MODULE III

SQL DATA DEFINITION

SQL uses the terms table, row, and column for the formal relational model terms relation, tuple, and attribute, respectively.

The name SQL is derived from Structured Query Language. Originally, SQL was called SEQUEL (for Structured English Query Language) and was designed and implemented at IBM Research.

SQL is now the standard language for commercial relational DBMSs.

SQL is a comprehensive database language: It has statements for data definition, query, and update. Hence, it is both a DOL and a DML. In addition, it has facilities for defining views on the database, for specifying security and authorization, for defining integrity constraints, and for specifying transaction controls.

ATTRIBUTE DATA TYPES AND DOMAINS IN SQL

(a) Numeric data types

- **integer** numbers of various sizes (INTEGER or INT, and SMALLINT)

-**floating-point** (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION).

- Formatted numbers can be declared by using **DECIMAL(i,j)**or DEC(i,j) or NUMERIC(i,j)- where **i, the precision**, is the total number of decimal digits and **j, the scale**, is the number of digits after the decimal point.

(b) Character-string data type

-either **fixed length**--CHAR(n) or CHARACTER(n), where n is the number of characters

- **varying** length-VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n), where n is the maximum number of characters.

(c) Bit-string data types

BIT(n)-or varying length-BIT VARYING(n),

Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B'10101'

(d) Boolean data type -TRUE ,FALSE,Unknown

In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a boolean data type is UNKNOWN.

(e) Date and Time datatypes

The DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS.

(f) Timestamp data type (TIMESTAMP)

A timestamp data type (TIMESTAMP) includes both the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds.

Example: TIMESTAMP '2002-09-27 09:12:4 7648302'.

SCHEMA AND CATALOG CONCEPTS IN SQL

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 tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier JSMITH:

CREATE SCHEMA COMPANY AUTHORIZATION JSMITH;

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

CREATE TABLE COMMAND

The syntax is given below:

```
CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>]
                          {, <column name> <column type> [<attribute constraint>] }
                          [<table constraint> {,<table constraint>}])
```

```
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),
          Super_ssn      CHAR(9),
          Dno            INT                NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );

CREATE TABLE DEPARTMENT
        ( Dname          VARCHAR(15)        NOT NULL,
          Dnumber        INT                NOT NULL,
          Mgr_ssn        CHAR(9)            NOT NULL,
          Mgr_start_date DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );

CREATE TABLE DEPENDENT
        ( Essn           CHAR(9)            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) );
```

SPECIFYING BASIC CONSTRAINTS IN SQL

1. Specifying Attribute Constraints and Attribute Defaults

Because SQL allows NULLs as attribute values, a constraint NOT NULL may be specified if NULL is not permitted for a particular attribute. This is always implicitly specified for the attributes that are part of the primary key of each relation, but it can be specified for any other attributes whose values are required not to be NULL.

It is also possible to define a default value for an attribute by appending the clause DEFAULT <value> to an attribute definition.

If no default clause is specified, the default, default value is NULL for attributes that do not have the NOT NULL constraint.

Another type of constraint can restrict attribute or domain values using the CHECK clause following an attribute or domain definition.6 For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of DNUMBER in the DEPARTMENT table

DNUMBER INT NOT NULL CHECK (DNUMBER > 0 AND DNUMBER < 21);

2. Specifying Key and Referential Integrity Constraints

The PRIMARY KEY clause specifies one or more attributes that make up the primary key of a relation. Ifa primary key has a single attribute, the clause can follow the attribute directly.

For example, the primary key of DEPARTMENT can be specified as follows:

DNUMBER INT PRIMARY KEY

The UNIQUE clause specifies alternate (secondary) keys

Referential integrity is specified via the FOREIGN KEY clause . a referential integrity constraint can be violated when tuples are inserted or deleted, or when a foreign key or primary key attribute value is modified.

The default action that SQL takes for an integrity violation is to reject the update operation that will cause a violation. However, the schema designer can specify an alternative action to be taken if a referential integrity constraint is violated, by attaching a referential triggered action clause to any foreign key constraint. The options include SET NULL, CASCADE, and SET DEFAULT. An option must be qualified with either ON DELETE or ON UPDATE.

In Figure below, the database designer chooses SET NULL ON DELETE and CASCADE ON UPDATE for the foreign key SUPERSSN of EMPLOYEE. This means that if the tuple for a supervising employee is deleted, the value of SUPERSSN is automatically set to NULL for all

employee tuples that were referencing the deleted employee tuple. On the other hand, if the SSN value for a supervising employee is updated (say, because it was entered incorrectly), the new value is cascaded to SUPERSSN for all employee tuples referencing the updated employee tuple.

The action for CASCADE ON DELETE is to delete all the referencing tuples, whereas the action for CASCADE ON UPDATE is to change the value of the foreign key to the updated (new) primary key value for all referencing tuples.

```
CREATE TABLE EMPLOYEE
    (  . . . ,
       Dno        INT             NOT NULL        DEFAULT 1,
    CONSTRAINT EMPPK
       PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
       FOREIGN KEY (Super_ssn) 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);
```

In addition to key and referential integrity constraints, which are specified by special keywords, other table constraints can be specified through additional CHECK clauses at the end of a CREATE TABLE statement.

Then we could add the following CHECK clause at the end of the CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date:

CHECK (DEPT_CREATE_DATE < MGRSTARTDATE);

Figure 5.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

THE SELECT-FROM-WHERE STRUCTURE OF BASIC SQL QUERIES

The syntax is given below:

```
SELECT    <attribute list>
FROM      <table list>
WHERE     <condition>;
```

• <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) expression that identifies the tuples to be retrieved by the query.

Query 0

Retrieve the birthdate and address of the employees) whose name is 'John B. Smith'.

```
Q0:   SELECT   Bdate, Address
      FROM     EMPLOYEE
      WHERE    Fname='John' AND Minit='B' AND Lname='Smith';
```

(a)

BDATE	ADDRESS
1965-01-09	731 Fondren, Houston, TX

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1:   SELECT   Fname, Lname, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    Dname='Research' AND Dnumber=Dno;
```

Output:

(b)

Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

```
Q2:   SELECT   Pnumber, Dnum, Lname, Address, Bdate
      FROM     PROJECT, DEPARTMENT, EMPLOYEE
      WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn AND
               Plocation='Stafford';
```

(c)

Pnumber	Dnum	Lname	Address	Bdate
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

AMBIGUOUS ATTRIBUTE NAMES, ALIASING

In SQL the same name can be used for two (or more) attributes as long as the attributes are in different relations. If this is 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(.).

For example ,let the DNO and LNAME attributes of the EMPLOYEE relation were called DNUMBER and NAME, and the DNAME attribute of DEPARTMENT was also called NAME; then, to prevent ambiguity,

Then the Query 1 above can be rewritten as

```
Q1A:  SELECT   Fname, EMPLOYEE.Name, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.Name='Research' AND
               DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;
```

Here,aliasing is done with employee and department table names.Rather than using "employee " and "department"everywhere we can make use of aliases "E " and "D"respectively.

Fully qualified attribute names can be used for clarity even if there is no ambiguity in attribute names. Q1 is shown in this manner as is Q1′ below. We can also create an *alias* for each table name to avoid repeated typing of long table names (see Q8 below).

```
Q1′:  SELECT   EMPLOYEE.Fname, EMPLOYEE.LName,
               EMPLOYEE.Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.DName='Research' AND
               DEPARTMENT.Dnumber=EMPLOYEE.Dno;
```

UNSPECIFIED WHERE CLAUSE

Select all EMPLOYEE SSNS (Q9), and all combinations of EMPLOYEE SSN and DEPARTMENT DNAME (Q10) in the database.

Queries 9 and 10. Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

```
Q9:     SELECT    Ssn
        FROM      EMPLOYEE;
```

(e)

E.Fname
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

```
Q10:    SELECT    Ssn, Dname
        FROM      EMPLOYEE, DEPARTMENT;
```

(f)

Ssn	Dname
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

USE OF THE ASTERISK (*)

```
Q1C:    SELECT    *
        FROM      EMPLOYEE
        WHERE     Dno=5;

Q1D:    SELECT    *
        FROM      EMPLOYEE, DEPARTMENT
        WHERE     Dname='Research' AND Dno=Dnumber;

Q10A:   SELECT    *
        FROM      EMPLOYEE, DEPARTMENT;
```

Output of Q1C

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

Query QIC retrieves all the attribute values of any EMPLOYEE who works in DEPARTMENT number 5.

query QID retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works for every employee of the 'Research' department, and

Query10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations.

Query 11 below 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,. If we are interested only in distinct salary values, we want each value to appear only once, regardless of how many employees earn that salary. By using the keyword DISTINCT as in QIIA, we can accomplish this.

Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:   SELECT   ALL Salary
       FROM     EMPLOYEE;

Q11A:  SELECT   DISTINCT Salary
       FROM     EMPLOYEE;
```

(a)output of Q11 (b)output of Q11A

(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

(b)

Salary
30000
40000
25000
43000
38000
55000

SETOPERATIONS

SQL has directly incorporated some of the set operations of relational algebra. There are set **union (UNION), set difference (EXCEPT), and set intersection (INTERSECT) operations**. The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result. Because these set operations apply only to union-compatible relations,

we must make sure that the two relations on which we apply theoperation have the same attributes and that the attributes appear in the same order in both relations.

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4A:  ( SELECT    DISTINCT Pnumber
        FROM      PROJECT, DEPARTMENT, EMPLOYEE
        WHERE     Dnum=Dnumber AND Mgr_ssn=Ssn
                  AND Lname='Smith' )
        UNION
      ( SELECT    DISTINCT Pnumber
        FROM      PROJECT, WORKS_ON, EMPLOYEE
        WHERE     Pnumber=Pno AND Essn=Ssn
                  AND Lname='Smith' );
```

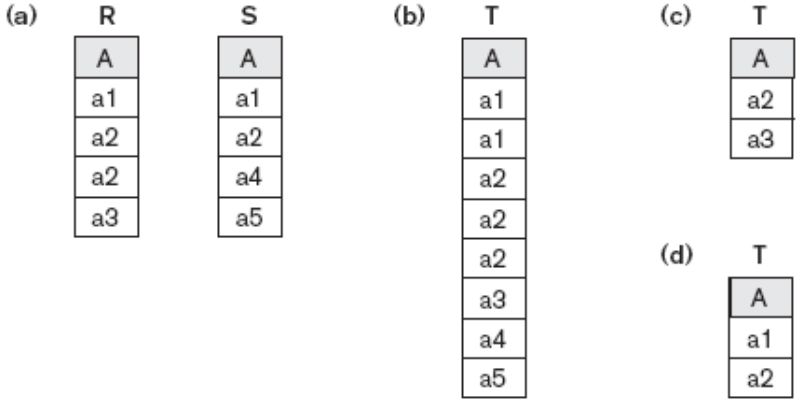

Figure 4.5
The results of SQL multiset operations. (a) Two tables, R(A) and S(A). (b) R(A) UNION ALL S(A). (c) R(A) EXCEPT ALL S(A). (d) R(A) INTERSECT ALL S(A).

SUBSTRING PATTERN MATCHING AND ARITHMETIC OPERATORS

Partial strings are specified using two reserved characters:

(a)% replaces an arbitrary number of zero or more characters,

(b)the underscore (_)replaces a single character.

For example, consider the following query.

Query 12. Retrieve all employees whose address is in Houston, Texas.

```
Q12:  SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    Address LIKE '%Houston,TX%';
```

Query 12A. Find all employees who were born during the 1950s.

```
Q12:  SELECT   Fname, Lname
      FROM     EMPLOYEE
      WHERE    Bdate LIKE '_ _ 5 _ _ _ _ _ _ _';
```

To retrieve all employees who were born during the 1950s, we can use Query 12A. Here, '5' must be the third character of the string (according to our format for date), so we use the value '_5 ', with each underscore serving as a placeholder for an arbitrary character.

If an underscore or % is needed as a literal character in the string, the character should be preceded by an escape character, which is specified after the string using the keyword ESCAPE. For example, 'AB_CD\%EF'. ESCAPE '\' represents the literal string 'AB_CD%EF', because \ is specified as the escape character.

Query 14. Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

```
Q14:  SELECT   *
      FROM     EMPLOYEE
      WHERE    (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

The condition (Salary **BETWEEN** 30000 **AND** 40000) in Q14 is equivalent to the condition ((Salary >= 30000) **AND** (Salary <= 40000)).

ORDERING OF QUERY RESULTS

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.

Query 15. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

```
Q15:  SELECT     D.Dname, E.Lname, E.Fname, P.Pname
      FROM       DEPARTMENT D, EMPLOYEE E, WORKS_ON W,
                 PROJECT P
      WHERE      D.Dnumber= E.Dno AND E.Ssn= W.Essn AND
                 W.Pno= P.Pnumber
      ORDER BY   D.Dname, E.Lname, E.Fname;
```

The default order is in ascending order of values. We can specify the keyword DESCif we want to see the result in a descending order of values. The keyword ASC can be usedto specify ascending order explicitly. For example, if we want descending order on DNAME and ascending order on LNAME, FNAME, the ORDER BY clause of Q15 can be written as

```
ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC
```

COMPARISONS INVOLVING NULL

SQL allows queries that check whether an attribute value is NULL. Rather than using =or< >(not equal) to compare an attribute value to NULL, SQL uses IS or IS NOT.

Consider the following examples to illustrate each of the meanings of NULL.

1. Unknown value. A person's date of birth is not known, so it is represented by NULL in the database.

2. Unavailable or withheld value. A person has a home phone but does not want it to be listed, so it is withheld and represented as NULL in the database.

3. Not applicable attribute. An attribute LastCollegeDegree would be NULL for a person who has no college degrees because it does not apply to that person.

Query 18. Retrieve the names of all employees who do not have supervisors.

```
Q18:  SELECT  Fname, Lname
      FROM    EMPLOYEE
      WHERE   Super_ssn IS NULL;
```

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-from-where blocks within the WHERE clause of another query. That other query is called the outer query.

Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4A:  SELECT   DISTINCT Pnumber
      FROM     PROJECT
      WHERE    Pnumber IN
               ( SELECT     Pnumber
                 FROM       PROJECT, DEPARTMENT, EMPLOYEE
                 WHERE      Dnum=Dnumber AND
                            Mgr_ssn=Ssn AND Lname='Smith' )
               OR
               Pnumber IN
               ( SELECT     Pno
                 FROM       WORKS_ON, EMPLOYEE
                 WHERE      Essn=Ssn AND Lname='Smith' );
```

The first nested query selects the project numbers of projects that have a 'Smith' involved as manager, while the second selects the project numbers of projects that have a 'Smith' involved as worker. In the outer query, we use the OR logical connective to retrieve aPROJECT tuple if the PNUMBER value of that tuple is in the result of either nested query.

If a nested query returns a single attribute and a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator. In general, the nested query will return a table (relation), which is a set or multiset of tuples.

SQL allows the use of **tuples** of values in comparisons by placing them within parentheses. To illustrate this, consider the following query:

```
SELECT   DISTINCT Essn
FROM     WORKS_ON
WHERE    (Pno, Hours) IN ( SELECT   Pno, Hours
                           FROM     WORKS_ON
                           WHERE    Essn='123456789' );
```

In addition to the IN operator, a number of other comparison operators can be used to compare a single value v (typically an attribute name) to a set or multiset v (typically a nested query). The = ANY (or = SOME) operator returns TRUE if the value v is equal to some value in the set V and is hence equivalent to IN. The two keywords ANY and SOME have the same effect. Other operators that can be combined with ANY (or SOME) include >, >=, <, <=, and <>. The keyword ALL can also be combined with each of these operators. For example, the comparison condition (v > ALL V) returns TRUE if the value v is greater than all the values in the set (or multiset) V.

An example is the following query, which returns the names of employees whose salary is greater than the salary of all the employees in department 5:

```
SELECT    Lname, Fname
FROM      EMPLOYEE
WHERE     Salary > ALL  ( SELECT    Salary
                          FROM      EMPLOYEE
                          WHERE     Dno=5 );
```

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:  SELECT    E.Fname, E.Lname
      FROM      EMPLOYEE AS E
      WHERE     E.Ssn IN  ( SELECT    Essn
                            FROM      DEPENDENT AS D
                            WHERE     E.Fname=D.Dependent_name
                            AND E.Sex=D.Sex );
```

In the nested query of Q16, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex. If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT. However, we would not have to qualify the attributes Fname and Ssn of EMPLOYEE if they appeared in the nested query because the DEPENDENT relation does not have attributes called Fname andSsn, so there is no ambiguity.

CORRELATED NESTED QUERIES

Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be correlated.

We can understand a correlated query better by considering that the nested query is evaluated once for each tuple (or combination of tuples) in the outer query.

In general, a query written with nested select-from-where blocks and using the = or IN comparison operators can *always* be expressed as a single block query. For example, Q16 may be written as in Q16A:

```
Q16A:  SELECT    E.Fname, E.Lname
       FROM      EMPLOYEE AS E, DEPENDENT AS D
       WHERE     E.Ssn=D.Essn AND E.Sex=D.Sex
                 AND E.Fname=D.Dependent_name;
```

THE EXISTS AND NOT EXISTS FUNCTIONS IN SQL

Q16 can be written with EXISTS function

```
Q16B:  SELECT   E.Fname, E.Lname
       FROM     EMPLOYEE AS E
       WHERE    EXISTS ( SELECT   *
                         FROM     DEPENDENT AS D
                         WHERE    E.Ssn=D.Essn AND E.Sex=D.Sex
                                  AND E.Fname=D.Dependent_name);
```

EXISTS and NOTEXISTS are usually used in conjunction with a correlated nested query. In general, EXISTS(Q) returns TRUE if there is at least onetuple in the result of the nested query Q, and it returns FALSE otherwise. On the other hand, NOTEXISTS(Q) returns TRUE if there are no tuples in the result of nested query Q, and it returns FALSE otherwise.

Query 6. Retrieve the names of employees who have no dependents.

```
Q6:    SELECT   Fname, Lname
       FROM     EMPLOYEE
       WHERE    NOT EXISTS ( SELECT   *
                             FROM     DEPENDENT
                             WHERE    Ssn=Essn );
```

EXPLICIT SETS

We have seen several queries with a nested query in the WHERE clause. It is also possible to use an explicit set of values in the WHERE clause, rather than a nested query. Such a set is enclosed in parentheses in SQL.

Query 17. Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17:   SELECT   DISTINCT Essn
       FROM     WORKS_ON
       WHERE    Pno IN (1, 2, 3);
```

JOINED TABLES IN SQL

Query 1 can be rewritten as

```
Q1A:   SELECT   Fname, Lname, Address
       FROM     (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
       WHERE    Dname='Research';
```

The concept of a joined table also allows the user to specify different types of join, such as

NATURAL JOIN and various types of OUTER JOIN. In a NATURAL JOIN on two relations Rand S, no join condition is specified.

If the names of the join attributes are not the same in the base relations, it is possible to rename the attributes so that they match, and then to apply NATURAL JOIN. In this case, the AS construct can be used to rename a relation and all its attributes in the FROM clause

```
Q1B:  SELECT   Fname, Lname, Address
      FROM     (EMPLOYEE NATURAL JOIN
               (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))
      WHERE    Dname='Research';
```

AGGREGATE FUNCTIONS IN SQL

A number of built-in functions exist: COUNT, SUM, MAX, MIN, and AVG. The COUNT function returns the number of tuples or values as specified in a query. The functions SUM, MAX, MIN, and AVG are applied to a set or 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 clause or in a HAVING clause.

Query 19. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q19:  SELECT  SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
      FROM    EMPLOYEE;
```

Query 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:  SELECT  SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
      FROM    (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
      WHERE   Dname='Research';
```

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21:  SELECT  COUNT (*)
      FROM    EMPLOYEE;

Q22:  SELECT  COUNT (*)
      FROM    EMPLOYEE, DEPARTMENT
      WHERE   DNO=DNUMBER AND DNAME='Research';
```

Here the asterisk (*) refers to the *rows* (tuples), so COUNT (*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples, as in the next example.

Query 23. Count the number of distinct salary values in the database.

```
Q23:  SELECT  COUNT (DISTINCT Salary)
      FROM    EMPLOYEE;
```

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) in Q23, then duplicate values will not be eliminated. However, any tuples with NULL for SALARY will not be counted. In general, NULL values are **discarded** when aggregate functions are applied to a particular column (attribute).

The GROUP BY and HAVING Clauses

In many cases we want to apply the aggregate functions *to subgroups of tuples in a relation*, where the subgroups are 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 **partition** the relation into nonoverlapping subsets (or **groups**) of tuples. Each group (partition) will consist of the tuples that have the same value of some attribute(s), called the **grouping attribute(s)**.
We can then apply the function to each such group independently produce summary information about each group. 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 aggregate 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 numbe of employees in the department, and their average salary.

```
Q24:   SELECT    Dno, COUNT (*), AVG (Salary)
       FROM      EMPLOYEE
       GROUP BY  Dno;
```

Results of GROUP BY and HAVING. (a) Q24. (b) Q26.

(a)

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

(b)

Pname	Pnumber	...	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

If NULLs exist in the grouping attribute, then a **separate group** is created for all tuples with a *NULL value in the grouping attribute*. For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result of Q24.

Query 25. For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q25:  SELECT    Pnumber, Pname, COUNT (*)
      FROM      PROJECT, WORKS_ON
      WHERE     Pnumber=Pno
      GROUP BY  Pnumber, Pname;
```

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 summary information regarding the group of tuples associated with each value of the grouping attributes. Only the groups that satisfy the condition are retrieved in the result of the query. This is illustrated by 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 the project.

```
Q26:  SELECT    Pnumber, Pname, COUNT (*)
      FROM      PROJECT, WORKS_ON
      WHERE     Pnumber=Pno
      GROUP BY  Pnumber, Pname
      HAVING    COUNT (*) > 2;
```

Notice that while selection conditions in the WHERE clause limit the *tuples* to which functions are applied, the HAVING clause serves to choose *whole groups*. Figure 5.1(b) illustrates the use of HAVING and displays the result of Q26.

Query 27. For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
Q27:  SELECT    Pnumber, Pname, COUNT (*)
      FROM      PROJECT, WORKS_ON, EMPLOYEE
      WHERE     Pnumber=Pno AND Ssn=Essn AND Dno=5
      GROUP BY  Pnumber, Pname;
```

to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work. Here, the condition (SALARY > 40000) applies only to the COUNT function in the SELECT clause. Suppose that we write the following *incorrect* query:

```
SELECT    Dname, COUNT (*)
FROM      DEPARTMENT, EMPLOYEE
WHERE     Dnumber=Dno AND Salary>40000
GROUP BY  Dname
HAVING    COUNT (*) > 5;
```

This is incorrect because it will select only departments that have more than five employees *who each earn more than $40,000*. The rule is that the WHERE clause is executed first, to select individual tuples or joined tuples; the HAVING clause is applied later, to select individual groups of tuples. Hence, the tuples are already restricted to employees who earn more than $40,000 *before* the function in the HAVING clause is applied. One way to write this query correctly is to use a nested query, as shown in Query 28.

Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28:  SELECT   Dnumber, COUNT (*)
      FROM     DEPARTMENT, EMPLOYEE
      WHERE    Dnumber=Dno AND Salary>40000 AND
               ( SELECT       Dno
                 FROM         EMPLOYEE
                 GROUP BY Dno
                 HAVING       COUNT (*) > 5)
```

THE INSERT COMMAND

```
INSERT INTO <table name> [( <column name>{, <column name>} ) ]
(VALUES ( <constant value> , { <constant value>} ){,(<constant value>{,<constant value>})}
| <select statement>)
```

```
INSERT INTO   EMPLOYEE
VALUES        ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                Oak Forest, Katy, TX', 'M', 37000, '653298653', 4 );
```

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.

To enter a tuple for a new EMPLOYEE for whom we know only the FNAME, LNAME, DNa, and SSN attributes, we can use

```
INSERT INTO   EMPLOYEE (Fname, Lname, Dno, Ssn)
VALUES        ('Richard', 'Marini', 4, '653298653');
```

The query would be rejected because no SSN value is provided

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 only one table at a time. However, the deletion may propagate to tuples in other relations if referential triggered actions are specified in the referential integrity constraints of the DDL.

```
U4A:  DELETE FROM  EMPLOYEE
      WHERE        Lname='Brown';

U4B:  DELETE FROM  EMPLOYEE
      WHERE        Ssn='123456789';

U4C:  DELETE FROM  EMPLOYEE
      WHERE        Dno=5;

U4D:  DELETE FROM  EMPLOYEE;
```

If applied independently to the database will delete zero, one, four, and all tuples, respectively, from the EMPLOYEE relation.

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.

An additional SET clause in the UPDATE command specifies the attributes to be modified and their new values.

Eg:

to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use U5:

```
U5:  UPDATE  PROJECT
     SET     Plocation = 'Bellaire', Dnum = 5
     WHERE   Pnumber=10;
```

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary, as shown in U6. In this request, the modified Salary value depends on the original Salary value in each tuple, so two references to the Salary attribute are needed. In the SET clause, the reference to the Salary attribute on the right refers to the old Salary value before modification, and the one on the left refers to the new Salary value after modification:

```
U6:     UPDATE      EMPLOYEE
        SET         Salary = Salary * 1.1
        WHERE       Dno = 5;
```

DROP COMMAND

DROPTABLE DEPENDENT CASCADE;

Ifthe RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is not referenced in any constraints (for example, 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.

One can also drop a schema. For example, 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 is chosen in place of CASCADE, the schema is dropped only if ithasnoelements in it; otherwise, the DROP command will not be executed.

THE ALTER COMMAND

For example, to add an attribute for keeping track of jobs of employees tothe EMPLOYEE base relations in the COMPANY schema

ALTER TABLE COMPANY.EMPLOYEE ADD JOB VARCHAR(12);

For example, the following command removes the attribute ADDRESS from the EMPLOYEE base table:

ALTER TABLE COMPANY.EMPLOYEE DROP ADDRESS CASCADE;

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause

ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN DROP DEFAULT;

ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN SET DEFAULT "333445555";

For example, to drop the constraint named EMPSUPERFK in Figure 8.2 from the EMPLOYEE relation, we write:

ALTER TABLE COMPANY.EMPLOYEE DROP CONSTRAINT EMPSUPERFK CASCADE;

VIEWS

- A view in SQL terminology is a single table that is **derived** from other tables.
- These other tables could be **base tables** or previously defined views.
- Think of a view as a way of specifying a table that we need to **reference frequently**, even though it may not exist physically.
- In SQL, the command to specify a view is CREATE VIEW.
- The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view.
- If none of the view attributes results from applying functions or arithmetic operations, we do not have to specify **attribute names for the view**, since they would be the **same as the names of the attributes of the defining tables** in the **default case.**

```
V1:   CREATE VIEW   WORKS_ON1
      AS SELECT     Fname, Lname, 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;
```

Figure 5.2
Two views specified on the database schema of Figure 3.5.

WORKS_ON1

Fname	Lname	Pname	Hours

DEPT_INFO

Dept_name	No_of_emps	Total_sal

Figure : Two views specified

- In V1, we did not specify any new attribute names for the view WORKS_ONI (although we could have); in this case, WORKS_ONI inherits the names of the view attributes from

the defining tables EMPLOYEE, PROJECT, and WORKS_ON.

- View V2 explicitly specifies new attribute names for the view DEPT_INFO, using a one-to-one correspondence between the attributes specified in the CREATE VIEW clause and those specified in the SELECT clause of the query that defines the view.
- If we do not need a view any more, we can use the DROP VIEW command to dispose of it.
- For example, to get rid of the view V1, we can use the SQL statement in V1A:

```
V1A:    DROP VIEW      WORKS_ON1;
```

An efficient strategy for **automatically updating** the **view table** when the **base tables** are **updated** must be developed in order to keep the view up to date.

- Techniques using the concept of ***incremental update*** have been developed for this purpose, where it is determined what new tuples must be inserted, deleted, or modified in a materialized view table when a change is applied to one of the defining base tables.
- The view is generally kept as long as it is being queried.
- If the view is **not queried for a certain period of time**, the system may **then automatically remove the physical view table** and recompute it from scratch when future queries reference the view.
- A **view** with a **single defining table is updatable** if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint that do not have default values specified.
- **Views** defined on multiple tables using **joins** are generally not updatable.
- **Views** defined using **grouping and aggregate functions** are not updatable.

ASSERTIONS

- Users can specify **general constraints**-those that do not fall into any of the categories like primary constraint, referential integrity constraint ,domain constraint **via declarative assertions**, using the CREATE ASSERTION statement of the DDL.

create assertion assertion-name **check** predicate

- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- **For example**, to specify the constraint that "the salary of an employee must not be greater than the salary of the manager of the department that the employee works for" in SQL, we can write the following assertion:

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS  ( SELECT     *
                      FROM       EMPLOYEE E, EMPLOYEE M,
                                 DEPARTMENT D
                      WHERE      E.Salary>M.Salary
                                 AND E.Dno=D.Dnumber
                                 AND D.Mgr_ssn=M.Ssn ) );
```

- The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a condition in parentheses that must hold **true** on every database state for the assertion to be **satisfied.**The constraint name can be used later to refer to the constraint or to modify or drop it.
- Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.
- The basic technique for writing such assertions is to specify a query that selects any tuples that violate the desired condition.
- By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be **empty**. Thus, the assertion is violated if the result of the query is not empty.
- In our example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.

Eg: Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.

Ensuring every loan customer keeps a minimum of $1000 in an account.

Two assertions mentioned above can be written as follows.

Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.

create assertion *sum-constraint* **check**(**not exists** (**select * from** *branch* **where** (**select sum**)*amount*) **from** *loan* **where** (*loan.bname = branch.bname* >=(**select sum**)*amount*) **from** *account* **where** (*account.bname = branch.bname*)))

Ensuring every loan customer keeps a minimum of $1000 in an account.

create assertion *balance-constraint* **check** (**not exists** (**select * from** *loan L*(**where not exists** (**select *from** *borrower B, depositor D, account A* **where** *L.loan# = B.loan#* **and** *B.cname = D.cname* **and** *D.account# = A.account#* **and** *A.balance* >= 1000)))

When an assertion is created, the system tests it for validity.

If the assertion is valid, any further modification to the database is allowed only if it does not cause that assertion to be violated.

This testing may result in significant overhead if the assertions are complex. Because of this, the **assert** should be used with great care.

Some system developer omits support for general assertions or provides specialized form of assertions that are easier to test.

TRIGGERS

- It may be useful to specify a condition that ,if violated, causes some user to be informed of the violation.
- For example, A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs.
- The action that the DBMS must take in this case is to send an appropriate message to that user.
- The CREATE TRIGGER statement is used to implement such actions in SQL.

COMPONENTS OF TRIGGERS

- event (inserting, changing)-(**before/after**)

These events are specified after the keyword BEFORE , which means that the trigger should be executed before the triggering operation is executed. An alternative is to use AFTER , which specifies that the trigger should be execute after the operation specified in the event is completed.

- condition(**when**)

The condition that determines whether the rule action should be executed.Once the triggering event has occurred, an *optional* condition may be evaluated. If a condition is specified, it is first evaluated, and only if it evaluates to true will the rule action be executed. The condition is specified in the **when** clause of the trigger.

- action(**rollback,update**)

The action is usually sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
        {INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.

- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will be executed. The INSTEAD OF clause is used for creating trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} − This specifies the DML operation.
- [OF col_name] − This specifies the column name that will be updated.
- [ON table_name] − This specifies the name of the table associated with the trigger.
- [REFERENCING OLD AS o NEW AS n] − This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- WHEN (condition) − This provides a condition for rows for which the trigger would fire. This clause is valid only for row-level triggers.

Advantages of using SQL triggers

- SQL triggers provide an alternative way to check the integrity of data.
- SQL triggers can catch errors in business logic in the database layer.
- SQL triggers provide an alternative way to run scheduled tasks. By using SQL triggers, you don't have to wait to run the scheduled tasks because the triggers are invoked automatically *before* or *after* a change is made to the data in the tables.
- SQL triggers are very useful to audit the changes of data in tables.

Disadvantages of using SQL triggers

- SQL triggers only can provide an extended validation and they cannot replace all the validations. Some simple validations have to be done in the application layer. For example, you can validate user's inputs in the client side by using JavaScript or on the server side using server-side scripting languages such as JSP, PHP, ASP.NET, Perl.
- SQL triggers are invoked and executed invisible from the client applications, therefore, it is difficult to figure out what happens in the database layer.
- SQL triggers may increase the overhead of the database server.

Example 1

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
        select time_slot_id
        from time_slot)) /* time_slot_id not present in time_slot */
begin
   rollback
end;
```

Timeslot_check1 is the name of the TRIGGER

Section and time_slot are the 2 TABLES . Both tables contain time_slot_id as the ATTRIBUTES.

Time_slot_id act as foreign key in the table *section* whereas primary key in the table *time_slot*

Trigger gets activates when foreign key(time_slot_id) in *section* tries to enter values not in primary key (time_slot_id) of time_slot since it violates referential integrity. If it violates ,rollback happens.

The first trigger definition in the figure specifies that the trigger is initiated after any insert on the relation section and it ensures that the time slot id value being inserted is valid.

An SQL insert statement could insert multiple tuples of the relation, and **the for each row** clause in the trigger code would then explicitly iterate over each inserted row.

The **referencing new row as** clause creates a variable **nrow**(called a transition variable)that **stores** the **value of an inserted row after the insertion**.

The **when** statement specifies a condition. The system executes the rest of the trigger body only for tuples that satisfy the condition.

Example 2

```
create trigger timeslot_check2 after delete on timeslot
referencing old row as orow
for each row
when (orow.time_slot_id not in (
        select time_slot_id
        from time_slot) /* last tuple for time_slot_id deleted from time_slot */
  and orow.time_slot_id in (
        select time_slot_id
        from section)) /* and time_slot_id still referenced from section*/
begin
  rollback
end;
```

Timeslot_check2 is the name of the TRIGGER

Section and time_slot are the 2 TABLES . Both tables contain time_slot_id as the ATTRIBUTES.

Time_slot_id act as foreign key in the table ***section*** whereas primary key in the table ***time_slot***

Trigger gets activates when primary key (time_slot_id) of time_slot tries to delete some of its value and foreign key(time_slot_id) of *section* still have references for this deleted values in its table(*section*),because it violates referential integrity .If it violates ,rollback happens.

Example 3

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
   and (orow.grade = 'F' or orow.grade is null)
begin atomic
   update student
   set tot_cred= tot_cred+
           (select credits
            from course
            where course.course_id= nrow.course_id)
   where student.id = nrow.id;
end;
```

Table❼takes, studentAttribute❼grade

Trigger activates when *grade* gets updated, which in turn updates the total credits.