 4.8.

**RESULT (3)**

<table>
<thead>
<tr>
<th>SSN</th>
<th>ENAME</th>
<th>SEX</th>
<th>DNO</th>
<th>SUPERSSN</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>343</td>
<td>Samr</td>
<td>F</td>
<td>5</td>
<td>686</td>
<td>3000</td>
</tr>
</tbody>
</table>

Figure 4.8 Result of Query (3)
**Query (4):**
Retrieve all information for all employees who work in department number 5 and their salary greater than 3000.

```sql
SELECT * 
FROM EMPLOYEE;
WHERE DNO = 5 AND SALARY > 3000;
```

- This query involves only the EMPLOYEE relation listed in the FROM clause. The query selects the EMPLOYEE tuples that satisfy the condition of the WHERE clause. The result of that query is a relation of the same schema structure as relation EMPLOYEE and it will contain the values of all attributes for all tuples representing employees satisfying the given condition (working for the department number 5 and having salary greater than 3000). The result of Query (3) is an empty relation because there are no tuples in relation employee that satisfy the specified conditions (Figure 4.9).

**RESULT (4):**

<table>
<thead>
<tr>
<th>SSN</th>
<th>ENAME</th>
<th>SEX</th>
<th>DNO</th>
<th>SUPERSSN</th>
<th>SALARY</th>
</tr>
</thead>
</table>

Figure 4.9 Result of Query (4)
**Query (5):**
Retrieve the name and salary of all employees who work in department number 5 and their salary greater than 1000.

```sql
SELECT ENAME, SALARY
FROM EMPLOYEE
WHERE DNO = 5 AND SALARY > 1000;
```

- This query involves only the EMPLOYEE relation listed in the FROM clause. The query selects the EMPLOYEE tuples that satisfy the condition of the WHERE clause, then projects the result on the ENAME, and SALARY attributes. The result of that query is a relation of two attributes ENAME and SALARY and it will contain the values of these attributes for all tuples representing employees satisfying the given condition (working for the department number 5 and having salary greater than 1000). Figure 4.10 illustrates the result of query (5).

**RESULT (5):**

<table>
<thead>
<tr>
<th>ENAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saleh</td>
<td>2000</td>
</tr>
<tr>
<td>Samr</td>
<td>3000</td>
</tr>
</tbody>
</table>

Figure 4.10 Result of Query (5)
**Query (6):**
Retrieve the name and salary of all employees who work for the ‘Research’.

```
SELECT ENAME, SALARY
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER
  AND DNAME = 'Research';
```

- This query involves two relations: the EMPLOYEE relation and the DEPARTMENT relation, where information will be collected from both. The condition DNO = DNUMBER is a **join condition**, which corresponds to the condition under which a **JOIN** is performed. Figure 4.11 illustrates the result of query (6).

**RESULT (6):**

<table>
<thead>
<tr>
<th>ENAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dalia</td>
<td>1000</td>
</tr>
<tr>
<td>Ahmed</td>
<td>2500</td>
</tr>
</tbody>
</table>

Figure 4.11 Result of Query (6)
**Query (7):**
Retrieve the name of the manager of the department controlling project number 1.

```
SELECT ENAME
FROM   PROJECT, DEPARTMENT, EMPLOYEE
WHERE  DNUM    = DNUMBER
       AND  MGRSSN  = SSN
       AND  PNUMBER = 1;
```

- This query involves three relations: the PROJECT, DEPARTMENT and EMPLOYEE relations, where information will be collected from the three relations. The condition DNUM = DNUMBER is a join condition to join relation PROJECT with relation DEPARTMENT, and the resulting relation is joined with relation EMPLOYEE via the join condition MGRSSN = SSN. Figure 4.12 illustrates the result of query (7).

**RESULT (7):**

<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saleh</td>
</tr>
</tbody>
</table>

Figure 4.12 Result of Query (7)
4.3.2 Dealing with Ambiguous Attribute Names

In SQL, the same name can be used for two (or more) attributes as long as the attributes are in different relations. If this the case, and a query refers to two or more attributes with the same name, we must qualify the attribute name with the relation name to prevent ambiguity. This is done by prefixing the relation name to the attribute name and separating the two by a period. To illustrate this, suppose that in Figure 4.5 the attribute ENAME in relation EMPLOYEE was called NAME and the DNAME attribute in relation DEPARTMENT was also called NAME; then to prevent ambiguity, Query (6) would be rephrased as shown in Query (6A). We must prefix the attribute NAME in Query (6A) to specify which one we are referring to, because the attributes names are used in both relations:

Query (6A):

```
SELECT EMPLOYEE.NAME, SALARY
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER
  AND DEPARTMENT.NAME = 'Research';
```

Ambiguity also arises in the case of queries that refer to the same relation twice, as in the following example.
**Query (8):**
For each employee, retrieve the employee’s name and the name of his or her direct supervisor.

```sql
SELECT E.ENAME, S.ENAME
FROM EMPLOYEE E, EMPLOYEE S
WHERE E.SUPERSSN = S.SSN
```

- In this case we are allowed to declare alternative relations E and S, called **aliases**, for the EMPLOYEE relation. An alias can directly follow the relation name, as in Query (8), or it follow the keyword **AS** - for example, EMPLOYEE AS E. It is also possible to rename the relation attributes within the query by giving them aliases; for example, EMPLOYEE AS E(SSN, EN, S, DN, SSSN, SAL).

- In Query (8), we can think of E and S as two different copies of the EMPLOYEE relation; the first, E, represents employees in the role of supervisees; and the second, S, represents employees in the role of supervisors. We can now join the two copies via the join condition E.SUPERSSN = S.SSN. Query (8) is an example of a one-level recursive query (Figure 4.13).

**RESULT (8):**

<table>
<thead>
<tr>
<th>E.ENAME</th>
<th>S.ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Badran</td>
<td>Khalil</td>
</tr>
<tr>
<td>Dalia</td>
<td>Khalil</td>
</tr>
<tr>
<td>Saleh</td>
<td>Samr</td>
</tr>
<tr>
<td>Samr</td>
<td>Ahmed</td>
</tr>
</tbody>
</table>

Figure 4.13 Result of Query (8)
4.3.3 Tables as Sets in SQL

SQL does not treat a relation as a set; duplicate tuples can appear more than once in a result of query. SQL does not automatically eliminate duplicate tuples in the result of queries, for the following reasons:

- Duplicate elimination is an expensive operation. One way to implement it is to sort the tuples first and then eliminate duplicates.
- The user may want to see duplicate tuples in the result of a query.
- When an aggregate function (see next sections) is applied to tuples, in most cases we do not want to eliminate duplicates.

If we want to eliminate duplicate tuples from a result of an SQL query, we use the keyword **DISTINCT** in the SELECT clause, meaning that only distinct tuples should remain in the result.

For example, Query (9) retrieves the salary of every employee; if several employees have the same salary, that salary value will appear as many times in the result of the query, as shown in Figure 4.14.
**Query (9):**
Retrieve the salary of every employee.

```
SELECT SALARY
FROM EMPLOYEE;
```

**RESULT (9):**

<table>
<thead>
<tr>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>3000</td>
</tr>
<tr>
<td>2500</td>
</tr>
</tbody>
</table>

Figure 4.14 Result of Query (9)

If we are interested only in distinct salary values, we want each value to appear only once. This can be done by using the keyword DISTINCT as in Query (9A).
Query (9A):
Retrieve the distinct salary values in the database.

SELECT DISTINCT SALARY
FROM EMPLOYEE;

RESULT (9A):

<table>
<thead>
<tr>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>2500</td>
</tr>
</tbody>
</table>

Figure 4.15  Result of Query (9A)
4.3.4 Sets Operators in SQL

SQL has directly incorporated some of the set operations of the relational algebra. SQL includes the following set operators:

- The union operation (UNION),
- The difference operation (EXCEPT),
- The intersection operation (INTERSECT),

The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result (unless the operation is followed by the keyword ALL). Because the set operations apply only to union-compatible relations, we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations. Query (10) illustrates the use of UNION. The first SELECT query retrieves the projects that involve a ‘Khalil’ as manager of the department that controls the project, and the second retrieves the projects that involve a ‘Khalil’ as a worker on the project.
Query (10):

Make a list of all projects that involve an employee whose name is ‘Khalil’, either as a worker or as a manager of the department that controls the project.

\[
\begin{align*}
(\text{SELECT} & \quad \text{PNUMBER} \\
\text{FROM} & \quad \text{PROJECT, DEPARTMENT, EMPLOYEE} \\
\text{WHERE} & \quad \text{DNUM} = \text{DNUMBER} \\
& \quad \text{AND} \quad \text{MGRSSN} = \text{SSN} \\
& \quad \text{AND} \quad \text{NAME} = \text{‘Khalil’}) \\
\text{UNION} & \\
(\text{SELECT} & \quad \text{PNUMBER} \\
\text{FROM} & \quad \text{PROJECT, WORKS_ON, EMPLOYEE} \\
\text{WHERE} & \quad \text{PNUMBER} = \text{PNO} \\
& \quad \text{AND} \quad \text{ESSN} = \text{SSN} \\
& \quad \text{AND} \quad \text{NAME} = \text{‘Khalil’});
\end{align*}
\]
4.3.5 Set Comparisons

The comparison operator **IN** compares a value \( v \) with a set (or multiset) of values \( V \) and evaluates to TRUE if \( v \) is one of the elements in \( V \).

*Query (11):*

Make a list of all employees that work on project number 1, 2, or 4.

```
SELECT DISTINCT ESSN
FROM WORKS_ON
WHERE PNO = 1
  OR PNO = 2
  OR PNO = 4;
```

Query (11) can be rephrased to an equivalent expression by using the **IN** comparison operator. Query (11A) illustrates this solution.
Query (11A):

```sql
SELECT DISTINCT ESSN
FROM WORKS_ON
WHERE PNO IN (1, 2, 4);
```

In addition to the IN operator, a number of other comparison operators can be used to compare a single value (typically an attribute name) to a set V. These comparison operators are:

- The = any (or = some) operator which returns TRUE if the value v is equal to some value in the set V and is hence equivalent to IN. The keyword ANY and SOME have the same meaning.

- Other operators that can be combined with ANY (or SOME) include <, >=, <, <=, and <>.

- The keyword ALL can also be combined with one of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than all values in the set V.

More examples on these comparison operators will be given in the next section.
4.3.6 Nested Queries

Some queries require that existing values in the database be fetched and
then used in a comparison condition. Such queries can be conveniently
formulated by using nested queries, which are complete SELECT queries
within the WHERE clause of another query. That other query is called the
outer query.

Query (10) can be rephrased by using nested queries as shown in Query
(10A).

Query (10A):

```
SELECT PNUMBER
FROM PROJECT
WHERE PNUMBER IN
  (SELECT PNUMBER
   FROM PROJECT, DEPARTMENT, EMPLOYEE
   WHERE DNUM = DNUMBER
     AND MGRSSN = SSN
     AND NAME = 'Khalil')
OR
  PNUMBER IN
  (SELECT PNO
   FROM WORKS_ON, EMPLOYEE
   WHERE ESSN = SSN
     AND NAME = 'Khalil')
```

The first nested query selects the project numbers of projects that have a
‘Khalil’ involved as a manager, while the second selects the project numbers
of projects that have a ‘Khalil’ involved as a worker. In the outer query, we
select a PROJECT tuple if the PNUMBER value of that tuple is in the result
of either nested query.

Query (6) can be rewritten using the concept of nested queries as shown in
Query (6B).

Query (6B):
Retrieve the name and salary of all employees who work for the ‘Research’.

```
SELECT ENAME, SALARY
```

FROM EMPLOYEE
WHERE DNO IN (SELECT DNUMBER
    FROM DEPARTMENT
    WHERE DNAME = 'Research');

Query (12):
Retrieve the names of employees whose salary is greater than the salary of all
the employees in department 5.

SELECT ENAME
FROM EMPLOYEE
WHERE SALARY > ALL (SELECT SALARY
    FROM EMPLOYEE
    WHERE DNO = 5);
4.3.7 The EXISTS Function in SQL

The function EXISTS in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples).

Query (13):
Retrieve the name of each employee who works on at least one project.

SELECT ENAME
FROM EMPLOYEE
WHERE EXISTS (SELECT *
               FROM WORKS_ON
               WHERE ESSN = SSN);

- In Query (13), the nested query references the SSN of the EMPLOYEE relation from the outer query. We can think of Query (13) as follows: for each EMPLOYEE tuple, evaluate the nested query, which retrieves all WORKS_ON tuples with the same SSN as the EMPLOYEE tuple; if at least one tuple exists in the result of the nested query, then select that EMPLOYEE tuple.
- EXISTS(Q) returns TRUE if there is at least one tuple in the result of query Q, and it returns FALSE otherwise.

RESULT (13):

<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Badran</td>
</tr>
<tr>
<td>Dalia</td>
</tr>
<tr>
<td>Saleh</td>
</tr>
</tbody>
</table>

Figure 4.16 Result of Query (13)

Query (14):
Retrieve the name of each employee who does not work on any project.

SELECT ENAME
FROM EMPLOYEE
WHERE NOT EXISTS (SELECT *
                   FROM WORKS_ON
                   WHERE ESSN = SSN);

- In Query (14), the nested query references the SSN of the EMPLOYEE relation from the outer query. We can think of Query (13) as follows: for each EMPLOYEE tuple, evaluate the nested query, which retrieves all
WORKS_ON tuples with the same SSN as the EMPLOYEE tuple; if the result of the nested query is *empty*, then select that EMPLOYEE tuple.

NOT EXISTS(Q) returns TRUE if there are no tuples in the result of query Q, and it returns FALSE otherwise.

**RESULT (14):**

<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Khalil</td>
</tr>
<tr>
<td>Samr</td>
</tr>
<tr>
<td>Ahmed</td>
</tr>
</tbody>
</table>

Figure 4.17 Result of Query (14)
Query (15):
Retrieve the names of managers who do not work on any project.

```
SELECT ENAME
FROM EMPLOYEE
WHERE EXISTS (SELECT *
               FROM DEPARTMENT
               WHERE MGRSSN = SSN)
   AND NOT EXISTS (SELECT *
                   FROM WORKS_ON
                   WHERE ESSN = SSN);
```

In Query (15), two nested queries are specified; the first selects all DEPARTMENT tuples managed by an EMPLOYEE, and the second selects all WORKS_ON tuples related to the EMPLOYEE. If at least one of the first and none of the second exist, we select the EMPLOYEE tuple and retrieve its ENAME.

RESULT (15):

<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Khalil</td>
</tr>
<tr>
<td>Ahmed</td>
</tr>
</tbody>
</table>

Figure 4.18 Result of Query (15)
4.3.8 Nulls in SQL

SQL allows queries that check whether a value is NULL - missing or undefined or not applicable. SQL uses IS or IS NOT to compare an attribute to NULL.

**Query (16):**
Retrieve the names of all employee who do not have supervisors.

```
SELECT ENAME
FROM EMPLOYEE
WHERE SUPERSSN IS NULL;
```

**RESULT (16):**

<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Khalil Ahmed</td>
</tr>
</tbody>
</table>

Figure 4.19 Result of Query (16)
**Query (17):**

Retrieve the names of all employee who have supervisors.

```
SELECT ENAME
FROM EMPLOYEE
WHERE SUPERSSN IS NOT NULL;
```

**RESULT (17):**

```
<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Badran</td>
</tr>
<tr>
<td>Dalia</td>
</tr>
<tr>
<td>Saleh</td>
</tr>
<tr>
<td>Samr</td>
</tr>
</tbody>
</table>
```

Figure 4.20 Result of Query (17)
4.3.9 Renaming Attributes

SQL allows to rename attributes that appears in the result of a query by adding the qualifier AS followed by the desired new name.

For example, Query (8A) below shows how Query (8) can be slightly changed to retrieve the name of each employee and the name of his or her supervisor, while renaming the resulting attribute names as EMPLOYEE_NAME and SUPERVISOR_NAME. The new names will appear as column headers in the query result.

**Query (8A):**
For each employee, retrieve the employee’s name and the name of his or her direct supervisor.

```
SELECT E.ENAME AS EMPLOYEE_NAME, S.ENAME AS SUPERVISOR_NAME
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.SUPERSSN = S.SSN;
```

**RESULT (8A):**

<table>
<thead>
<tr>
<th>EMPLOYEE_NAME</th>
<th>SUPERVISOR_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Badran</td>
<td>Khalil</td>
</tr>
<tr>
<td>Dalia</td>
<td>Khalil</td>
</tr>
<tr>
<td>Saleh</td>
<td>Samr</td>
</tr>
<tr>
<td>Samr</td>
<td>Ahmed</td>
</tr>
</tbody>
</table>

Figure 4.21 Result of Query (8A)
4.3.10 Joined Tables in SQL

The concept of a joined table (or joined relation) was incorporated into SQL2 to permit users to specify a table resulting from a join operation in the FROM clause of a query. This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause.

For example, consider Query (6), which retrieves the name and salary of every employee who works for the ‘Research’ department. For some users, it may be easier first to specify the join of the EMPLOYEE and DEPARTMENT relations, and then select the desired tuples and attributes. This can be written in SQL2 as in Query (6B).

\(\text{Query (6B):}\)

\[
\text{SELECT ENAME, SALARY} \\
\text{FROM (EMPLOYEE JOIN DEPARTMENT ON DNO = DNUMBER)} \\
\text{WHERE DNAME = 'Research';}
\]

- The FROM clause in Query (6B) contains a single joined table. The attributes of such a table are all the attributes of the first table, EMPLOYEE, followed by all the attributes of the second table, DEPARTMENT.
4.3.11 Aggregate Functions and Grouping

SQL provides a set of built-in functions: COUNT, SUM, MAX, MIN, and AVE.

- **COUNT** functions returns the number of tuples in a query.
- **SUM, MAX, MIN,** and **AVE** are applied to a multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values.

These functions can be used in the SELECT or in a HAVING clause (which will be introduced later).

**Query (18)**
Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
SELECT SUM (SALARY), MAX (SALARY), MIN (SALARY), AVE (SALARY)
FROM EMPLOYEE;
```

**RESULT (18):**

<table>
<thead>
<tr>
<th>SUM(SALARY)</th>
<th>MAX(SALARY)</th>
<th>MIN(SALARY)</th>
<th>AVE(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12500</td>
<td>3000</td>
<td>1000</td>
<td>2083</td>
</tr>
</tbody>
</table>

Figure 4.22 Result of Query (18)
**Query (19)**

Find the sum of the salaries of all employees of the department number 5, as well as the maximum salary, the minimum salary, and the average salary.

```
SELECT SUM(SALARY), MAX(SALARY), MIN(SALARY), AVE(SALARY)
FROM EMPLOYEE
WHERE DNO = 5;
```

**RESULT (19):**

<table>
<thead>
<tr>
<th>SUM(SALARY)</th>
<th>MAX(SALARY)</th>
<th>MIN(SALARY)</th>
<th>AVE(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>3000</td>
<td>2000</td>
<td>2500</td>
</tr>
</tbody>
</table>

Figure 4.23 Result of Query (19)

**Query (20)**

Retrieve the total number of employees in the company

```
SELECT COUNT(*)
FROM EMPLOYEE;
```

**RESULT (20):**

<table>
<thead>
<tr>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Figure 4.24 Result of Query (20)
**Query (21)**
Retrieve the total number of employees working for department number 5.

```sql
SELECT COUNT (*)
FROM EMPLOYEE
WHERE DNO = 5;
```

**RESULT (21):**

<table>
<thead>
<tr>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Figure 4.25 Result of Query (21)

- The asterisk (*), in the previous queries, refers to the tuples, so COUNT returns the number of tuples in the result of the query.

- We may also use the COUNT function to count values in an attribute rather than tuples, as in the next example.

**Query (22)**
Count the number of distinct salary values in the database.

```sql
SELECT COUNT (DISTINCT SALARY)
FROM EMPLOYEE;
```

**RESULT (22):**

<table>
<thead>
<tr>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Figure 4.26 Result of Query (22)

**Query (23)**
Retrieve the names of all employees who work on two or more projects.

```sql
SELECT ENAME
FROM EMPLOYEE
WHERE (SELECT COUNT (*)
        FROM WORKS_ON
        WHERE SSN = ESSN) >= 2;
```

- In Query (23), the COUNT function is used to select particular tuples. We specify a correlated nested query with the COUNT function, and we use that
nested query in the WHERE clause of an outer query. The correlated nested query counts the number of projects that each employee works on; if this is greater than or equal to 2, the employee tuple is selected.

**RESULT (23):**

<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Badran</td>
</tr>
<tr>
<td>Dalia</td>
</tr>
<tr>
<td>Saleh</td>
</tr>
</tbody>
</table>

Figure 4.27 Result of Query (23)

In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, based on some attribute values. For example, we may want to find the average salary of employees in each department or the number of employees who work on each project. In these cases we need to group the tuples that have the same value of some attribute(s), called grouping attribute(s), and we need to apply the function on each such group independently. SQL has a GROUP BY clause for this purpose. The GROUP BY clause specifies the **grouping attributes**, which should also appear in the SELECT clause, so that the value resulting from applying each function to a group of tuples appears along with the value of the grouping attribute(s).

**Query (24)**

For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT DNO, COUNT (*), AVE (SALARY)
FROM EMPLOYEE
GROUP BY DNO;
```

- In Query (24), the EMPLOYEE tuples are divided into groups - each group having the same value for the grouping attribute DNO. The COUNT and AVE functions are applied to each such group of tuples.

**RESULT (24):**

<table>
<thead>
<tr>
<th>DNO</th>
<th>COUNT (*)</th>
<th>AVE (SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1750</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2500</td>
</tr>
</tbody>
</table>
Figure 4.28 Result of Query (24)
Query (25)

For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
SELECT    PNUMBER, PNAME, COUNT (*)
FROM       PROJECT, WORKS_ON
WHERE      PNUMBER = PNO
GROUP BY   PNUMBER, PNAME;
```

- Query (25) shows how we can use a join condition in conjunction with GROUP BY. In this case, the grouping and functions are applied after the joining of the two relations.

RESULT (25):

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>COUNT (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProjectX</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>ProjectY</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>ProjectZ</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 4.29 Result of Query (25)

Sometimes we want to retrieve the values of some aggregate functions only for groups that satisfy certain conditions. For example, suppose that we want to modify Query (25) so that only projects with more than two employees appear in the result. SQL provides a HAVING clause, which can appear in conjunction with a GROUP BY clause. for this purpose. HAVING provides a condition on the group of tuples associated with each value of the grouping attributes; and only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated in Query (26).

Query (26)

For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on that project.

```
SELECT    PNUMBER, PNAME, COUNT (*)
FROM       PROJECT, WORKS_ON
WHERE      PNUMBER = PNO
GROUP BY   PNUMBER, PNAME
HAVING     COUNT (*) > 2;
```
• Notice that, while WHERE selects tuples to which functions are applied, the HAVING clause selects whole groups.

RESULT (26):

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>COUNT (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ProjectY</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 4.30 Result of Query (26)
Query (27)  
For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on that project.

```sql
SELECT PNUMBER, PNAME, COUNT(*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE PNUMBER = PNO AND SSN = ESSN AND DNO = 5
GROUP BY PNUMBER, PNAME;
```

RESULT (27):

<table>
<thead>
<tr>
<th>PNUMBER</th>
<th>PNAME</th>
<th>COUNT (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProjectX</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ProjectY</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.31 Result of Query (27)
Query (28)
Find the total number of employees whose salaries exceed 2000 in each department, but only for departments where more than 2 employees work.

\[
\text{SELECT DNAME, COUNT (*) FROM DEPARTMENT, EMPLOYEE WHERE DNUMBER = DNO AND SALARY > 2000 AND DNO IN (SELECT DNO FROM EMPLOYEE GROUP BY DNO HAVING COUNT(*) > 2)}
\]

\text{GROUP BY PNAME;}

RESULT (28):

<table>
<thead>
<tr>
<th>DNAME</th>
<th>COUNT (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.32 Result of Query (28)
4.3.12 Substring Comparisons

SQL allows comparison conditions on only parts of character string, using the LIKE comparison operator.

*Query (29)*
Retrieve all employees whose name start with ‘S’.

```
SELECT ENAME
FROM EMPLOYEE
WHERE ENAME LIKE 'S%';
```

- Partial strings are specified using two reserved characters: ‘%’ replaces an arbitrary number of characters, and ‘_’ replaces a single arbitrary character.

*RESULT (29):*

<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Saleh</td>
</tr>
<tr>
<td>Samr</td>
</tr>
</tbody>
</table>

Figure 4.33 Result of Query (29)
**Query (30)**
Retrieve all employees whose name has character ‘m’ in the third position of the name.

```sql
SELECT ENAME
FROM EMPLOYEE
WHERE ENAME LIKE '__m%';
```

**RESULT (30):**

<table>
<thead>
<tr>
<th>ENAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samr</td>
</tr>
<tr>
<td>Ahmed</td>
</tr>
</tbody>
</table>

Figure 4.34 Result of Query (30)
4.3.13 Arithmetic Operators

SQL allows the use of arithmetic in queries. The standard operators ‘+’, ‘-’, ‘*’, and ‘/’ (for addition, subtraction, multiplication, and division, respectively) can be applied to numeric values in a query. For example, suppose that we want to see the effect of giving all employees who work on the ‘ProjectX’ project a 10% raise; we can issue Query (31) to see what their salaries would become.

Query (31)
Show the resulting salaries if every employee working on the ‘ProjectX’ project is given a 10% raise.

SELECT ENAME, 1.1*SALARY
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE SSN = ESSN AND PNO = PNUMBER AND PNAME = ‘ProjectX’;

RESULT (31):

<table>
<thead>
<tr>
<th>ENAME</th>
<th>1.1*SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Badran</td>
<td>1100</td>
</tr>
<tr>
<td>Saleh</td>
<td>2200</td>
</tr>
</tbody>
</table>

Figure 4.35 Result of Query (31)
4.3.14 Ordering

SQL allows the user to order the tuples in the result of a query by the values of one or more attributes, using the `ORDER BY` clause. The default order is in ascending order of values. We can specify the keyword `DESC` if we want a descending order of values. The keyword `ASC` can be used to specify ascending order explicitly. For example, suppose that we want to retrieve a list of employees ordered by name of employee.

**Query (32)**
Get a list of all employees ordered by employee’s name.

```sql
SELECT * FROM EMPLOYEE ORDER BY ENAME;
```

**RESULT (32):**

<table>
<thead>
<tr>
<th>SSN</th>
<th>ENAME</th>
<th>SEX</th>
<th>DNO</th>
<th>SUPERSSN</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>686</td>
<td>Ahmed</td>
<td>M</td>
<td>4</td>
<td>NULL</td>
<td>2500</td>
</tr>
<tr>
<td>123</td>
<td>Badran</td>
<td>M</td>
<td>1</td>
<td>111</td>
<td>1000</td>
</tr>
<tr>
<td>444</td>
<td>Dalia</td>
<td>F</td>
<td>4</td>
<td>111</td>
<td>1000</td>
</tr>
<tr>
<td>111</td>
<td>Khalil</td>
<td>M</td>
<td>1</td>
<td>NULL</td>
<td>3000</td>
</tr>
<tr>
<td>525</td>
<td>Saleh</td>
<td>M</td>
<td>5</td>
<td>343</td>
<td>2000</td>
</tr>
<tr>
<td>343</td>
<td>Samr</td>
<td>F</td>
<td>5</td>
<td>686</td>
<td>3000</td>
</tr>
</tbody>
</table>

Figure 4.36 Result of Query (32)
**Query (33)**
Get a list of all employees ordered by department number and, within each department, alphabetically by employee’s name descendingly.

```sql
SELECT *  
FROM EMPLOYEE  
ORDER BY DNO, ENAME DESC;
```

**RESULT (33):**

<table>
<thead>
<tr>
<th>SSN</th>
<th>ENAME</th>
<th>SEX</th>
<th>DNO</th>
<th>SUPERSSN</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Khalil</td>
<td>M</td>
<td>1</td>
<td>NULL</td>
<td>3000</td>
</tr>
<tr>
<td>123</td>
<td>Badran</td>
<td>M</td>
<td>1</td>
<td>111</td>
<td>1000</td>
</tr>
<tr>
<td>444</td>
<td>Dalia</td>
<td>F</td>
<td>4</td>
<td>111</td>
<td>1000</td>
</tr>
<tr>
<td>686</td>
<td>Ahmed</td>
<td>M</td>
<td>4</td>
<td>NULL</td>
<td>2500</td>
</tr>
<tr>
<td>343</td>
<td>Samr</td>
<td>F</td>
<td>5</td>
<td>686</td>
<td>32000</td>
</tr>
<tr>
<td>525</td>
<td>Saleh</td>
<td>M</td>
<td>5</td>
<td>343</td>
<td>2000</td>
</tr>
</tbody>
</table>

Figure 4.37 Result of Query (33)
4.4 Update Statements in SQL

In SQL three commands can be used to modify the database:

- **INSERT**,
- **DELETE**, and
- **UPDATE**.

4.4.1 The INSERT Command

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE relation, we can use U1:

\[ U1: \]

**INSERT INTO** EMPLOYEE  
**VALUES** ('666', 'Sherif', 'M', 1, '111', 1500);

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values in the INSERT command. In this case, attributes with NULL or DEFAULT values can be left out. This is illustrated in U2.
4.4.2 The DELETE Command

The DELETE command removes tuples from a relation. It includes a WHERE clause, similar to that used in an SQL query, to select the tuples to be deleted. Tuples are explicitly deleted from one table at a time. Hereafter some examples of DELETE operations:

U3:
DELETE FROM EMPLOYEE
WHERE ENAME = ‘Saleh’;

U4:
DELETE FROM EMPLOYEE
WHERE SSN = ‘123’;

U5:
DELETE FROM EMPLOYEE
WHERE DNO IN (SELECT DNUMBER
FROM DEPARTMENT
WHERE DNAME = ‘Research’);

U6:
DELETE FROM EMPLOYEE;

4.4.2 The UPDATE Command

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE clause in the UPDATE command selects the tuples to be modified from a single relation. Hereafter some examples of DELETE operations:

U7:
UPDATE PROJECT
SET DNUM = 4
WHERE PNUMBER = 1;

$U8:$

UPDATE EMPLOYEE
SET SALARY = 1.1 * SALARY
WHERE DNO IN (SELECT DNUMBER
               FROM DEPARTMENT
               WHERE DNAME = 'Research');
4.5 Views in SQL

A view in SQL is a single virtual table that is derived from other tables. These other tables could be base tables or previously defined views. A view does not exist in physical form; it is considered a virtual table, in contrast to base tables whose tuples are actually stored in the database. Views are defined for the following reasons:

- To satisfy the informational needs for a particular user or group of users.
- To simplify writing queries.
- A view provides some level of data security as it allows to user a subset of the database and hides the rest.

The command to specify a view is CREATE VIEW. The view is given a (virtual) table name, a list of attribute names, and a query to specify the contents of the view. If none of the view attributes result from applying functions or arithmetic operations, we do not have to specify attribute names for the view, as they will be the same as the names of the attributes of the defining tables. V1 and V2 are examples of views that can be created against the COMPANY schema of Figure 4.5.
V1:

```
CREATE VIEW WORKS_ON1
AS
SELECT ENAME, PNAME, HOURS
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE SSN = ESSN AND PNO = PNUMBER;
```

V2:

```
CREATE VIEW DEPT_INFO (DEPT_NAME,
                NO_OF_EMPS, TOTAL_SAL)
AS
SELECT DNAME, COUNT (*), SUM (SALARY)
FROM DEPARTMENT, EMPLOYEE
WHERE DNUMBER = DNO
GROUP BY DNAME;
```

We can specify SQL queries on the defined views in the same way we specify
queries involving base tables. For example, to retrieve the names of all
employees who work on ‘ProjectX’, we can utilize the WORKS_ON1 view and
specify the query in QV1:

QV1:

```
SELECT DNAME
FROM WORKS_ON1
WHERE PNAME = ‘ProjectX’;
```

If we want to know for the ‘Research’ department, the total number of
employees and their total salaries, we can write down the following query
against the defined view V2.

QV2:

```
SELECT NO_OF_EMPS, TOTAL_SAL
FROM DEPT_INFO
WHERE DEPT_NAME = ‘Research’;
```
If we do not view any more, we can use the DROP VIEW command to dispose of it. For example, to get rid of the two views in V1 and V2, we can use the SQL statements in V1A and V2A:

V1A:  DROP VIEW WORKS_ON1;

V2A:  DROP VIEW DEPT_INFO;
KEY POINTS

- SQL is a comprehensive relational database language, it has its own DDL (Data Definition Language) component including statements for data definition and DML (Data Manipulation Language) component including statements for query and update operations.

- It has facilities for defining views on the database, for creating and dropping indexes on the files that represent relations, and for embedding SQL statements into a high-level general-purpose programming language such as C or PASCAL.

- The SQL commands for data definition are CREATE, ALTER, and DROP commands.

- SQL has one basic statement for retrieving information from a database: the SELECT statement. That statement is used, for example, to select tuples from individual relations and to combine related tuples from several relations for the purpose of specifying a query - a retrieval request - on the database. The result of each query is a new relation, which can be further manipulated.
• The basic form of the SELECT statement is formed of the three clauses SELECT, FROM, and WHERE and has the following general structure:

```
SELECT       <Attribute list>
FROM          <table list>
[WHERE         <condition>]
[GROUP BY     <grouping attribute(s)> [HAVING
                      <group condition>]]
[ORDER BY     <attribute list>]
```

• In SQL, three commands can be used to modify the database: INSERT, DELETE, and UPDATE.